Object Oriented Programming Concepts

Prepared using following Resources:

- Herbert Schildt, “Java: The Complete Reference”, Tata McGrawHill Education
- E Balagurusamy, Programming with Java - A Tata McGraw Hill Education
- https://www.geeksforgeeks.org/java/
- https://www.javatpoint.com/java-tutorial
- https://www.tutorialspoint.com/java/index.htm
- https://www.w3schools.com/java/

By: DIVAKARA .N

Object Oriented Concepts

- Background
- Programming Paradigms
- Concepts of OOPL
- Major and Minor elements
- Class, Object and relationships among objects
- Encapsulation
- Polymorphism
- Inheritance
- Message passing
- Difference between OOP and other conventional programming
- Advantages, Disadvantages and Applications

Object Oriented Concepts

Background:

- The process problem solving using a computer is complicated process requiring careful planning, logical precision, persistence and attention to detail.
- A programmers primary task is to write software to solve a problem.
- Many programming models have evolved to help programmers in being more effective such as Modular, Top-down, Bottom-up, Structured, Object-Oriented programs etc.

Object Oriented Concepts

Programming Paradigms:

- All computer programs consists of two elements:
  Code and
  Data
- A program can be conceptually organised around its code or around its data.
- ***Major Programming Paradigms are:***
  Procedure Oriented
  Object Oriented

Object Oriented Concepts

Programming Paradigms: … POP

- The interdependent functions cannot be used in other programs. As a result, even for a similar task across programs, the entire function has to be recoded. This made program development a more complex task.
- A change means rewriting huge portions of the code. As a result of this, software maintenance costs are very high.
- This approach failed to show the desired result in terms bug free, easy-to-maintain and reusable programs.

Object Oriented Concepts

Programming Paradigms: … OOP

- An approach that provides a way of modularizing programs by creating partitioned memory area for both data and code that can be used as templates for creating copies of such modules on demand.
- OOP is a programming methodology that helps organizing complex programs through the use of the three principles – ***Encapsulation, Polymorphism and Inheritance.***
- OOP enables you to consider a real-world entity as an object.
- OOP combines user-defined data and instructions into a single entity called an ***Object.***

Object Oriented Concepts

Programming Paradigms: … OOP…

It is a well suited paradigm for the following:

- Modelling the real world problem as close as possible to the perspective of the user.
- Constructing reusable software components and easily extendable libraries.
- Easily modifying and extending implementations of components without having to recode everything from scratch.

Object Oriented Concepts

Concepts of OOP Languages

- Classes
- Objects
- Encapsulation
- Polymorphism
- Inheritance
- Dynamic Binding
- Message Communication

Object Oriented Concepts

Concepts of OOP Languages…

Classes: *"An user-defined data type/ADT"*

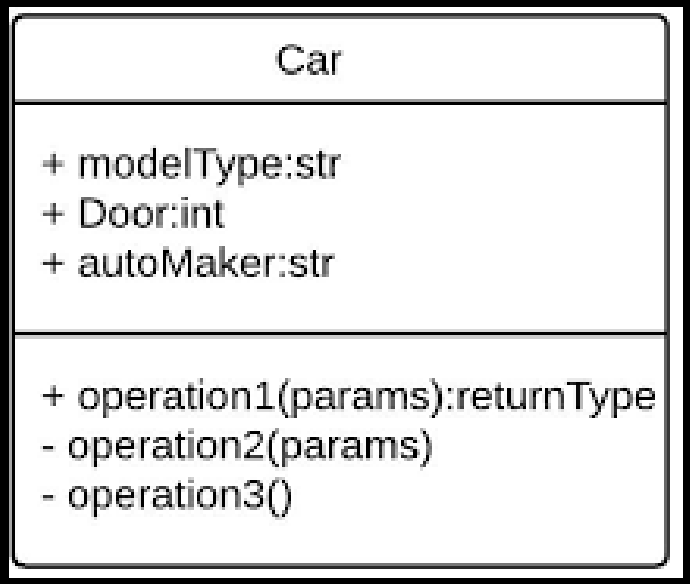
- A blue print used to instantiate many objects.
- Defines set of data members (States/Attributes) and set of methods (Behaviors).
- A ***template*** for an object / Collection of objects of similar type.
- No memory is allocated when a class is created. Memory is allocated only when an object is created, i.e., when an instance of a class is created.

Object Oriented Concepts

Concepts of OOP Languages…

Classes: …

- **Eg:** A Class of **Cars** under which Santro Xing, Alto and WaganR represents individual Objects. In this context each Car Object will have its own, ***Model, Year of Manufacture, Colour, Top Speed, Engine Power etc., which form Properties of the Car class*** and the associated actions i.e., object functions like ***Start, Move, Stop form the Methods*** of Car Class.



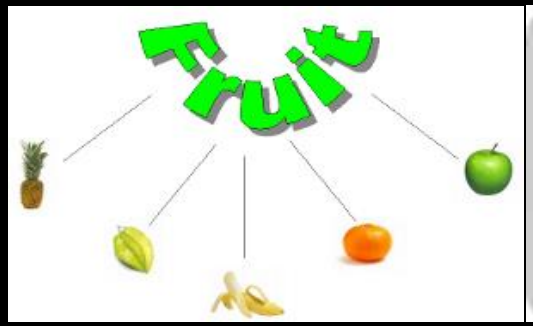
Object Oriented Concepts

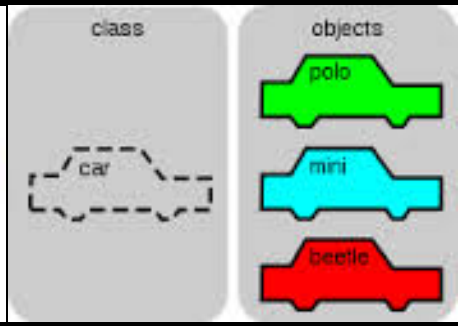
Concepts of OOP Languages…

Objects: *"Instances"*

- The basic Building Blocks/Run-time Entities having two characteristics: ***State and Behavior***.
- An Object is a collection of data members and associated member functions also known as methods.
- Each instance of an object can hold its own relevant data.
- Memory is allocated only when an object is created, i.e., when an instance of a class is created.
- **Eg:** Employee emp1; emp1 = new Employee(); ***OR***

  Employee emp1 = new Employee();



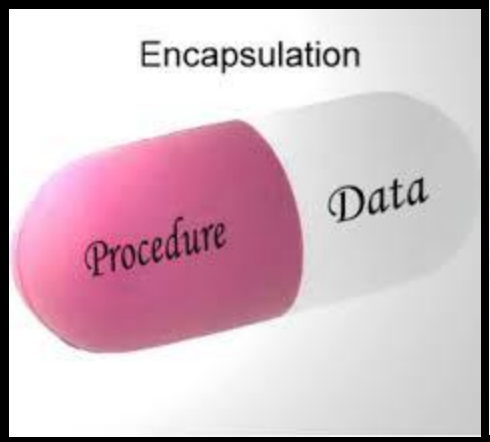


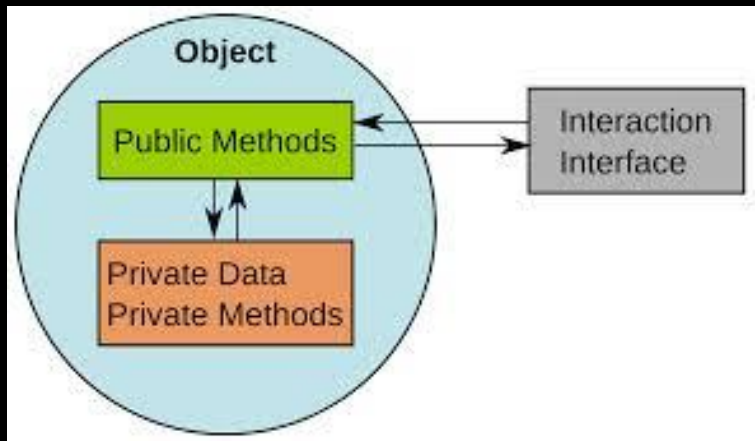
Object Oriented Concepts

Concepts of OOP Languages…

Encapsulation: *"Wraps Data and Method"*

- The mechanism that binds together Code and the Data it manipulates.
- Keeps both safe from outside interference and misuse.
- The wrapping up of data and methods into a single unit called ***class***, access is tightly controlled through a well defined interfaces.





Object Oriented Concepts

Concepts of OOP Languages…

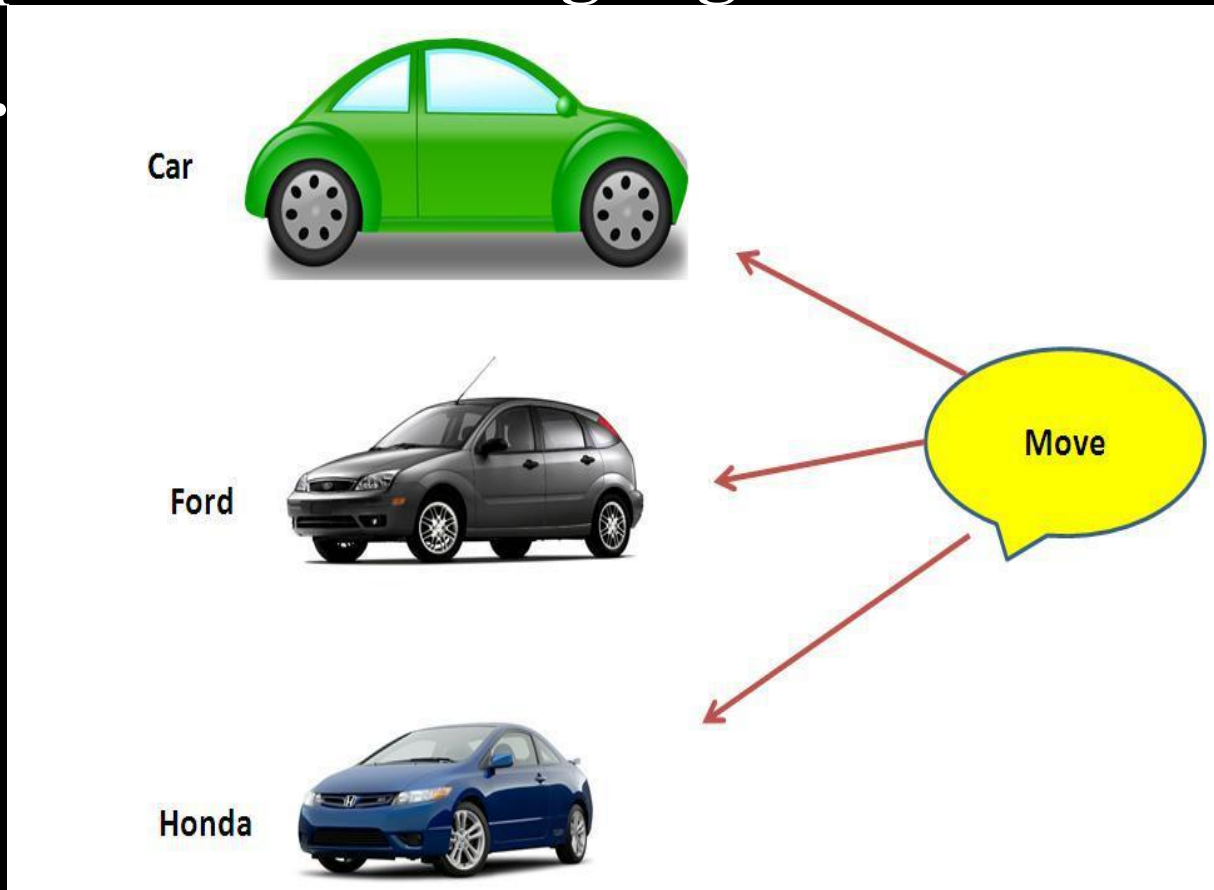
Polymorphism: *"ability to take more than one form"*

- A feature that allows one interface to be used for a general class of actions.
- It allows an object to have different meanings, depending on its context.
- ***"One Interface, Multiple Methods"*** to design a generic interface to a group of related activities.
- **Eg:** Coffee day vending machine

Object Oriented Concepts

Concepts of OOP Languages…

Polymorphism: …



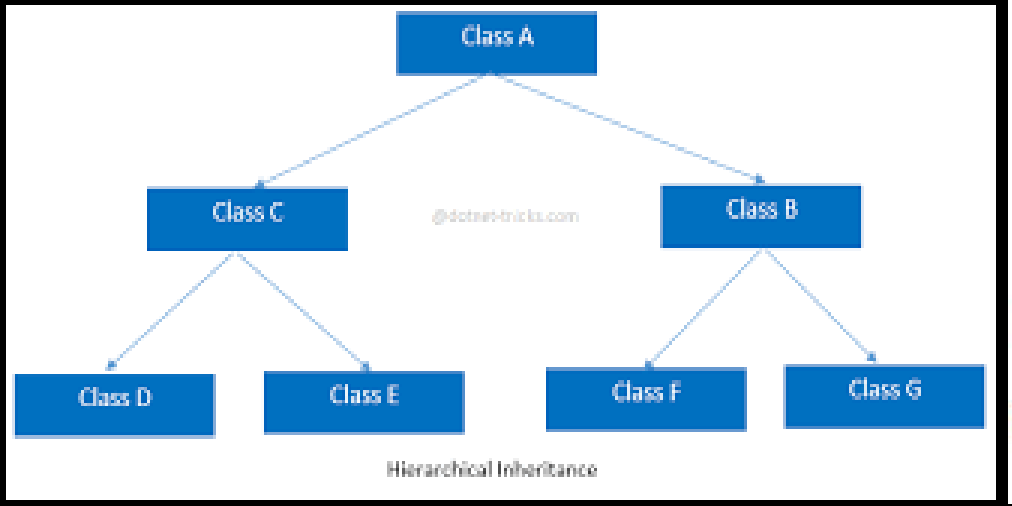
- Car uses normal engine to move
- Ford uses V engine to move
- Honda uses i-vtec technology to move

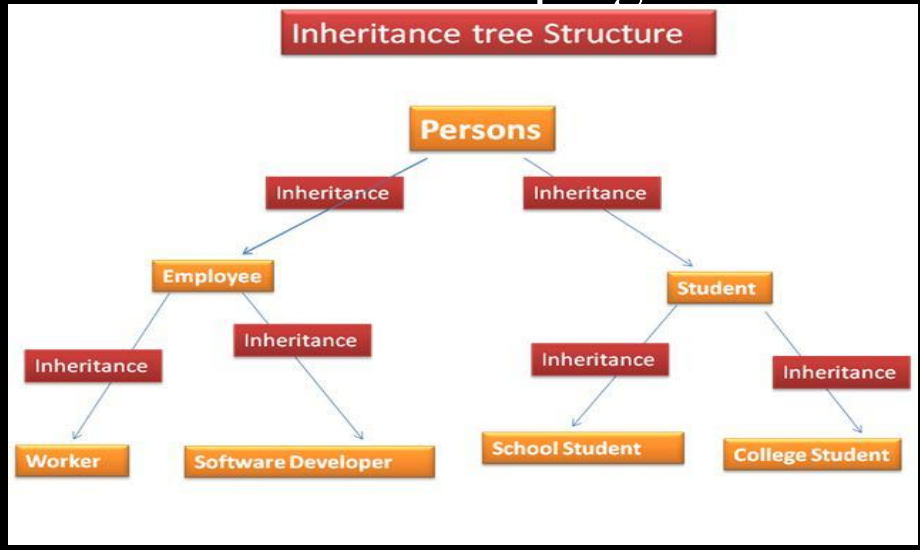
Object Oriented Concepts

Concepts of OOP Languages…

Inheritance: *"The Idea of Reusability"*

- The process by which one object acquires the properties of another object **OR** the process of forming a new class from an existing class or base class.
- Supports the concepts of hierarchical classifications.
- Inheritance helps in reducing the overall code size of the program.





Object Oriented Concepts

Concepts of OOP Languages…

Dynamic Binding: *"Associated with the concept of Inheritance and Polymorphism"*

- Binding means link between procedure call and code to be execute.
- It is the process of linking of a function call to the actual code of the function at run-time.
- That is, in dynamic binding, the actual code to be executed is not known to the compiler until run-time.
- It is also known late binding.

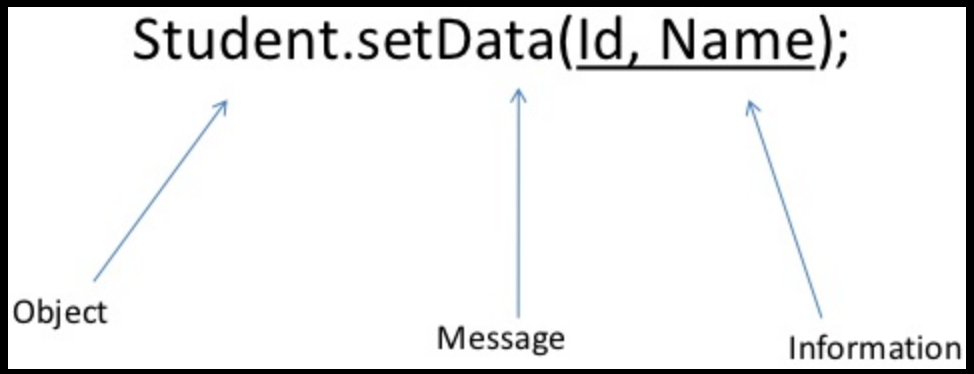
- At run-time the code matching the object under current reference will be called.

Object Oriented Concepts

Concepts of OOP Languages…

Message Communication: *"A request for execution of a method"*

- Objects communicate with one another by sending and receiving information.
- A message of an object is a request for execution of a procedure/method that generates desired result.
- **Eg:** ***student1.examResult(4JC10CS901);***



Object Oriented Concepts

Difference between OOP and other conventional programming

Procedure Oriented	Object Oriented
Emphasis is on procedure rather than data, characterises a program as a series of linear steps.	Emphasis is on data, data is hidden and cannot be accessed by external functions.
Process-centric approach	***Data-centric approach***
Programs are written around "What is happening" – Code acting on data.	Programs are written around "Who is being affected" – Data controlling access to code.
Programs are divided into smaller parts called functions/modules.	Programs are divided into objects, may communicate with each other through methods.
Function can call one from anther and difficult to separate due to interdependency between modules.	Objects are independent used for different programs.
Follows Top-down approach in program design.	Follows Bottom-up approach.
Eg: Basic, Pascal, COBOL, Fortran, C, etc.	**Eg:** Smalltalk, C++, Objective C, Ada, Objective Pascal, Java(Pure OOL), etc.

Object Oriented Concepts

Advantages and Disadvantages

Advantages:

Simplicity: software objects model ***real world objects***, so the complexity is reduced and the program structure is very clear.

Modularity: each object forms a separate entity whose internal workings are decoupled from other parts of the system.

Modifiability: it is easy to make minor changes in the data representation or the procedures in an OO program. Changes inside a class do not affect any other part of a program, since the only public interface that the external world has to a class is through the use of methods.

Object Oriented Concepts

Advantages and Disadvantages

Advantages:...

Extensibility(Resilience – System can be allowed to evolve): adding new features or responding to changing operating environments can be solved by introducing a few new objects and modifying some existing ones.

Maintainability: objects can be maintained separately, making locating and fixing problems easier.

Re-usability: objects can be reused in different programs.

Design Benefits: Large programs are very difficult to write. OOPs force designers to go through an extensive planning phase, which makes for better designs with less flaws. In addition, once a program reaches a certain size, Object Oriented Programs are actually easier to program than non-Object Oriented ones.

Object Oriented Concepts

Advantages and Disadvantages…

Disadvantages:

- **Size:** Programs are much larger than other programs.
- **Effort:** Require a lot of work to create, coders spent more time actually writing the program.
- **Speed:** Slower than other programs, partially because of their size also demand more system resources.
- Not all programs can be modelled accurately by the objects model.
- One programmer's concept of what constitutes an abstract object might not match the vision of another programmer.

However; many novice programmers do not like Object Oriented Programming because of the great deal of work required to produce minimal results.

Object Oriented Concepts

Applications:

- **User-Interface Design** (CUI/CLI, GUI...WIN), Games, CAD/CAM/CIM systems
- **Real-Time System**
- **Simulation and Modeling**
- **Object-Oriented Database**
- **Artificial Intelligence and Expert Systems**
- **Neural Networks and Parallel Programs**
- **Decision Support and Office Automation System**

A software that is easy to use hard to build. OOP changes the way software engineers will Think, Analyse, Design and Implement systems in the future.

Overview of Java

Prepared using following Resources:

- Herbert Schildt, “Java: The Complete Reference”, Tata McGrawHill Education
- E Balagurusamy, Programming with Java - A Tata McGraw Hill Education
- https://www.geeksforgeeks.org/java/
- https://www.javatpoint.com/java-tutorial
- https://www.tutorialspoint.com/java/index.htm
- https://www.w3schools.com/java/

By: DIVAKARA .N

Object Oriented Programming using Java

- The History and Evolution
- Overview and basic concepts
- Features, Advantages and Applications
- Java Development Kit (JDK) and JVM
- Simple Java programs
- Data types and Variables, dynamic initialization , the scope and lifetime of variables
- Type conversion and casting
- Operators and Expressions: Operator Precedence, Logical expression, Access specifiers
- Control statements & Loops and
- Arrays

Object Oriented Programming using Java

The History and Evolution:

- Java is a blend of the best elements of its rich heritage combined with the innovative concepts required by its unique mission.
- Although Java has become inseparably linked with the online environment of the Internet, it is important to remember that Java is first and foremost a programming language.
- Computer language innovation and development occurs for two fundamental reasons:
  - To adapt to changing environments and uses
  - To implement refinements and improvements in the art of programming

Object Oriented Programming using Java

The History and Evolution: ...

- Java is related to C++, which is a direct descendant of C.
- Much of the character of Java is inherited from these two languages.
  - From C, Java derives its syntax.
  - Object-oriented features were influenced by C++.

By the end of the 1980s and the early 1990s, object-oriented programming using C++ took hold. Indeed, for a brief moment it seemed as if programmers had finally found the perfect language. Because C++ blended the high efficiency and stylistic elements of C with the object-oriented paradigm, it was a language that could be used to create a wide range of programs.

Object Oriented Programming using Java

The History and Evolution: ...

- However, just as in the past, forces were brewing that would, once again, drive computer language evolution forward. ***Within a few years, the World Wide Web and the Internet would reach critical mass. This event would bring about abruptly another revolution in programming.***
- Java was conceived by ***James Gosling***, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan ***at Sun Microsystems, Inc. in 1991***. It was initially called "Oak" but was renamed "Java" in 1995.
- Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lindholm were key contributors to the maturing of the original prototype.

Object Oriented Programming using Java

The History and Evolution: ...

- Original impetus for Java was not the Internet! Instead, the primary motivation was the need for a platform-independent (i.e., architecture-neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls.
- Gosling and others began work on a portable, platform-independent language that could be used to produce code that would run on a variety of CPUs under differing environments.
- Java was not designed to replace C++. Java was designed to solve a certain set of problems. C++ was designed to solve a different set of problems.

Object Oriented Programming using Java

The History and Evolution: ...

- Today, with technology such a part of our daily lives, we take it for granted that we can be connected and access applications and content anywhere, anytime. Because of Java, we expect digital devices to be smarter, more functional, and way more entertaining.
- Today, Java not only permeates the Internet, but also is the invisible force behind many of the applications and devices that power our day-to-day lives.
- From mobile phones to handheld devices, games and navigation systems to e-business solutions, ***Java is everywhere!***

Object Oriented Programming using Java

The History and Evolution: ...

- The first version of Java Development Kit, The JDK 1.0 was released on January 23, 1996. From the few hundred class files the JDK 1.0 had, it has now grown dramatically into more than 3000 classes in J2SE 5.0, J2SE6, . . .
- Now Java is extensively used to program stand alone applications to Multi tier web applications.
  - ***JDK 1.0 (January 23, 1996)***
  - ***JDK 1.1 (February 19, 1997)***
  - ***J2SE 1.2 (December 8, 1998)***
  - ***J2SE 1.3 (May 8, 2000)***
  - ***J2SE 1.4 (February 6, 2002)***
  - ***J2SE 5.0 (September 30, 2004)***
  - ***Java SE 6 (December 11, 2006)***
  - ***Java SE 7 (July 28, 2011)***
  - ***Java SE 8 (March 18, 2014)***

Object Oriented Programming using Java

The History and Evolution: ...*Brief Summary*

- In 1990, Sun Microsystems initiated a team to develop software for consumer electronics devices headed by ***James Gosling.***
- In 1991, the team announced the ***"Oak"*** from C++
- In 1992, GreenProject team by Sun demonstrated the application to home appliances.
- In 1993, Transformation of text-based internet into a graphical-rich environment. (Applets)
- In 1994, HotJava – web browser to locate & run applets
- In 1995, Oak renamed as Java.
- In 1996, Leader as General Purpose Programming Language / OOPL.

Object Oriented Programming using Java

Overview and Basic Concepts:

- Originally developed by Sun Microsystems, initiated by James Gosling. With the advancement of Java and its widespread popularity, multiple configurations were built to suite various types of platforms. ***Ex:*** J2EE for Enterprise Applications, J2ME for Mobile Applications. Sun Microsystems has renamed the new J2 versions as Java SE, Java EE and Java ME, respectively.
- Java is guaranteed to be ***Write Once, Run Anywhere***.
- Java is easy to learn. Java was designed to be easy to use and is therefore easy to write, compile, debug, and learn than other programming languages.
- Java is object-oriented. This allows you to create modular programs and reusable code.
- Java is platform-independent.

Object Oriented Programming using Java

Features: - ***The Java Buzzwords***

The key considerations were summed up by the Java team in the following list of buzzwords:

Simple, Secure, Portable, Object-oriented, Robust, Multithreaded, Architecture-neutral, Interpreted, High performance, Distributed and Dynamic

Object Oriented Programming using Java

Features: - *The Java Buzzwords...*

- **Simple**: Java was designed to be easy for the professional programmer to learn and use effectively. It contains many features of other Languages like c and C++ and Java Removes Complexity because it doesn't use pointers.
- **Secure:** Java achieved protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer.
- **Portable:** The same code must work on all computers. Being architectural-neutral and having no implementation dependent aspects of the specification makes Java portable.
- **Object-oriented**: In Java, everything is an Object. Java can be easily extended since it is based on the Object model.

Object Oriented Programming using Java

Features: - *The Java Buzzwords...*

- **Robust:** Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking. Also auto garbage collection mechanism.
- **Multithreaded:** Enables us to write programs that can do many tasks simultaneously/concurrently. This design feature allows developers to construct smoothly running interactive applications.
- **Architecture-neutral:** A central issue for the Java designers was that of code longevity and portability. It follows 'Write-once-run-anywhere' ***WORA*** approach. To a great extent, this goal was accomplished.
- **Interpreted**: Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine (JVM).

Object Oriented Programming using Java

Features: - *The Java Buzzwords...*

- **High performance:** Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time (JIT) compiler.
- **Distributed:** Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. Also supports Remote Method Invocation (RMI) feature it enables a program to invoke methods across a network.
- **Dynamic:** It is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

Object Oriented Programming using Java

Advantages:

- ***Write Once, Run Anywhere*** - You only have to write your application once--for the Java platform--and then you'll be able to run it anywhere. Java support is becoming ubiquitous.
- ***Security -*** Allows users to download untrusted code over a network and run it in a secure environment in which it cannot do any harm: it cannot infect the host system with a virus, cannot read or write files from the hard drive, and so forth. This capability alone makes the Java platform unique.
- ***Network-centric Programming -*** Java makes it unbelievably easy to work with resources across a network and to create network-based applications using client/server or multitier architectures. ***"The network is the computer"***

Object Oriented Programming using Java

Advantages: ...

- ***Dynamic, Extensible Programs -*** Java application can dynamically extend itself by loading new classes over a network.
- ***Internationalization -*** Java uses 16-bit Unicode characters that represent the phonetic alphabets and ideographic character sets of the entire world.
- ***Performance -*** The VM has been highly tuned and optimized in many significant ways. Using sophisticated JIT compilers, Java programs can execute at speeds comparable to the speeds of native C and C++ applications.
- ***Programmer Efficiency and Time-to-Market -*** Java is an elegant language combined with a powerful and well-designed set of APIs.

Object Oriented Programming using Java

Applications:

- ✓ Standalone/Desktop/Console –based Applications
  - Stand alone CUI/GUI based applications
- ✓ Web Applications
  - Applets and Servlets
- ✓ Distributed Applications
  - Database applications
- ✓ Client/Server Applications

Object Oriented Programming using Java

DIFFERENCES BETWEEN C , C++ AND JAVA

- C Uses header Files but java uses Packages
- C Uses Pointers but java doesn't supports pointers .
- Java doesn't supports storage classes like auto, external etc.
- The Code of C Language is Converted into the Machine code after Compilation But in Java Code First Converted into the Bytes Codes then after it is converted into the Machine Code.
- C++ supports Operator Overloading but java doesn't Supports Operator Overloading .
- In C++ Multiple Inheritance is Possible but in java A Class Can not Inherit the features from the two classes in other words java doesn't supports Multiple Inheritance The Concept of Multiple Inheritances is Introduced in the Form of Interfaces.
- Java Uses import statement for including the contents of screen instead of #include.
- Java Doesn't uses goto.
- Java Doesn't have Destructor like C++ Instead Java Has finalize Method.
- Java Doesn't have Structure Union , enum data types.

Object Oriented Programming using Java

DIFFERENCES BETWEEN C++ and JAVA

C++ and Java both are Object Oriented Languages but some of the features of both languages are different from each other. Some of these features are:

- All stand-alone C++ programs require a function named main and can have numerous other functions, including both stand-alone functions and functions, which are members of a class. There are no stand-alone functions in Java. Instead, there are only functions that are members of a class, usually called methods. Global functions and global data are not allowed in Java.
- All classes in Java ultimately inherit from the Object class. This is significantly different from C++ where it is possible to create inheritance trees that are completely unrelated to one another.
- Java does not support multiple inheritance. To some extent, the interface feature provides the desirable features of multiple inheritance to a Java program without some of the underlying problems.
- Java does not support operator overloading.

Object Oriented Programming using Java

Java Runtime Environment (JRE):

- The smallest set of executables and files that constitute the standard java plotform.
- It provides the libraries, the JVM, and other components to run applets and applications written in the Java programming language.
- In addition, two key deployment technologies are part of the JRE: ***Java Plug-in***, which enables applets to run in popular browsers; and ***Java Web Start***, which deploys standalone applications over a network.

Object Oriented Programming using Java

Java Development Kit (JDK):

- The Java Development Kit (JDK) is a package that includes a large number of development tools and hundreds of classes, Java Standard library APIs and methods.
- The JDK is developed by Sun Microsystem's. it contain JRE and development tools.
- JDK provides tools for users to integrate and execute applications and Applets.
- **Eg.** javac, java, javap, jdb, javah, javadoc, appletviwer …

Object Oriented Programming using Java

Java Development Kit (JDK): ...

- **javac -** The Java Compiler, it compiles Java source code into Java bytecodes.

  javac filename.java

- **java -** The Java Interpreter, it executes Java class files created by a Java compiler.

  java classfilename

- **javap -** The Java Class File Disassembler, Disassembles class files. Its output depends on the options used.

  javap [options] classfilename

Object Oriented Programming using Java

Java Development Kit (JDK): ...

- **jdb -** The Java Debugger, helps you find and fix bugs in Java language programs.

  jdb [options]

- **javah -** C Header and Stub File Generator, produces C header files and C source files from a Java class. These files provide the connective glue that allow your Java and C code to interact.

  javah [options] classfilename

Object Oriented Programming using Java

Java Development Kit (JDK): ...

- **javadoc -** The Java API Documentation Generator Generates HTML pages of API documentation from Java source files, parses the declarations and documentation comments in a set of Java source files and produces a set of HTML pages.

  javadoc [options] [package | source.java]*

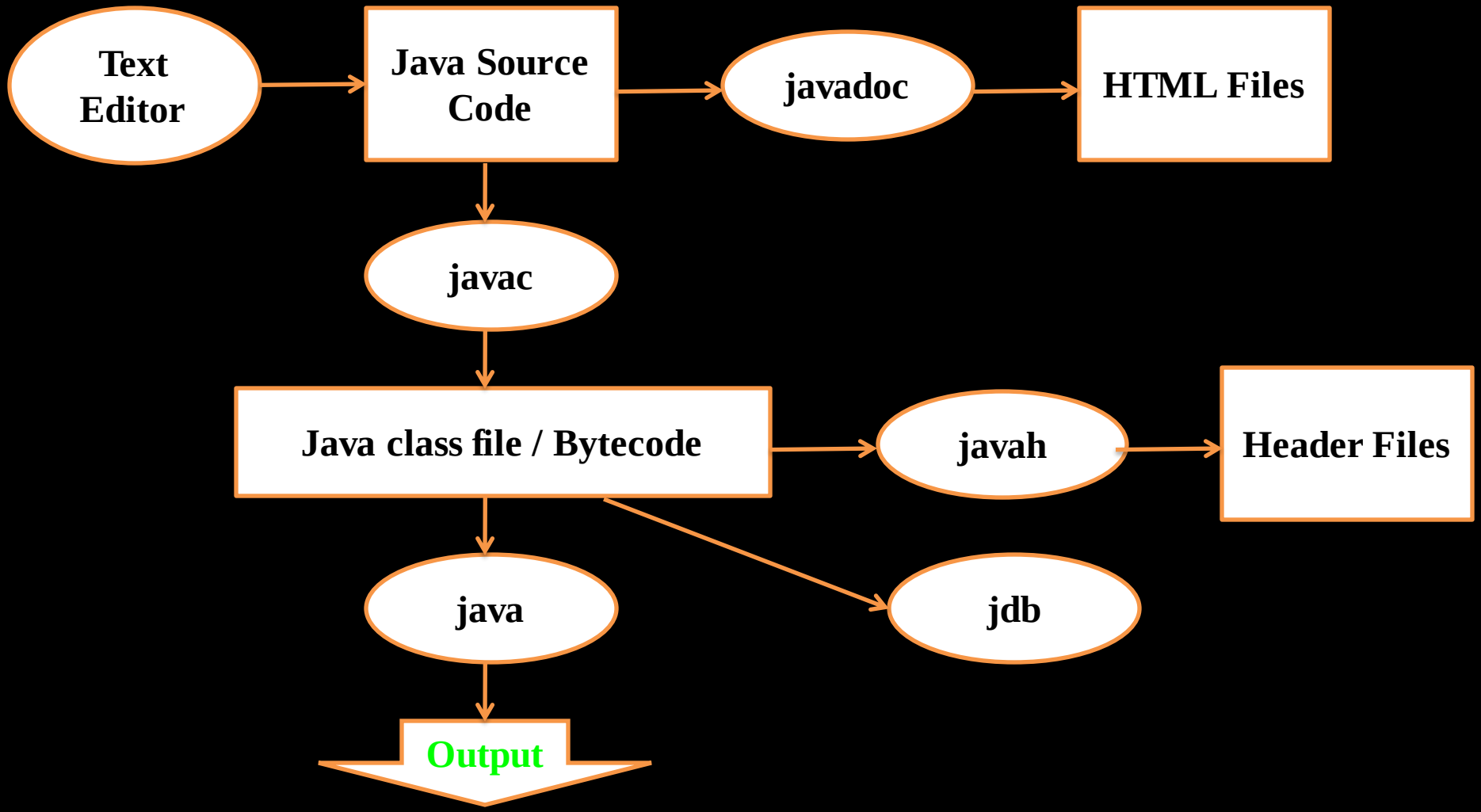
- **appletviewer -** The appletviewer command allows you to run applets outside of a web browser.

  appletviewer [options] filename.html [URLs]

Object Oriented Programming using Java

Java Development Kit (JDK): ...

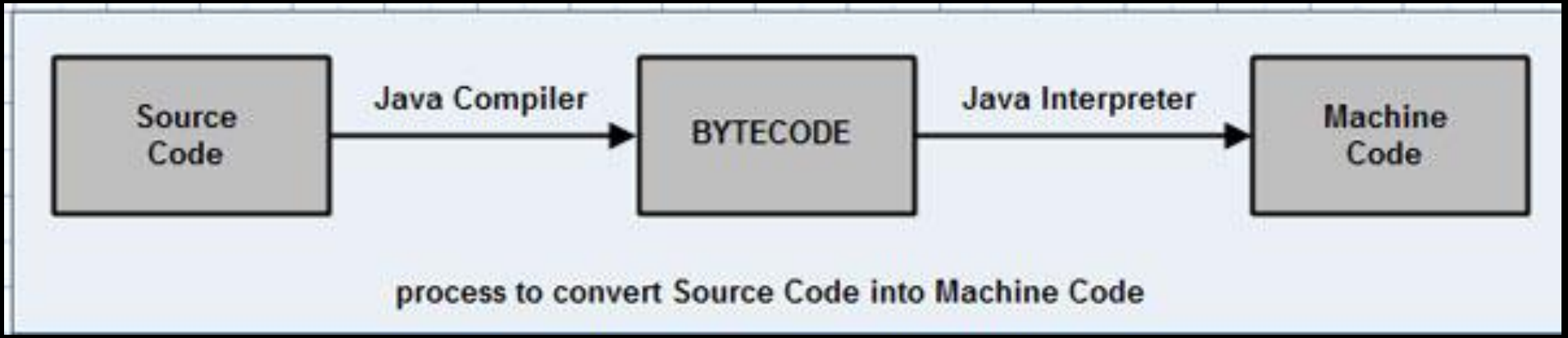
The Java Programming Environment



Object Oriented Programming using Java

Java Virtual Machine(JVM): - 'Simulated computer within the computer'

- JVM is the virtual machine that run the Java byte code. It's also the entity that allows Java to be a "portable language" (write once, run anywhere). Indeed there are specific implementations of the JVM for different systems (Windows, Linux, MacOS,..), the aim is that with the same byte code they all give the same results (***i.e. JVM is platform dependent***).



Object Oriented Programming using Java

Basic Structure of Java programs:

Documentation Section
Package Statements
Import Statements
Interface Statements
Class Definitions

main method class{
 	
//Definitions
 	
}

Object Oriented Programming using Java

Simple Java programs:

```
/*
        This is a simple Java program.
        Call this file "Example.java".
*/
class Example {
    // Your program begins with a call to main().
    public static void main(String args[]) {
        System.out.println("This is a simple Java program.");
    }
}
```

Object Oriented Programming using Java

Simple Java programs: ...

Entering the Program:

- For most computer languages, the name of the file that holds the source code to a program is immaterial. For this example, the name of the source file should be Example.java.
- In Java, a source file is officially called a compilation unit. It is a text file that contains one or more class definitions. The Java compiler requires that a source file use the .java filename extension.
- ***In Java, all code must reside inside a class. By convention, the name of that class should match the name of the file that holds the program.***
- You should also make sure that the capitalization of the filename matches the class name.
- The Java is case-sensitive. The convention that filenames correspond to class names may seem arbitrary. However, this convention makes it easier to maintain and organize your programs.

Simple Java programs: ...

Compiling the Program:

- To compile the Example program, execute the compiler, **javac**, specifying the name of the source file on the command line, as shown here:

C:\>javac Example.java

- *The **javac** compiler creates a file called Example.class that contains the bytecode version of the program. The Java bytecode is the intermediate representation of your program that contains instructions the Java Virtual Machine will execute. Thus, the output of **javac** is not code that can be directly executed.*
- To run the program, you must use the Java application launcher, called **java**

C:\>java Example

Simple Java programs: ...

Compiling the Program: ...

- When the program is run, the following output is displayed:

  This is a simple Java program.

- ***When Java source code is compiled, each individual class is put into its own output file named after the class and using the .class extension.*** This is why it is a good idea to give your Java source files the same name as the class they contain-the name of the source file will match the name of the .class file. When you execute java as just shown, you are actually specifying the name of the class that you want to execute. It will automatically search for a file by that name that has the .class extension. If it finds the file, it will execute the code contained in the specified class.

Object Oriented Programming using Java

Simple Java programs: ...

A Closer Look at the First Sample Program:

Example.java includes several key features that are common to all Java programs.
The program begins with the following lines:

```
/*
    This is a simple Java program.
    Call this file "Example.java".
*/
```

- This is a comment. Like most other programming languages, Java lets you enter a remark into a program's source file. The contents of a comment are ignored by the compiler. Instead, a comment describes or explains the operation of the program to anyone who is reading its source code.
- In real applications, comments generally explain how some part of the program works or what a specific feature does.

Object Oriented Programming using Java

Simple Java programs: ...

A Closer Look at the First Sample Program: ...

- Java supports three styles of comments.
- *//* Single line comment
- */* and end with */* A multiline comment
- */** documentation */* called doc comment, the JDK **javadoc** tool uses doc comments when preparing automatically generated documentation.
- Any comments are ignored by the compiler.

- **class Example {**
- This line uses the keyword **class** to declare that a new class is being defined. The entire class definition, including all of its members, will be between the opening curly brace ({) and the closing curly brace (}).

Object Oriented Programming using Java

Simple Java programs: ...

A Closer Look at the First Sample Program: ...

public static void main(String args[]) {

- The **main( )** method, at which the program will begin executing. All Java applications begin execution by calling main( ).
- The **public** keyword is an access specifier, which allows the programmer to control the visibility of class members. When a class member is preceded by **public**, then that member may be accessed by code outside the class in which it is declared.
- The opposite of **public** is **private**, which prevents a member from being used by code defined outside of its class.

Object Oriented Programming using Java

Simple Java programs: ...

A Closer Look at the First Sample Program: ...

public static void main(String args[]) {

- In this case, **main( )** must be declared as **public**, since it must be called by code outside of its class when the program is started.
- The keyword **static** allows main( ) to be called without having to instantiate a particular instance of the class. This is necessary since main( ) is called by the JVM before any objects are made.
- The keyword **void** simply tells the compiler that **main( )** does not return a value.

Object Oriented Programming using Java

Simple Java programs: ...

```
/*
   Here is another short example.
   Call this file "Example2.java".
*/
class Example2 {
   public static void main(String args[]) {
      int num; // this declares a variable called num
      num = 100; // this assigns num the value 100
      System.out.println("This is num: " + num);
      num = num * 2;
      System.out.print("The value of num * 2 is ");
      System.out.println(num);
   }
}
```

Object Oriented Programming using Java

Two Control Statements:

- **The if Statement:** Syntactically identical to the **if** statements in C, C++, and C#. Its simplest form is shown here:

if(condition) statement;

- Here, condition is a Boolean expression. If condition is true, then the statement is executed. If condition is false, then the statement is bypassed.

  Example: if(num < 100) System.out.println("num is less than 100");

- Java defines a full complement of relational operators which may be used in a conditional expression. Here are a few: **>, < and ==**

Object Oriented Programming using Java

Two Control Statements:

- **The if Statement: ...**

```
/*
  Demonstrate the if.

  Call this file "IfSample.java".
*/
class IfSample {
  public static void main(String args[]) {
    int x, y;

    x = 10;
    y = 20;

    if(x < y) System.out.println("x is less than y");

    x = x * 2;
    if(x == y) System.out.println("x now equal to y");

    x = x * 2;
    if(x > y) System.out.println("x now greater than y");

    // this won't display anything
    if(x == y) System.out.println("you won't see this");
  }
}
```

Object Oriented Programming using Java

Two Control Statements: ...

- **The for Loop:** Java supplies a powerful assortment of loop constructs. Perhaps the most versatile is the for loop. The simplest form of the for loop is shown here:

for(initialization; condition; iteration)
statement;

- In its most common form, the initialization portion of the loop sets a loop control variable to an initial value. The condition is a Boolean expression that tests the loop control variable.

```
/*
  Demonstrate the for loop.

  Call this file "ForTest.java".
*/
class ForTest {
  public static void main(String args[]) {
    int x;

    for(x = 0; x<10; x = x+1)
      System.out.println("This is x: " + x);
  }
}
```

Two Control Statements: ...

- **Using Blocks of Code:** Java allows two or more statements to be grouped into blocks of code, also called code blocks. This is done by enclosing the statements between opening and closing curly braces, it becomes a logical unit that can be used any place that a single statement can.
- **Eg:**

```
if(x < y) { // begin a block
   x = y;
   y = 0;
} // end of block
```

Two Control Statements: ...

- **Using Blocks of Code: ...**

```
/*
  Demonstrate a block of code.

  Call this file "BlockTest.java"
*/
class BlockTest {
  public static void main(String args[]) {
    int x, y;

    y = 20;

    // the target of this loop is a block
    for(x = 0; x<10; x++) {
      System.out.println("This is x: " + x);
      System.out.println("This is y: " + y);
      y = y - 2;
    }
  }
}
```

Object Oriented Programming using Java

Lexical Issues :

- Java programs are a collection of whitespace, identifiers, literals, comments, operators, separators, and keywords - *The atomic elements of Java*
-
- **Whitespace:** Java is a free-form language. In Java, whitespace is a space, tab, or newline.
- **Identifiers:** Used for class names, method names, and variable names. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. They must not begin with a number, again, Java is case-sensitive.

Object Oriented Programming using Java

Lexical Issues : ...

- **Identifiers:...**
  - ✓ AvgTemp, count, a4, $test, this_is_ok are Valid
  - ✓ 2count, high-temp, Not/ok are Invalid
- **Literals:** A constant value in Java is created by using a literal representation of it. It can be used anywhere a value of its type is allowed.
  - ✓ 100, 98.6, 'X', "This is a test"
- **Comments:** As mentioned, there are three types of comments defined by Java.

Lexical Issues : ...

- **Separators:**

Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a **for** statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.

Object Oriented Programming using Java

Lexical Issues : ...

- **The Java Keywords:** 50 keywords currently defined, these combined with the syntax of the operators and separators, form the foundation of the Java language. These keywords cannot be used as names for a variable, class, or method.
- The keywords **const** and **goto** are reserved but not used. In the early days of Java, several other keywords were reserved for possible future use.
- In addition to the keywords, Java reserves the following: **true, false,** and **null**. These are values defined by Java.
- You may not use these words for the names of variables, classes, and so on.

Object Oriented Programming using Java

Lexical Issues : ...

- **The Java Keywords: ...**

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Object Oriented Programming using Java

- The History and Evolution
- Overview and basic concepts
- Features, Advantages and Applications
- Java Development Kit (JDK) and JVM
- Simple Java programs
- Data types and Variables, dynamic initialization , the scope and lifetime of variables
- Type conversion and casting
- Operators and Expressions: Operator Precedence, Logical expression, Access specifiers
- Control statements & Loops and
- Arrays

Object Oriented Programming using Java

Data types and Variables

- The Java is a strongly typed language, part of Java's safety and robustness.
- First, every variable has a type, every expression has a type, and every type is strictly defined.
- Second, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.
- There are no automatic coercions or conversions of conflicting types as in some languages.
- The Java compiler checks all expressions and parameters to ensure that the types are compatible. Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.

Object Oriented Programming using Java

Data types and Variables ...

The domains which determine what type of contents can be stored in a variable. In Java, there are two types of data types:

- ***Primitive /Simple data types:*** Defines eight types of data: *byte, short, int, long, char, float, double, and boolean*.
- ***Reference data types: or Abstract datatypes***

  arrays, objects, interfaces, enum, Srting etc.

Object Oriented Programming using Java

Data types and Variables ...

Type	Length	Min. Value	Max. Value
byte	8 bits	-128	127
short	16 bits	-32768	32767
int	32 bits	-2,147,483,648	2,147,483,647
long	64 bits	-9,223,372,036,854,775,808	9,223,372,036,854,775,80
float	32 bits	-3.4E+38	+3.4E+38
double	64 bits	-1.7E+308	+1.7E+308
boolean	1 bit	Possible values are ***true*** or ***false***	
char	16 bits	Unicode / International character set	

Object Oriented Programming using Java

Data types and Variables:...

- **Integers:** Java defines four integer types: *byte, short, int, and long*. All of these are *signed, positive and negative values*. Java does not support unsigned, positive-only integers. **Eg: *int num1, num2,...***
- However, the concept of unsigned was used to specify the behavior of the high-order bit, which defines the sign of an integer value.
- Java manages the meaning of the high-order bit by adding a special *"unsigned right shift"* operator. Thus, the need for an unsigned integer type was eliminated.

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

Object Oriented Programming using Java

Data types and Variables:...

- **byte:** The smallest integer type, a signed 8-bit type, has a range from –128 to 127.
- Useful when you're working with a stream of data from a network or file. **Eg:** ***byte Byte1, Byte2,...***
- **short:** A signed 16-bit type, has a range from –32,768 to 32,767. It is probably the least-used type. **Eg: s*hort s1, s2;***
- **int:** Most commonly used integer type, a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647.
- Variables of type *int* are commonly employed to control loops and to index arrays. *byte* and *short* values are used in an expression are promoted to *int* when the expression is evaluated. **Eg:** ***int num1, num2,...***

Object Oriented Programming using Java

Data types and Variables:...

- **long:** A signed 64-bit type, useful for those occasions where an **int** type is not large enough to hold the desired value.
- This makes it useful when big, whole numbers are needed.

```
// Compute distance light travels using long variables
class Light {
  public static void main(String args[]) {
    int lightspeed;
    long days;
    long seconds;
    long distance;

    // approximate speed of light in miles per second
    lightspeed = 186000;

    days = 1000; // specify number of days here
    seconds = days * 24 * 60 * 60; // convert to seconds

    distance = lightspeed * seconds; // compute distance

    System.out.print("In " + days);
    System.out.print(" days light will travel about ");
    System.out.println(distance + " miles.");
  }
}
```

This program generates the following output:

```
In 1000 days light will travel about 16070400000000 miles.
```

Object Oriented Programming using Java

Data types and Variables:...

- **Floating-Point Types:** Also known as real numbers, are used when evaluating expressions that require fractional precision. There are two kinds of floating-point types, float and double, which represent single- and double-precision numbers.
- **float**: Specifies a single-precision value that uses 32 bits of storage, Single precision is faster on some processors, Useful when you need a fractional component, but don't require a large degree of precision.
  Eg: float hightemp, lowtemp;

Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

Object Oriented Programming using Java

Data types and Variables:...

- **double**: Uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All transcendental math functions, such as sin( ), cos( ), and sqrt( ), return double values.
- **char**: A 16-bit type, the range of a char is 0 to 65,536. Java uses Unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages. It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more.

Object Oriented Programming using Java

Data types and Variables:...

- **char:...** An *Eg*

```
// char variables behave like integers.
class CharDemo2 {
        public static void main(String args[]) {
                char ch1;
                ch1 = 'X';
                System.out.println("ch1 contains " + ch1);
                ch1++; // increment ch1
                System.out.println("ch1 is now " + ch1);
        }
}
```

- **boolean:** a primitive type, for logical values. It can have only one of two possible values, true or false. This is the type returned by all relational operators.

Object Oriented Programming using Java

Data types and Variables:...

- **boolean:...**

the control statements such as if and for.

Here is a program that demonstrates the boolean type:

```
// Demonstrate boolean values.
class BoolTest {
  public static void main(String args[]) {
    boolean b;

    b = false;
    System.out.println("b is " + b);
    b = true;
    System.out.println("b is " + b);

    // a boolean value can control the if statement
    if(b) System.out.println("This is executed.");

    b = false;
    if(b) System.out.println("This is not executed.");

    // outcome of a relational operator is a boolean value
    System.out.println("10 > 9 is " + (10 > 9));
  }
}
```

The output generated by this program is shown here:

```
b is false
b is true
This is executed.
10 > 9 is true
```

Object Oriented Programming using Java

Data types and Variables:...

A Closer Look at Literals

- **Integer Literals:** The most commonly used type in the typical program. Any whole number value is an integer literal. Eg: 1, 2, 3, and 42. all decimal values, describing a base 10 number.
- There are two other bases which can be used in integer literals, octal (base eight) and hexadecimal (base 16).
- Octal values are denoted in Java by a leading zero. Normal decimal numbers cannot have a leading zero.
- You signify a hexadecimal constant with a leading zero-x, (0x or 0X). The range of a hexadecimal digit is 0 to 15, so **A** through **F** (or **a** through **f**) are substituted for 10 through 15.

Data types and Variables:...

A Closer Look at Literals...

- **Integer Literals:...**
- Integer literals create an **int** value, which in Java is a 32-bit integer value.
- it is possible to assign an integer literal to one of Java's other integer types, such as **byte** or **long**, without causing a type mismatch error.
- When a literal value is assigned to a **byte** or **short** variable, no error is generated if the literal value is within the range of the target type.
- An integer literal can always be assigned to a **long** variable. However, to specify a **long** literal, you will need to explicitly tell the compiler that the literal value is of type **long**. You do this by appending an upper- or lowercase **L** to the literal. For example, 0x7fffffffffffffL or ***9223372036854775807L is the largest* long.** An integer can also be assigned to a char as long as it is within range.

Object Oriented Programming using Java

Data types and Variables:...

A Closer Look at Literals...

- **Floating-Point Literals:** Represent decimal values with a fractional component. They can be expressed in either standard or scientific notation. Standard notation consists of a whole number component followed by a decimal point followed by a fractional component. ***For example,*** 2.0, 3.14159, and 0.6667 represent valid standard-notation floating-point numbers.
- The exponent is indicated by an **E** or **e** followed by a decimal number, which can be positive or negative. Examples include 6.022E23, 314159E–05, and 2e+100.
- Floating-point literals in Java default to **double** precision. To specify a float literal, you must append an F or f to the constant.
- You can also explicitly specify a **double** literal by appending a **D** or **d**. Doing so is, of course, redundant.

Object Oriented Programming using Java

Data types and Variables:...

A Closer Look at Literals...

- **Boolean Literals:** Simple, only two logical values ***true*** and ***false***. These values do not convert into any numerical representation. The true literal in Java does not equal 1, nor does the false literal equal 0. In Java, they can only be assigned to variables declared as boolean, or used in expressions with Boolean operators.
- **Character Literals:** Indices into the Unicode character set, are 16-bit values that can be converted into integers and manipulated with the integer operators, such as the addition and subtraction operators. A ***literal character is represented inside a pair of single quotes***.

Object Oriented Programming using Java

Data types and Variables:...

A Closer Look at Literals...

- **String Literals:** Specified by enclosing a sequence of characters between a pair of double quotes. **Eg:** "Hello World"

Escape Sequence	Description
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal Unicode character (xxxx)
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Backspace

Object Oriented Programming using Java

Data types and Variables:...

- **Variables:** The basic unit of storage, defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime.

The basic form of a variable declaration:

type identifier [= value][, identifier [= value] ...] ;

```
int a, b, c;  // declares three ints, a, b, and c.
int d = 3, e, f = 5; // declares three more ints, initializing
// d and f.
byte z = 22;          // initializes z.
double pi = 3.14159;          // declares an approximation of pi.
char x = 'x';          // the variable x has the value 'x'.
```

Dynamic initialization:

- Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.
- **Eg:**

```
// Demonstrate dynamic initialization.
class DynInit {
    public static void main(String args[]) {
        double a = 3.0, b = 4.0;
        // c is dynamically initialized
        double c = Math.sqrt(a * a + b * b);
        System.out.println("Hypotenuse is " + c);
    }
}
```

Note: The key point here is that the initialization expression may use any element valid at the time of the initialization, including calls to methods, other variables, or literals.

The Scope and Lifetime of Variables:

- Java allows variables to be declared within any block.
- A block is begun with an opening curly brace and ended by a closing curly brace.
- A block defines a scope. Thus, each time you start a new block, you are creating a new scope.
- A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.
- Many other computer languages define two general categories of scopes: ***global and local***. However, these traditional scopes do not fit well with Java's strict, object-oriented model.

Object Oriented Programming using Java

The Scope and Lifetime of Variables:...

```
// Demonstrate block scope.
class Scope {
  public static void main(String args[]) {
    int x; // known to all code within main

    x = 10;
    if(x == 10) { // start new scope
     int y = 20; // known only to this block

      // x and y both known here.
      System.out.println("x and y: " + x + " " + y);
      x = y * 2;
    }
    // y = 100; // Error! y not known here

    // x is still known here.
    System.out.println("x is " + x);
  }
}
```

The Scope and Lifetime of Variables:...

```
// Demonstrate lifetime of a variable.
class LifeTime {
  public static void main(String args[]) {
    int x;

    for(x = 0; x < 3; x++) {
      int y = -1; // y is initialized each time block is entered
      System.out.println("y is: " + y); // this always prints -1
      y = 100;
      System.out.println("y is now: " + y);
    }
  }
}
```

One last point: Although blocks can be nested, you cannot declare a variable to have the same name as one in an outer scope. For example, the following program is illegal:

```
// This program will not compile
class ScopeErr {
        public static void main(String args[]) {
                int bar = 1;
                { // creates a new scope
                int bar = 2; // Compile-time error – bar already defined!
                }
        }
}
```

The Scope and Lifetime of Variables:...

- In Java, the two major scopes are those ***defined by a class and those defined by a method***.
- The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method's scope.
- As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope.
- Scopes can be nested. For example, each time you create a block of code, you are creating a new, nested scope. When this occurs, the outer scope encloses the inner scope. This means that objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true.

Type Conversion and Casting:

- You already know that it is fairly common to assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically.
- For example, it is always possible to assign an **int** value to a **long** variable.
- Fortunately, it is still possible to obtain a conversion between incompatible types. To do so, you must use a cast, which performs an explicit conversion between incompatible types.
- Let's look at both automatic type conversions and casting.

Type Conversion and Casting:...

Java's Automatic Conversions - *Widening Conversion*

- When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:
  - The two types are compatible.
  - The destination type is larger than the source type.
- When these two conditions are met, a ***widening conversion*** takes place.
- For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other.
- Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, long, or char.
- ***However, there are no automatic conversions from the numeric types to char or boolean. Also, char and boolean are not compatible with each other.***

Type Conversion and Casting:...

Casting Incompatible Types - *Narrowing Conversion*

- To create a conversion between two incompatible types, you must use a cast. A *cast* is simply an explicit type conversion.

(target-type) value

- **Eg:**
```
int a;
byte b;
// ...
b = (byte) a;
```
- If the integer's value is larger than the range of a **byte**, it will be reduced modulo (the remainder of an integer division by the byte's range)

Type Conversion and Casting:...

Casting Incompatible Types - *Narrowing Conversion…*

- A different type of conversion will occur when a floating-point value is assigned to an integer type: ***truncation.***

Automatic Type Promotion in Expressions

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

- To handle this kind of problem, Java automatically promotes each ***byte, short,*** *or* ***char*** operand to **int** when evaluating an expression.
- As useful as the automatic promotions are, they can cause confusing compile-time errors. For example, this seemingly correct code causes a problem:

```
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
```

```
byte b = 50;
b = (byte)(b * 2);
```

which yields the correct value of 100.

Type Conversion and Casting:...

Casting Incompatible Types - *Narrowing Conversion*…

```
// Demonstrate casts.
class Conversion {
  public static void main(String args[]) {
    byte b;
    int i = 257;
    double d = 323.142;

    System.out.println("\nConversion of int to byte.");
    b = (byte) i;
    System.out.println("i and b " + i + " " + b);

    System.out.println("\nConversion of double to int.");
    i = (int) d;
    System.out.println("d and i " + d + " " + i);

    System.out.println("\nConversion of double to byte.");
    b = (byte) d;
    System.out.println("d and b " + d + " " + b);
  }
}
```

output:
```
Conversion of int to byte.
i and b 257 1
Conversion of double to int.
d and i 323.142 323
Conversion of double to byte.
d and b 323.142 67
```

Type Conversion and Casting:...

The Type Promotion Rules

- Java defines several type promotion rules that apply to expressions.
- ***They are as follows:*** First, all **byte**, **short**, and **char** values are promoted to **int**, as just described. Then, if one operand is a **long**, the whole expression is promoted to **long**. If one operand is a **float**, the entire expression is promoted to **float**. If any of the operands is **double**, the result is **double**.

```
class Promote {
  public static void main(String args[]) {
    byte b = 42;
    char c = 'a';
    short s = 1024;
    int i = 50000;
    float f = 5.67f;
    double d = .1234;
    double result = (f * b) + (i / c) - (d * s);
    System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
    System.out.println("result = " + result);
  }
}
```

Operators and Expressions:

- Java provides a rich operator environment. Most of its operators can be divided into the following four groups: ***arithmetic, bitwise, relational, and logical***.

Classification of Operators

(1) Arithmetic Operators
(2) Relational Operators
(3) Logical Operators
(4) Assignment Operators
(5) Increments and Decrement Operators
(6) Conditional Operators
(7) Bitwise Operators
(8) Special Operators

Operators and Expressions:...

- **Arithmetic Operators:**

Operator	Result
+	Addition
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
–=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

The Basic Arithmetic Operators: addition, subtraction, multiplication, and division

The Modulus Operator: It can be applied to floating-point types as well as integer types.

Arithmetic Compound Assignment Operators: Used to combine an arithmetic operation with an assignment.

var = var op expression; can be written as var op= expression;

Increment and Decrement: ++ and --

Operators and Expressions:...

- **The Bitwise Operators:** Java defines several bitwise operators that can be applied to the integer types, long, int, short, char, and byte. These operators act upon the individual bits of their operands.

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

The Bitwise Logical Operators: &, |, ^, and ~

The Bit Shift: <<, >> , >>>
value << num, value >> num,

Bitwise Operator Compound Assignments:

a = a >> 4;
a >>= 4;

Operators and Expressions:...

- **The Bitwise Operators:** ***The Unsigned Right Shift***

The Left Shift: For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right., multiply by 2.

value << num

The Right Shift: Causes the two low-order bits to be lost, divide by 2.

value >> num

The Unsigned Right Shift/Shift Right Zero Fill: Always shifts zeros into the high-order bit. For example, if you are shifting something that does not represent a numeric value, you may not want sign extension to take place. ***This situation is common when you are working with pixel-based values and graphics.*** In these cases, you will generally want to shift a zero into the high-order bit no matter what its initial value was.

value >>> num

```
int a = -1;
a = a >>> 24;
```

Here is the same operation in binary form to further illustrate what is happening:

11111111 11111111 11111111 11111111 –1 in binary as an int

>>>24

00000000 00000000 00000000 11111111 255 in binary as an int

Operators and Expressions:...

Relational Operators: The outcome of these operations is a ***boolean*** value. Any type in Java, including integers, floating-point numbers, characters, and Booleans can be compared using the equality test, ==, and the inequality test, !=. Only integer, floating-point, and character operands compared to see which is greater or less than the other.

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Eg:

```
int a = 4;
int b = 1;
boolean c = a < b;
```

Operators and Expressions:...

- **Boolean Logical Operators:** operate only on boolean operands.

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

Object Oriented Programming using Java

Operators and Expressions:...

- **Short-Circuit Logical Operators:** **&&** and **||** secondary versions of the Boolean AND and OR operators (& and |).
- **The Assignment Operator:**

var = expression;

- **The ? Operator:** Java includes a special ternary (three-way) operator that can replace certain types of ***if-then-else*** statements. **expression1 ? expression2 : expression3**
  Eg: ratio = denom == 0 ? 0 : num / denom;

Object Oriented Programming using Java

Operator Precedence:

Category	Operator	Associativity
Postfix	() [] . (dot operator)	Left to right
Unary	++ - - ! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Object Oriented Programming using Java

Access specifiers:

- Encapsulation links data with the code that manipulates it. However, it provides another important attribute: ***access control***.
- Through encapsulation, you can control what parts of a program can access the members of a class. By controlling access, you can prevent misuse.
- **Eg:** Allowing access to data only through a well defined set of methods, you can prevent the misuse of that data.
- How a member can be accessed is determined by the access specifier that modifies its declaration. Java supplies a rich set of access specifiers.

Object Oriented Programming using Java

Access specifiers: ...

- Java's access specifiers are ***public***, ***private***, ***protected*** and *default*.
- ***public*:** Member can be accessed by any other code.
- ***private*:** Member can only be accessed by other members of its class.
- ***protected*:** Applies only when inheritance is involved.
- ***When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.***

Object Oriented Programming using Java

Control Statements & Loops:

- Programming language uses control statements to Cause the flow of execution to advance and branch based on changes to the state of a program.

Three Categories

- **Selection:** Allows program to choose different paths of execution based upon the outcome of an expression or the state of a variable. **Eg:** ***if*** *and* ***switch***
- **Iteration:** Enables program execution to repeat one or more statements (form loops). **Eg:** ***for, while*** *and* ***do-while***
- **Jump:** Allows program to execute in a nonlinear fashion. **Eg:** ***break, continue,*** *and* ***return.***

Java's Selection Statements:

- ***The if statement:*** Java's conditional branch statement. It can be used to route program execution through two different paths.
- The general form: ***if (condition) statement1;***
  else statement2;
- Here, each *statement* may be a single statement or a compound statement enclosed in curly braces (that is, a block). The *condition* is any expression that returns a *boolean* value. The *else* clause is optional.

```
int a, b;
// ...
if(a < b) a = 0;
else b = 0;
```

```
boolean dataAvailable;
// ...
if (dataAvailable)
  ProcessData();
else
  waitForMoreData();
```

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
    ProcessData();
    bytesAvailable -= n;
} else
    waitForMoreData();
```

Java's Selection Statements: ...

- ***Nested ifs:*** A *nested if* is an ***if*** statement that is the target of another ***if*** or ***else***. The main thing to remember is that an ***else*** statement always refers to the nearest ***if*** statement that is within the same block as the ***else*** and that is not already associated with an ***else***.

```
if(i == 10) {
     if(j < 20) a = b;
     if(k > 100) c = d; // this if is
        else a = c;              // associated with this else
}
else a = d; // this else refers to if(i == 10)
```

Java's Selection Statements: ...

- ***The if-else-if Ladder:*** A common programming construct that is based upon a sequence of nested **ifs** is the ***if-else-if ladder***.

```
if(condition)
        statement;
else if(condition)
        statement;
else if(condition)
        statement;
.
.
.
else
        statement;
```

```
// Demonstrate if-else-if statements.
class IfElse {
  public static void main(String args[]) {
    int month = 4; // April
    String season;

    if(month == 12 || month == 1 || month == 2)
      season = "Winter";
    else if(month == 3 || month == 4 || month == 5)
      season = "Spring";
    else if(month == 6 || month == 7 || month == 8)
      season = "Summer";
    else if(month == 9 || month == 10 || month == 11)
      season = "Autumn";
    else
      season = "Bogus Month";

    System.out.println("April is in the " + season + ".");
  }
}
```

Java's Selection Statements: ...

- ***switch:*** Java's *multiway* branch statement, provides an easy way to dispatch execution to different parts of your code based on the value of an expression, provides a better alternative than a large series of ***if-else-if*** statements.
- ***The general form:***

```
switch (expression) {
  case value1:
    // statement sequence
    break;
  case value2:
    // statement sequence
    break;
.
.
.
  case valueN:
    // statement sequence
    break;
  default:
    // default statement sequence
}
```

The *expression must be of type byte, short, int, or char; each of the values specified in the* case statements must be of a type compatible with the expression. (An enumeration value can also be used to control a switch statement. Each case value must be a unique literal (that is, it must be a constant, not a variable). Duplicate case values are not allowed. the default statement is optional. If no case matches and no default is present, then no further action is taken.

Object Oriented Programming using Java

Java's Selection Statements: ...

- *switch*: ...

```
// A simple example of the switch.
class SampleSwitch {
  public static void main(String args[]) {
    for(int i=0; i<6; i++)
      switch(i) {
        case 0:
          System.out.println("i is zero.");
          break;
        case 1:
          System.out.println("i is one.");
          break;
        case 2:
          System.out.println("i is two.");
          break;
        case 3:
          System.out.println("i is three.");
          break;
        default:
          System.out.println("i is greater than 3.");
      }
  }
}
```

```
// In a switch, break statements are optional.
class MissingBreak {
  public static void main(String args[]) {
    for(int i=0; i<12; i++)
      switch(i) {
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
          System.out.println("i is less than 5");
          break;
        case 5:
        case 6:
        case 7:
        case 8:
        case 9:
          System.out.println("i is less than 10");
          break;
        default:
          System.out.println("i is 10 or more");
      }
  }
}
```

Java's Selection Statements: ...

- *switch*: ...

```
// An improved version of the season program.
class Switch {
    public static void main(String args[]) {
        int month = 4;
        String season;
        switch (month) {
          case 12:
          case 1:
          case 2:
            season = "Winter";
            break;
          case 3:
          case 4:
          case 5:
            season = "Spring";
            break;
          case 6:
          case 7:
          case 8:
            season = "Summer";
            break;
          case 9:
          case 10:
          case 11:
            season = "Autumn";
            break;
          default:
            season = "Bogus Month";
        }
        System.out.println("April is in the " + season + ".");
    }
}
```

You can use a switch as part of the statement sequence of an outer switch. This is called a ***nested switch.*** *Since a switch statement defines its own block, no conflicts arise between the* case constants in the inner switch and those in the outer switch.

```
switch(count) {
  case 1:
    switch(target) { // nested switch
      case 0:
        System.out.println("target is zero");
        break;
      case 1: // no conflicts with outer switch
        System.out.println("target is one");
        break;
    }
    break;
  case 2: // ...
```

Java's Iteration Statements: ***for, while,*** and ***do-while,*** these statements create loops. A loop repeatedly executes the same set of instructions until a termination condition is met.

while	*do-while*	*for*
while(*condition*) { ** // body of loop** **}**	**do {** ** // body of loop** **} while (*condition*);**	**for(initialization; *condition*; iteration) {** ** // body** **}**

The *condition can be any Boolean expression. The body of the loop will be executed as long* as the conditional expression is true. When *condition becomes false, control passes to the* next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated. The body of the ***while*** (or any other of Java's loops) can be empty. This is because a ***null*** *statement (one that consists only of a semicolon) is syntactically valid in Java.*

Object Oriented Programming using Java

Java's Iteration Statements: ...

```
// Demonstrate the while loop.
class While {
  public static void main(String args[]) {
    int n = 10;

    while(n > 0) {
      System.out.println("tick " + n);
      n--;
    }
  }
}
```

```
// Demonstrate the do-while loop.
class DoWhile {
  public static void main(String args[]) {
    int n = 10;

    do {
      System.out.println("tick " + n);
      n--;
    } while(n > 0);
  }
}
```

```
// Demonstrate the for loop.
class ForTick {
  public static void main(String args[]) {
    int n;

    for(n=10; n>0; n--)
      System.out.println("tick " + n);
  }
}
```

```
// Declare a loop control variable inside the for
class ForTick {
  public static void main(String args[]) {

    // here, n is declared inside of the for loop
    for(int n=10; n>0; n--)
      System.out.println("tick " + n);
  }
}
```

Java's Iteration Statements: ... *for*

- Beginning with JDK 5, there are two forms of the **for** loop. The first is the traditional form that has been in use since the original version of Java. The second is the new ***"for-each"*** form.

- **Declaring Loop Control Variables Inside the for Loop:**

```
for(int n=10; n>0; n--)
```

- **Using the Comma:**

```
for(a=1, b=4; a<b; a++, b--)
```

- **Some for Loop Variations:**

```
boolean done = false;
for(int i=1; !done; i++) {
    // ...
    if(interrupted()) done = true;
}
```

Java's Iteration Statements: ... *for*...

- Some for Loop Variations:...

```
boolean done = false;
for(int i=1; !done; i++) {
    // ...
    if(interrupted()) done = true;
}

// Parts of the for loop can be empty.
for( ; !done; )

for( ; ; ) {
    // ...
}
```

Java's Iteration Statements: ... *for*...

- **The For-Each Version of the for Loop:**

Beginning with JDK 5, a second form of **for** was defined that implements a **"for-each"** style loop.

- A f***oreach*** style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish. Unlike some languages, such as C#, that implement a for-each loop by using the keyword **foreach,** Java adds the for-each capability by enhancing the **for** statement.
- The advantage of this approach is that no new keyword is required, and no pre-existing code is broken. The for-each style of **for** is also referred to as the ***enhanced for loop.***
- General form:

for(*type itr-var : collection) statement-block*

Java's Iteration Statements: ... *for*...

- **The For-Each Version of the for Loop:...*Examples***

int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int i=0; i < 10; i++) sum += nums[i];

Can be expressed as:

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int x: nums) sum += x;
```

```
// Use break with a for-each style for.
class ForEach2 {
	public static void main(String args[]) {
		int sum = 0;
		int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
		// use for to display and sum the values
		for(int x : nums) {
			System.out.println("Value is: " + x);
			sum += x;
			if(x == 5) break; // stop the loop when 5 is obtained
		}
		System.out.println("Summation of first 5 elements: " + su
	}
}
```

Java's Iteration Statements: ... *for*...

- **The For-Each Version of the for Loop:...**
- ***Iterating Over Multidimensional Arrays... Examples***

```
// Use for-each style for on a two-dimensional array.
class ForEach3 {
  public static void main(String args[]) {
    int sum = 0;
    int nums[][] = new int[3][5];

    // give nums some values
    for(int i = 0; i < 3; i++)
      for(int j=0; j < 5; j++)
        nums[i][j] = (i+1)*(j+1);

    // use for-each for to display and sum the values
    for(int x[] : nums) {
      for(int y : x) {
        System.out.println("Value is: " + y);
        sum += y;
      }
    }
    System.out.println("Summation: " + sum);
  }
}
```

Java's Iteration Statements: ... *for*...

- **The For-Each Version of the for Loop:...**
- ***Applying the Enhanced for... Examples***

```
// Search an array using for-each style for.
class Search {
  public static void main(String args[]) {
    int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 };
    int val = 5;
    boolean found = false;

    // use for-each style for to search nums for val
    for(int x : nums) {
      if(x == val) {
        found = true;
        break;
      }
    }

    if(found)
      System.out.println("Value found!");
  }
}
```

Java's Iteration Statements: ... *for*...

- **The For-Each Version of the for Loop:...**
- ***Nested Loops... Examples***

```
// Loops may be nested.
class Nested {
  public static void main(String args[]) {
    int i, j;

    for(i=0; i<10; i++) {
      for(j=i; j<10; j++)
        System.out.print(".");
      System.out.println();
    }
  }
}
```

Java's Jump Statements: *break, continue, and return.*

- These statements transfer control to another part of your program.
- ***Using break:*** Statement has three uses:
  - Terminates a statement sequence in a switch statement.
  - Used to exit a loop
  - Used as a "civilized" form of ***goto.***

break was not designed to provide the normal means by which a loop is terminated. *The loop's conditional expression serves this purpose. The* ***break statement should be used to*** *cancel a loop only when some sort of special situation occurs.*

Java's Jump Statements:...*Using break: ...*

- ***Using break to Exit a Loop:*** force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.

```
// Using break to exit a loop.
class BreakLoop {
  public static void main(String args[]) {
    for(int i=0; i<100; i++) {
      if(i == 10) break; // terminate loop if i is 10
      System.out.println("i: " + i);
    }
    System.out.println("Loop complete.");
  }
}
```

- ***Using break as a form of goto:*** **break *label;***

Java does not have a goto statement because it provides a way to branch in an arbitrary and unstructured manner. This usually makes goto-ridden code hard to understand and hard to maintain. It also prohibits certain compiler optimizations.

```
// Using break to exit from nested loops
class BreakLoop4 {
  public static void main(String args[]) {
    outer: for(int i=0; i<3; i++) {
      System.out.print("Pass " + i + ": ");
      for(int j=0; j<100; j++) {
        if(j == 10) break outer; // exit both loops
        System.out.print(j + " ");
      }
      System.out.println("This will not print");
    }
    System.out.println("Loops complete.");
  }
}
```

```
public static void main(String args[]) {

  one: for(int i=0; i<3; i++) {
    System.out.print("Pass " + i + ": ");
  }

  for(int j=0; j<100; j++) {
    if(j == 10) break one; // WRONG
    System.out.print(j + " ");
  }
}
```

Java's Jump Statements:...

Using continue: Causes control to be transferred directly to the conditional expression that controls the loop. Sometimes it is useful to force an early iteration of a loop.

```
// Demonstrate continue.
class Continue {
  public static void main(String args[]) {
    for(int i=0; i<10; i++) {
      System.out.print(i + " ");
      if (i%2 == 0) continue;
      System.out.println("");
    }
  }
}
```

```
// Using continue with a label.
class ContinueLabel {
  public static void main(String args[]) {
outer: for (int i=0; i<10; i++) {
         for(int j=0; j<10; j++) {
           if(j > i) {
             System.out.println();
             continue outer;
           }
           System.out.print(" " + (i * j));
         }
       }
       System.out.println();
  }
}
```

As with the ***break*** statement, ***continue*** may specify a label to describe which enclosing loop to continue.

Java's Jump Statements:... *Return*

- The last control statement is **return.** The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.

```
// Demonstrate return.
class Return {
  public static void main(String args[]) {
    boolean t = true;
    System.out.println("Before the return.");

    if(t) return; // return to caller

    System.out.println("This won't execute.");
  }
}
```

Arrays:

- A group of ***like-typed*** variables that are referred to by a common name.
- Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index.
- Arrays offer a convenient means of grouping related information.
- **Eg:**
  - ***One-Dimensional Arrays:*** Essentially, a list of like-typed variables.
  - ***Multidimensional Arrays:*** Arrays of arrays.

One-Dimensional Arrays: A list of like-typed variables. To create an array, you first must create an array variable of the desired type.

- General form: ***type var-name[ ];***
- **Eg:** int month_days[];

month_days is an array variable, no array actually exists. In fact, the value of **month_days** is set to **null,** which represents ***an array with no value.***

- To allocate memory for arrays, the general form:

array-var = new type[size]; or

type array-var[] = new type[size];

- **Eg:** month_days = new int[12];
- It is possible to combine the declaration of the array variable with the allocation of the array itself, as shown here: ***int month_days[] = new int[12];***

One-Dimensional Arrays: ...

```
// An improved version.
class AutoArray {
    public static void main(String args[]) {
        int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
        System.out.println("April has " + month_days[3] + "
        days.");
    }
}
```

Multidimensional Arrays: *arrays of arrays*

Eg: int twoD[][] = new int[4][5];

```
// Demonstrate a two-dimensional array.
class TwoDArray {
    public static void main(String args[]) {
        int twoD[][]= new int[4][5];
        int i, j, k = 0;
        for(i=0; i<4; i++)
        for(j=0; j<5; j++) {
            twoD[i][j] = k;
            k++;
        }
        for(i=0; i<4; i++) {
            for(j=0; j<5; j++)
            System.out.print(twoD[i][j] + " ");
            System.out.println();
        }
    }
}
```

Multidimensional Arrays: …

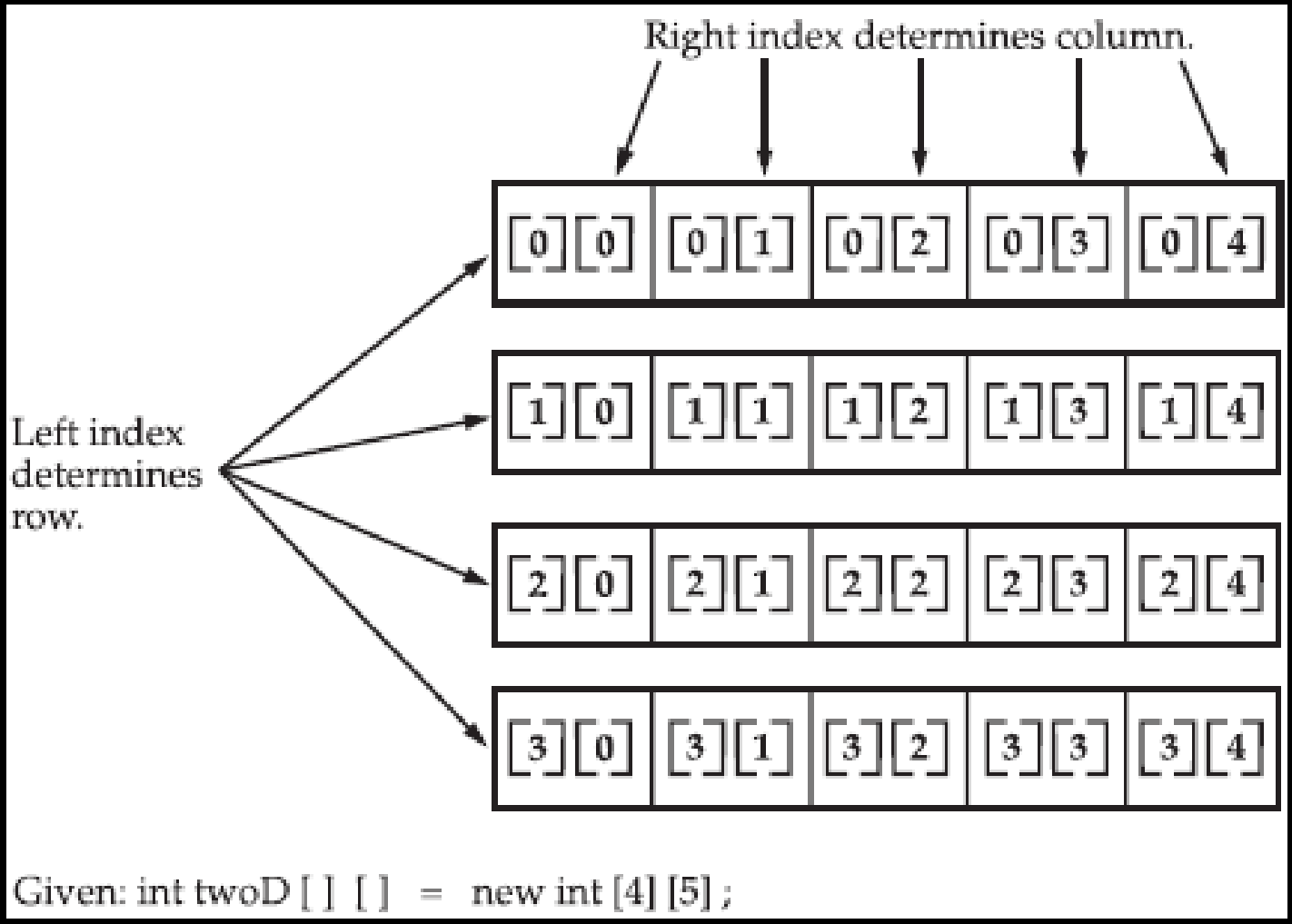
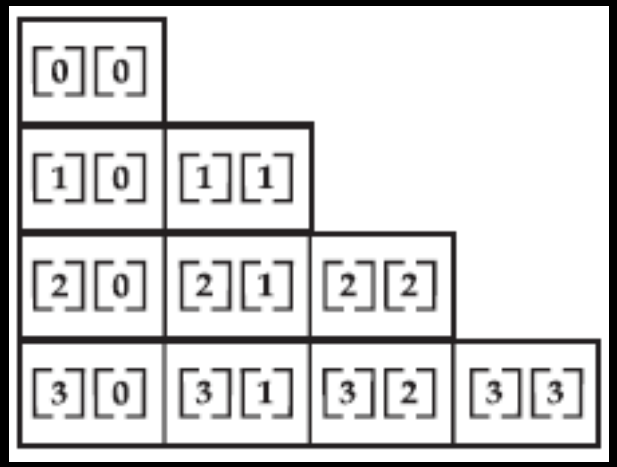


Figure : ***A conceptual view of a 4 by 5, two-dimensional array***

Multidimensional Arrays: ... *uneven/irregular*

```
// Manually allocate differing size second dimensions.
class TwoDAgain {
    public static void main(String args[]) {
    int twoD[][] = new int[4][];
        twoD[0] = new int[1];
        twoD[1] = new int[2];
        twoD[2] = new int[3];
        twoD[3] = new int[4];
        . . .
```



- The use of uneven (or, irregular) multidimensional arrays may not be appropriate for many applications, because it runs contrary to what people expect to find when a multidimensional array is encountered. However, irregular arrays can be used effectively in some situations. For example, if you need a very large two-dimensional array that is sparsely populated (that is, one in which not all of the elements will be used), then an irregular array might be a perfect solution.

Object Oriented Programming using Java

- **Alternative Array Declaration Syntax:**

type[ ] var-name;

int al[] = new int[3]; **OR** int[] a2 = new int[3];

char twod1[][] = new char[3][4];

OR

char[][] twod2 = new char[3][4];

int[] nums, nums2, nums3; // create three arrays

OR

int nums[], nums2[], nums3[]; // create three arrays

- ***The alternative declaration form is also useful when specifying an array as a return type for a method.***

Introducing Classes

Prepared using following Resources:

- Herbert Schildt, “Java: The Complete Reference”, Tata McGrawHill Education
- E Balagurusamy, Programming with Java - A Tata McGraw Hill Education
- https://www.geeksforgeeks.org/java/
- https://www.javatpoint.com/java-tutorial
- https://www.tutorialspoint.com/java/index.htm
- https://www.w3schools.com/java/

By: DIVAKARA .N

Introducing Classes

- Class Fundamentals
- Declaring objects
- Introducing Methods
- Constructors
- ***this*** keyword
- Use of objects as parameter & Methods returning objects
- Call by value & Call by reference
- Static variables & methods
- Garbage collection
- Nested & Inner classes.

Introducing Classes

Class Fundamentals:

- Class is the logical construct upon which the entire Java language is built because *it defines the shape and nature of an object*. It defines a new data type.
- The class forms the basis for OOP in Java. Any concept you wish to implement in a Java program must be encapsulated within a class.
- Used to create objects of that type. Thus, a class is a template for an object, and an object is an instance of a class.
- The two words ***object*** and ***instance*** used interchangeably.

Introducing Classes

Class Fundamentals: ...

- When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data. While very simple classes may contain only code or only data, ***most real-world classes contain both***.
- A class is declared by use of the ***class*** keyword.

Introducing Classes

Class Fundamentals: ... *A simplified general form*

```
class classname {
    type instance-variable1;
    type instance-variable2;
    // ...
    type instance-variableN;

    type methodname1(parameter-list) {
      // body of method
    }
    type methodname2(parameter-list) {
      // body of method
  }
  // ...
  type methodnameN(parameter-list) {
      // body of method
  }
}
```

- The data, or variables, defined within a **class** are called *instance variables*. *The code* ***is*** contained within *methods*. *Collectively, the methods and variables defined within a class are* called *members of the class. In most classes, the instance variables are acted upon and accessed* by the methods defined for that class.
- Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another.

Introducing Classes

Class Fundamentals: ...

NOTE:

- *C++ programmers will notice that the class declaration and the implementation of the methods are stored in the same place and not defined separately. This sometimes makes for very large .java files, since any class must be entirely defined in a single source file.*
- *This design feature was built into Java because it was felt that in the long run, having specification, declaration, and implementation all in one place makes for code that is easier to maintain.*

Introducing Classes

Class Fundamentals: ... *Introducing Access Control*

```
/* This program demonstrates the difference between
   public and private.
*/
class Test {
  int a; // default access
  public int b; // public access
  private int c; // private access

  // methods to access c
  void setc(int i) { // set c's value
    c = i;
  }
  int getc() { // get c's value
    return c;
  }
}

class AccessTest {
  public static void main(String args[]) {
    Test ob = new Test();

    // These are OK, a and b may be accessed directly
    ob.a = 10;
    ob.b = 20;

    // This is not OK and will cause an error
//  ob.c = 100; // Error!

    // You must access c through its methods
    ob.setc(100); // OK
    System.out.println("a, b, and c: " + ob.a + " " +
                       ob.b + " " + ob.getc());
  }
}
```

- Java's access specifiers are **public, private,** and **protected.** Java also defines a default access level.
- **protected** applies only when inheritance is involved.

Introducing Classes

Class Fundamentals:

Simple Class: ...

```
class Box {
        double width;
        double height;
        double depth;
}

Box mybox = new Box();

mybox.width = 100;
```

```
/* A program that uses the Box class.

   Call this file BoxDemo.java
*/
class Box {
  double width;
  double height;
  double depth;
}

// This class declares an object of type Box.
class BoxDemo {
  public static void main(String args[]) {
    Box mybox = new Box();
    double vol;

    // assign values to mybox's instance variables
    mybox.width = 10;
    mybox.height = 20;
    mybox.depth = 15;

    // compute volume of box
    vol = mybox.width * mybox.height * mybox.depth;

    System.out.println("Volume is " + vol);
  }
}
```

Introducing Classes

Class Fundamentals: ...

Simple Class: ...

```
// This program declares two Box objects.

class Box {
  double width;
  double height;
  double depth;
}

class BoxDemo2 {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;

    // assign values to mybox1's instance variables
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;

    /* assign different values to mybox2's
       instance variables */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;

    // compute volume of first box
    vol = mybox1.width * mybox1.height * mybox1.depth;
    System.out.println("Volume is " + vol);

    // compute volume of second box
    vol = mybox2.width * mybox2.height * mybox2.depth;
    System.out.println("Volume is " + vol);
  }
}
```

Introducing Classes

Declaring objects:

- Obtaining objects of a class is a two-step process.
  - Declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer to an object.*
  - *Acquire an actual*, physical copy of the object and assign it to that variable. (using the **new** operator)
- The **new** operator dynamically allocates (allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by **new.**
- This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.

```
1. Box mybox; // declare reference to object
2. mybox = new Box(); // allocate a Box object
```

ClassName Object_Name = new ClassName(); OR

ClassName class-var = new ClassName();

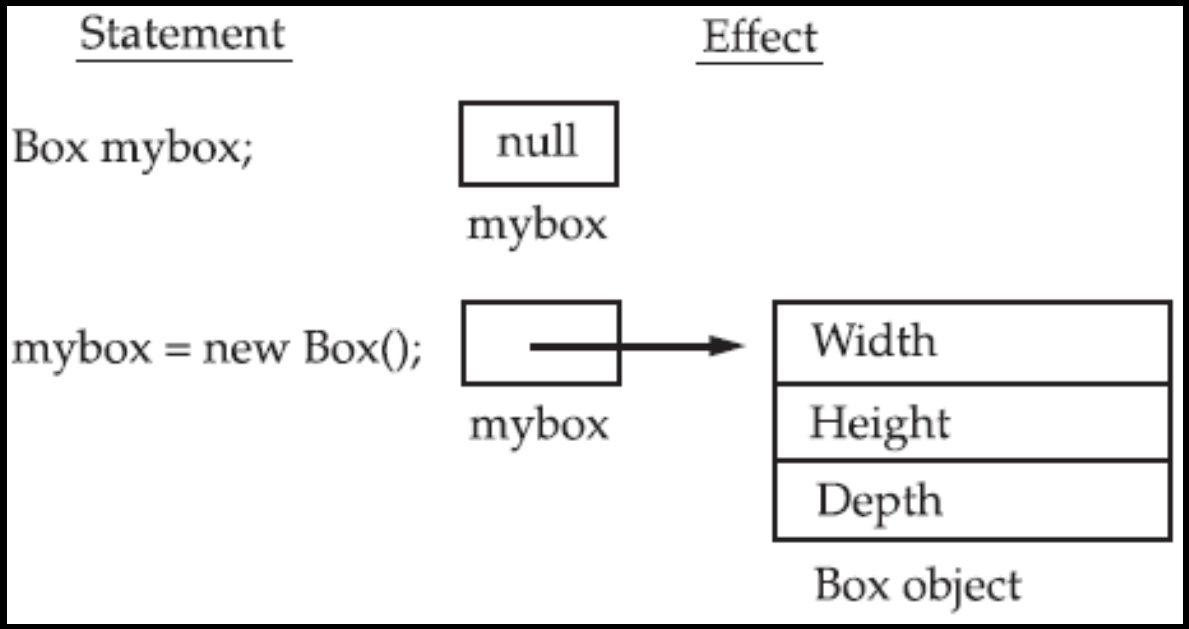
Box mybox = new Box();

Introducing Classes

Declaring objects: ...

A Closer Look at ***new :*** operator dynamically allocates memory for an object.

General form: ***class-var = new classname( );***



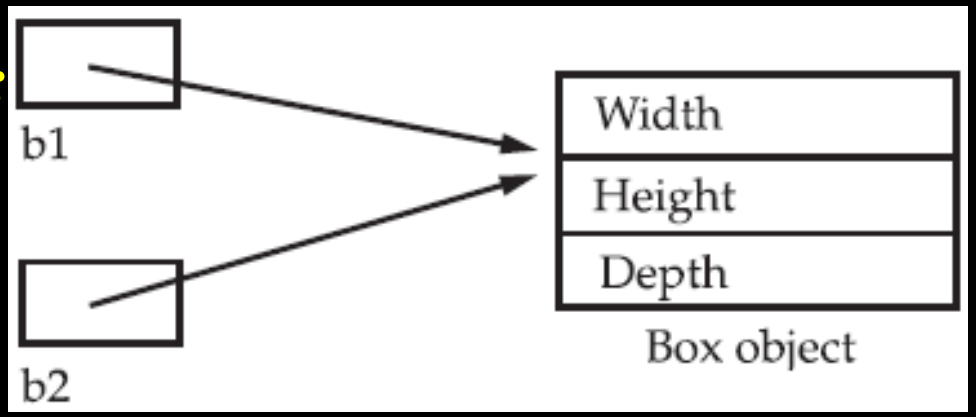
Note: *An object reference is similar to a memory pointer. The main difference—and the key to Java's safety—is that you cannot manipulate references as you can actual pointers. Thus, you cannot cause an object reference to point to an arbitrary memory location or manipulate it like an integer.*

Introducing Classes

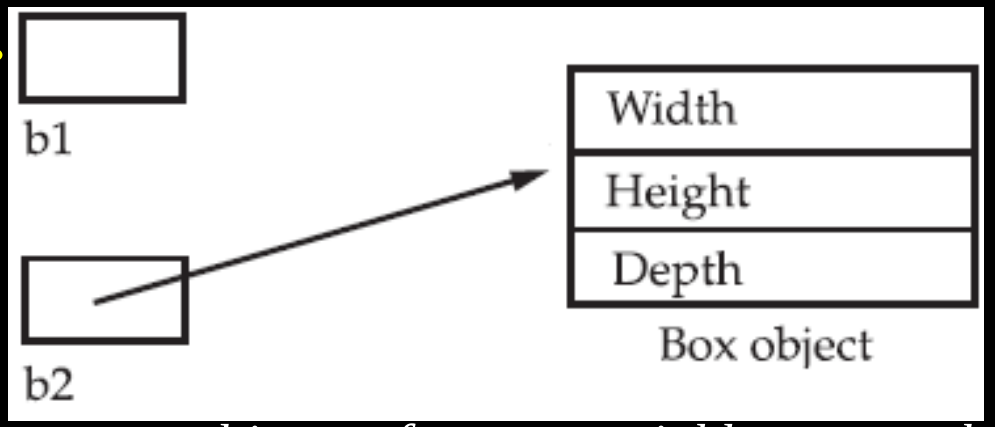
Declaring objects: ...

Assigning Object Reference Variables:

```
Box b1 = new Box();
Box b2 = b1;
```




```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```



REMEMBER *When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.*

Introducing Classes

Introducing Methods:

- Classes consist of two things*: instance variables and methods.*

General form: ***type name(parameter-list) {***

// body of method

}

- Here, *type specifies the type of data returned by the method. This can be any valid type,* including class types that you create. If the method does not return a value, its return type must be **void.** The name of the method is specified by *name. This can be any legal identifier* other than those already used by other items within the current scope.
- The *parameter-list is a* sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the *arguments passed to the method when it is called. If the method* has no parameters, then the parameter list will be empty.
- Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement:

return *value;*

Introducing Classes

Introducing Methods: ...

Adding a Method to the Box Class:

```
// This program includes a method inside the box cla

class Box {
  double width;
  double height;
  double depth;

  // display volume of a box
  void volume() {
    System.out.print("Volume is ");
    System.out.println(width * height * depth);
  }
}
```

```
class BoxDemo3 {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();

    // assign values to mybox1's instance variables
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;

    /* assign different values to mybox2's
       instance variables */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;

    // display volume of first box
    mybox1.volume();

    // display volume of second box
    mybox2.volume();
  }
}
```

Introducing Classes

Introducing Methods:...

Returning a Value:

```
// Now, volume() returns the volume of a box.

class Box {
  double width;
  double height;
  double depth;

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
```

```
class BoxDemo4 {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;

    // assign values to mybox1's instance variables
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;

    /* assign different values to mybox2's
       instance variables */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);
    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
}
```

vol = mybox1.volume(); can be replaced by

System.out.println("Volume is " + mybox1.volume());

Introducing Classes

Introducing Methods: ...

- **Adding a Method That Takes Parameters:** Parameters allow a method to be generalized. A parameterized method can operate on a variety of data and/or be used in a number of slightly different situations.

```
int square()
{
  return 10 * 10;
}
```

```
int square(int i)
{
  return i * i;
}
```

Example:

```
int x, y;
x = square(5); // x equals 25
x = square(9); // x equals 81
y = 2;
x = square(y); // x equals 4
```

```
// This program uses a parameterized method.

class Box {
  double width;
  double height;
  double depth;

  // compute and return volume
  double volume() {
    return width * height * depth;
  }

  // sets dimensions of box
  void setDim(double w, double h, double d) {
    width = w;
```

Introducing Classes

Introducing Methods: ...

- **Adding a Method That Takes Parameters: ...**

```
// This program uses a parameterized method.

class Box {
  double width;
  double height;
  double depth;

  // compute and return volume
  double volume() {
    return width * height * depth;
  }

  // sets dimensions of box
  void setDim(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }
}

class BoxDemo5 {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;

    // initialize each box
    mybox1.setDim(10, 20, 15);
    mybox2.setDim(3, 6, 9);

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
}
```

Note: ***The concepts of the method invocation, parameters, and return values are fundamental to Java programming.***

Introducing Classes

Introducing Methods: ... Recursion:

- Java supports *recursion*. Recursion is ***the process of defining something in terms of itself***. *As* it relates to Java programming, recursion is the attribute that ***allows a method to call itself***. A method that calls itself is said to be *recursive*.

```
// A simple example of recursion.
class Factorial {
  // this is a recursive method
  int fact(int n) {
    int result;

    if(n==1) return 1;
    result = fact(n-1) * n;
    return result;
  }
}

class Recursion {
  public static void main(String args[]) {
    Factorial f = new Factorial();

    System.out.println("Factorial of 3 is " + f.fact(3));
    System.out.println("Factorial of 4 is " + f.fact(4));
    System.out.println("Factorial of 5 is " + f.fact(5));
  }
}
```

Introducing Classes

Introducing Methods: ... *Overloading*

- In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different.
- Method overloading is one of the ways that Java supports polymorphism.
- Method overloading is one of Java's most exciting and useful features.
- Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters.

Introducing Classes

Introducing Methods: ... *Overloading* ...

Method Overloading by changing the no. of arguments

```
class Calculation{

    void sum(int a, int b){
        System.out.println(a+b);
    }

    void sum(int a, int b, int c){
        System.out.println(a+b+c);
    }
    public static void main(String args[]){
        Calculation obj = new Calculation();
        obj.sum(10,10,10);
        obj.sum(20,20);
    }
}
```

Introducing Classes

Introducing Methods: ... *Overloading* ...

Method Overloading by changing data type of argument

```
class Calculation2{
  void sum(int a, int b){
          System.out.println(a + b);
  }

  void sum(double a, double b){
          System.out.println(a + b);
  }

  public static void main(String args[]){
        Calculation2 obj = new Calculation2();
        obj.sum(10.5,10.5);
        obj.sum(20,20);
   }
}
```

Introducing Classes

Introducing Methods: ... *Overloading* ...

```
class MyClass {
   int height;
   MyClass() {
      System.out.println("bricks");
      height = 0;
   }
   MyClass(int i) {
      System.out.println("Building new House that is "
      + i + " feet tall");
      height = i;
   }
   void info() {
      System.out.println("House is " + height
      + " feet tall");
   }
   void info(String s) {
      System.out.println(s + ": House is "
      + height + " feet tall");
   }
}
public class MainClass {
   public static void main(String[] args) {
      MyClass t = new MyClass(0);
      t.info();
      t.info("overloaded method");
      //Overloaded constructor:
      new MyClass();
   }
}
```

Method/Constructor Overloading by changing number of argument

Introducing Classes

Introducing Methods: ... *Overloading* ...

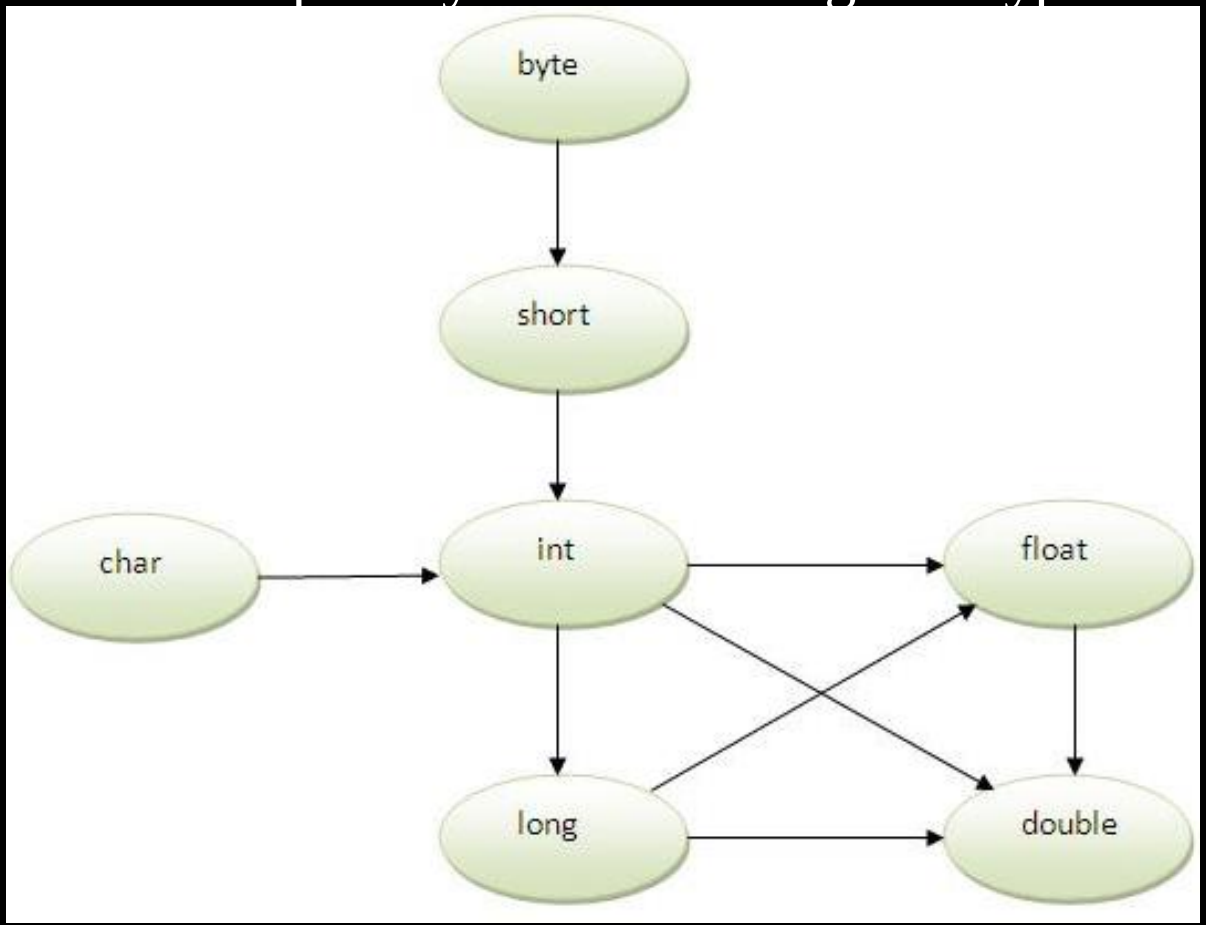
Can we overload main() method?

```
class Overloading1{
  public static void main(int a){
  System.out.println(a);
  }

  public static void main(String args[]){
  System.out.println("main() method invoked");
  main(10);
  }
}
```

Introducing Classes

Introducing Methods: ... *Overloading* ...

Method Overloading and Type Promotion: One type is promoted to another implicitly if no matching datatype is found.



byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int,long,float or double. The char datatype can be promoted to int, long, float or double and so on.

Introducing Classes

Introducing Methods: ... *Overloading* ...

Method Overloading and Type Promotion:...

```
class OverloadingCalculation1{
  void sum(int a,long b){System.out.println(a+b);}
  void sum(int a,int b,int c){System.out.println(a+b+c);}

  public static void main(String args[]){
  OverloadingCalculation1 obj=new OverloadingCalculation1();
  obj.sum(20,20);//now second int literal will be promoted to long
  obj.sum(20,20,20);

  }
}
```

```
class OverloadingCalculation3{
  void sum(int a,long b){System.out.println("a method invoked");}
  void sum(long a,int b){System.out.println("b method invoked");}

  public static void main(String args[]){
  OverloadingCalculation3 obj=new OverloadingCalculation3();
  obj.sum(20,20);//now ambiguity
  }
}
```

```
class OverloadingCalculation2{
  void sum(int a,int b){System.out.println("int arg method invoked");}
  void sum(long a,long b){System.out.println("long arg method invoked");}

  public static void main(String args[]){
  OverloadingCalculation2 obj=new OverloadingCalculation2();
  obj.sum(20,20);//now int arg sum() method gets invoked
  }
}
```

Introducing Classes

Introducing Methods: ... *Overloading* ...

```
public class MainClass {
   public static void printArray(Integer[] inputArray) {
      for (Integer element : inputArray){
         System.out.printf("%s ", element);
         System.out.println();
      }
   }
   public static void printArray(Double[] inputArray) {
      for (Double element : inputArray){
         System.out.printf("%s ", element);
         System.out.println();
      }
   }
   public static void printArray(Character[] inputArray) {
      for (Character element : inputArray){
         System.out.printf("%s ", element);
         System.out.println();
      }
   }
   public static void main(String args[]) {
      Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };
      Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4,
      5.5, 6.6, 7.7 };
      Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' };
      System.out.println("Array integerArray contains:");
      printArray(integerArray);
      System.out.println("\nArray doubleArray contains:");
      printArray(doubleArray);
      System.out.println("\nArray characterArray contains:");
      printArray(characterArray);
   }
}
```

Introducing Classes

Constructors:

- It can be tedious to initialize all of the variables in a class each time an instance is created. It would be simpler and more concise to have all of the setup done at the time the object is first created. Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a ***constructor***.
- A *constructor initializes an object immediately upon creation. It has the same name as the* class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called immediately after the object is created, *before the* ***new*** *operator completes.* Constructors look a little strange because they have no return type, not even **void.**

```
class Programming {
  //constructor method
  Programming() {
    System.out.println("Constructor method called.");
  }

  public static void main(String[] args) {
    Programming object = new Programming(); //creating object
  }
}
```

Introducing Classes

Constructors: ...

class-var = new classname( );

Box mybox1 = new Box();

- When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class.
- The default constructor automatically initializes all instance variables to zero.
- Once you define your own constructor, the default constructor is no longer used.

```
/* Here, Box uses a constructor to initialize the
   dimensions of a box.
*/
class Box {
  double width;
  double height;
  double depth;

  // This is the constructor for Box.
  Box() {
    System.out.println("Constructing Box");
    width = 10;
    height = 10;
    depth = 10;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}

class BoxDemo6 {
  public static void main(String args[]) {
    // declare, allocate, and initialize Box objects
    Box mybox1 = new Box();
    Box mybox2 = new Box();

    double vol;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
}
```

Introducing Classes

Constructors: ... *Parameterized Constructors*

```
/* Here, Box uses a parameterized constructor to
   initialize the dimensions of a box.
*/
class Box {
  double width;
  double height;
  double depth;

  // This is the constructor for Box.
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
```

```
class BoxDemo7 {
  public static void main(String args[]) {
    // declare, allocate, and initialize Box objects
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box(3, 6, 9);

    double vol;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
}
```

Introducing Classes

Constructors: ... *Overloaded Constructors*

```
class Student4{
    int id;
    String name;

    Student4(int i,String n){
    id = i;
    name = n;
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    Student4 s1 = new Student4(111,"Karan");
    Student4 s2 = new Student4(222,"Aryan");
    s1.display();
    s2.display();
   }
}
```

```
class Student5{
    int id;
    String name;
    int age;
    Student5(int i,String n){
    id = i;
    name = n;
    }
    Student5(int i,String n,int a){
    id = i;
    name = n;
    age=a;
    }
    void display(){System.out.println(id+" "+name+" "+age);}

    public static void main(String args[]){
    Student5 s1 = new Student5(111,"Karan");
    Student5 s2 = new Student5(222,"Aryan",25);
    s1.display();
    s2.display();
   }
}
```

Introducing Classes

Constructors: ... *Overloaded Methods...*

```
class Volume {

   public void findVolume ( int s) {
      System.out.println ( "Volume of cube is "+ ( s * s * s ) );
   }

   public void findVolume ( int r, int h ) {
      System.out.println ( "Volume of cylinder is "+ ( 3.14 * r * r * h) );
   }

   public void findVolume ( int l, int b, int h) {
      System.out.println ("Volume of cuboid is " + ( l * b * h ) );
   }
}

class VolumeTest {

   public static void main(String[] args) {
      Volume v=new Volume();
      v.findVolume(3);
      v.findVolume(3,4);
      v.findVolume(3,4,7);
   }
}
```

```
byte b = 3;
v.findVolume(b);
findVolume ( short a) // version 1
findVolume ( int a) // version 2
byte a=4;
v.findVolume(a); // version 1 is called and not version 2
```

Introducing Classes

Constructors: ... *Overloaded Constructors...*

Java's Copy Constructor

```
class Student6{
    int id;
    String name;
    Student6(int i,String n){
    id = i;
    name = n;
    }

    Student6(Student6 s){
    id = s.id;
    name =s.name;
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    Student6 s1 = new Student6(111,"Karan");
    Student6 s2 = new Student6(s1);
    s1.display();
    s2.display();
   }
}
```

Introducing Classes

Use of objects as parameter & Methods returning objects

- So far, we have only been using simple types as parameters to methods. However, it is both correct and common to pass objects to methods.

```
// Objects may be passed to methods.
class Test {
  int a, b;

  Test(int i, int j) {
    a = i;
    b = j;
  }

  // return true if o is equal to the invoking object
  boolean equals(Test o) {
    if(o.a == a && o.b == b) return true;
    else return false;
  }
}

class PassOb {
  public static void main(String args[]) {
    Test ob1 = new Test(100, 22);
    Test ob2 = new Test(100, 22);
    Test ob3 = new Test(-1, -1);

    System.out.println("ob1 == ob2: " + ob1.equals(ob2));
    System.out.println("ob1 == ob3: " + ob1.equals(ob3));
  }
}
```

Introducing Classes

Use of objects as parameter & Methods returning objects

- One of the most common uses of object parameters involves constructors.
- Frequently, you will want to construct a new object so that it is initially the same as some existing object.
- ***Example….***

Introducing Classes

Use of objects as parameter & Methods returning objects

```
class Box {
  double width;
  double height;
  double depth;

  // Notice this constructor. It takes an object of type Box.
  Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }

  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }
  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
```

```
class OverloadCons2 {
  public static void main(String args[]) {
    // create boxes using the various constructors
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box();
    Box mycube = new Box(7);

    Box myclone = new Box(mybox1); // create copy of mybox1

    double vol;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);
    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);

    // get volume of cube
    vol = mycube.volume();
    System.out.println("Volume of cube is " + vol);

    // get volume of clone
    vol = myclone.volume();
    System.out.println("Volume of clone is " + vol);
  }
}
```

Introducing Classes

Use of objects as parameter & Methods returning objects

- **Returning Objects:** A method can return any type of data, including class types that you create.

```
// Returning an object.
class Test {
  int a;

  Test(int i) {
    a = i;
  }

  Test incrByTen() {
    Test temp = new Test(a+10);
    return temp;
  }
}

class RetOb {
  public static void main(String args[]) {
    Test ob1 = new Test(2);
    Test ob2;

    ob2 = ob1.incrByTen();
    System.out.println("ob1.a: " + ob1.a);
    System.out.println("ob2.a: " + ob2.a);

    ob2 = ob2.incrByTen();
    System.out.println("ob2.a after second increase: "
                       + ob2.a);
  }
}
```

Introducing Classes

Call by value & Call by reference:

A Closer Look at Argument Passing

- In general, there are two ways that a computer language can pass an argument to a subroutine: *call-by-value and call-by-reference.*
- *call-by-value: Copies the value of an argument into the formal*parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument.
- *call-by-reference:* a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.

REMEMBER *When a primitive type is passed to a method, it is done by use of call-by-value. Objects are implicitly passed by use of call-by-reference.*

Introducing Classes

Call by value & Call by reference: ...

A Closer Look at Argument Passing ...

- *In Java, when you pass a primitive type to a method, it is passed by value. Thus, what occurs to the parameter that receives the argument has no effect outside the method.*

```
// Primitive types are passed by value.
class Test {
  void meth(int i, int j) {
    i *= 2;
    j /= 2;
  }
}
class CallByValue {
  public static void main(String args[]) {
    Test ob = new Test();

    int a = 15, b = 20;

    System.out.println("a and b before call: " +
                       a + " " + b);

    ob.meth(a, b);

    System.out.println("a and b after call: " +
                       a + " " + b);
  }
}
```

Introducing Classes

Call by value & Call by reference: ...

A Closer Look at Argument Passing ...

- *The objects are passed to methods by use of call-by-reference. Changes to the object inside the method do affect the object used as an argument.*

```
// Objects are passed by reference.

class Test {
  int a, b;

  Test(int i, int j) {
    a = i;
    b = j;
  }
  // pass an object
  void meth(Test o) {
    o.a *=  2;
    o.b /= 2;
  }
}

class CallByRef {
  public static void main(String args[]) {
    Test ob = new Test(15, 20);

    System.out.println("ob.a and ob.b before call: " +
                       ob.a + " " + ob.b);

    ob.meth(ob);

    System.out.println("ob.a and ob.b after call: " +
                       ob.a + " " + ob.b);
  }
}
```

Introducing Classes

this keyword:

- Sometimes a method will need to refer to the object that invoked it.
- **this** can be used inside any method to refer to the *current object. That is,* **this** is always a reference to the object on which the method was invoked. You can use **this** anywhere a reference to an object of the current class' type is permitted.

```
// A redundant use of this.
Box(double w, double h, double d) {
        this.width = w;
        this.height = h;
        this.depth = d;
}
```

Instance Variable Hiding

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
        this.width = width;
        this.height = height;
        this.depth = depth;
}
```

Introducing Classes

this keyword: ...

A word of caution: The use of **this** in such a context can sometimes be confusing, and some programmers are careful not to use local variables and formal parameter names that hide instance variables. ***Of course, other programmers believe the contrary—that it is a good convention to use the same names for clarity, and use this to overcome the instance variable hiding.*** It is a matter of taste which approach you adopt.

Introducing Classes

Varargs: Variable-Length Arguments:

- Beginning with JDK 5, Java has included a feature that simplifies the creation of methods that need to take a variable number of arguments. This feature is called *varargs* and it is short for *variable-length arguments.*
- A method that takes a variable number of arguments is called a *variable-arity method,* or simply a *varargs method.*
- Situations that require that a variable number of arguments be passed to a method are not unusual.
- For example, a method that opens an Internet connection might take a user name, password, filename, protocol, and so on, but supply defaults if some of this information is not provided.

Introducing Classes

Varargs: Variable-Length Arguments:...

```
// Use an array to pass a variable number of
// arguments to a method. This is the old-style
// approach to variable-length arguments.
class PassArray {
  static void vaTest(int v[]) {
    System.out.print("Number of args: " + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  public static void main(String args[])
  {
    // Notice how an array must be created to
    // hold the arguments.
    int n1[] = { 10 };
    int n2[] = { 1, 2, 3 };
    int n3[] = { };

    vaTest(n1); // 1 arg
    vaTest(n2); // 3 args
    vaTest(n3); // no args
  }
}
```

The output from the program is shown here:

```
Number of args: 1 Contents: 10
Number of args: 3 Contents: 1 2 3
Number of args: 0 Contents:
```

Introducing Classes

Varargs: Variable-Length Arguments:...

- Avariable-length argument is specified by three periods (...)
- **Eg:** vaTest( ) is written using a **vararg:** ***static void vaTest(int ... v) {***

```
// Demonstrate variable-length arguments.
class VarArgs {

  // vaTest() now uses a vararg.
  static void vaTest(int ... v) {
    System.out.print("Number of args: " + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  public static void main(String args[])
  {
    // Notice how vaTest() can be called with a
    // variable number of arguments.
    vaTest(10);       // 1 arg
    vaTest(1, 2, 3); // 3 args
    vaTest();         // no args
  }
}
```

The output from the program is shown here:

```
Number of args: 1 Contents: 10
Number of args: 3 Contents: 1 2 3
Number of args: 0 Contents:
```

Introducing Classes

Varargs: Variable-Length Arguments:...

- A method can have "normal" parameters along with a variable-length parameter. However, the variable-length parameter must be the last parameter declared by the method.
- **Eg:** This method declaration is perfectly acceptable:

int doIt(int a, int b, double c, int ... vals) {

- Remember, the varargs parameter must be last.
- For example, the following declaration is incorrect:

int doIt(int a, int b, double c, int ... vals, boolean stopFlag) { // Error!

int doIt(int a, int b, double c, int ... vals, double ... morevals) { // Error!

Introducing Classes

Varargs: Variable-Length Arguments:...

```
// Use varargs with standard arguments.
class VarArgs2 {

  // Here, msg is a normal parameter and v is a
  // varargs parameter.
  static void vaTest(String msg, int ... v) {
    System.out.print(msg + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }
  public static void main(String args[])
  {
    vaTest("One vararg: ", 10);
    vaTest("Three varargs: ", 1, 2, 3);
    vaTest("No varargs: ");
  }
}
```

The output from this program is shown here:

```
One vararg: 1 Contents: 10
Three varargs: 3 Contents: 1 2 3
No varargs: 0 Contents:
```

Introducing Classes

Static Variables & Methods:

Understanding static

- Normally, a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance.
- To create such a member, precede its declaration with the keyword **static.** When a member is declared **static,** it can be accessed before any objects of its class are created, and without reference to any object.
- Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.

Introducing Classes

Static Variables & Methods: ...

Understanding static...

- Methods declared as **static** have several restrictions:
  - They can only call other **static** methods.
  - They must only access **static** data.
  - They cannot refer to **this** or **super** in any way. (The keyword **super** relates to inheritance)
- If you need to do computation in order to initialize your **static** variables, you can declare a **static** block that gets executed exactly once, when the class is first loaded.

Introducing Classes

Static Variables & Methods: ...

Understanding static...

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
  static int a = 3;
  static int b;

  static void meth(int x) {
    System.out.println("x = " + x);
    System.out.println("a = " + a);
    System.out.println("b = " + b);
  }

  static {
    System.out.println("Static block initialized.");
    b = a * 4;
  }

  public static void main(String args[]) {
    meth(42);
  }
}
```

Introducing Classes

Static Variables & Methods: ...

Understanding static...

- Outside of the class in which they are defined, **static** methods and variables can be used independently of any object.

classname.method( )

```
class StaticDemo {
  static int a = 42;
  static int b = 99;
  static void callme() {
    System.out.println("a = " + a);
  }
}

class StaticByName {
  public static void main(String args[]) {
    StaticDemo.callme();
    System.out.println("b = " + StaticDemo.b);
  }
}
```

Introducing Classes

Introducing final:

- A variable can be declared as **final. it** prevents its contents from being modified. This means that you must initialize a **final** variable when it is declared**.**
- **For example:**

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

- It is a common coding convention to choose all uppercase identifiers for **final** variables.
- Variables declared as **final** do not occupy memory on a per-instance basis. Thus, a **final** variable is essentially a constant.

Introducing Classes

Garbage collection:

- Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. The technique that accomplishes this is called *garbage collection.*
- **It works like this:** when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.
- Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used.

Introducing Classes

Garbage collection: ...

The finalize( ) Method:

- Sometimes an object will need to perform some action when it is destroyed.
- For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed.
- Java provides a mechanism called *finalization*. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the ***garbage collector.***

Introducing Classes

Garbage collection: ...

The finalize( ) Method:...

- To add a finalizer to a class, you simply define the **finalize( )** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize( )** method, you will specify those actions that must be performed before an object is destroyed.
- The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the **finalize( )** method on the object.

The **finalize( ) method has this general form:**

```
protected void finalize( ) {
        // finalization code here
}
```

Introducing Classes

Garbage collection: ...

The finalize( ) Method:...

- It is important to understand that **finalize( )** is only called just prior to garbage collection.
- It is not called when an object goes out-of-scope. This means that you cannot know when—or even if—**finalize( )** will be executed.
- Therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on **finalize( )** for normal program operation.

Introducing Classes

Nested & Inner classes:

- It is possible to define a class within another class; such classes are known as ***nested classes*.**
- The scope of a nested class is bounded by the scope of its enclosing class.
- ***Eg:*** If class B is defined within class A, then B does not exist independently of A.
- A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.
- A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block.

Introducing Classes

Nested & Inner classes: ...

There are two types of nested classes: ***static and non-static***

- ***Static:*** A static nested class is one that has the **static** modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.
- ***Non-static:*** The most important type of nested class is the *inner* class. An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

Introducing Classes

Nested & Inner classes: ... *Inner/Non-Static classes*

```
// Demonstrate an inner class.
class Outer {
  int outer_x = 100;

  void test() {
    Inner inner = new Inner();
    inner.display();
  }

  // this is an inner class
  class Inner {
    void display() {
      System.out.println("display: outer_x = " + outer_x);
    }
  }
}

class InnerClassDemo {
  public static void main(String args[]) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

Introducing Classes

Nested & Inner classes: ... *Inner/Non-Static classes...*

```
// This program will not compile.
class Outer {
  int outer_x = 100;

  void test() {
    Inner inner = new Inner();
    inner.display();
  }

  // this is an inner class
  class Inner {
    int y = 10; // y is local to Inner
    void display() {
      System.out.println("display: outer_x = " + outer_x);
    }
  }

  void showy() {
    System.out.println(y); // error, y not known here!
  }
}

class InnerClassDemo {
  public static void main(String args[]) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

Introducing Classes

Nested & Inner classes: ... *Inner/Non-Static classes...*

```
// Define an inner class within a for loop.
class Outer {
  int outer_x = 100;

  void test() {
    for(int i=0; i<10; i++) {
      class Inner {
        void display() {
          System.out.println("display: outer_x = " + outer_x);
        }
      }
      Inner inner = new Inner();
      inner.display();
    }
  }
}

class InnerClassDemo {
  public static void main(String args[]) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

While nested classes are not applicable to all situations, they are particularly helpful when handling events.

One final point: Nested classes were not allowed by the original 1.0 specification for Java. They were added by Java 1.1

Introducing Classes

The Stack class: *An example...*

```
// This class defines an integer stack that can hold 10 values.
class Stack {
  int stck[] = new int[10];
  int tos;

  // Initialize top-of-stack
  Stack() {
    tos = -1;
  }

  // Push an item onto the stack
  void push(int item) {
    if(tos==9)
      System.out.println("Stack is full.");
    else
      stck[++tos] = item;
  }

  // Pop an item from the stack
  int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}
```

```
class TestStack {
  public static void main(String args[]) {
    Stack mystack1 = new Stack();
    Stack mystack2 = new Stack();

    // push some numbers onto the stack
    for(int i=0; i<10; i++) mystack1.push(i);
    for(int i=10; i<20; i++) mystack2.push(i);

    // pop those numbers off the stack
    System.out.println("Stack in mystack1:");
    for(int i=0; i<10; i++)
      System.out.println(mystack1.pop());

    System.out.println("Stack in mystack2:");
    for(int i=0; i<10; i++)
      System.out.println(mystack2.pop());
  }
}
```

Introducing Classes

The Stack class: *An example...Modified*

```
// Improved Stack class that uses the length array member.
class Stack {
  private int stck[];
  private int tos;

  // allocate and initialize stack
  Stack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // Push an item onto the stack
  void push(int item) {
    if(tos==stck.length-1) // use length member
      System.out.println("Stack is full.");
    else
      stck[++tos] = item;
  }

  // Pop an item from the stack
  int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}
```

```
class TestStack2 {
  public static void main(String args[]) {
    Stack mystack1 = new Stack(5);
    Stack mystack2 = new Stack(8);
    // push some numbers onto the stack
    for(int i=0; i<5; i++) mystack1.push(i);
    for(int i=0; i<8; i++) mystack2.push(i);

    // pop those numbers off the stack
    System.out.println("Stack in mystack1:");
    for(int i=0; i<5; i++)
      System.out.println(mystack1.pop());

    System.out.println("Stack in mystack2:");
    for(int i=0; i<8; i++)
      System.out.println(mystack2.pop());
  }
}
```

String Handling

Prepared using following Resources:

- Herbert Schildt, “Java: The Complete Reference”, Tata McGrawHill Education
- E Balagurusamy, Programming with Java - A Tata McGraw Hill Education
- https://www.geeksforgeeks.org/java/
- https://www.javatpoint.com/java-tutorial
- https://www.tutorialspoint.com/java/index.htm
- https://www.w3schools.com/java/

By: DIVAKARA .N

String Handling

- The String Constructors
- String Length
- Special String Operations
- Character Extraction
- String Comparison
- Searching Strings
- Modifying Strings
- String Buffer
- concept of mutable and immutable string
- Command line arguments and
- basics of I/O operations – keyboard input using BufferedReader & Scanner classes.

String Handling

A Few Words About Strings – *A brief overview*:

- **String,** is not a simple type. Nor is it simply an array of characters. Rather, **String** defines an object.
- The **String** type is used to declare string variables. You can also declare arrays of strings.
- A quoted string constant can be assigned to a **String** variable.
- A variable of type **String** can be assigned to another variable of type **String.** You can use an object of type **String** as an argument to **println( ).**

```
String str = "this is a test";
System.out.println(str);
```

- *Here,* ***str*** *is an object of type* ***String****. It is assigned the string "this is a test". This string is displayed by the* ***println( )*** *statement.*
- **String** objects have many special features and attributes that make them quite powerful and easy to use.

String Handling

Overview:

- As is the case in most other programming languages, in Java a *string* is a *sequence of characters*. But, unlike many other languages that *implement strings as character arrays*, *Java implements* strings as ***objects*** of type **String.**
- Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient.
- ***For example***, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string.

String Handling

Overview: ...

- Once a **String** object has been created, you cannot change the characters that comprise that string. You can still perform all types of string operations.
- Each time you need an altered version of an existing string, a new **String** object is created that contains the modifications. The original string is left unchanged(Immutable).
- Immutable strings can be implemented more efficiently than changeable ones(Mutable).
- Java provides two options: **StringBuffer** and **StringBuilder.** Both hold strings that can be modified after they are created.

String Handling

Overview: ...

- The **String, StringBuffer,** and **StringBuilder** classes are defined in ***java.lang,*** they are available to all programs automatically. All are declared **final**, (none of these classes may be sub-classed)
- **One last point:** The strings within objects of type **String** are unchangeable means that the contents of the **String** instance cannot be changed after it has been created. However, a variable declared as a **String** reference can be changed to point at some other **String** object at any time.

String Handling

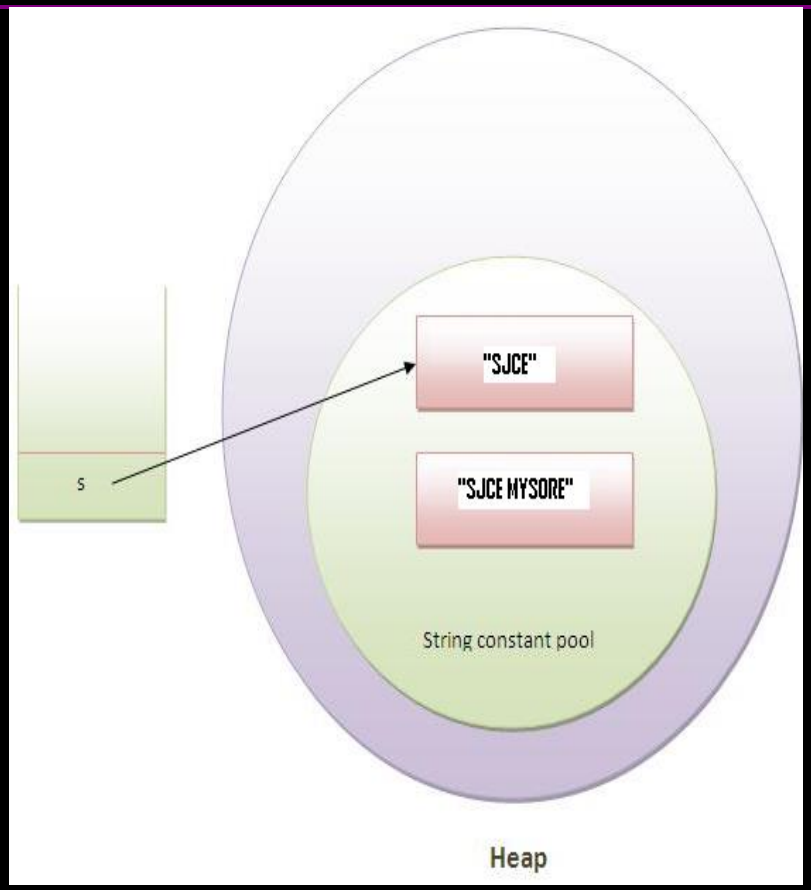
Overview: ...

- String is a sequence of characters.
- Java implements strings as objects of type **String.**
- It belongs to **java.lang (*java.lang.String*)**
- Once a String object is created, it is not possible to change the characters that comprise the string.
- When a modifiable string is needed, java provides two options:
  - ***java.lang.StringBuffer***
  - ***java.lang.StringBuilder***

String Handling

Overview: ...

```
class Simple{
    public static void main(String args[]){
        String s = "SJCE";
        s.concat("  MYSORE");

        // s = s.concat("   MYSORE");

        System.out.println(s);
    }
}
```



Why string objects are immutable in java?

Because java uses the concept of string literal. Suppose there are 5 reference variables, all refers to one object "SJCE". If one reference variable changes the value of the object, it will be affected to all the reference variables. That is why string objects are immutable in java.

String Handling

The String Constructors:

- The **String** class supports several constructors.

- *To create an empty* **String**, you call the default constructor.
  Eg: String s = new String(); // instance with no characters

- To create strings that have initial values by an array of characters, **String(char chars[ ])**
  Eg:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
```

- Specify a sub-range of a character array as an initializer using, **String(char chars[ ], int startIndex, int numChars)**

String Handling

The String Constructors: ...

- specify a sub-range...

Eg:
```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars, 2, 3);
```

- To construct a **String** object that contains the same character sequence as another **String** object using, **String(String strObj)**

```
// Construct one String from another.
class MakeString {
    public static void main(String args[]) {
        char c[] = {'J', 'a', 'v', 'a'};
        String s1 = new String(c);
        String s2 = new String(s1);
        System.out.println(s1);
        System.out.println(s2);
    }
}
```

String Handling

The String Constructors: ...

- Even though Java's **char** type uses 16 bits to represent the basic Unicode character set, the typical format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set.
- Because 8-bit ASCII strings are common, the String class provides constructors that initialize a string when given a byte array. Their forms are shown here:

  String(byte asciiChars[ ])

  String(byte asciiChars[ ], int startIndex, int numChars)
- Here, *asciiChars* specifies the array of bytes. The second form allows you to specify a sub-range.
- In each of these constructors, the byte-to-character conversion is done by using the default character encoding of the platform.

String Handling

The String Constructors: ...

- 8-bit ASCII strings....

```
// Construct string from subset of char array.
class SubStringCons {
        public static void main(String args[]) {
                byte ascii[] = {65, 66, 67, 68, 69, 70 };
                String s1 = new String(ascii);
                System.out.println(s1);
                String s2 = new String(ascii, 2, 3);
                System.out.println(s2);
        }
}
```

***NOTE** :The contents of the array are copied whenever you create a **String** object from an array. If you modify the contents of the array after you have created the string, the **String** will be unchanged.*

You can construct a **String** from a **StringBuffer** by using the constructor, **String(StringBuffer *strBufObj)***

String Handling

The String Constructors: ...

- To create an empty **String**, you call the default constructor, **String();**
  String s = new String();
- To create strings that have initial values by an array of characters,
  String(char chars[ ])
- To specify a sub-range of a character array as an initializer using,
  String(char chars[ ], int startIndex, int numChars)
- To construct a String object that contains the same character sequence as another String object using,
  String(String strObj)
- The String class provides constructors that initialize a string when given a byte array. Their forms are shown here: **String(byte asciiChars[ ])**
  String(byte asciiChars[ ], int startIndex, int numChars)
- To construct a **String** from a **StringBuffer** by using the constructor,
  String(StringBuffer *strBufObj)*

String Handling

String Length:

- The length of a string is the number of characters that it contains.
- To obtain this value, call the **length( )** method, shown here:

int length( )

Eg:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length()); // What is the Size?
```

String Handling

Special String Operations:

- Because strings are a common and important part of programming, Java has added special support for several string operations within the syntax of the language.
- Operations include the automatic creation of new **String** instances from string literals, concatenation of multiple **String** objects by use of the + operator, and the conversion of other data types to a string representation.
- There are explicit methods available to perform all of these functions, but Java does them automatically as a convenience for the programmer and to add clarity.

String Handling

Special String Operations: ...

String Literals: Explicitly create a **String** instance using a string literal. Thus, you can use a string literal to initialize a **String** object.

```
char chars[] = { 'a', 'b', 'c' };
String s1 = new String(chars);

String s2 = "abc"; // use string literal
```

Because a **String** object is created for every string literal, you can use a string literal any place you can use a **String** object.

```
System.out.println("abc".length());
```

String Handling

Special String Operations: ...

String Concatenation: In general, Java does not allow operators to be applied to **String** objects. The one exception to this rule is the + operator, which concatenates two strings, producing a **String** object as the result. This allows you to chain together a series of + operations.

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);
```

String Handling

Special String Operations: ...

String Concatenation: ...

One practical use of string concatenation is found when you are creating very long strings. Instead of letting long strings wrap around within your source code, you can break them into smaller pieces, using the + to concatenate them.

```
// Using concatenation to prevent long lines.
class ConCat {
  public static void main(String args[]) {
    String longStr = "This could have been " +
      "a very long line that would have " +
      "wrapped around.  But string concatenation " +
      "prevents this.";

    System.out.println(longStr);
  }
}
```

String Handling

Special String Operations: ...

String Concatenation with other Data Types:

```
int age = 9;
String s = "He is " + age + " years old.";
System.out.println(s);
```

- The compiler will convert an operand to its string equivalent whenever the other operand of the **+** is an instance of **String.**
- Be careful when you mix other types of operations with string concatenation expressions.

```
String s = "four: " + 2 + 2;
System.out.println(s); // Output: ...
```

String s = "four: " + (2 + 2); // Output: ...

String Handling

Special String Operations: ...

String Conversion and toString( ):

- When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method **valueOf( )** defined by **String**.
- **valueOf( )** is overloaded for all the simple types and for type **Object**.
- *For the simple types,* ***valueOf( )*** *returns a string that contains the human-readable equivalent of the value with which it is called. For objects,* ***valueOf( )*** *calls the* ***toString( )*** *method on the object.*
- Every class implements **toString( )** because it is defined by **Object.**
- For most important classes that you create, you will want to override **toString( )** and provide your own string representations.
- The **toString( )** method has this general form: **String toString( )**

String Handling

Special String Operations: ...

String Conversion and toString( ): ....

```
// Override toString() for Box class.
class Box {
  double width;
  double height;
  double depth;

  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  public String toString() {
    return "Dimensions are " + width + " by " +
           depth + " by " + height + ".";
  }
}
```

```
class toStringDemo {
  public static void main(String args[]) {
    Box b = new Box(10, 12, 14);
    String s = "Box b: " + b; // concatenate Box object

    System.out.println(b); // convert Box to string
    System.out.println(s);
  }
}
```

Box's toString( ) method is automatically invoked when a **Box** object is used in a concatenation expression or in a call to **println( ).**

String Handling

Character Extraction:

- The **String** class provides a number of ways in which characters can be extracted from a **String** object.

- **charAt( ):** To extract a single character from a **String,** you can refer directly to an individual character.

char charAt(int *where)*

Eg: char ch; ch = “abc”.charAt(1); // b is extracted

- **getChars( ):** To extract more than one character at a time.

void getChars(int *sourceStart, int sourceEnd,*
char target[ ], int targetStart)

String Handling

Character Extraction: ...

- **getChars( ): ...**

```
class getCharsDemo {
    public static void main(String args[]) {
        String s = "This is a demo of the getChars method.";
        int start = 10;
        int end = 14;
        char buf[] = new char[end - start];
        s.getChars(start, end, buf, 0);
        System.out.println(buf);
    }
}
```

- **getBytes( ) :** Alternative to **getChars( )** that stores the characters in an array of bytes. It uses the default character-to-byte conversions provided by the platform.

byte[ ] getBytes( )

- most useful when you are exporting a **String** value into an environment that does not support 16-bit Unicode characters. **Eg:** Most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

String Handling

Character Extraction: ...

- **toCharArray( )**: Converts all the characters in a **String** object into a character array, it returns an array of characters for the entire string. **char[ ] toCharArray( )**

- This function is provided as a convenience, since it is possible to use getChars( ) to achieve the same result.

String Handling

String Comparison:

- The **String** class includes several methods that compare strings or substrings within strings.

- **equals( ) and equalsIgnoreCase( ):** To compare two strings for equality.

  - **equals( )** //case-sensitive

    boolean equals(Object str)

  - **equalsIgnoreCase( )** //ignores case differences

    boolean equalsIgnoreCase(String str)

String Handling

String Comparison: ...

- equals( ) and equalsIgnoreCase( ): ...

```
// Demonstrate equals() and equalsIgnoreCase().
class equalsDemo {
  public static void main(String args[]) {
    String s1 = "Hello";
    String s2 = "Hello";
    String s3 = "Good-bye";
    String s4 = "HELLO";
    System.out.println(s1 + " equals " + s2 + " -> " +
                       s1.equals(s2));
    System.out.println(s1 + " equals " + s3 + " -> " +
                       s1.equals(s3));
    System.out.println(s1 + " equals " + s4 + " -> " +
                       s1.equals(s4));
    System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +
                       s1.equalsIgnoreCase(s4));
  }
}
```

String Handling

String Comparison: ...

- **regionMatches( ):**
  - **boolean regionMatches(int startIndex, String str2, int str2StartIndex, int numChars)**
  - **boolean regionMatches(boolean ignoreCase, int startIndex, String str2, int str2StartIndex, int numChars)**

- For both versions, startIndex specifies the index at which the region begins within the invoking String object. The String being compared is specified by str2. The index at which the comparison will start within str2 is specified by str2StartIndex. The length of the substring being compared is passed in numChars.
- In the second version, if ignoreCase is true, the case of the characters is ignored. Otherwise, case is significant.

String Handling

String Comparison: ...

- **startsWith( ) and endsWith( ):** Specialized forms of **regionMatches( ).** These methods determines whether a given String begins/ends with a specified string.

  boolean startsWith(String str)

  boolean endsWith(String str)

  boolean startsWith(String str, int startIndex)

  Eg:

  ```
  "Foobar".endsWith("bar")
  "Foobar".startsWith("Foo")
  "Foobar".startsWith("bar", 3)
  ```

String Handling

String Comparison: ...

equals( ) Versus ==( )

- The ***equals( )*** method compares the characters inside a String object.
- The == operator compares two object references to see whether they refer to the same instance.

```
// equals() vs ==
class EqualsNotEqualTo {
  public static void main(String args[]) {
    String s1 = "Hello";
    String s2 = new String(s1);

    System.out.println(s1 + " equals " + s2 + " -> " +
                       s1.equals(s2));
    System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
  }
}
```

String Handling

String Comparison: ...

- **compareTo( ):** It is not enough to simply know whether two strings are identical. For sorting applications, you need to know which is *less than, equal to, or greater than the next. A string* is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order.

int compareTo(String str)

Value	Meaning
Less than zero	The invoking string is less than *str*.
Greater than zero	The invoking string is greater than *str*.
Zero	The two strings are equal.

String Handling

String Comparison: ...

- **compareTo( ): *An Exmaple***

```
// A bubble sort for Strings.
class SortString {
  static String arr[] = {
    "Now", "is", "the", "time", "for", "all", "good", "men",
    "to", "come", "to", "the", "aid", "of", "their", "country"
  };
  public static void main(String args[]) {
    for(int j = 0; j < arr.length; j++) {
      for(int i = j + 1; i < arr.length; i++) {
        if(arr[i].compareTo(arr[j]) < 0) {
          String t = arr[j];
          arr[j] = arr[i];
          arr[i] = t;
        }
      }
      System.out.println(arr[j]);
    }
  }
}
```

String Handling

Searching Strings:

- The **String** class provides two methods that allow you to search a string for a specified character or substring:
  - **indexOf( )** Searches for the first occurrence of a character or substring.
  - **lastIndexOf( )** Searches for the last occurrence of a character or substring.
- These two methods are overloaded in several different ways.
- *In all cases, the methods return the index at which the character or substring was found, or –1 on failure.*

String Handling

Searching Strings: ...

- To search for the first/last occurrence of a character, use
  int indexOf(int *ch)*
  int lastIndexOf(int *ch)*
- To search for the first/last occurrence of a substring, use
  int indexOf(String *str)*
  int lastIndexOf(String *str)*
- You can specify a starting point for the search using
  int indexOf(int *ch, int startIndex)*
  int lastIndexOf(int *ch, int startIndex)*
  int indexOf(String *str, int startIndex)*
  int lastIndexOf(String *str, int startIndex)*
- Here, *startIndex specifies the index at which point the search begins. For* ***indexOf( )****, the* search runs from *startIndex to the end of the string. For* ***lastIndexOf( ),*** *the search runs from startIndex to zero.*

String Handling

Searching Strings: ...*An Example*

```
// Demonstrate indexOf() and lastIndexOf().
class indexOfDemo {
  public static void main(String args[]) {
    String s = "Now is the time for all good men " +
               "to come to the aid of their country.";

    System.out.println(s);
    System.out.println("indexOf(t) = " +
                       s.indexOf('t'));
    System.out.println("lastIndexOf(t) = " +
                       s.lastIndexOf('t'));
    System.out.println("indexOf(the) = " +
                       s.indexOf("the"));
    System.out.println("lastIndexOf(the) = " +
                       s.lastIndexOf("the"));
    System.out.println("indexOf(t, 10) = " +
                       s.indexOf('t', 10));
    System.out.println("lastIndexOf(t, 60) = " +
                       s.lastIndexOf('t', 60));
    System.out.println("indexOf(the, 10) = " +
                       s.indexOf("the", 10));
    System.out.println("lastIndexOf(the, 60) = " +
                       s.lastIndexOf("the", 60));
  }
}
```

String Handling

Modifying Strings:

- Because **String** objects are immutable, whenever you want to modify a **String,** you must either copy it into a **StringBuffer** or **StringBuilder,** or use one of the following **String** methods, Which will construct a new copy of the string with your modifications complete.
- **substring( )**: To extract a substring use, **substring( ).**
- It has two forms:

  String substring(int *startIndex)*

  String substring(int *startIndex, int endIndex)*
- Here, startIndex specifies the beginning index, and endIndex specifies the stopping point. The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

String Handling

Modifying Strings: ... The following program uses **substring( )** to replace all instances of one substring with another within a string:

```
// Substring replacement.
class StringReplace {
  public static void main(String args[]) {
    String org = "This is a test. This is, too.";
    String search = "is";
    String sub = "was";
    String result = "";
    int i;

    do { // replace all matching substrings
      System.out.println(org);
      i = org.indexOf(search);
      if(i != -1) {
        result = org.substring(0, i);
        result = result + sub;
        result = result + org.substring(i + search.length());
        org = result;
      }
    } while(i != -1);

  }
}
```

String Handling

Modifying Strings: ...

- **concat( )**: To concatenate two stings use, **String substring( ),** performs the same function as **+.**

  Eg:

  ```
  String s1 = "one";
  String s2 = s1.concat("two");   // String s2 = s1 + "two";
  ```

- **replace( )**: To replaces all occurrences of one character in the invoking string with another character.

  String replace(char *original, char replacement)*

  Eg: String str = “NEW".replace(‘E', ‘O');

- The second form of **replace( )** replaces one character sequence with another. It has this general form:

  String replace(CharSequence *original, CharSequence replacement)*

- This form was added by J2SE 5.

String Handling

Modifying Strings: ...

- **trim( ):** returns a copy of the invoking string from which any leading and trailing whitespace has been removed.

  String trim( )

  Eg: String s = " Hello World ".trim();

String Handling

StringBuffer:

- **StringBuffer** is a peer class of **String** that provides much of the functionality of strings.
- **String** represents fixed-length, immutable character sequences. *In contrast,*
- **StringBuffer** represents growable and writeable character sequences**.**
- **StringBuffer** may have characters and substrings inserted in the middle or appended to the end.
- **StringBuffer** will automatically grow to make room for such additions and often has more characters pre-allocated than are actually needed, to allow room for growth.

String Handling

StringBuffer: ... StringBuffer Constructors

StringBuffer defines these four constructors:

- StringBuffer( ): Reserves room for 16 characters without reallocation.
- StringBuffer(int *size):* Accepts an integer argument that explicitly sets the size of the buffer.
- StringBuffer(String *str):* Accepts a **String** argument that sets the initial contents of the **StringBuffer** object and reserves room for 16 more characters without reallocation.
- StringBuffer(CharSequence *chars):* Creates an object that contains the character sequence contained in *chars.*
- **StringBuffer** allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time. Also, frequent reallocations can fragment memory. By allocating room for a few extra characters, **StringBuffer** reduces the number of reallocations that take place.

String Handling

StringBuffer: ...

- **length( ) and capacity( ):** The current length of a **StringBuffer** can be found via the *length( )* method, while the total allocated capacity can be found through the *capacity( )* method. **int length( )** **int capacity( )**

```
// StringBuffer length vs. capacity.
class StringBufferDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("Hello");

    System.out.println("buffer = " + sb);
    System.out.println("length = " + sb.length());
    System.out.println("capacity = " + sb.capacity());
  }
}
```

```
buffer = Hello
length = 5
capacity = 21
```

Since **sb** is initialized with the string **"Hello"** when it is created, its length is 5. Its capacity is 21 because room for 16 additional characters is automatically added.

String Handling

StringBuffer: ... ***Some useful methods***

- **charAt( ) and setCharAt( )**
  char charAt(int *where)*
  void setCharAt(int *where, char ch)*
- **getChars( )**
  void getChars(int sourceStart, int sourceEnd,
  char target[ ], int targetStart)
- **append( )**
  StringBuffer append(String str)
  StringBuffer append(int num)
  StringBuffer append(Object obj)

String Handling

StringBuffer: ... ***Some useful methods***

- **insert( )**
  StringBuffer insert(int index, String str)
  StringBuffer insert(int index, char ch)
  StringBuffer insert(int index, Object obj)
- **reverse( ):** StringBuffer reverse( )
- **delete( ) and deleteCharAt( )**
  StringBuffer delete(int startIndex, int endIndex)
  StringBuffer deleteCharAt(int loc)
- **replace( ):**
  StringBuffer replace(int startIndex, int endIndex, String str)
- **replace( ):** String substring(int startIndex)
  String substring(int startIndex, int endIndex)

String Handling

StringBuilder:

- J2SE 5 adds a new string class to Java's already powerful string handling capabilities, called **StringBuilder.**
- It is identical to **StringBuffer** except for one important difference: it is not synchronized, which means that it is not thread-safe.
- The advantage of **StringBuilder** is faster performance.
- However, in cases in which you are using multithreading, you must use **StringBuffer** rather than **StringBuilder.**

String Handling

Concept of Mutable and Immutable String:

- Java's String is designed to be ***immutable*** *i.e*, once a String is constructed, its contents cannot be modified.

- The Strings within objects of type **String** are unchangeable means the content of the **String** instance cannot be changes after it has been created. However, a variable declared as a **String** reference can be changed to point at some other **String** object at any time.

String Handling

Concept of Mutable and Immutable String:...

- The **StringBuffer** and **StringBuilder** classes are used when there is a necessity to make a lot of modifications to Strings of characters. (*Mutable*)

- The ***String, StringBuffer*** and ***StringBuilder*** classes are defined in ***java.lang***

String Handling

Command -line arguments:

- Sometimes you will want to pass information into a program when you run it. This is accomplished by passing *command-line arguments to* ***main( ).***
- It is the information that directly follows the program's name on the command line when it is executed.
- To access the command-line arguments inside a Java program is quite easy - *they are stored as strings in a* ***String*** *array passed to the args parameter of* ***main( ).*** *The first command-line argument is stored at* ***args[0]****, the second at* ***args[1]****, and so on.*
- ***REMEMBER*** *All command-line arguments are passed as strings. You must convert numeric values to their internal forms manually.*

String Handling

Command -line arguments: ...*Example*

```
// Display all command-line arguments.
class CommandLine {
    public static void main(String args[]) {
        for(int i=0; i<args.length; i++)
        System.out.println("args[" + i + "]: " + args[i]);
    }
}
```

javac CommandLine.java
java CommandLine this is a test 100 -1

```
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

String Handling

Basics of I/O operations: Keyboard input using ***BufferedReader class***

- In Java, console input is accomplished by reading from **System.in**. To obtain a character-based stream that is attached to the console, wrap **System.in** in a **BufferedReader** object.
- **BufferedReader** supports a buffered input stream. Its most commonly used constructor is shown here:

  BufferedReader(Reader *inputReader*)
- Here, *inputReader* is the stream that is linked to the instance of **BufferedReader** that is being created. **Reader** is an abstract class.
- One of its concrete subclasses is **InputStreamReader**, which converts bytes to characters. To obtain an **InputStreamReader** object that is linked to **System.in**, use the following constructor:

  InputStreamReader(InputStream *inputStream*)

String Handling

Basics of I/O operations: Keyboard input using ***BufferedReader class...***

- Because **System.in** refers to an object of type **InputStream**, it can be used for *inputStream*.
- Putting it all together, the following line of code creates a **BufferedReader** that is connected to the keyboard:

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

- After this statement executes, **br** is a character-based stream that is linked to the console through **System.in**.
- **Reading Characters:** To read a character from a **BufferedReader**, use **read( )**. The version of **read( )** that we will be using is

int read( ) throws IOException

String Handling

Basics of I/O operations: Keyboard input using ***BufferedReader class...***

int read( ) throws IOException...*Example*

```
// Use a BufferedReader to read characters from the console.
import java.io.*;
class BRRead {
    public static void main(String args[])
    throws IOException
    {
        char c;
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter characters, 'q' to quit.");

        // read characters
        do {
            c = (char) br.read();
            System.out.println(c);
        } while(c != 'q');
    }
}
```

String Handling

Basics of I/O operations: Keyboard input using ***BufferedReader class...***

Declaration: **public class BufferedReader extends Reader**

Class constructors

BufferedReader(Reader in) : Create a new BufferedReader that will read from the specified subordinate stream with a default buffer size of 8192 chars.

BufferedReader(Reader in, int size) : Create a new BufferedReader that will read from the specified subordinate stream with a buffer size that is specified by the caller.

String Handling

Basics of I/O operations: Keyboard input using *BufferedReader class...*

Method Summary	
void	close() This method closes the underlying stream and frees any associated resources.
void	mark(int readLimit) Mark a position in the input to which the stream can be "reset" by calling the reset() method.
boolean	markSupported() Returns true to indicate that this class supports mark/reset functionality.
int	read() Reads an char from the input stream and returns it as an int in the range of 0-65535.
int	read(char[] buf, int offset, int count) This method read chars from a stream and stores them into a caller supplied buffer.
String	readLine() This method reads a single line of text from the input stream, returning it as a String.
boolean	ready() This method determines whether or not a stream is ready to be read.
void	reset() Reset the stream to the point where the mark() method was called.
long	skip(long count) This method skips the specified number of chars in the stream.

String Handling

Basics of I/O operations:

Keyboard input using ***Scanner class:***

- It is the complement of **Formatter,** reads formatted input and converts it into its binary form.
- Read all types of numeric values, strings, and other types of data, whether it comes from a disk file, the keyboard, or another source.
- **Scanner** can be used to read input from the console, a file, a string, or any source that implements the **Readable interface** or **ReadableByteChannel.**

String Handling

Basics of I/O operations:

Keyboard input using ***Scanner class*:...**

The Scanner Constructors

- **Scanner** defines the constructors, it can be created for a **String,** an **InputStream,** a **File,** or any object that implements the **Readable** or **ReadableByteChannel** interfaces.
- **Eg:** The following sequence creates a **Scanner** that reads the file Test.txt:

```
FileReader fin = new FileReader("Test.txt");
Scanner src = new Scanner(fin);
```

String Handling

Basics of I/O operations:

Keyboard input using ***Scanner class*:...**

The Scanner Constructors...

Method	Description
Scanner(File *from*) throws FileNotFoundException	Creates a **Scanner** that uses the file specified by *from* as a source for input.
Scanner(File *from*, String *charset*) throws FileNotFoundException	Creates a **Scanner** that uses the file specified by *from* with the encoding specified by *charset* as a source for input.
Scanner(InputStream *from*)	Creates a **Scanner** that uses the stream specified by *from* as a source for input.
Scanner(InputStream *from*, String *charset*)	Creates a **Scanner** that uses the stream specified by *from* with the encoding specified by *charset* as a source for input.
Scanner(Readable *from*)	Creates a **Scanner** that uses the **Readable** object specified by *from* as a source for input.
Scanner (ReadableByteChannel *from*)	Creates a **Scanner** that uses the **ReadableByteChannel** specified by *from* as a source for input.
Scanner(ReadableByteChannel *from*, String *charset*)	Creates a **Scanner** that uses the **ReadableByteChannel** specified by *from* with the encoding specified by *charset* as a source for input.
Scanner(String *from*)	Creates a **Scanner** that uses the string specified by *from* as a source for input.

String Handling

Basics of I/O operations:

Keyboard input using ***Scanner class*:...**

Scanning Basics

- Once you have created a **Scanner,** it is a simple matter to use it to read formatted input. It reads *tokens from the underlying source that you specified when the* **Scanner** was created.
- As it relates to **Scanner**, a token is a portion of input that is delineated by a set of delimiters, which is whitespace by default.

String Handling

Basics of I/O operations:

Keyboard input using ***Scanner class*:...**

Scanning Basics...

In general, to use **Scanner,** follow this procedure:

1. Determine if a specific type of input is available by calling one of **Scanner's hasNext*X*** *methods, where **X** is the type of data desired.*
2. If input is available, read it by calling one of **Scanner's next*X*** *methods.*
3. Repeat the process until input is exhausted.

The following sequence shows how to read a list of integers from the keyboard.

```
Scanner conin = new Scanner(System.in);
int i;
// Read a list of integers.
while(conin.hasNextInt()) {
        i = conin.nextInt();
        // ...
}
```

String Handling

Basics of I/O operations:

Keyboard input using ***Scanner class*:...**

Scanning Basics... The Scanner **hasNext** Methods

Method	Description
boolean hasNext( )	Returns **true** if another token of any type is available to be read. Returns **false** otherwise.
boolean hasNext(Pattern *pattern*)	Returns **true** if a token that matches the pattern passed in *pattern* is available to be read. Returns **false** otherwise.
boolean hasNext(String *pattern*)	Returns **true** if a token that matches the pattern passed in *pattern* is available to be read. Returns **false** otherwise.
boolean hasNextBigDecimal( )	Returns **true** if a value that can be stored in a **BigDecimal** object is available to be read. Returns **false** otherwise.
boolean hasNextBigInteger( )	Returns **true** if a value that can be stored in a **BigInteger** object is available to be read. Returns **false** otherwise. The default radix is used. (Unless changed, the default radix is 10.)
boolean hasNextBigInteger(int *radix*)	Returns **true** if a value in the specified radix that can be stored in a **BigInteger** object is available to be read. Returns **false** otherwise.
boolean hasNextBoolean( )	Returns **true** if a **boolean** value is available to be read. Returns **false** otherwise.
boolean hasNextByte( )	Returns **true** if a **byte** value is available to be read. Returns **false** otherwise. The default radix is used. (Unless changed, the default radix is 10.)
boolean hasNextByte(int *radix*)	Returns **true** if a **byte** value in the specified radix is available to be read. Returns **false** otherwise.
boolean hasNextDouble( )	Returns **true** if a **double** value is available to be read. Returns **false** otherwise.
boolean hasNextFloat( )	Returns **true** if a **float** value is available to be read. Returns **false** otherwise.
boolean hasNextInt( )	Returns **true** if an **int** value is available to be read. Returns **false** otherwise. The default radix is used. (Unless changed, the default radix is 10.)
boolean hasNextInt(int *radix*)	Returns **true** if an **int** value in the specified radix is available to be read. Returns **false** otherwise.
boolean hasNextLine( )	Returns **true** if a line of input is available.
boolean hasNextLong( )	Returns **true** if a **long** value is available to be read. Returns **false** otherwise. The default radix is used. (Unless changed, the default radix is 10.)
boolean hasNextLong(int *radix*)	Returns **true** if a **long** value in the specified radix is available to be read. Returns **false** otherwise.
boolean hasNextShort( )	Returns **true** if a **short** value is available to be read. Returns **false** otherwise. The default radix is used. (Unless changed, the default radix is 10.)
boolean hasNextShort(int *radix*)	Returns **true** if a **short** value in the specified radix is available to be read. Returns **false** otherwise.

String Handling

Basics of I/O operations:

Keyboard input using

Scanner class:...

Scanning Basics...

The Scanner **next** Methods

Method	Description
String next( )	Returns the next token of any type from the input source.
String next(Pattern *pattern*)	Returns the next token that matches the pattern passed in *pattern* from the input source.
String next(String *pattern*)	Returns the next token that matches the pattern passed in *pattern* from the input source.
BigDecimal nextBigDecimal( )	Returns the next token as a **BigDecimal** object.
BigInteger nextBigInteger( )	Returns the next token as a **BigInteger** object. The default radix is used. (Unless changed, the default radix is 10.)
BigInteger nextBigInteger(int *radix*)	Returns the next token (using the specified radix) as a **BigInteger** object.
boolean nextBoolean( )	Returns the next token as a **boolean** value.
byte nextByte( )	Returns the next token as a **byte** value. The default radix is used. (Unless changed, the default radix is 10.)
byte nextByte(int *radix*)	Returns the next token (using the specified radix) as a **byte** value.
double nextDouble( )	Returns the next token as a **double** value.
float nextFloat( )	Returns the next token as a **float** value.
int nextInt( )	Returns the next token as an **int** value. The default radix is used. (Unless changed, the default radix is 10.)
int nextInt(int *radix*)	Returns the next token (using the specified radix) as an **int** value.
String nextLine( )	Returns the next line of input as a string.
long nextLong( )	Returns the next token as a **long** value. The default radix is used. (Unless changed, the default radix is 10.)
long nextLong(int *radix*)	Returns the next token (using the specified radix) as a **long** value.
short nextShort( )	Returns the next token as a **short** value. The default radix is used. (Unless changed, the default radix is 10.)
short nextShort(int *radix*)	Returns the next token (using the specified radix) as a **short** value.

Inheritance

[Reusable Properties]

Prepared using following Resources:

- Herbert Schildt, “Java: The Complete Reference”, Tata McGrawHill Education
- E Balagurusamy, Programming with Java - A Tata McGraw Hill Education
- https://www.geeksforgeeks.org/java/
- https://www.javatpoint.com/java-tutorial
- https://www.tutorialspoint.com/java/index.htm
- https://www.w3schools.com/java/

By: DIVAKARA .N

Inheritance - *Reusable Properties*

INHERITANCE: Super class & subclasses including multilevel hierarchy, process of constructor calling in inheritance, use of 'super' and 'final' keywords with super() method, dynamic method dispatch, use of abstract classes & methods, Method call binding, Overriding vs. overloading, Abstract classes and methods, Constructors and polymorphism, Order of constructor calls.

Inheritance

- Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications.
- Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it.
- In the terminology of Java, a class that is inherited is called a *superclass*. The class that does the inheriting is called a *subclass.*
- Therefore, a *subclass* is a specialized version of a *superclass.* It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.

Inheritance

- **Inheritance Basics**
- To inherit a class, you simply incorporate the definition of one class into another by using the ***extends*** keyword.
- The general form of a **class** declaration that inherits a superclass:

```
class subclass-name extends superclass-name {
    // body of class
}
```

- You can only specify one superclass for any subclass that you create. Java does not support the inheritance of multiple superclasses into a single subclass. You can, as stated, create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass. However, no class can be a superclass of itself.

Inheritance

- **Inheritance Basics ...**

```
// A simple example of inheritance.

// Create a superclass.
class A {
  int i, j;

  void showij() {
    System.out.println("i and j: " + i + "
  }
}

// Create a subclass by extending class A.
class B extends A {
  int k;

  void showk() {
    System.out.println("k: " + k);
  }
  void sum() {
    System.out.println("i+j+k: " + (i+j+k))
  }
}
```

```
class SimpleInheritance {
  public static void main(String args[]) {
    A superOb = new A();
    B subOb = new B();

    // The superclass may be used by itself.
    superOb.i = 10;
    superOb.j = 20;
    System.out.println("Contents of superOb: ");
    superOb.showij();
    System.out.println();

    /* The subclass has access to all public members of
       its superclass. */
    subOb.i = 7;
    subOb.j = 8;
    subOb.k = 9;
    System.out.println("Contents of subOb: ");
    subOb.showij();
    subOb.showk();
    System.out.println();

    System.out.println("Sum of i, j and k in subOb:");
    subOb.sum();
  }
}
```

Inheritance

- **Inheritance Basics ...**
- **Member Access and Inheritance:** Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private.**

```
/* In a class hierarchy, private members remain
   private to their class.

   This program contains an error and will not
   compile.
*/

// Create a superclass.
class A {
  int i; // public by default
  private int j; // private to A

  void setij(int x, int y) {
    i = x;
    j = y;
  }
}
```

```
// A's j is not accessible here.
class B extends A {
  int total;
  void sum() {
    total = i + j; // ERROR,
  }
}
```

REMEMBER *A class member that has been declared as* ***private*** *will remain private to its class. It is not accessible by any code outside its class, including subclasses.*

Inheritance

- **Inheritance Basics ...** ***More Practical Example***

```
// This program uses inheritance to extend Box.
class Box {
  double width;
  double height;
  double depth;

  // construct clone of an object
  Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }

  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
```

```
// Here, Box is extended to include weight.
class BoxWeight extends Box {
  double weight; // weight of box

  // constructor for BoxWeight
  BoxWeight(double w, double h, double d, double m) {
    width = w;
    height = h;
    depth = d;
    weight = m;
  }
}

class DemoBoxWeight {
  public static void main(String args[]) {
    BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
    BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
    double vol;

    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);
    System.out.println("Weight of mybox1 is " + mybox1.weight);
    System.out.println();

    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);
    System.out.println("Weight of mybox2 is " + mybox2.weight);
  }
}
```

Inheritance

- **Inheritance Basics ...** ***More Practical Example***
- A major advantage of inheritance is that once you have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses. Each subclass can precisely tailor its own classification. For example, the following class inherits **Box and adds a color attribute:**

```
// Here, Box is extended to include color.
class ColorBox extends Box {
  int color; // color of box

  ColorBox(double w, double h, double d, int c) {
    width = w;
    height = h;
    depth = d;
    color = c;
  }
}
```

- **Remember:** Once you have created a superclass that defines the general aspects of an object, that superclass can be inherited to form specialized classes. Each subclass simply adds its own unique attributes. This is the essence of inheritance.

Inheritance

- **Inheritance Basics ...**
- **A Superclass Variable Can Reference a Subclass Object:**
- A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.
- It is important to understand that it is the type of the reference variable — not the type of the object that it refers to — that determines what members can be accessed. That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass, *because the superclass has no knowledge of what a subclass adds to it.*

Inheritance

- **Inheritance Basics ...**
- **A Superclass Variable Can Reference a Subclass Object: ...**

```
class RefDemo {
  public static void main(String args[]) {
    BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
    Box plainbox = new Box();
    double vol;

    vol = weightbox.volume();
    System.out.println("Volume of weightbox is " + vol);
    System.out.println("Weight of weightbox is " +
                       weightbox.weight);
    System.out.println();

    // assign BoxWeight reference to Box reference
    plainbox = weightbox;

    vol = plainbox.volume(); // OK, volume() defined in Box
    System.out.println("Volume of plainbox is " + vol);

    /* The following statement is invalid because plainbox
       does not define a weight member. */
//  System.out.println("Weight of plainbox is " + plainbox.weight);
  }
}
```

Inheritance

- Using **super**
- Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super.**
- **super** has two general forms.
  - The first calls the superclass' constructor.
  - The second is used to access a member of the superclass that has been hidden by a member of a subclass.

Inheritance

- Using **super ...**

Using super to Call Superclass Constructors

- A subclass can call a constructor defined by its superclass by use of the following form of **super:**

super(*arg-list*);

- Here, *arg-list* specifies any arguments needed by the constructor in the superclass. **super( )** must always be the first statement executed inside a subclass' constructor.

```
// BoxWeight now uses super to initialize its Box attributes.
class BoxWeight extends Box {
  double weight; // weight of box

  // initialize width, height, and depth using super()
  BoxWeight(double w, double h, double d, double m) {
    super(w, h, d); // call superclass constructor
    weight = m;
  }
}
```

Inheritance

- Using **super ...**

Using super to Call Superclass Constructors...

- Since constructors can be overloaded, **super( )** can be called using any form defined by the superclass.
- **The** constructor executed will be the one that matches the arguments.

Inheritance

- Using **super ...**

Using super to Call Superclass Constructors...

```
// A complete implementation of BoxWeight.
class Box {
  private double width;
  private double height;
  private double depth;

  // construct clone of an object
  Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }

  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
```

```
// BoxWeight now fully implements all constructors.
class BoxWeight extends Box {
  double weight; // weight of box

  // construct clone of an object
  BoxWeight(BoxWeight ob) { // pass object to constructor
    super(ob);
    weight = ob.weight;
  }

  // constructor when all parameters are specified
  BoxWeight(double w, double h, double d, double m) {
    super(w, h, d); // call superclass constructor
    weight = m;
  }

  // default constructor
  BoxWeight() {
    super();
    weight = -1;
  }

  // constructor used when cube is created
  BoxWeight(double len, double m) {
    super(len);
    weight = m;
  }
}
```

Inheritance

- Using **super ...**

Using super to Call Superclass Constructors...

```
class DemoSuper {
  public static void main(String args[]) {
    BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
    BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
    BoxWeight mybox3 = new BoxWeight(); // default
    BoxWeight mycube = new BoxWeight(3, 2);
    BoxWeight myclone = new BoxWeight(mybox1);
    double vol;

    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);
    System.out.println("Weight of mybox1 is " + mybox1.weight);
    System.out.println();

    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);
    System.out.println("Weight of mybox2 is " + mybox2.weight);
    System.out.println();

    vol = mybox3.volume();
    System.out.println("Volume of mybox3 is " + vol);
    System.out.println("Weight of mybox3 is " + mybox3.weight);
    System.out.println();

    vol = myclone.volume();
    System.out.println("Volume of myclone is " + vol);
    System.out.println("Weight of myclone is " + myclone.weight);
    System.out.println();

    vol = mycube.volume();
    System.out.println("Volume of mycube is " + vol);
    System.out.println("Weight of mycube is " + mycube.weight);
    System.out.println();
  }
}
```

This program generates the following output:

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3

Volume of mybox2 is 24.0
Weight of mybox2 is 0.076

Volume of mybox3 is -1.0
Weight of mybox3 is -1.0

Volume of myclone is 3000.0
Weight of myclone is 34.3

Volume of mycube is 27.0
Weight of mycube is 2.0
```

Inheritance

- Using **super ...**

Using super to Call Superclass Constructors...

```
// construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
  super(ob);
  weight = ob.weight;
}
```

- Notice that **super( )** is passed an object of type **BoxWeight**—not of type **Box**. This still invokes the constructor **Box(Box ob)**. As mentioned earlier, *a superclass variable can be used to reference any object derived from that class*.
- Thus, we are able to pass a **BoxWeight** object to the **Box** constructor. *Of course*, ***Box*** *only has knowledge of its own members*.
- When a subclass calls **super( )**, it is calling the constructor of its immediate superclass. Thus, **super( )** always refers to the superclass immediately above the calling class.
- This is true even in a multileveled hierarchy. Also, **super( )** must always be the first statement executed inside a subclass constructor.

Inheritance

- Using **super ...**

A Second Use for *super*

- Acts somewhat like *this*, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

super.*member*

Here, *member can be either a method or an instance variable.*

- This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

Inheritance

- Using **super ...**

A Second Use for *super* ...

- Although the instance variable **i** in **B** hides the **i** in **A**, **super** allows access to the **i** defined in the superclass.
- As you will see, **super** can also be used to call methods that are hidden by a subclass.

```
// Using super to overcome name hiding.
class A {
  int i;
}

// Create a subclass by extending class A.
class B extends A {
  int i; // this i hides the i in A

  B(int a, int b) {
    super.i = a; // i in A
    i = b; // i in B
  }

  void show() {
    System.out.println("i in superclass: " + super.i);
    System.out.println("i in subclass: " + i);
  }
}

class UseSuper {
  public static void main(String args[]) {
    B subOb = new B(1, 2);

    subOb.show();
  }
}
```

This program displays the following:

```
i in superclass: 1
i in subclass: 2
```

Inheritance

- **Creating a Multilevel Hierarchy**
- Builds hierarchies that contain as many layers of inheritance, uses a subclass as a superclass of another.

Inheritance

- **Creating a Multilevel Hierarchy:...**

```
// Start with Box.
class Box {
  private double width;
  private double height;
  private double depth;

  // construct clone of an object
  Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }

  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
```

Inheritance

- **Creating a Multilevel Hierarchy:...**

```
// Add weight.
class BoxWeight extends Box {
  double weight; // weight of box

  // construct clone of an object
  BoxWeight(BoxWeight ob) { // pass object to constructor
    super(ob);
    weight = ob.weight;
  }
  // constructor when all parameters are specified
  BoxWeight(double w, double h, double d, double m) {
    super(w, h, d); // call superclass constructor
    weight = m;
  }

  // default constructor
  BoxWeight() {
    super();
    weight = -1;
  }
  // constructor used when cube is created
  BoxWeight(double len, double m) {
    super(len);
    weight = m;
  }
}
```

```
// Add shipping costs.
class Shipment extends BoxWeight {
  double cost;

  // construct clone of an object
  Shipment(Shipment ob) { // pass object to constructor
    super(ob);
    cost = ob.cost;
  }

  // constructor when all parameters are specified
  Shipment(double w, double h, double d,
           double m, double c) {
    super(w, h, d, m); // call superclass constructor
    cost = c;
  }

  // default constructor
  Shipment() {
    super();
    cost = -1;
  }

  // constructor used when cube is created
  Shipment(double len, double m, double c) {
    super(len, m);
    cost = c;
  }
}
```

Inheritance

- **Creating a Multilevel Hierarchy:...**

```
class DemoShipment {
  public static void main(String args[]) {
    Shipment shipment1 =
               new Shipment(10, 20, 15, 10, 3.41);
    Shipment shipment2 =
               new Shipment(2, 3, 4, 0.76, 1.28);

    double vol;

    vol = shipment1.volume();
    System.out.println("Volume of shipment1 is " + vol);
    System.out.println("Weight of shipment1 is "
                       + shipment1.weight);
    System.out.println("Shipping cost: $" + shipment1.cost);
    System.out.println();
    vol = shipment2.volume();
    System.out.println("Volume of shipment2 is " + vol);
    System.out.println("Weight of shipment2 is "
                       + shipment2.weight);
    System.out.println("Shipping cost: $" + shipment2.cost);
  }
}
```

```
Volume of shipment1 is 3000.0
Weight of shipment1 is 10.0
Shipping cost: $3.41

Volume of shipment2 is 24.0
Weight of shipment2 is 0.76
Shipping cost: $1.28
```

*The entire class hierarchy, including **Box, BoxWeight, and Shipment, is shown all in one file.** In Java, all three classes could have been placed into their own files and compiled separately. In fact, using separate files is the norm, not the exception, in creating class hierarchies.*

Inheritance

- In a class hierarchy, constructors are called in order of derivation, from superclass to subclass.
- Further, since **super( )** must be the first statement executed in a subclass' constructor, this order is the same whether or not **super( )** is used.
- If **super( )** is not used, then the default or parameterless constructor of each superclass will be executed.

Inheritance

```
// Demonstrate when constructors are called.

// Create a super class.
class A {
  A() {
    System.out.println("Inside A's constructor.");
  }
}
// Create a subclass by extending class A.
class B extends A {
  B() {
    System.out.println("Inside B's constructor.");
  }
}

// Create another subclass by extending B.
class C extends B {
  C() {
    System.out.println("Inside C's constructor.");
  }
}

class CallingCons {
  public static void main(String args[]) {
    C c = new C();
  }
}
```

The output from this program is shown here:

```
Inside A's constructor
Inside B's constructor
Inside C's constructor
```

The constructors are called in order of derivation, it makes sense that constructors are executed in order of derivation.

Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it must be executed first.

Inheritance

Method Overriding:

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override the method in* the superclass.
- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

Inheritance

Method Overriding:...

```
// Method overriding.
class A {
  int i, j;
  A(int a, int b) {
    i = a;
    j = b;
  }

  // display i and j
  void show() {
    System.out.println("i and j: " + i + " " + j);
  }
}
```

When show( ) is invoked on an object of type B, the version of show( ) defined within B is used. That is, the version of show( ) inside B overrides the version declared in A. To access the superclass version of an overridden method, you can do so by using super.

```
class B extends A {
  int k;

  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }
  void show() {
    super.show(); // this calls A's show()
    System.out.println("k: " + k);
  }
}
```

```
class B extends A {
  int k;

  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }

  // display k - this overrides show() in A
  void show() {
    System.out.println("k: " + k);
  }
}

class Override {
  public static void main(String args[]) {
    B subOb = new B(1, 2, 3);

    subOb.show(); // this calls show() in B
  }
}
```

Inheritance

Method Overriding:...

```
// Methods with differing type signatures are overloaded - not
// overridden.
class A {
  int i, j;

  A(int a, int b) {
    i = a;
    j = b;
  }

  // display i and j
  void show() {
    System.out.println("i and j: " + i + " " + j);
  }
}

// Create a subclass by extending class A.
class B extends A {
  int k;

  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }

  // overload show()
  void show(String msg) {
    System.out.println(msg + k);
  }
}
```

```
class Override {
  public static void main(String args[]) {
    B subOb = new B(1, 2, 3);

    subOb.show("This is k: "); // this calls show() in B
    subOb.show(); // this calls show() in A
  }
}
```

- Method overriding occurs *only when the names and the type signatures of the two* methods are identical.
- If they are not, then the two methods are simply overloaded.

Inheritance

Dynamic Method Dispatch:

- Method overriding forms the basis for one of Java's most powerful concepts: *dynamic method dispatch.*
- *Dynamic method dispatch* is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- Dynamic method dispatch is important because this is how Java implements run-time polymorphism.
- **An important principle:** ***A superclass reference variable can refer to a subclass object.***
- ***Java uses this fact to resolve calls to overridden methods at run time.***

Inheritance

Dynamic Method Dispatch:...

- When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs.
- Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called.

Inheritance

Dynamic Method Dispatch:...

- **In other words, *it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.***
- Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

Inheritance

Dynamic Method Dispatch:...

```
// Dynamic Method Dispatch
class A {
   void callme() {
     System.out.println("Inside A's callme method");
  }
}

class B extends A {
  // override callme()
  void callme() {
    System.out.println("Inside B's callme method");
  }
}

class C extends A {
  // override callme()
  void callme() {
    System.out.println("Inside C's callme method");
  }
}
```

```
class Dispatch {
  public static void main(String args[]) {
    A a = new A(); // object of type A
    B b = new B(); // object of type B
    C c = new C(); // object of type C
    A r; // obtain a reference of type A
    r = a; // r refers to an A object
    r.callme(); // calls A's version of callme

    r = b; // r refers to a B object
    r.callme(); // calls B's version of callme

    r = c; // r refers to a C object
    r.callme(); // calls C's version of callme
  }
}
```

Inheritance

Dynamic Method Dispatch:...

Why Overridden Methods?

- Overridden methods allow Java to support run-time polymorphism. Polymorphism is essential to object-oriented programming for one reason: ***It allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.***
- Overridden methods are another way that Java implements the **"One interface, Multiple methods"** aspect of polymorphism.

Inheritance

Dynamic Method Dispatch:...*Applying Method Overriding*

```
// Using run-time polymorphism.
class Figure {
  double dim1;
  double dim2;

  Figure(double a, double b) {
    dim1 = a;
    dim2 = b;
  }

  double area() {
    System.out.println("Area for Figure is undefined.");
    return 0;
  }
}

class Rectangle extends Figure {
  Rectangle(double a, double b) {
    super(a, b);
  }

  // override area for rectangle
  double area() {
    System.out.println("Inside Area for Rectangle.");
    return dim1 * dim2;
  }
}

class Triangle extends Figure {
  Triangle(double a, double b) {
    super(a, b);
  }

  // override area for right triangle
  double area() {
    System.out.println("Inside Area for Triangle.");
    return dim1 * dim2 / 2;
  }
}
```

```
class FindAreas {
  public static void main(String args[]) {
    Figure f = new Figure(10, 10);
    Rectangle r = new Rectangle(9, 5);
    Triangle t = new Triangle(10, 8);
    Figure figref;

    figref = r;
    System.out.println("Area is " + figref.area());

    figref = t;
    System.out.println("Area is " + figref.area());

    figref = f;
    System.out.println("Area is " + figref.area());
  }
}
```

Inheritance

- **Using Abstract Classes:**
- There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.
- Sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement.
- One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method.

Inheritance

- **Using Abstract Classes:...**
- You can require that certain methods be overridden by subclasses by specifying the **abstract** type modifier. These methods are sometimes referred to as ***subclasser responsibility*** because they have no implementation specified in the superclass. Thus, a subclass must override them—it cannot simply use the version defined in the superclass.
- To declare an abstract method, use this general form:

abstract *type name(parameter-list);*

No method body is present.

Inheritance

- **Using Abstract Classes:...**
- ***Any class that contains one or more abstract methods must also be declared abstract.***
- To declare a class abstract, you simply use the **abstract** keyword in front of the class keyword at the beginning of the class declaration.
- There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator. Such objects would be useless, because an abstract class is not fully defined. ***Also, you cannot declare abstract constructors, or abstract static methods.***
- Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared **abstract.**

Inheritance

- **Using Abstract Classes:...** ***An Example***

```
// A Simple demonstration of abstract.
abstract class A {
  abstract void callme();

  // concrete methods are still allowed in abstract classes
  void callmetoo() {
    System.out.println("This is a concrete method.");
  }
}

class B extends A {
  void callme() {
    System.out.println("B's implementation of callme.");
  }
}

class AbstractDemo {
  public static void main(String args[]) {
    B b = new B();

    b.callme();
    b.callmetoo();
  }
}
```

Abstract classes can include as much implementation as they see fit.

Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references.

Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object.

Inheritance

- **Using Abstract Classes:...** ***An Example***

```
// Using abstract methods and classes.
abstract class Figure {
  double dim1;
  double dim2;

  Figure(double a, double b) {
    dim1 = a;
    dim2 = b;
  }

  // area is now an abstract method
  abstract double area();
}

class Rectangle extends Figure {
  Rectangle(double a, double b) {
    super(a, b);
  }

  // override area for rectangle
  double area() {
    System.out.println("Inside Area for Rectangle.");
    return dim1 * dim2;
  }
}

class Triangle extends Figure {
  Triangle(double a, double b) {
    super(a, b);
  }

  // override area for right triangle
  double area() {
    System.out.println("Inside Area for Triangle.");
    return dim1 * dim2 / 2;
  }
}
```

```
class AbstractAreas {
  public static void main(String args[]) {
  // Figure f = new Figure(10, 10); // illegal now
    Rectangle r = new Rectangle(9, 5);
    Triangle t = new Triangle(10, 8);
    Figure figref; // this is OK, no object is created

    figref = r;
    System.out.println("Area is " + figref.area());
    figref = t;
    System.out.println("Area is " + figref.area());
  }
```

Inheritance

- **Using *final* with Inheritance:**
- The keyword **final** has three uses.
  - First, it can be used to create the equivalent of a named constant.
  - The other two uses of **final** apply to inheritance.

Using final to Prevent Overriding

- To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden.

Inheritance

- **Using *final* with Inheritance:...**

Using final to Prevent Overriding

- To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden.

```
class A {
  final void meth() {
    System.out.println("This is a final method.");
  }
}

class B extends A {
  void meth() { // ERROR! Can't override.
    System.out.println("Illegal!");
  }
}
```

Inheritance

- **Using *final* with Inheritance:...**

Using final to Prevent Overriding...

- Methods declared as **final** can sometimes provide a performance enhancement: The compiler is free to *inline* calls to them because it "knows" they will not be overridden by a subclass. When a small **final** method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call.
- Inlining is only an option with **final** methods. Normally, Java resolves calls to methods dynamically, at run time. This is called *late binding*. However, since **final** methods cannot be overridden, a call to one can be resolved at compile time. This is called *early binding*.

Inheritance

- **Using *final* with Inheritance:...**

Using final to Prevent Inheritance

- Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final.** Declaring a class as **final** implicitly declares all of its methods as **final**, too.
- It is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

```
final class A {
  // ...
}

// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
  // ...
}
```

Inheritance

- **The Object Class:**
- There is one special class, **Object**, defined by Java. All other classes are subclasses of **Object.**
- That is, **Object** is a superclass of all other classes. This means that a reference variable of type **Object** can refer to an object of any other class.
- Also, since arrays are implemented as classes, a variable of type **Object** can also refer to any array.

Inheritance

- **The Object Class:...**
- **Object** defines the following methods, which means that they are available in every object.

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

Inheritance

- **Additional Coverage:**
- **Inheritance in java** is a mechanism in which one object acquires all the properties and behaviors of parent object.
- The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.
- Inheritance represents the **IS-A relationship**, also known as parent-child relationship. **IS-A is a way of saying: This object *is a* type of that object.**

Inheritance

- **Additional Coverage:...**
- The **extends** keyword is used to achieve inheritance.

```
public class Animal{

}
    public class Mammal extends Animal{

    }
    public class Reptile extends Animal{

    }
        public class Dog extends Mammal{

        }
```

Inheritance

- **Additional Coverage:...*Example***

```
public class Animal{
}
public class Mammal extends Animal{
}
public class Dog extends Mammal{

   public static void main(String args[]){

      Animal a = new Animal();
      Mammal m = new Mammal();
      Dog d = new Dog();

      System.out.println(m instanceof Animal);
      System.out.println(d instanceof Mammal);
      System.out.println(d instanceof Animal);
   }
}
```

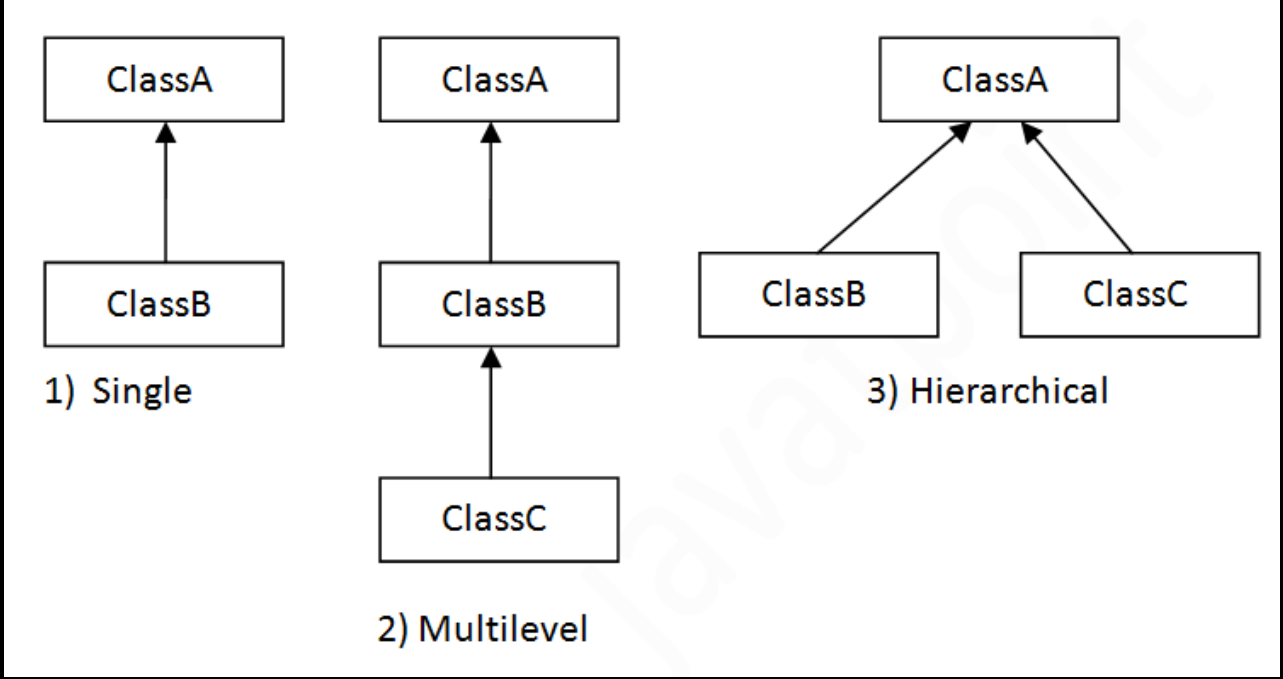
Inheritance

- **Additional Coverage:...**

Why use inheritance in java?

- ✓ For Method Overriding (so runtime polymorphism can be achieved).
- ✓ For Code Reusability.

- **Types:**



Inheritance

- **Additional Coverage:...*Examples***

```
class Parent
{
    public void p1()
    {
        System.out.println("Parent method");
    }
}
public class Child extends Parent {
    public void c1()
    {
        System.out.println("Child method");
    }
    public static void main(String[] args)
    {
        Child cobj = new Child();
        cobj.c1();    //method of Child class
        cobj.p1();    //method of Parent class
    }
}
```

```
class Vehicle
{
    String vehicleType;
}
public class Car extends Vehicle {

    String modelType;
    public void showDetail()
    {
        vehicleType = "Car";        //accessing Vehicle class member
        modelType = "sports";
        System.out.println(modelType+" "+vehicleType);
    }
    public static void main(String[] args)
    {
        Car car =new Car();
        car.showDetail();
    }
}
```

Inheritance

- **Additional Coverage:...*Examples***

Example of Child class refering Parent class `property` using `super` keyword

```
class Parent
{
    String name;

}
public class Child extends Parent {
    String name;
    public void details()
    {
        super.name = "Parent";       //refers to parent class member
        name = "Child";
        System.out.println(super.name+" and "+name);
    }
    public static void main(String[] args)
    {
        Child cobj = new Child();
        cobj.details();
    }
}
```

Inheritance

- **Additional Coverage:...*Examples***

Example of Child class calling Parent class `constructor` using `super` keyword

```
class Parent
{
    String name;

    public Parent(String n)
    {
        name = n;
    }

}
public class Child extends Parent {
    String name;

    public Child(String n1, String n2)
    {

        super(n1);        //passing argument to parent class constructor
        this.name = n2;
    }
    public void details()
    {
        System.out.println(super.name+" and "+name);
    }
     public static void main(String[] args)
    {
        Child cobj = new Child("Parent","Child");
        cobj.details();
    }
}
```

Packages & Interfaces

Prepared using following Resources:

- Herbert Schildt, “Java: The Complete Reference”, Tata McGrawHill Education
- E Balagurusamy, Programming with Java - A Tata McGraw Hill Education
- https://www.geeksforgeeks.org/java/
- https://www.javatpoint.com/java-tutorial
- https://www.tutorialspoint.com/java/index.htm
- https://www.w3schools.com/java/

By: DIVAKARA .N

Packages & Interfaces

Packages: Defining a Package, Finding Packages and CLASSPATH, Access Protection, Importing Packages.

Interfaces: Defining an Interface, Implementing Interfaces, Nested Interfaces, Applying Interfaces, Variables in Interfaces, Interfaces Can Be Extended.

Packages and Interfaces

Packages:

- Defining a Package
- Finding Packages and CLASSPATH
- Access Protection
- Importing Packages

Interfaces:

- Defining an Interface
- Implementing Interfaces
- Nested Interfaces Applying Interfaces
- Variables in Interfaces
- Interfaces Can Be Extended

Packages

Basics:

- *Packages* are containers for classes that are used to keep the class name space compartmentalized.
- **Eg:** A package allows you to create a class named **List,** which you can store in your own package without concern that it will collide with some other class named **List** stored elsewhere.
- Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.
- A Java's mechanism for partitioning the class name space into more manageable chunks.

Packages

Basics: ...

- The package is both a naming and a visibility control mechanism.
- One can define classes inside a package that are not accessible by code outside that package.
- Also define class members that are only exposed to other members of the same package.
- This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

Packages

Basics: ...

The benefits of organising classes into packages are:

- The classes contained in the packages of other programs/applications can be reused.
- In packages classes can be unique compared with classes in other packages. That two classes in two different packages can have the same name. If there is a naming clash, then classes can be accessed with their fully qualified name.
- Classes in packages can be hidden if we don't want other packages to access them.
- Also provide a way for separating "design" from coding.
- ***Packages enable grouping of functionally related classes.***

Packages

Basics: ...

The Java Foundation Packages

- Java provides a large number of classes groped into different packages based on their functionality.

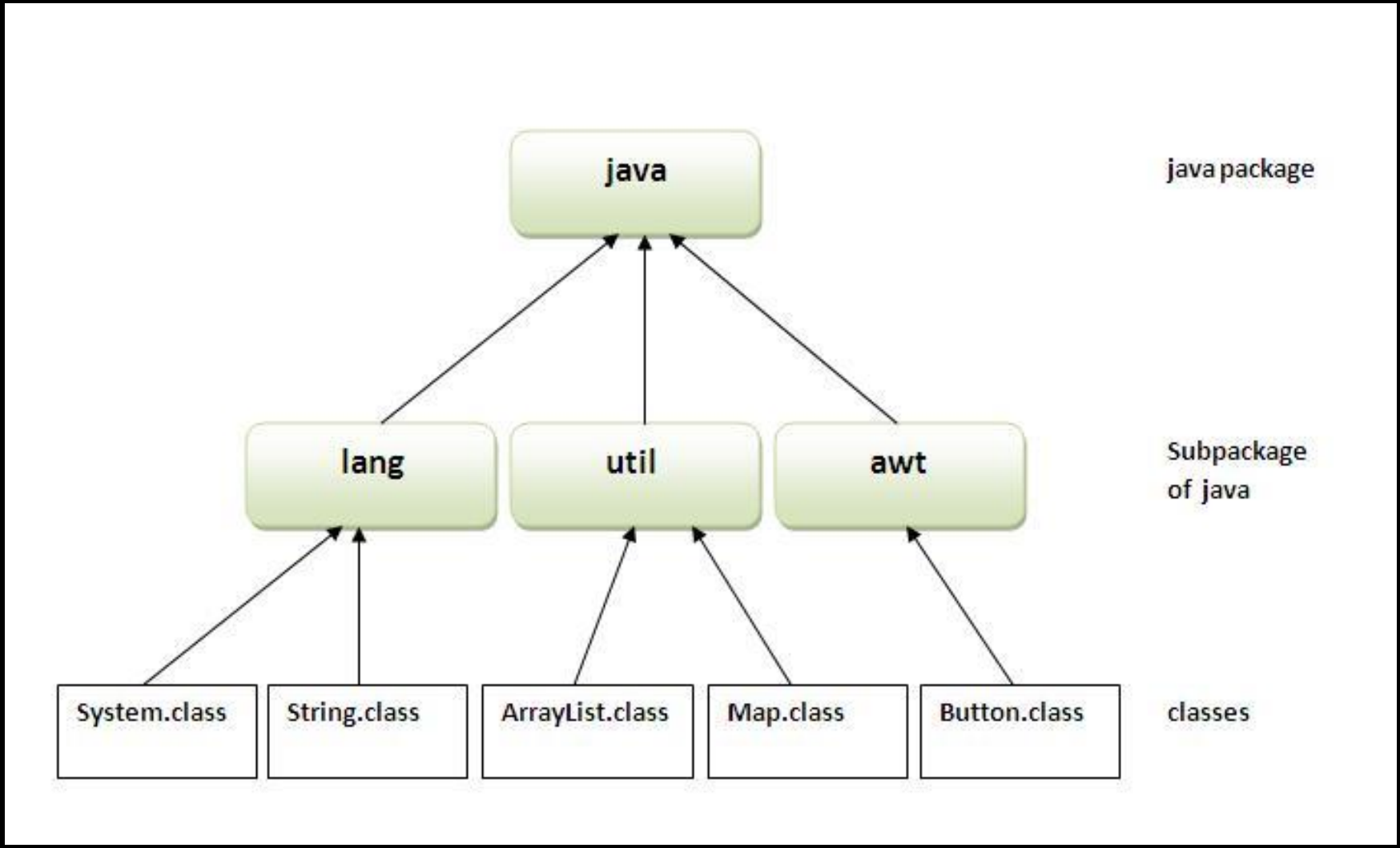
The six foundation Java packages are:

- ***java.lang:*** Classes for primitive types, strings, math functions, threads, and exception
- ***java.util:*** Classes such as vectors, hash tables, date etc.
- ***java.io:*** Stream classes for I/O
- ***java.awt:*** Classes for implementing GUI – windows, buttons, menus etc.
- ***java.net:*** Classes for networking
- ***java.applet:*** Classes for creating and implementing applets

Packages

Basics: ...

The Java Foundation Packages



Packages

Defining a Package:

- Include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package.
- The **package** statement defines a name space in which classes are stored. If you omit the package statement, the class names are put into the **default** package, which has no name.
- While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code.
- General Form: **package *pkg;*** **Eg:** package MyPackage;

Packages

Defining a Package: ...

- Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage.**
- ***Remember that case is significant, and the directory name must match the package name exactly.***
- More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package.
- Most real-world packages are spread across many files.

Packages

Defining a Package: ...

- You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement **package *pkg1[.pkg2[.pkg3]];***
- A package hierarchy must be reflected in the file system of your Java development system.
- **Eg:** A package declared as **package java.awt.image;** needs to be stored in **java\awt\image** in a Windows environment.
- Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

Packages

Finding Packages and CLASSPATH:

- Packages are mirrored by directories. This raises an important question: ***How does the Java run-time system know where to look for packages that you create?***
- The answer has three parts.
  - ✓ First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found.
  - ✓ Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable.
  - ✓ Third, you can use the **-classpath** option with **java** and **javac** to specify the path to your classes.

Packages

Finding Packages and CLASSPATH: ...

- **Eg:** ***package MyPack;*** In order for a program to find **MyPack,** one of three things must be true.
  - ✓ Either the program can be executed from a directory immediately above **MyPack,** *or*
  - ✓ the **CLASSPATH** must be set to include the path to **MyPack,** *or*
  - ✓ the **-classpath** option must specify the path to **MyPack** when the program is run via **java.**
- When the second two options are used, the class path *must not include **MyPack***, itself. It must simply specify the *path to* **MyPack.** For example, in a Windows environment, if the path to **MyPack** is C:\MyPrograms\Java\MyPack Then the class path to **MyPack** is C:\MyPrograms\Java\MyPack

Packages

Finding Packages and CLASSPATH: ...

- **A Short Package Example**

```
// A simple package
package MyPack;

class Balance {
  String name;
  double bal;

  Balance(String n, double b) {
    name = n;
    bal = b;
  }

  void show() {
    if(bal<0)
      System.out.print("--> ");
    System.out.println(name + ": $" + bal);
  }
}
class AccountBalance {
  public static void main(String args[]) {
    Balance current[] = new Balance[3];
    current[0] = new Balance("K. J. Fielding", 123.23);
    current[1] = new Balance("Will Tell", 157.02);
    current[2] = new Balance("Tom Jackson", -12.33);

    for(int i=0; i<3; i++) current[i].show();
  }
}
```

Call this file **AccountBalance.java** and put it in a directory called **MyPack.** Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory. Then, try executing the **AccountBalance** class, using the following command line:

java MyPack.AccountBalance

Remember, you will need to be in the directory above **MyPack** when you execute this command.

AccountBalance is now part of the package **MyPack.** This means that it cannot be executed by itself.

Packages

Finding Packages and CLASSPATH: ...

- **A Short Package Example ...**

```
package name_of_folder_given
public class A
 {
  public void display()
   {
    System.ou.println("Hello world");
   }
 }
```

```
import name_of_the_folder_given.*;
class packagedemo
 {
  public static void main(String arg[])
   {
    A ob = new A();
    ob.display();
   }
 }
```

```
package mypackage;
public class A
{
  public void display()
   {
     System.out.println("hello world");
   }
}
```

```
import mypackage.*;
class packagedemo
{
public static void main(String arg[])
  {
    A ob = new A();
    ob.display();
  }
}
```

Packages

Access Protection:

- Packages add another dimension to access control. Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages.
- Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods.
- Packages act as containers for classes and other subordinate packages.
- Classes act as containers for data and code. The class is Java's smallest unit of abstraction.

Packages

Access Protection: ...

- Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:
  - Subclasses in the same package
  - Non-subclasses in the same package
  - Subclasses in different packages
  - Classes that are neither in the same package nor subclasses
- The three access specifiers, **private, public,** and **protected,** provide a variety of ways to produce the many levels of access required by these categories.

Packages

Access Protection: ...

- While Java's access control mechanism may seem complicated, we can simplify it as follows:
  - Anything declared **public** can be accessed from anywhere.
  - Anything declared **private** cannot be seen outside of its class.
  - When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access.
  - If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element **protected**.

Packages

Access Protection: ...

- Class Member Access - ***Applies only to members of classes.***

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

A non-nested class has only two possible access levels: default and public. When a class is declared as **public**, it is accessible by any other code.

If a class has default access, then it can only be accessed by other code within its same package.

When a class is **public**, it must be the only public class declared in the file, and the file must have the same name as the class.

Packages

Access Protection: ...

- *An Access Example*

This is file **Protection.java**:

```
package p1;

public class Protection {
  int n = 1;
  private int n_pri = 2;
  protected int n_pro = 3;
  public int n_pub = 4;

  public Protection() {
    System.out.println("base constructor");
    System.out.println("n = " + n);
    System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

This is file **Derived.java**:

```
package p1;

class Derived extends Protection {
  Derived() {
    System.out.println("derived constructor");
    System.out.println("n = " + n);

//  class only
//  System.out.println("n_pri = "4 + n_pri);

    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

This is file **SamePackage.java**:

```
package p1;

class SamePackage {
  SamePackage() {

    Protection p = new Protection();
    System.out.println("same package constructor");
    System.out.println("n = " + p.n);

//  class only
//  System.out.println("n_pri = " + p.n_pri);
    System.out.println("n_pro = " + p.n_pro);
    System.out.println("n_pub = " + p.n_pub);
  }
}
```

Packages

Access Protection: ...

- *An Access Example ...*

This is file **Protection2.java**:

```
package p2;

class Protection2 extends p1.Protection {
  Protection2() {
    System.out.println("derived other package constructor");

//  class or package only
//  System.out.println("n = " + n);

//  class only
//  System.out.println("n_pri = " + n_pri);

    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

This is file **OtherPackage.java**:

```
package p2;

class OtherPackage {
  OtherPackage() {
    p1.Protection p = new p1.Protection();
    System.out.println("other package constructor");

//  class or package only
//  System.out.println("n = " + p.n);

//  class only
//  System.out.println("n_pri = " + p.n_pri);

//  class, subclass or package only
//  System.out.println("n_pro = " + p.n_pro);

    System.out.println("n_pub = " + p.n_pub);
  }
}
```

Packages

Access Protection: ...

- *An Access Example ...*

If you wish to try these two packages, here are two test files you can use. The one for package **p1** is shown here:

```
// Demo package p1.
package p1;

// Instantiate the various classes in p1.
public class Demo {
  public static void main(String args[]) {
    Protection ob1 = new Protection();
    Derived ob2 = new Derived();
    SamePackage ob3 = new SamePackage();
  }
}
```

The test file for **p2** is shown next:

```
// Demo package p2.
package p2;

// Instantiate the various classes in p2.
public class Demo {
  public static void main(String args[]) {
    Protection2 ob1 = new Protection2();
    OtherPackage ob2 = new OtherPackage();
  }
}
```

Packages

Importing Packages:

- Given that packages exist and are a good mechanism for compartmentalizing diverse classes from each other, it is easy to see why all of the built-in Java classes are stored in packages.
- There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package. Since classes within packages must be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package path name for every class you want to use.
- For this reason, Java includes the ***import*** statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name.

Packages

Importing Packages: ...

- The **import** statement is a convenience to the programmer and is not technically needed to write a complete Java program. If you are going to refer to a few dozen classes in your application, however, the **import** statement will save a lot of typing.
- In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions. The general form of the import statement:

import pkg1[.pkg2].(classname|*);

- Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (.).

Packages

Importing Packages: ...

- There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify either an explicit *classname* or a star (*), which indicates that the Java compiler should import the entire package.
- This code fragment shows both forms in use:

```
import java.util.Date;
import java.io.*;
```

- ***CAUTION*** *The star form may increase compilation time-especially if you import several large packages. For this reason it is a good idea to explicitly name the classes that you want to use rather than importing whole packages. However, the star form has absolutely no effect on the run-time performance or size of your classes.*

Packages

Importing Packages: ...

- All of the standard Java classes included with Java are stored in a package called **java.**
- The basic language functions are stored in a package inside of the **java** package called **java.lang**
- Normally, you have to import every package or class that you want to use, but since Java is useless without much of the functionality in **java.lang**, it is implicitly imported by the compiler for all programs. This is equivalent to the following line being at the top of all of your programs: ***import java.lang.*;***
- If a class with the same name exists in two different packages that you import using the star form, the compiler will remain silent, unless you try to use one of the classes. In that case, you will get a compile-time error and have to explicitly name the class specifying its package.

Packages

Importing Packages: ...

- It must be emphasized that the **import** statement is optional. Any place you use a class name, you can use its *fully qualified name*, which includes its full package hierarchy.
- For example, this fragment uses an import statement:

```
import java.util.*;
class MyDate extends Date {
}
```

- The same example without the **import** statement looks like this:

```
class MyDate extends java.util.Date {
}
```

- In this version, **Date** is fully-qualified.

Packages

Importing Packages: ...

- When a package is imported, only those items within the package declared as **public** will be available to non-subclasses in the importing code.
- For example, if you want the **Balance** class of the package **MyPack** shown earlier to be available as a stand-alone class for general use outside of **MyPack**, then you will need to declare it as **public** and put it into its own file.

Packages

Importing Packages: ...

```
package MyPack;

/* Now, the Balance class, its constructor, and its
   show() method are public.  This means that they can
   be used by non-subclass code outside their package.
*/
public class Balance {
  String name;
  double bal;

  public Balance(String n, double b) {
    name = n;
    bal = b;
  }

  public void show() {
    if(bal<0)
      System.out.print("--> ");
    System.out.println(name + ": $" + bal);
  }
}
```

```
import MyPack.*;

class TestBalance {
  public static void main(String args[]) {

    /* Because Balance is public, you may use Balance
       class and call its constructor. */
    Balance test = new Balance("J. J. Jaspers", 99.88);

    test.show(); // you may also call show()
  }
}
```

Interfaces

Basics:

- Through the use of the **interface** keyword, Java allows you to fully abstract the interface from its implementation.
- Using **interface**, you can specify a set of methods that can be implemented by one or more classes. The **interface,** itself, does not actually define any implementation.
- Although they are similar to abstract classes, **interfaces** have an additional capability: A class can implement more than one interface. By contrast, a class can only inherit a single superclass (abstract or otherwise).

Interfaces

Basics: ...

- Using the keyword **interface**, you can fully abstract a class' interface from its implementation. Also you can specify what a class must do, but not how it does it.
- **Interfaces** are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.
- Once it is defined, any number of classes can implement an **interface.** Also, one class can implement any number of interfaces.

Interfaces

Basics: ...

- To implement an **interface**, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation.
- By providing the **interface** keyword, Java allows you to fully utilize the ***"one interface, multiple methods"*** aspect of polymorphism.
- Interfaces are designed to support dynamic method resolution at run time.
- ***NOTE*** *Interfaces add most of the functionality that is required for many applications that would normally resort to using multiple inheritance in a language such as C++.*

Interfaces

Basics: ...

Why use Java interface? *There are mainly three reasons:*

It is used to achieve fully abstraction.

By interface, we can support the functionality of multiple inheritance.

It can be used to achieve loose coupling.

- The java compiler adds **public** and **abstract** keywords before the interface method and **public, static** and **final** keywords before data members.

Interfaces

Basics: ...

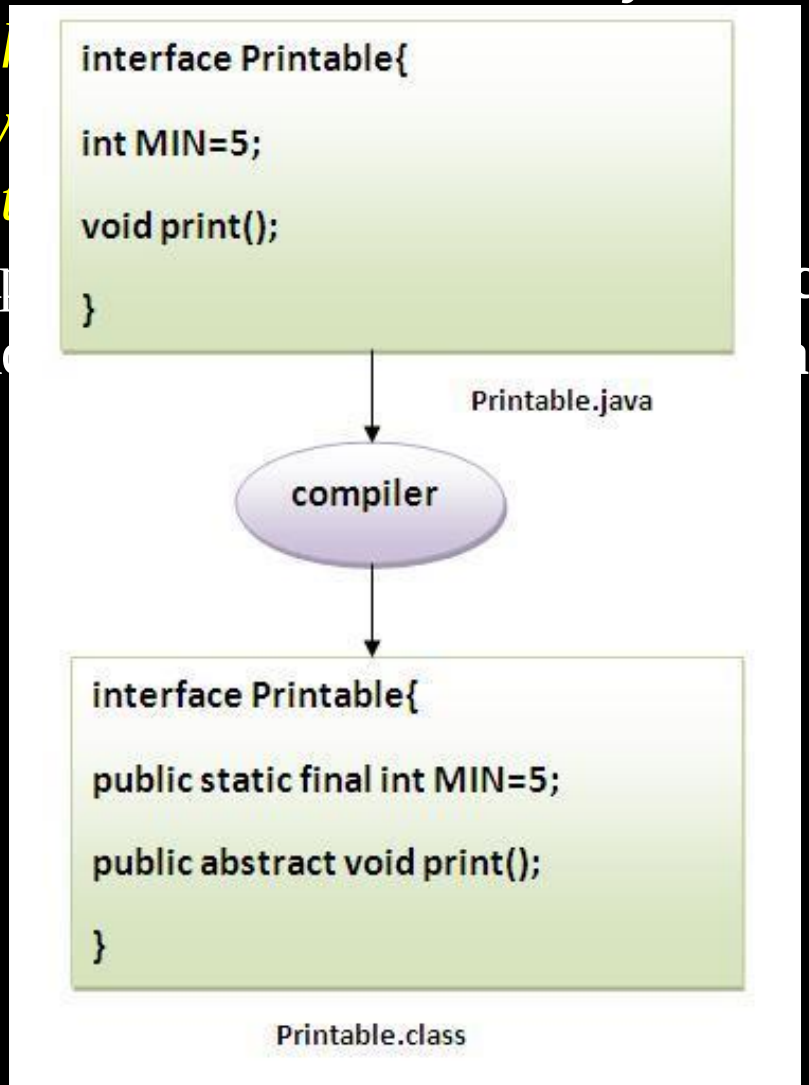
Why use Java interface? *There are mainly three reasons:*

It is used to ac

By interface, w *of multiple inheritance.*

It can be used

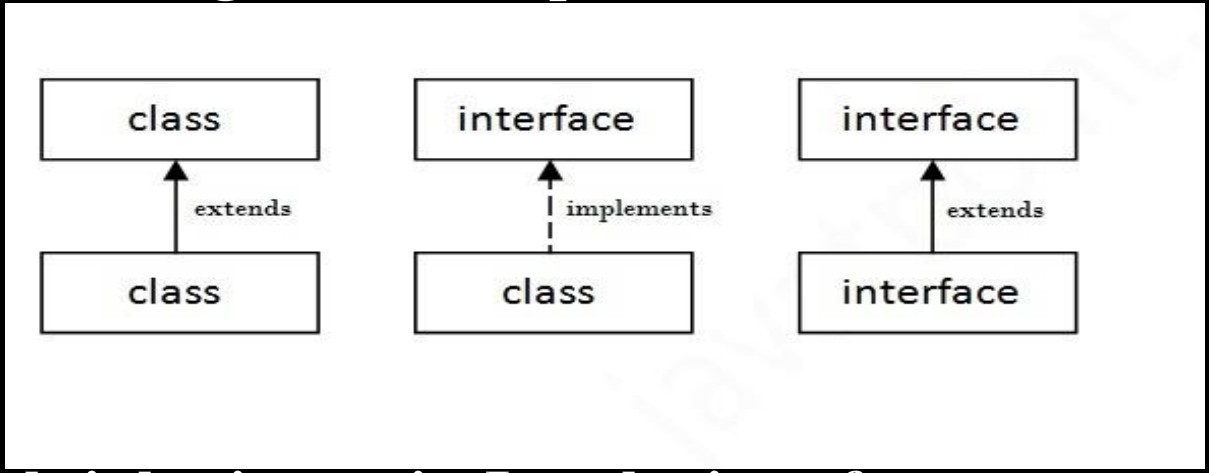
- The java comp *ct* keywords before the interface meth *l* keywords before data members.



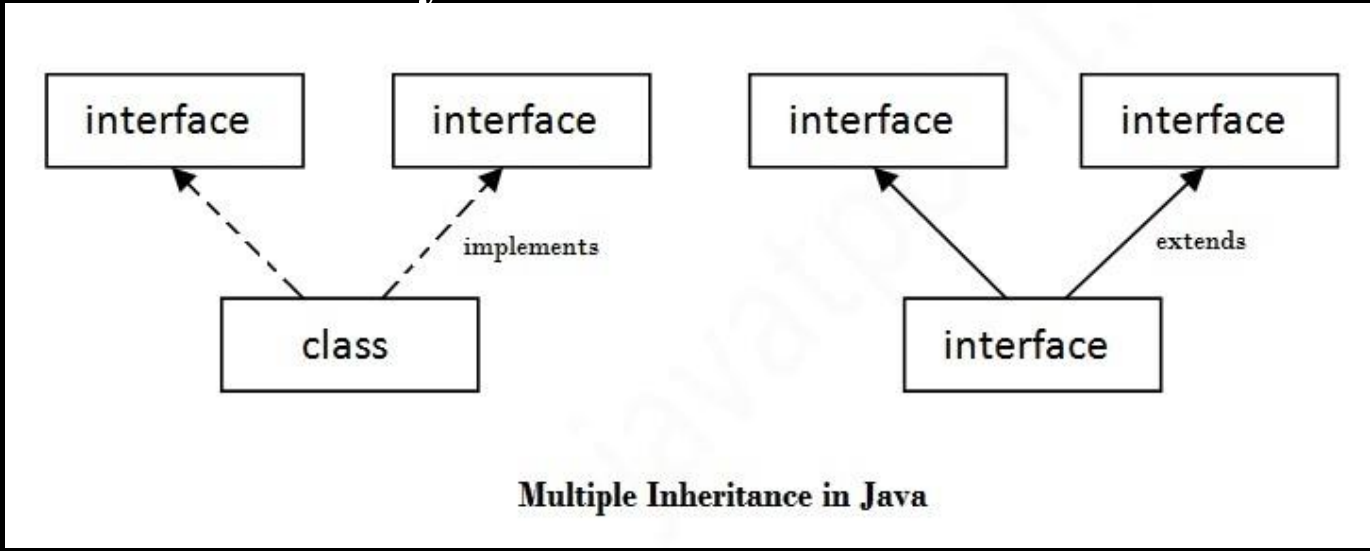
Interfaces

Basics: ...

- Understanding relationship between classes and interfaces



- Multiple inheritance in Java by interface



Interfaces

Defining an Interface:

- An interface is defined much like a class. The General form:

```
access interface name {
    return-type method-name1(parameter-list);
    return-type method-name2(parameter-list);
    type final-varname1 = value;
    type final-varname2 = value;
    // ...
    return-type method-nameN(parameter-list);
    type final-varnameN = value;
}
```

- When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as **public,** the interface can be used by any other code. In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface.

Interfaces

Defining an Interface: ...

- Notice that the methods that are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface.
- Each class that includes an interface must implement all of the methods.
- Variables can be declared inside of interface declarations. They are implicitly **final** and **static,** meaning they cannot be changed by the implementing class. They must also be initialized.
- All methods and variables are implicitly **public.**

Interfaces

Defining an Interface: ...

- Examples:

```
interface Callback {
  void callback(int param);
}
```

```
interface MyInterface
{
  /* All the methods are public abstract by default
   * Note down that these methods are not having body
   */
  public void method1();
  public void method2();
}
```

Interfaces

Implementing Interfaces:

- Once an **interface** has been defined, one or more classes can implement that interface.
- To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface.

```
class classname [extends superclass] [implements interface [,interface...]] {
    // class-body
}
```

- If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface.
- The methods that implement an interface must be declared **public**. Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

Interfaces

Implementing Interfaces: ...

```
class Client implements Callback {
  // Implement Callback's interface
  public void callback(int p) {

    System.out.println("callback called with " + p);
  }
}
```

REMEMBER *When you implement an interface method, it must be declared as public.*

- It is both permissible and common for classes that implement interfaces to define additional members of their own.

```
class Client implements Callback {
  // Implement Callback's interface
  public void callback(int p) {
    System.out.println("callback called with " + p);
  }

  void nonIfaceMeth() {
    System.out.println("Classes that implement interfaces " +
                       "may also define other members, too.");
  }
}
```

Interfaces

Implementing Interfaces: ...

- Accessing Implementations Through Interface References

```
class TestIface {
  public static void main(String args[]) {
    Callback c = new Client();
    c.callback(42);
  }
}
```

```
// Another implementation of Callback.
class AnotherClient implements Callback {
  // Implement Callback's interface
  public void callback(int p) {
    System.out.println("Another version of callback");
    System.out.println("p squared is " + (p*p));
  }
}
```

```
class TestIface2 {
  public static void main(String args[]) {
    Callback c = new Client();
    AnotherClient ob = new AnotherClient();

    c.callback(42);

    c = ob; // c now refers to AnotherClient object
    c.callback(42);
  }
}
```

Interfaces

Implementing Interfaces: ...

- **Partial Implementations**
- If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as **abstract.**

```
abstract class Incomplete implements Callback {
  int a, b;
  void show() {
    System.out.println(a + " " + b);
  }
  // ...
}
```

- Here, the class **Incomplete** does not implement **callback( )** and must be declared as **abstract.**
- Any class that inherits **Incomplete** must implement **callback( )** or be declared **abstract** itself.

Interfaces

Nested Interfaces:

- An interface can be declared a member of a class or another interface. Such an interface is called a *member interface* or *a nested interface.*
- A nested interface can be declared as **public, private, or protected.** This differs from a top-level interface, which must either be declared as **public** or use the **default** access level.
- When a **nested interface** is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member. Thus, outside of the class or interface in which a nested interface is declared, its name must be fully qualified.

Interfaces

Nested Interfaces: ...

```
// A nested interface example.

// This class contains a member interface.
class A {
  // this is a nested interface
  public interface NestedIF {
    boolean isNotNegative(int x);
  }
}

// B implements the nested interface.
class B implements A.NestedIF {
  public boolean isNotNegative(int x) {
    return x < 0 ? false : true;
  }
}
```

```
class NestedIFDemo {
  public static void main(String args[]) {

    // use a nested interface reference
    A.NestedIF nif = new B();

    if(nif.isNotNegative(10))
      System.out.println("10 is not negative");
    if(nif.isNotNegative(-12))
      System.out.println("this won't be displayed");
  }
}
```

- Notice that the name is fully qualified by the enclosing class' name. Inside the **main( )** method, an **A.NestedIF** reference called **nif** is created, and it is assigned a reference to a **B** object. Because **B** implements **A.NestedIF,** this is legal.

Interfaces

Applying Interfaces:

- The interface to the stack remains the same. That is, the methods **push( )** and **pop( )** define the interface to the stack independently of the details of the implementation.
- Because the interface to a stack is separate from its implementation, it is easy to define a stack interface, leaving it to each implementation to define the specifics.

```
// Define an integer stack interface.
interface IntStack {
  void push(int item); // store an item
  int pop(); // retrieve an item
}
```

Interfaces

Applying Interfaces: ...

```
// An implementation of IntStack that uses fixed storage.
class FixedStack implements IntStack {
  private int stck[];
  private int tos;

  // allocate and initialize stack
  FixedStack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // Push an item onto the stack
  public void push(int item) {
    if(tos==stck.length-1) // use length member
      System.out.println("Stack is full.");
    else
      stck[++tos] = item;
  }

  // Pop an item from the stack
  public int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}
```

```
class IFTest {
  public static void main(String args[]) {
    FixedStack mystack1 = new FixedStack(5);
    FixedStack mystack2 = new FixedStack(8);

    // push some numbers onto the stack
    for(int i=0; i<5; i++) mystack1.push(i);
    for(int i=0; i<8; i++) mystack2.push(i);

    // pop those numbers off the stack
    System.out.println("Stack in mystack1:");
    for(int i=0; i<5; i++)
       System.out.println(mystack1.pop());

    System.out.println("Stack in mystack2:");
    for(int i=0; i<8; i++)
       System.out.println(mystack2.pop());
  }
}
```

Interfaces

Applying Interfaces: ...

```
// Implement a "growable" stack.
class DynStack implements IntStack {
  private int stck[];
  private int tos;

  // allocate and initialize stack
  DynStack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // Push an item onto the stack
  public void push(int item) {
    // if stack is full, allocate a larger stack
    if(tos==stck.length-1) {
      int temp[] = new int[stck.length * 2]; // double size
      for(int i=0; i<stck.length; i++) temp[i] = stck[i];
      stck = temp;
      stck[++tos] = item;
    }
    else
      stck[++tos] = item;
  }

  // Pop an item from the stack
  public int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}
```

```
class IFTest2 {
  public static void main(String args[]) {
    DynStack mystack1 = new DynStack(5);
    DynStack mystack2 = new DynStack(8);

    // these loops cause each stack to grow
    for(int i=0; i<12; i++) mystack1.push(i);
    for(int i=0; i<20; i++) mystack2.push(i);

    System.out.println("Stack in mystack1:");
    for(int i=0; i<12; i++)
      System.out.println(mystack1.pop());

    System.out.println("Stack in mystack2:");
    for(int i=0; i<20; i++)
      System.out.println(mystack2.pop());
  }
}
```

Interfaces

Applying Interfaces: ...

```
/* Create an interface variable and
   access stacks through it.
*/
class IFTest3 {
  public static void main(String args[]) {
    IntStack mystack; // create an interface reference variable
    DynStack ds = new DynStack(5);
    FixedStack fs = new FixedStack(8);

    mystack = ds; // load dynamic stack
    // push some numbers onto the stack
    for(int i=0; i<12; i++) mystack.push(i);

    mystack = fs; // load fixed stack
    for(int i=0; i<8; i++) mystack.push(i);


    mystack = ds;
    System.out.println("Values in dynamic stack:");
    for(int i=0; i<12; i++)
       System.out.println(mystack.pop());

    mystack = fs;
    System.out.println("Values in fixed stack:");
    for(int i=0; i<8; i++)
       System.out.println(mystack.pop());
  }
}
```

Interfaces

Variables in Interfaces:

- You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values. When you include that interface in a class (that is, when you “implement” the interface), all of those variable names will be in scope as constants.
- It is as if that class were importing the constant fields into the class name space as **final** variables.

Interfaces

Variables in Interfaces: ...

```
import java.util.Random;

interface SharedConstants {
  int NO = 0;
  int YES = 1;
  int MAYBE = 2;
  int LATER = 3;
  int SOON = 4;
  int NEVER = 5;
}

class Question implements SharedConstants {
  Random rand = new Random();
  int ask() {
    int prob = (int) (100 * rand.nextDouble());
    if (prob < 30)
      return NO;            // 30%
    else if (prob < 60)
      return YES;           // 30%
    else if (prob < 75)
      return LATER;         // 15%
    else if (prob < 98)
      return SOON;          // 13%

    else
      return NEVER;         // 2%
  }
}
```

```
class AskMe implements SharedConstants {
  static void answer(int result) {
    switch(result) {
      case NO:
        System.out.println("No");
        break;
      case YES:
        System.out.println("Yes");
        break;
      case MAYBE:
        System.out.println("Maybe");
        break;
      case LATER:
        System.out.println("Later");
        break;
      case SOON:
        System.out.println("Soon");
        break;
      case NEVER:
        System.out.println("Never");
        break;
    }
  }
  public static void main(String args[]) {
    Question q = new Question();
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
  }
}
```

Interfaces

Interfaces Can Be Extended:

- One interface can inherit another by use of the keyword **extends**.
- The syntax is the same as for inheriting classes.
- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

Interfaces

Interfaces Can Be Extended:

```
// One interface can extend another.
interface A {
  void meth1();
  void meth2();
}

// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A {
  void meth3();
}

// This class must implement all of A and B
class MyClass implements B {
  public void meth1() {
    System.out.println("Implement meth1().");
  }

  public void meth2() {
    System.out.println("Implement meth2().");
  }

  public void meth3() {
    System.out.println("Implement meth3().");
  }


}
```

```
class IFExtend {
  public static void main(String arg[]) {
    MyClass ob = new MyClass();

    ob.meth1();
    ob.meth2();
    ob.meth3();
  }
}
```

Exception-Handling

Prepared using following Resources:

- Herbert Schildt, “Java: The Complete Reference”, Tata McGrawHill Education
- E Balagurusamy, Programming with Java - A Tata McGraw Hill Education
- https://www.geeksforgeeks.org/java/
- https://www.javatpoint.com/java-tutorial
- https://www.tutorialspoint.com/java/index.htm
- https://www.w3schools.com/java/

By: DIVAKARA .N

Exception-Handling

- Basics
- Different types of exception classes
- Use of try & catch with throw
- throws & finally
- Creation of user defined exception classes

Exception-Handling

Basics:

The three categories of errors

- ***Syntax errors*** arise because the rules of the language have not been followed. They are detected by the compiler.
- ***Runtime errors*** occur while the program is running if the environment detects an operation that is impossible to carry out.
- ***Logic errors*** occur when a program doesn't perform the way it was intended to.

Exception-Handling

Basics: ...

- An *exception* is an abnormal condition that arises in a code sequence at run time. In other words, ***an exception is a run-time error.***
- In computer languages that do not support exception handling, errors must be checked and handled manually - typically through the use of error codes, and so on.
- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error.

Exception-Handling

Basics: ...

- Exceptions can be generated by the Java run-time system, or they can be manually generated by our code.
- Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.
- Manually generated exceptions are typically used to report some error condition to the caller of a method.
- Java exception handling is managed via five keywords: **try, catch, throw, throws,** and **finally**.

Exception-Handling

Basics: ...

- Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. Our code can catch this exception (using **catch**) and handle it in some rational manner.
- System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword **throw**. Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.

Exception-Handling

Basics: ...

- The general form of an exception-handling block:

```
try {
        // block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
        // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
        // exception handler for ExceptionType2
}
// ...
finally {
        // block of code to be executed after try block ends
}
```

Exception-Handling

Exception Types:

- All exception types are subclasses of the built-in class **Throwable.** Thus, **Throwable** is at the top of the exception class hierarchy. Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches.
- One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create our own custom exception types. There is an important subclass of **Exception**, called **RuntimeException.** Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

Exception-Handling

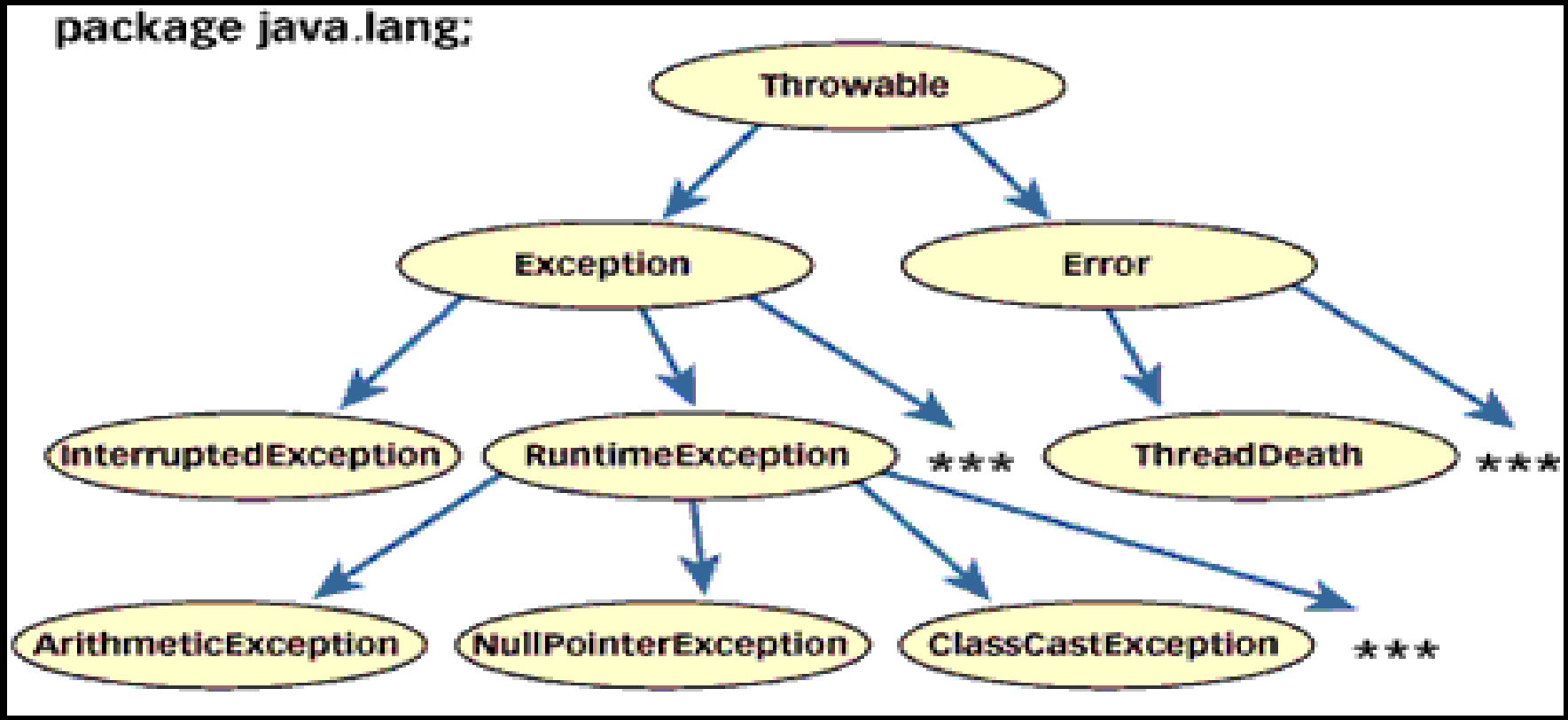
Exception Types: ...

- The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by our program. Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.
- This chapter will not be dealing with exceptions of type **Error**, because these are typically created in response to **catastrophic** failures that cannot usually be handled by our program.

Exception-Handling

Exception Types: ...

- *Every Exception type is basically an object belonging to class* ***Exception***
- ***Throwable*** *class is the root class of Exceptions.*
- ***Throwable*** *class has two direct subclasses named* ***Exception, Error***



Exception-Handling

Exception Types: ...

Checked Exceptions

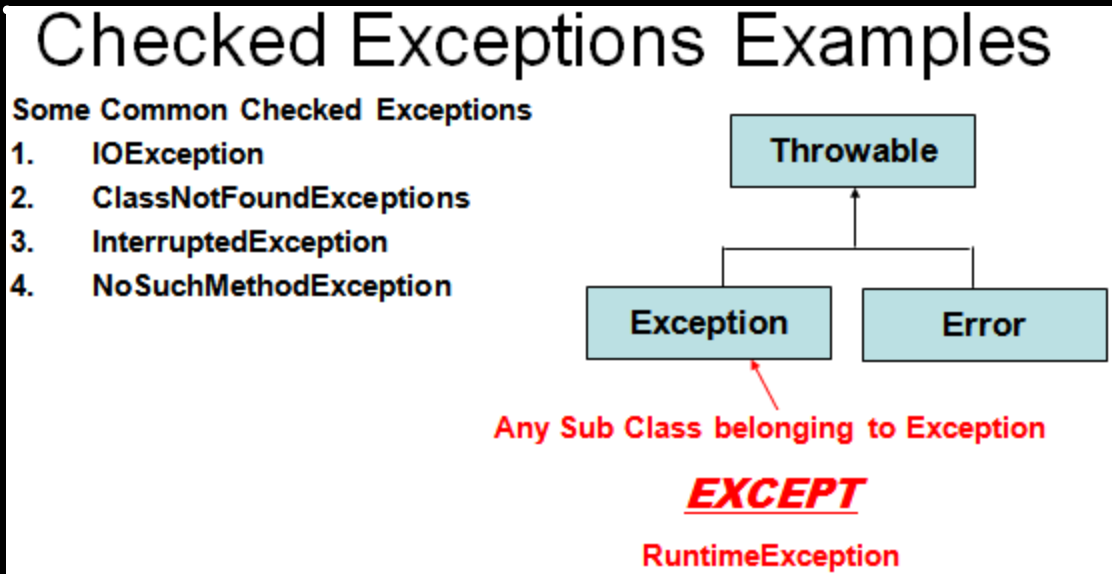
- All Exceptions that extends the Exception or any one its subclass except RunTimeException class are checked exceptions.
- Checked Exceptions are checked by the Java compiler.
- There are two approaches that a user can follow to deal with checked exceptions.
  - Inform the compiler that a method can throw an Exception.
  - Catch the checked exception in try catch block.

Exception-Handling

Exception Types: ...

Checked Exceptions…

- If Checked exception is caught then exception handling code will be executed and program's execution continues.
- If Checked exception is not caught then java interpreter will provide the default handler. But in this case execution of the program will be stopped by displaying the name of the exceptions object.



Exception-Handling

Exception Types: ...

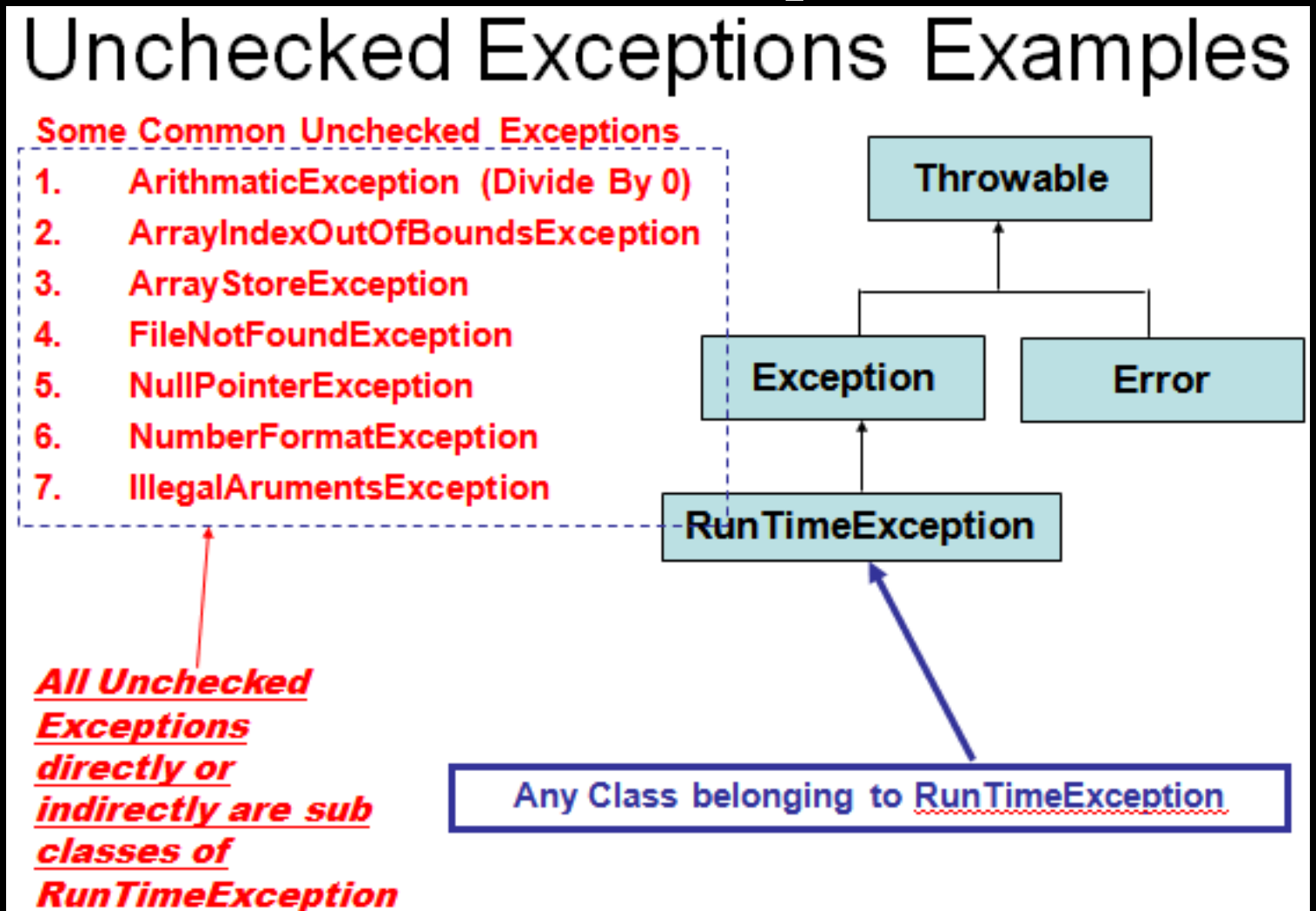
Unchecked Exceptions

- All Exceptions that extend the RuntimeException or any one of its subclass are unchecked exceptions.
- Unchecked Exceptions are unchecked by compiler.
- Whether you catch the exception or not compiler will pass the compilation process.
- If Unchecked exception is caught then exception handling code will be executed and program's execution continues.
- If Unchecked exception is not caught then java interpreter will provide the default handler. But in this case execution of the program will be stopped by displaying the name of the exceptions object.

Exception-Handling

Exception Types: ...

Unchecked Exceptions…



Exception-Handling

Exception Types: ...

Unchecked Exceptions…

UncheckedExceptions Example

```
class Exceptiondemo1
{
public static void main(String arhs[])
{
int a=10;
int b= 5;
int c =5;
int x = a/(b-c); // Dynamic Initilization
System.out.println("c="+c);
int y = a/(b+c);
System.out.println("y="+y);
}
}
```

throws ArithmeticException

No Need to mention for Unchecked Exceptions

Can Throw an Exception

```
D:\java\bin>javac Exceptiondemo1.java << Compilation Step Pass>>
D:\java\bin>java Exceptiondemo1
Exception in thread "main"
java.lang.ArithmeticException: / by zero
        at Exceptiondemo1.main(Exceptiondemo1.java:8)
```

Exception-Handling

Exception Types: ...

Unchecked Exceptions…

Example 2 (Unchecked Exceptions)

```
class Exceptiondemo2
{
public static void main(String args[])
{
double a= Double.parseDouble(args[0]);
}
}
```

Can throw either ArrayIndexOutOfBoundsException OR NumberFormatException

```
D:\java\bin>javac  Exceptiondemo2.java
D:\java\bin>java  Exceptiondemo2
Exception in thread "main"  java.lang.ArrayIndexOutOfBoundsException:  0
      at Exceptiondemo2.main(Exceptiondemo2.java:5)

D:\java\bin>java  Exceptiondemo2  pankaj
Exception in thread "main" java.lang.NumberFormatException:  For input
string: "pankaj“     at
sun.misc.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:1
2 24)   at java.lang.Double.parseDouble(Double.java:482)
      at Exceptiondemo2.main(Exceptiondemo2.java:5)
```

Exception-Handling

Exception Types: ...

Checked Exceptions vs. Unchecked Exceptions

- RuntimeException, Error and their subclasses are known as *unchecked exceptions*.
- All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with the exceptions.

Exception-Handling

Exception Types: ...

Checked Exceptions vs. Unchecked Exceptions…

- Exceptions which are checked for during compile time are called checked exceptions. **Eg:** SQLException or any userdefined exception extending the Exception class.
- Exceptions which are not checked for during compile time are called unchecked exception. **Eg:** NullPointerException or any class extending the RuntimeException class.

- All the checked exceptions must be handled in the program.
- The exceptions raised, if not handled will be handled by the Java Virtual Machine. The Virtual machine will print the stack trace of the exception indicating the stack of exception and the line where it was caused.

Exception-Handling

Uncaught Exceptions:

```
class Exc0 {
    public static void main(String args[]) {
        int d = 0;
        int a = 42 / d;
    }
}
```

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception. This causes the execution of **Exc0** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately.
- In this example, we haven't supplied any exception handlers of our own, so ***the exception is caught by the default handler provided by the Java run-time system.***

Exception-Handling

Uncaught Exceptions: ...

- Any exception that is not caught by our program will ultimately be processed by the default handler.
- The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.
- Here is the exception generated when this example is executed:

java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)

Exception-Handling

Uncaught Exceptions: ...

- The stack trace will always show the sequence of method invocations that led up to the error.

```
class Exc1 {
    static void subroutine() {
        int d = 0;
        int a = 10 / d;
    }
    public static void main(String args[]) {
        Exc1.subroutine();
    }
}
```

The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

```
java.lang.ArithmeticException: / by zero
at Exc1.subroutine(Exc1.java:4)
at Exc1.main(Exc1.java:7)
```

Exception-Handling

Using try and catch:

- Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception ourself.
- Doing so provides two benefits:
  - First, it allows you to fix the error.
  - Second, it prevents the program from automatically terminating.

Exception-Handling

Using try and catch: ...

- To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a **try** block.
- Immediately following the **try** block, include a **catch** clause that specifies the exception type that you wish to catch.

```
class Exc2 {
    public static void main(String args[]) {
        int d, a;
        try {        // monitor a block of code.
            d = 0;
            a = 42 / d;
            System.out.println("This will not be printed.");
        } catch (ArithmeticException e) {        // catch divide-by-zero error
                System.out.println("Division by zero.");
          }
        System.out.println("After catch statement.");
    }
}
```

Program output:

```
Division by zero.
After catch statement.
```

Exception-Handling

Using try and catch: ...

- Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try/catch** mechanism.
- A **try** and its **catch** statement form a unit. The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement. A **catch** statement cannot catch an exception thrown by another **try** statement.
- The goal of most well-constructed ***catch*** clauses should be to resolve the exceptional condition and then continue on as if the error had never happened.

Exception-Handling

Using try and catch: ...

```
// Handle an exception and move on.
import java.util.Random;
class HandleError {
    public static void main(String args[]) {
        int a=0, b=0, c=0;
        Random r = new Random();
        for(int i=0; i<32000; i++) {
            try {
            b = r.nextInt();
            c = r.nextInt();
            a = 12345 / (b/c);
            } catch (ArithmeticException e) {
                System.out.println("Division by zero.");
                a = 0; // set a to zero and continue
              }
            System.out.println("a: " + a);
        }
    }
}
```

For example, in this program each iteration of the **for** loop obtains two random integers. Those two integers are
divided by each other, and the result is used to divide the value 12345.
The final result is put into **a**. If either division operation causes a divide-by-zero error, it is caught, the value of a is set to zero, and the program continues.

Exception-Handling

Using try and catch: ...

Displaying a Description of an Exception

- You can display this description in a **println( )** statement by simply passing the exception as an argument. For example, the **catch** block in the preceding program can be rewritten like this:

```
catch (ArithmeticException e) {
    System.out.println("Exception: " + e);
    a = 0; // set a to zero and continue
}
```

- When this version is substituted in the program, and the program is run, each divide-by-zero error displays the following message:

```
Exception: java.lang.ArithmeticException: / by zero
```

Exception-Handling

Multiple catch Clauses:

- In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception.
- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed.
- After one **catch** statement executes, the others are bypassed, and execution continues after the **try/catch** block.

Exception-Handling

Multiple catch Clauses: ...

- **Eg:** To trap two different exceptions

```
// Demonstrate multiple catch statements.
class MultiCatch {
        public static void main(String args[]) {
                try {
                        int a = args.length;
                        System.out.println("a = " + a);
                        int b = 42 / a;
                        int c[] = { 1 };
                        c[42] = 99;
                } catch(ArithmeticException e) {
                        System.out.println("Divide by 0: " + e);
                  }
                  catch(ArrayIndexOutOfBoundsException e) {
                        System.out.println("Array index oob: " + e);
                  }
                System.out.println("After try/catch blocks.");
        }
}
```

The output generated by running it both ways:

C:\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.

C:\>java MultiCatch TestArg
a = 1
Array index oob:
java.lang.ArrayIndexOutOfBoundsException:42
After try/catch blocks.

Exception-Handling

Multiple catch Clauses: ...

- When you use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses.
- This is because a **catch** statement that uses *a superclass will catch exceptions of that type plus any of its subclasses*. Thus, a subclass would never be reached if it came after its superclass.
- Further, in Java, unreachable code is an error.

Exception-Handling

Multiple catch Clauses: ...

```
/* This program contains an error.

   A subclass must come before its superclass in
   a series of catch statements. If not,
   unreachable code will be created and a
   compile-time error will result.
*/
class SuperSubCatch {
  public static void main(String args[]) {
    try {
      int a = 0;
      int b = 42 / a;
    } catch(Exception e) {
        System.out.println("Generic Exception catch.");
      }
      /* This catch is never reached because
         ArithmeticException is a subclass of Exception. */
      catch(ArithmeticException e) { // ERROR - unreachable
        System.out.println("This is never reached.");
      }
    }
  }
```

Since **ArithmeticException** is a subclass of **Exception**, the first **catch** statement will handle all **Exception**-based errors, including **ArithmeticException.** This means that the second **catch** statement will never execute.

To fix the problem, reverse the order of the **catch** statements.

Exception-Handling

Nested try Statements:

- A **try** statement can be inside the block of another **try.** Each time a **try** statement is entered, the context of that exception is pushed on the stack. If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match.
- This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted. If no **catch** statement matches, then the Java run-time system will handle the exception.

Exception-Handling

Nested try Statements: ...

```
// An example of nested try statements.
class NestTry {
  public static void main(String args[]) {
    try {
      int a = args.length;

      /* If no command-line args are present,
         the following statement will generate
         a divide-by-zero exception. */
      int b = 42 / a;

      System.out.println("a = " + a);

      try { // nested try block
        /* If one command-line arg is used,
           then a divide-by-zero exception
           will be generated by the following code. */
        if(a==1) a = a/(a-a); // division by zero

        /* If two command-line args are used,
           then generate an out-of-bounds exception. */
        if(a==2) {
          int c[] = { 1 };
          c[42] = 99; // generate an out-of-bounds exception
        }
      } catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Array index out-of-bounds: " + e);
      }

    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    }
  }
```

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
  java.lang.ArrayIndexOutOfBoundsException:42
```

Exception-Handling

Nested try Statements: ...

```
/* Try statements can be implicitly nested via
   calls to methods. */
class MethNestTry {
  static void nesttry(int a) {
    try { // nested try block
      /* If one command-line arg is used,
         then a divide-by-zero exception
         will be generated by the following code. */
      if(a==1) a = a/(a-a); // division by zero
      /* If two command-line args are used,
         then generate an out-of-bounds exception. */
      if(a==2) {
        int c[] = { 1 };
        c[42] = 99; // generate an out-of-bounds exception
      }
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Array index out-of-bounds: " + e);
    }
  }

  public static void main(String args[]) {
    try {
      int a = args.length;

      /* If no command-line args are present,
         the following statement will generate
         a divide-by-zero exception. */
      int b = 42 / a;
      System.out.println("a = " + a);

      nesttry(a);
    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    }
  }
```

The output of this program is identical to that of the preceding example.

Exception-Handling

throw:

- So far, you have only been catching exceptions that are thrown by the Java run-time system. However, it is possible for your program to throw an exception explicitly, using the **throw** statement.
- The general form: **throw *ThrowableInstance;***
- Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable.** Primitive types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions.
- There are two ways to obtain a **Throwable** object:
  - using a parameter in a **catch** clause, or
  - creating one with the **new** operator.

Exception-Handling

throw: ...

- The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed. The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of exception.
- If it does find a match, control is transferred to that statement. If not, then the next enclosing **try** statement is inspected, and so on.
- If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace.
- Aa sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

Exception-Handling

throw: ...

```
// Demonstrate throw.
class ThrowDemo {
  static void demoproc() {
    try {
      throw new NullPointerException("demo");
    } catch(NullPointerException e) {
      System.out.println("Caught inside demoproc.");
      throw e; // rethrow the exception
    }
  }

  public static void main(String args[]) {
    try {
      demoproc();
    } catch(NullPointerException e) {
      System.out.println("Recaught: " + e);
    }
  }
}
```

The resulting output:

Caught inside demoproc.

Recaught: java.lang.NullPointerException: demo

Exception-Handling

throw: ...

- The program also illustrates how to create one of Java's standard exception objects. Pay close attention to this line:

throw new NullPointerException("demo");

- Here, new is used to construct an instance of **NullPointerException.** Many of Java's builtin run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter.
- When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to **print( )** or **println( ).** It can also be obtained by a call to **getMessage( )**, which is defined by **Throwable.**

Exception-Handling

throws:

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. By including a **throws** clause in the method's declaration. A **throws** clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses.
- All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result.

Exception-Handling

throws:

- The general form of a method declaration that includes a **throws clause:**

  type method-name(parameter-list) throws exception-list

  {

  // body of method

  }

  Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

Exception-Handling

throws: ...

- An example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a **throws** clause to declare this fact, the program will not compile.

```
// This program contains an error and will not compile.
class ThrowsDemo {
  static void throwOne() {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    throwOne();
  }
}
```

To make this example compile, you need to make two changes. First, you need to declare that **throwOne( ) throws IllegalAccessException. Second, main( ) must define a try/catch** statement that catches this exception.

```
// This is now correct.
class ThrowsDemo {
  static void throwOne() throws IllegalAccessException {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    try {
      throwOne();
    } catch (IllegalAccessException e) {
      System.out.println("Caught " + e);
    }
  }
}
```

Exception-Handling

finally:

- When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method.
- Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods.
- **Eg:** If a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The **finally** keyword is designed to address this contingency.

Exception-Handling

finally: ...

- **finally** creates a block of code that will be executed after a **try/catch** block has completed and before the code following the **try/catch** block.
- The **finally** block will execute whether or not an exception is thrown. If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception.
- Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause.

Exception-Handling

finally: ...

```
// Demonstrate finally.
class FinallyDemo {
  // Through an exception out of the method.
  static void procA() {
    try {
      System.out.println("inside procA");
      throw new RuntimeException("demo");
    } finally {
      System.out.println("procA's finally");
    }
  }

  // Return from within a try block.
  static void procB() {
    try {
      System.out.println("inside procB");
      return;
    } finally {
      System.out.println("procB's finally");
    }
  }
 // Execute a try block normally.
  static void procC() {
    try {
      System.out.println("inside procC");
    } finally {
      System.out.println("procC's finally");
    }
  }

  public static void main(String args[]) {
    try {
      procA();
    } catch (Exception e) {
      System.out.println("Exception caught");
    }
    procB();
    procC();
  }
}
```

In this example, **procA( )** prematurely breaks out of the **try** by throwing an exception. The **finally** clause is executed on the way out. procB( )'s **try** statement is exited via a **return** statement.

The **finally** clause is executed before **procB( )** returns. **In procC( ),** the **try** statement executes normally, without error. However, the **finally** block is still executed.

REMEMBER** If a **finally** block is associated with a **try**, the **finally** block will be executed upon conclusion of the **try.

Exception-Handling

Java's Built-in Exceptions:

- Inside the standard package **java.lang**, Java defines several exception classes. A few have been used by the preceding examples. The most general of these exceptions are subclasses of the standard type **RuntimeException**. As previously explained, these exceptions need not be included in any method's throws list. In the language of Java, these are called *unchecked* exceptions because the compiler does not check to see if a method handles or throws these exceptions.
- The unchecked exceptions defined in **java.lang** are listed, lists those exceptions defined by **java.lang** that must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself. These are called *checked exceptions*. Java defines several other types of exceptions that relate to its various class libraries.

Exception-Handling

Java's Built-in Exceptions: ...

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Java's Unchecked **RuntimeException** Subclasses Defined in **java.lang**

Exception-Handling

Java's Built-in Exceptions: ...

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

Java's Checked Exceptions Defined in **java.lang**

Exception-Handling

Creating our Own Exception Subclasses:

- A user defined exception should be a subclass of the **exception** class.
- The **Exception** class does not define any methods of its own. It does, of course, inherit those methods provided by **Throwable**. Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them.

Exception-Handling

Creating our Own Exception Subclasses: ...

Method	Description
Throwable fillInStackTrace( )	Returns a **Throwable** object that contains a completed stack trace. This object can be rethrown.
Throwable getCause( )	Returns the exception that underlies the current exception. If there is no underlying exception, **null** is returned.
String getLocalizedMessage( )	Returns a localized description of the exception.
String getMessage( )	Returns a description of the exception.
StackTraceElement[ ] getStackTrace( )	Returns an array that contains the stack trace, one element at a time, as an array of **StackTraceElement**. The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The **StackTraceElement** class gives your program access to information about each element in the trace, such as its method name.
Throwable initCause(Throwable *causeExc*)	Associates *causeExc* with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.
void printStackTrace( )	Displays the stack trace.
void printStackTrace(PrintStream *stream*)	Sends the stack trace to the specified stream.
void printStackTrace(PrintWriter *stream*)	Sends the stack trace to the specified stream.
void setStackTrace(StackTraceElement *elements*[ ])	Sets the stack trace to the elements passed in *elements*. This method is for specialized applications, not normal use.
String toString( )	Returns a **String** object containing a description of the exception. This method is called by **println( )** when outputting a **Throwable** object.

Exception-Handling

Using Exceptions:

- Exception handling provides a powerful mechanism for controlling complex programs that have many dynamic run-time characteristics.
- It is important to think of **try, throw**, and **catch** as clean ways to handle errors and unusual boundary conditions in our program's logic.
- Unlike some other languages in which error return codes are used to indicate failure, Java uses exceptions. Thus, when a method can fail, have it throw an exception. This is a cleaner way to handle failure modes.
- **One last point:** Java's exception-handling statements should not be considered a general mechanism for nonlocal branching.

Multithreading

Prepared using following Resources:

- Herbert Schildt, “Java: The Complete Reference”, Tata McGrawHill Education
- E Balagurusamy, Programming with Java - A Tata McGraw Hill Education
- https://www.geeksforgeeks.org/java/
- https://www.javatpoint.com/java-tutorial
- https://www.tutorialspoint.com/java/index.htm
- https://www.w3schools.com/java/

By: DIVAKARA .N

Multithreading

- Basics of Multithreading
- Main thread
- Thread life cycle
- Creation of multiple threads
- Thread priorities
- Thread synchronization
- Inter-thread communication
- Deadlocks for threads
- Suspending & Resuming threads

Multithreading

Basics:

- Unlike many other computer languages, Java provides built-in support for *multithreaded programming*.
- A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a *thread*, and each thread defines a separate path of execution.
- Thus, multithreading is a specialized form of multitasking.
- You are almost certainly acquainted with multitasking, because it is supported by virtually all modern operating systems. However, there are two distinct types of multitasking: ***process based*** and ***thread-based***.

Multithreading

Basics: …

- ***PROCESS-BASED* MULTITASKING**
- A *process* is a program that is executing. Thus, *process-based* multitasking is the feature that allows our computer to run two or more programs concurrently.
- For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor.
- In process based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

Multithreading

Basics: …

- ***THREAD-BASED* MULTITASKING**
- In a *thread-based multitasking environment, the thread is the smallest unit of dispatchable* code. This means that a single program can perform two or more tasks simultaneously.
- For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

Multithreading

Basics: …

THREAD-BASED vs.PROCESS-BASED MULTITASKING

- Multitasking threads require less overhead than multitasking processes.
- Processes are heavyweight tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is also costly.
- Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is low cost.
- While Java programs make use of process based multitasking environments, process-based multitasking is not under the control of Java.

Multithreading

Basics: …

THREAD-BASED vs.PROCESS-BASED MULTITASKING…

- Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum. This is especially important for the interactive, networked environment in which Java operates, because idle time is common.
- For example, the transmission rate of data over a network is much slower than the rate at which the computer can process it. Even local file system resources are read and written at a much slower pace than they can be processed by the CPU.
- And, of course, user input is much slower than the computer. In a single-threaded environment, our program has to wait for each of these tasks to finish before it can proceed to the next one—even though the CPU is sitting idle most of the time. Multithreading lets you gain access to this idle time and put it to good use.

Multithreading

- The Java Thread Model
- The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind. In fact, Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.
- The value of a multithreaded environment is best understood in contrast to its counterpart. Single-threaded systems use an approach called an event loop with polling.
- The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of your program.

Multithreading

- The Java Thread Model ...
- **Eg:** The idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere.
- Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause. When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

Advantage of Java Multithreading:

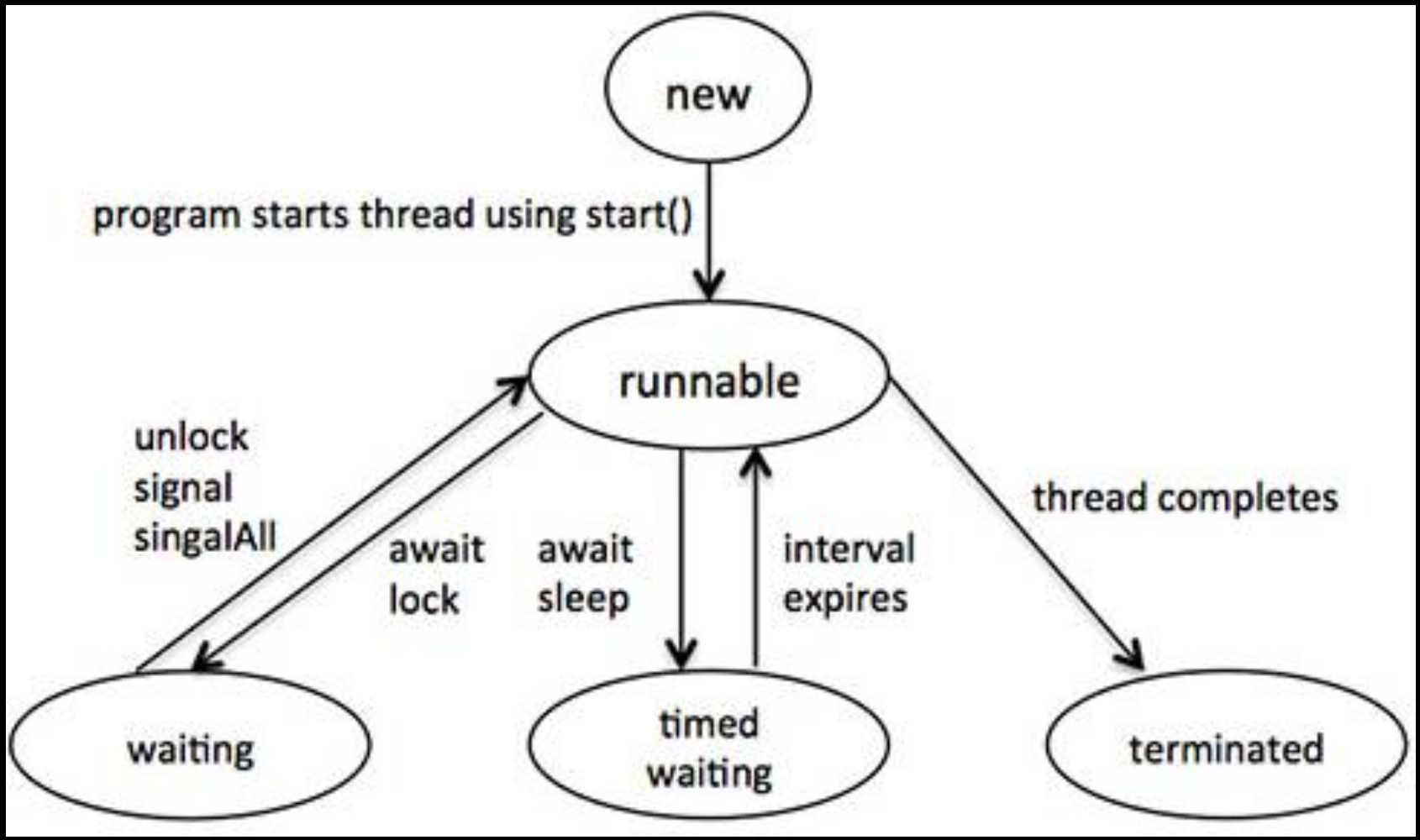
- It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.
- You **can perform many operations together so it saves time**.
- Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

Multithreading

- The Java Thread Model ...
- **Life Cycle of a Thread:**
  - Threads exist in several states. A thread can be ***running***.
  - It can be ***ready to run*** as soon as it gets CPU time.
  - A running thread can be ***suspended***, which temporarily suspends its activity.
  - A suspended thread can then be ***resumed***, allowing it to pick up where it left off.
  - A thread can be ***blocked*** when waiting for a resource.
  - At any time, a thread can be **terminated,** which halts its execution immediately.
  - Once terminated, a thread cannot be resumed.

Multithreading

- The Java Thread Model ...
- **Life Cycle of a Thread: ...**



Multithreading

- The Java Thread Model ...

Thread Priorities

- Java assigns to each thread a priority that determines how that thread should be treated with respect to the others.
- Thread priorities are integers that specify the relative priority of one thread to another. As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running.
- Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a *context switch*.

Multithreading

- The Java Thread Model ...

Thread Priorities ...

- The rules that determine when a context switch takes place are simple:
  - ***A thread can voluntarily relinquish control.*** This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
  - ***A thread can be preempted by a higher-priority thread.*** In this case, a lower-priority thread that does not yield the processor is simply preempted - no matter what it is doing - by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called *preemptive multitasking*.

Multithreading

- The Java Thread Model ...

Thread Priorities ...

- *In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated. For operating systems such as Windows, threads of equal priority are time-sliced automatically in round-robin fashion. For other types of operating systems, threads of equal priority must voluntarily yield control to their peers. If they don't, the other threads will not run.*
- Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

Multithreading

- The Java Thread Model ...

Synchronization

- Because multithreading introduces an asynchronous behavior to your programs, there must be a way for you to enforce synchronicity when you need it.
- For example, if you want two threads to communicate and share a complicated data structure, such as a linked list, you need some way to ensure that they don't conflict with each other. That is, you must prevent one thread from writing data while another thread is in the middle of reading it.
- For this purpose, Java implements an elegant twist on an age-old model of inter-process synchronization: ***the monitor***.
- **The monitor is a control mechanism first defined by C.A.R. Hoare.**

Multithreading

- The Java Thread Model ...

Synchronization ...

- A monitor is a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.
- Most multithreaded systems expose monitors as objects that your program must explicitly acquire and manipulate.
- **The synchronization is mainly used to,**

  To prevent thread interference.

  To prevent consistency problem.

Multithreading

- The Java Thread Model ...

Synchronization ...

- Java provides a cleaner solution. There is no class "Monitor"; instead, each object has its own implicit monitor that is automatically entered when **one of the object's synchronized methods** is called.
- **Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. This enables you to write very clear and concise multithreaded code, because synchronization support is built into the language.**

Multithreading

- The Java Thread Model ...

Messaging

- After you divide your program into separate threads, you need to define how they will communicate with each other. When programming with most other languages, you must depend on the operating system to establish communication between threads. This, of course, adds overhead.
- By contrast, Java provides a clean, low-cost way for two or more ***threads to talk to each other, via calls to predefined methods that all objects have.***
- Java's messaging system ***allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.***

Multithreading

- The Java Thread Model ...

The Thread Class and the Runnable Interface

- Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**.
- **Thread** encapsulates a thread of execution. Since you can't directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the **Thread** instance that spawned it.
- To create a new thread, your program will either extend **Thread** or implement the **Runnable interface.**

Multithreading

- The Java Thread Model ...

The Thread Class and the Runnable Interface ...

- The **Thread** class defines several methods that help manage threads.

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

Multithreading

- The Main thread
- When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program, because it is the one that is executed when your program begins.
- The main thread is important for two reasons:
  - ✓ It is the thread from which other "child" threads will be spawned.
  - ✓ Often, it must be the last thread to finish execution because it performs various shutdown actions.

Multithreading

- The Main thread ...
- Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object. To do so, you must obtain a reference to it by calling the method **currentThread( )**, which is a **public static** member of **Thread.**
- The General form of the method currentThread( )

static Thread currentThread( )

- This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

Multithreading

- The Main thread ...
- **Eg:**

```
// Controlling the main Thread.
class CurrentThreadDemo {
  public static void main(String args[]) {
    Thread t = Thread.currentThread();

    System.out.println("Current thread: " + t);

    // change the name of the thread
    t.setName("My Thread");
    System.out.println("After name change: " + t);

    try {
      for(int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted");
    }
  }
}
```

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

Multithreading

- The Main thread ...
- **Eg:**
- In this program, a reference to the current thread (the main thread, in this case) is obtained by calling **currentThread( ), and this reference is stored in the local variable t. Next, the program** displays information about the thread. The program then calls **setName( ) to change the** internal name of the thread. Information about the thread is then redisplayed. Next, a loop counts down from five, pausing one second between each line. The pause is accomplished by the **sleep( ) method. The argument to sleep( ) specifies the delay period in milliseconds.** Notice the **try/catch block around this loop. The sleep( ) method in Thread might throw** an **InterruptedException. This would happen if some other thread wanted to interrupt this** sleeping one. This example just prints a message if it gets interrupted. In a real program, you would need to handle this differently.

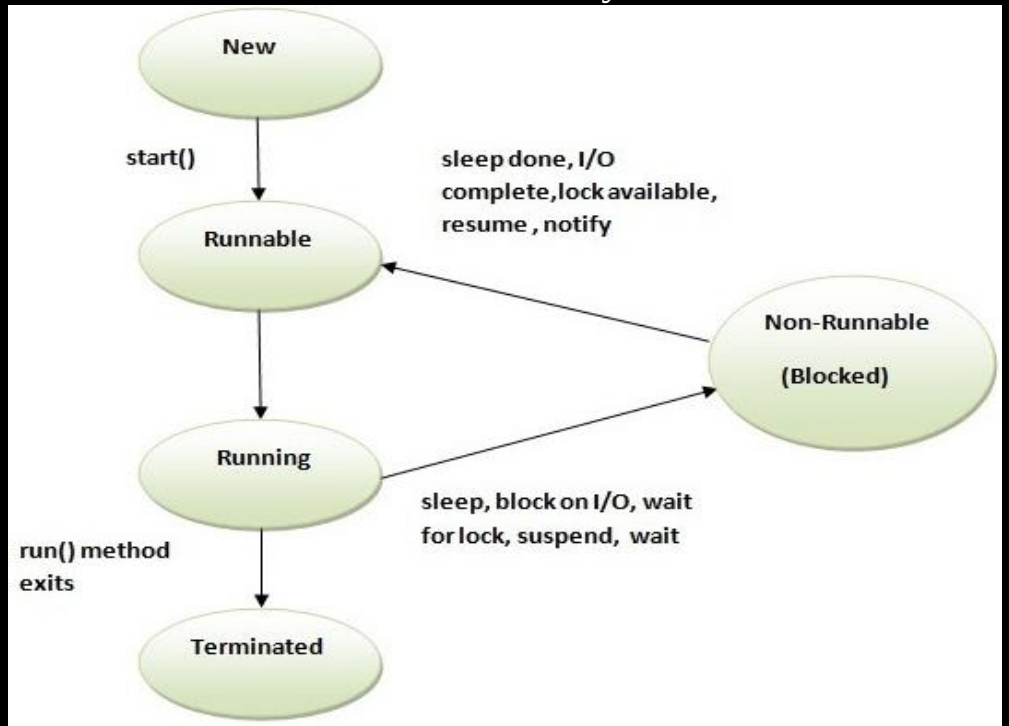
Multithreading

- The Main thread ...
- **Eg:**
- Notice the output produced when **t is used as an argument to println( ).**
- **This displays, in** order: the name of the thread, its priority, and the name of its group. By default, the name of the main thread is **main. Its priority is 5, which is the default value, and main is also the** name of the group of threads to which this thread belongs.
- A *thread group is a data structure* that controls the state of a collection of threads as a whole.
- After the name of the thread is changed, **t is again output. This time, the new name of the thread is displayed.**

Multithreading

- Thread life cycle

- As the process has several states, similarly a thread exists in several states. A thread can be in the following states:
  - ✓ ***Ready to run (New):*** First time as soon as it gets CPU time.
  - ✓ ***Running:*** Under execution.
  - ✓ ***Suspended:*** Temporarily not active or under execution.
  - ✓ ***Blocked:*** Waiting for resources.
  - ✓ ***Resumed:*** Suspended thread resumed, and start from where it left off.
  - ✓ ***Terminated:*** Halts the execution immediately and never resumes.



Multithreading

- Creating a Thread
  - Implementing Runnable
  - Extending Thread
  - Choosing an Approach

Creating a Thread

- One can create a thread by instantiating an object of type **Thread.**
- Java defines two ways in which thread can be accomplished: by implementing the **Runnable interface** and by extending the **Thread class.**

Multithreading

- Creating a Thread ...

Implementing Runnable:

- The easiest way to create a thread is to create a class that implements the **Runnable** interface.
- **Runnable** abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable.**
- To implement **Runnable,** a class need only implement a single method called **run( )**, which is declared like this:

```
public void run( )
```

Multithreading

- Creating a Thread ...

Implementing Runnable: ...

- Inside **run( )**, you will define the code that constitutes the new thread. It is important to understand that **run( )** can call other methods, use other classes, and declare variables, just like the main thread can.
- The only difference is that **run( )** establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run( )** returns.
- After you create a class that implements **Runnable**, you will instantiate an object of type **Thread** from within that class. **Thread** defines several constructors. The one that we will use is shown here: Thread(Runnable *threadOb*, *String threadName)*

Multithreading

- Creating a Thread ...

Implementing Runnable: ...

- After the new thread is created, it will not start running until you call its **start( )** method, which is declared within **Thread**. In essence, **start( )** executes a call to **run( ).**
- **The start( )** method is shown here: **void start( )**

Multithreading

- Creating a Thread ...

Implementing Runnable: ...

```
// Create a second thread.
class NewThread implements Runnable {
  Thread t;

  NewThread() {
    // Create a new, second thread
    t = new Thread(this, "Demo Thread");
    System.out.println("Child thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
}
```

```
class ThreadDemo {
  public static void main(String args[]) {
    new NewThread(); // create a new thread

    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {

      System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
  }
}
```

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Multithreading

- Creating a Thread ...

Extending Thread:

- The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class.
- The extending class must override the **run( )** method, which is the entry point for the new thread.
- It must also call **start( )** to begin execution of the new thread.

Multithreading

- Creating a Thread ...

Extending Thread: ...

```
// Create a second thread by extending Thread
class NewThread extends Thread {

  NewThread() {
    // Create a new, second thread
    super("Demo Thread");
    System.out.println("Child thread: " + this);
    start(); // Start the thread
  }

  // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
}
```

```
class ExtendThread {
  public static void main(String args[]) {
    new NewThread(); // create a new thread

    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
  }
}
```

Multithreading

- Creating a Thread ...

Choosing an Approach:

- At this point, you might be wondering why Java has two ways to create child threads, and which approach is better. The answers to these questions turn on the same point.
- The **Thread** class defines several methods that can be overridden by a derived class. Of these methods, the only one that *must be overridden is* ***run( )***. This is, of course, the same method required when you implement **Runnable.**
- Many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way. So, if you will not be overriding any of **Thread's** other methods, it is probably best simply to implement **Runnable**.

Multithreading

- Creation of multiple threads

```
// Create multiple threads.
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("New thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for thread.
  public void run() {
   try {
      for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println(name + "Interrupted");
    }
    System.out.println(name + " exiting.");
  }
}
```

```
class MultiThreadDemo {
  public static void main(String args[]) {
    new NewThread("One"); // start threads
    new NewThread("Two");
    new NewThread("Three");
    try {
      // wait for other threads to end
      Thread.sleep(10000);
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }

    System.out.println("Main thread exiting.");
  }
}
```

The output from this program is shown here:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

Multithreading

- Using isAlive( ) and join( )
- How can one thread know when another thread has ended? Fortunately, **Thread** provides a means by which you can answer this question. Two ways exist to determine whether a thread has finished.
- First, you can call **isAlive( )** on the thread. This method is defined by **Thread**, and its general form is shown here:

final boolean isAlive( )

- The **isAlive(**) method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise.

Multithreading

- Using isAlive( ) and join( ) ...
- While **isAlive( )** is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called **join(**), shown here:

  final void join( ) throws InterruptedException
- This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread *joins* it.
- Additional forms of **join( )** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

Multithreading

- Using isAlive( ) and join( ) ...

```
// Using join() to wait for threads to finish.
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("New thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println(name + " interrupted.");
    }
    System.out.println(name + " exiting.");
  }
}
```

```
class DemoJoin {
  public static void main(String args[]) {
    NewThread ob1 = new NewThread("One");
    NewThread ob2 = new NewThread("Two");
    NewThread ob3 = new NewThread("Three");
    System.out.println("Thread One is alive: "
                       + ob1.t.isAlive());
    System.out.println("Thread Two is alive: "
                       + ob2.t.isAlive());
    System.out.println("Thread Three is alive: "
                       + ob3.t.isAlive());
    // wait for threads to finish
   try {
      System.out.println("Waiting for threads to finish.");
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }

    System.out.println("Thread One is alive: "
                       + ob1.t.isAlive());
    System.out.println("Thread Two is alive: "
                       + ob2.t.isAlive());
    System.out.println("Thread Three is alive: "
                       + ob3.t.isAlive());

    System.out.println("Main thread exiting.");
  }
}
```

Multithreading

- Using isAlive( ) and join( ) ...

Sample output from this program is shown here. (Your output may vary based on processor speed and task load.)

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two exiting.
Three exiting.
One exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```

Multithreading

- Thread priorities
- To set a thread's priority, use the **setPriority( )** method, which is a member of **Thread.** This is its general form:

final void setPriority(int *level)*

- Here, *level* specifies the new priority setting for the calling thread.
- The value of level must be within the range **MIN_PRIORITY** and **MAX_PRIORITY. Currently**, these values are 1 and 10, respectively.
- To return a thread to default priority, specify **NORM_PRIORITY,** which is currently 5. These priorities are defined as **static final** ariables within **Thread.**
- To obtain the current priority setting by calling the **getPriority( )** method of **Thread,** shown here: ***final int getPriority( )***

Multithreading

- Thread priorities ...

```
// Demonstrate thread priorities.
class clicker implements Runnable {
  long click = 0;
  Thread t;
  private volatile boolean running = true;

  public clicker(int p) {
    t = new Thread(this);
    t.setPriority(p);
  }

  public void run() {
    while (running) {
      click++;
    }
  }

  public void stop() {
    running = false;
  }

  public void start() {
    t.start();
  }
}
```

```
class HiLoPri {
  public static void main(String args[]) {
    Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
    clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
    clicker lo = new clicker(Thread.NORM_PRIORITY - 2);

    lo.start();
    hi.start();
    try {
      Thread.sleep(10000);
    } catch (InterruptedException e) {
        System.out.println("Main thread interrupted.");
    }
    lo.stop();
    hi.stop();

    // Wait for child threads to terminate.
    try {
      hi.t.join();
      lo.t.join();
    } catch (InterruptedException e) {
      System.out.println("InterruptedException caught");
    }

    System.out.println("Low-priority thread: " + lo.click);
    System.out.println("High-priority thread: " + hi.click);
  }
}
```

Multithreading

- Thread synchronization
- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization*.
- If you have worked with synchronization when using other languages, such as C or C++, you know that it can be a bit tricky to use. This is because these languages do not, themselves, support synchronization. Instead, to synchronize threads, your programs need to utilize operating system primitives.
- Fortunately, because Java implements synchronization through language elements, most of the complexity associated with synchronization has been eliminated. You can synchronize your code in either of two ways. Both involve the use of the **synchronized** keyword.

Multithreading

- Thread synchronization ...

Using Synchronized Methods:

- Synchronization is easy in Java, because all objects have their own implicit monitor associated with them.
- To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword.
- While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait.
- To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

Multithreading

- Thread synchronization ...

Using Synchronized Methods: ... *Not Synchronized Program*

```
// This program is not synchronized.
class Callme {
  void call(String msg) {
    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
    System.out.println("]");
  }
}

class Caller implements Runnable {
  String msg;
  Callme target;
  Thread t;
  public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start();
  }

  public void run() {
    target.call(msg);
  }
}
```

```
class Synch {
  public static void main(String args[]) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Hello");
    Caller ob2 = new Caller(target, "Synchronized");
    Caller ob3 = new Caller(target, "World");

    // wait for threads to end
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
  }
}
```

Here is the output produced by this program:

```
Hello[Synchronized[World]
]
]
```

Multithreading

- Thread synchronization ...

Using Synchronized Methods: ...

- To fix the preceding program, you must *serialize access to* ***call( )***. That is, you must restrict its access to only one thread at a time.
- To do this, you simply need to precede **call( )**'s definition
- with the keyword **synchronized**, as shown here:

```
class Callme {
    synchronized void call(String msg) {
    ...
```

- This prevents other threads from entering **call( )** while another thread is using it. After **synchronized** has been added to **call( )**, the output of the program is as follows:

```
[Hello]
[Synchronized]
[World]
```

Multithreading

- Thread synchronization ...

Using Synchronized Methods: ...

- Any time that you have a method, or group of methods, that manipulates the internal state of an object in a multithreaded situation, you should use the **synchronized** keyword to guard the state from race conditions.
- Remember, once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance.
- However, nonsynchronized methods on that instance will continue to be callable.

Multithreading

- Thread synchronization ...

The synchronized Statement:

- While creating **synchronized** methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases.
- To understand why, consider the following. Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods.
- Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add **synchronized** to the appropriate methods within the class.
- How can access to an object of this class be synchronized? Fortunately, the solution to this problem is quite easy: You simply **put calls to the methods defined by this class inside a synchronized block.**

Multithreading

- Thread synchronization ...

The synchronized Statement: ...

- This is the general form of the **synchronized statement:**

```
synchronized(object) {
        // statements to be synchronized
}
```

Here, *object* is a reference to the object being synchronized.

- A synchronized block ensures that a call to a method that is a member of *object* occurs only after the current thread has successfully entered *object's monitor.*

Multithreading

- Thread synchronization …

The synchronized Statement: …

```
// This program uses a synchronized block.
class Callme {
  void call(String msg) {
    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      System.out.println("Interrupted");
    }
    System.out.println("]");
  }
}

class Caller implements Runnable {
  String msg;
  Callme target;
  Thread t;

  public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start();
  }

  // synchronize calls to call()
  public void run() {
    synchronized(target) { // synchronized block
      target.call(msg);
    }
  }
}
```

```
class Synch1 {
  public static void main(String args[]) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Hello");
    Caller ob2 = new Caller(target, "Synchronized");
    Caller ob3 = new Caller(target, "World");

    // wait for threads to end
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
  }
}
```

Multithreading

- Inter-thread communication
- **Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.
- Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.
- It is implemented by following methods of **Object** class:
  - wait()
  - notify()
  - notifyAll()

Multithreading

- Inter-thread communication ...
- These methods have been implemented as **final** methods in Object, so they are available in all the classes.
- All three methods can be called only from within a **synchronized** context.

Methods with Description
public void wait() Causes the current thread to wait until another thread invokes the notify().
public void notify() Wakes up a single thread that is waiting on this object's monitor.
public void notifyAll() Wakes up all the threads that called wait( ) on the same object.

Multithreading

- Inter-thread communication ...

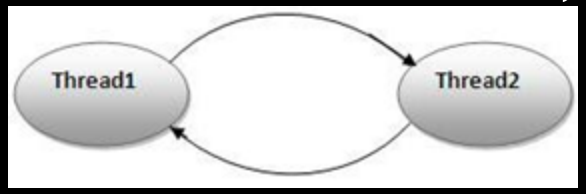
```
class Chat {
   boolean flag = false;

   public synchronized void Question(String msg) {
      if (flag) {
         try {
            wait();
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
      System.out.println(msg);
      flag = true;
      notify();
   }

   public synchronized void Answer(String msg) {
      if (!flag) {
         try {
            wait();
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }

      System.out.println(msg);
      flag = false;
      notify();
   }
}
```

```
class T1 implements Runnable {
   Chat m;
   String[] s1 = { "Hi", "How are you ?", "I am also doing fine!" };
   public T1(Chat m1) {
      this.m = m1;
      new Thread(this, "Question").start();
   }
   public void run() {
      for (int i = 0; i < s1.length; i++) {
         m.Question(s1[i]);
      }
   }
}
class T2 implements Runnable {
   Chat m;
   String[] s2 = { "Hi", "I am good, what about you?", "Great!" };

   public T2(Chat m2) {
      this.m = m2;
      new Thread(this, "Answer").start();
   }
   public void run() {
      for (int i = 0; i < s2.length; i++) {
         m.Answer(s2[i]);
      }
   }
}
public class TestThread {
   public static void main(String[] args) {
      Chat m = new Chat();
      new T1(m);
      new T2(m);
   }
}
```

```
Hi
Hi
How are you ?
I am good, what about you?
I am also doing fine!
Great!
```

Multithreading

- Deadlocks for threads
- Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



- ***Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Deadlock occurs when multiple threads need the same locks but obtain them in different order. A Java multithreaded program may suffer from the deadlock condition because the synchronized keyword causes the executing thread to block while waiting for the lock, or monitor, associated with the specified object.***

Multithreading

- Deadlocks for threads ...*Deadlock Situation*

```
public class TestThread {
   public static Object Lock1 = new Object();
   public static Object Lock2 = new Object();

   public static void main(String args[]) {

      ThreadDemo1 T1 = new ThreadDemo1();
      ThreadDemo2 T2 = new ThreadDemo2();
      T1.start();
      T2.start();
   }

   private static class ThreadDemo1 extends Thread {
      public void run() {
         synchronized (Lock1) {
            System.out.println("Thread 1: Holding lock 1...");
            try { Thread.sleep(10); }
            catch (InterruptedException e) {}
            System.out.println("Thread 1: Waiting for lock 2...");
            synchronized (Lock2) {
               System.out.println("Thread 1: Holding lock 1 & 2...");
            }
         }
      }
   }
   private static class ThreadDemo2 extends Thread {
      public void run() {
         synchronized (Lock2) {
            System.out.println("Thread 2: Holding lock 2...");
            try { Thread.sleep(10); }
            catch (InterruptedException e) {}
            System.out.println("Thread 2: Waiting for lock 1...");
            synchronized (Lock1) {
               System.out.println("Thread 2: Holding lock 1 & 2...");
            }
         }
      }
   }
}
```

When you compile and execute above program, you find a deadlock situation and below is the output produced by the program:

Thread 1: Holding lock 1...
Thread 2: Holding lock 2...
Thread 1: Waiting for lock 2...
Thread 2: Waiting for lock 1...

The program will hang forever because neither of the threads in position to proceed and waiting for each other to release the lock, so you can come out of the program by pressing CTRL-C.

Multithreading

- Deadlocks for threads ... *Solution*

```
public class TestThread {
   public static Object Lock1 = new Object();
   public static Object Lock2 = new Object();

   public static void main(String args[]) {

      ThreadDemo1 T1 = new ThreadDemo1();
      ThreadDemo2 T2 = new ThreadDemo2();
      T1.start();
      T2.start();
   }

   private static class ThreadDemo1 extends Thread {
      public void run() {
         synchronized (Lock1) {
            System.out.println("Thread 1: Holding lock 1...");
            try { Thread.sleep(10); }
            catch (InterruptedException e) {}
            System.out.println("Thread 1: Waiting for lock 2...");
            synchronized (Lock2) {
               System.out.println("Thread 1: Holding lock 1 & 2...");
            }
         }
      }
   }
   private static class ThreadDemo2 extends Thread {
      public void run() {
         synchronized (Lock1) {
            System.out.println("Thread 2: Holding lock 1...");
            try { Thread.sleep(10); }
            catch (InterruptedException e) {}
            System.out.println("Thread 2: Waiting for lock 2...");
            synchronized (Lock2) {
               System.out.println("Thread 2: Holding lock 1 & 2...");
            }
         }
      }
   }
}
```

So just changing the order of the locks prevent the program in going deadlock situation and completes with the following result:

Thread 1: Holding lock 1...
Thread 1: Waiting for lock 2...
Thread 1: Holding lock 1 & 2...
Thread 2: Holding lock 1...
Thread 2: Waiting for lock 2...
Thread 2: Holding lock 1 & 2...

Multithreading

- Suspending & Resuming threads / Suspending, Resuming, and Stopping Threads
- Core Java provides a complete control over multithreaded program. You can develop a multithreaded program which can be suspended, resumed or stopped completely based on your requirements.
- There are various static methods which you can use on thread objects to control their behavior.

Methods with Description
public void suspend() This method puts a thread in suspended state and can be resumed using resume() method.
public void stop() This method stops a thread completely.
public void resume() This method resumes a thread which was suspended using suspend() method.
public void wait() Causes the current thread to wait until another thread invokes the notify().
public void notify() Wakes up a single thread that is waiting on this object's monitor.

Multithreading

- Suspending & Resuming threads / Suspending, Resuming, and Stopping Threads ...

```
class MyThread implements Runnable {
 Thread thrd;
 boolean suspended;
 boolean stopped;
 MyThread(String name) {
  thrd = new Thread(this, name);
  suspended = false;
  stopped = false;
  thrd.start();
 }
 public void run() {
  try {
   for (int i = 1; i < 10; i++) {
    System.out.print(".");
    Thread.sleep(50);
    synchronized (this) {
     while (suspended)
      wait();
     if (stopped)
      break;
    }
   }
  } catch (InterruptedException exc) {
   System.out.println(thrd.getName() + " interrupted.");
  }
  System.out.println("\n" + thrd.getName() + " exiting.");
 }
 synchronized void stop() {
  stopped = true;
  suspended = false;
  notify();
 }
 synchronized void suspend() {
  suspended = true;
 }
 synchronized void resume() {
  suspended = false;
  notify();
 }
}
```

```
public class Main {
 public static void main(String args[]) throws Exception {
  MyThread mt = new MyThread("MyThread");
  Thread.sleep(100);
  mt.suspend();
  Thread.sleep(100);

  mt.resume();
  Thread.sleep(100);

  mt.suspend();
  Thread.sleep(100);

  mt.resume();
  Thread.sleep(100);

  mt.stop();
 }
}
```

Multithreading

- Using Multithreading
- The key to utilizing Java's multithreading features effectively is to think concurrently rather than serially. For example, when you have two subsystems within a program that can execute concurrently, make them individual threads.
- With the careful use of multithreading, you can create very efficient programs. A word of caution is in order, however: If you create too many threads, you can actually degrade the performance of your program rather than enhance it.
- Remember, some overhead is associated with context switching. If you create too many threads, more CPU time will be spent changing contexts than executing your program!

Applet Programming

Prepared using following Resources:

- Herbert Schildt, “Java: The Complete Reference”, Tata McGrawHill Education
- E Balagurusamy, Programming with Java - A Tata McGraw Hill Education
- https://www.geeksforgeeks.org/java/
- https://www.javatpoint.com/java-tutorial
- https://www.tutorialspoint.com/java/index.htm
- https://www.w3schools.com/java/

By: DIVAKARA .N

Applet Programming

- Introduction, How Applets Differ from Applications, Preparing to Write Applets, Building Applet Code, Applet Life Cycle, Creating an Executable Applet, Designing a Web Page, Applet Tag, Adding Applet to HTML File, Running the Applet, More About Applet Tag, Passing Parameters to Applets, Aligning the Display, More About HTML Tags, Displaying Numerical Values, Getting Input from the User, Event Handling.

Applet Programming

Introduction

Applet Fundamentals

- Applets are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a web document.
- After an applet arrives on the client, it has limited access to resources so that it can produce a graphical user interface and run complex computations without introducing the risk of viruses or breaching data integrity.

Applet Programming

Introduction ...

Applet Fundamentals

- ***The simple applet:***

```
import java.awt.*;
import java.applet.*;
public class SimpleApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("A Simple Applet", 20, 20);
    }
}
```

Applet Programming

Introduction ...

Applet Fundamentals

- ***The simple applet: ...***
- This applet begins with two **import** statements. The first imports the Abstract Window Toolkit (AWT) classes. Applets interact with the user (either directly or indirectly) through the AWT, not through the console-based I/O classes. The AWT contains support for a window-based, GUI. Fortunately, this simple applet makes very limited use of the AWT. (Applets can also use Swing to provide the GUI.)
- The second import statement **imports** the **applet** package, which contains the class **Applet**. Every applet that you create must be a subclass of **Applet**.
- The next line in the program declares the class **SimpleApplet.** This class must be declared as **public**, because it will be accessed by code that is outside the program.

Applet Programming

Introduction ...

Applet Fundamentals

- ***The simple applet: ...***
- Inside **SimpleApplet**, **paint( )** is declared. This method is defined by the AWT and must be overridden by the **applet**. **paint( )** is called each time that the applet must redisplay its output. whenever the applet must redraw its output, **paint( )** is called.
- The **paint( )** method has one parameter of type **Graphics**. This parameter contains the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.
- Inside **paint( )** is a call to **drawString( )**, which is a member of the Graphics class. This method outputs a string beginning at the specified X,Y location.

Applet Programming

Introduction ...

Applet Fundamentals

- ***The simple applet: ...***
- The general form: **void drawString(String message, int x, int y)**
- Here, message is the string to be output beginning at x,y. In a Java window, the upper-left corner is location 0,0. The call to **drawString( )** in the applet causes the message "A Simple Applet" to be displayed beginning at location 20,20.
- Notice that the applet does not have a **main( )** method. Unlike Java programs, applets do not begin execution at **main( )**. In fact, most applets don't even have a **main( )** method. Instead, an applet begins execution when the name of its class is passed to an applet viewer or to a network browser.

Applet Programming

Introduction ...

Applet Fundamentals

- ***The simple applet: ...***
- After you enter the source code for **SimpleApplet**, compile in the same way that you have been compiling programs. However, running SimpleApplet involves a different process. In fact, there are two ways in which you can run an applet:
  - Exceuting the applet within a Java-compatible web browser.
  - Using an applet viewer, such as the standard tool, ***appletviewer***. An applet viewer executes your applet in a window. This is generally the fastest and easiest way to test your applet.

Applet Programming

Introduction ...

Applet Fundamentals

- ***The simple applet: ...***
- To execute an applet in a web browser, you need to write a short HTML text file that contains a tag that loads the applet. Currently, Sun recommends using the APPLET tag for this purpose. Here is the HTML file that executes **SimpleApplet**:

  <applet code="SimpleApplet" width=200 height=60>

  </applet>

- The **width** and **height** statements specify the dimensions of the display area used by the applet. (The APPLET tag contains several other options.) After you create this file, you can execute your browser and then load this file, which causes **SimpleApplet** to be executed.

Applet Programming

Introduction ...

Applet Fundamentals

- ***The simple applet: ...***
- To execute **SimpleApplet** with an applet viewer, you may also execute the HTML file shown earlier. For example, if the preceding HTML file is called **RunApp.html**, then the following command line will run **SimpleApplet**:

```
C:\>appletviewer RunApp.html
```

Applet Programming

Introduction ...

Applet Fundamentals

- *The simple applet: ...*

However, a more convenient method exists that you can use to speed up testing. Simply include a comment at the head of your Java source code file that contains the APPLET tag.

By doing so, your code is documented with a prototype of the necessary HTML statements, and you can test your compiled applet merely by starting the applet viewer with your Java source code file. If you use this method, the **SimpleApplet** source file looks like this:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleApplet" width=200 height=60>
</applet>
*/

public class SimpleApplet extends Applet {
  public void paint(Graphics g) {
    g.drawString("A Simple Applet", 20, 20);
  }
}
```

Applet Programming

Introduction ...

Applet Fundamentals

- ***The simple applet: ...***
- With this approach, we can quickly iterate through applet development by using these three steps:

  1) Edit a Java source file.
  2) Compile your program.
  3) Execute the applet viewer, specifying the name of your applet's source file. The applet viewer will encounter the APPLET tag within the comment and execute your applet.

Applet Programming

Introduction ...

Applet Fundamentals

- ***The simple applet: ...***
- The window produced by **SimpleApplet,** as displayed by the applet viewer, is shown in the following illustration:

- **The key points that you should remember now:**
  - Applets do not need a **main( )** method.
  - Applets must be run under an applet viewer or a Java-compatible browser.
  - User I/O is not accomplished with Java's stream I/O classes. Instead, applets use the interface provided by the AWT or Swing.

Applet Programming

Introduction

- Java programs are divided into two main categories, applets and applications.
- An application is an ordinary Java program.
- An applet is a kind of Java program that can be run across the Internet.
- Applets are small Java programs that are embedded in Web pages.
- They can be transported over the Internet from one computer (web server) to another (client computers).
- They transform web into rich media and support the delivery of applications via the Internet.

Applet Programming

Introduction ...

- All applets are subclasses (either directly or indirectly) of **Applet**. Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer.
- The illustrations shown in this chapter were created with the standard applet viewer, called **appletviewer**, provided by the **JDK**.
- But you can use any applet viewer or browser you like. Execution of an applet does not begin at **main( ).** Actually, few applets even have **main( )** methods.
- Instead, execution of an applet is started and controlled with an entirely different mechanism.

Applet Programming

Advantages

- There are many advantages:
  - It works at client side so less response time.
  - Secured
  - It can be executed by browsers running under many platforms, including Linux, Windows, Mac Os etc.

Drawback

- Plug-in is required at client browser to execute applet.

Applet Programming

How Applets Differ from Applications

- Although both the Applets and stand-alone applications are Java programs, there are certain restrictions are imposed on Applets due to security concerns:
  - Applets don't use the main() method, but when they are load, automatically call certain methods (init, start, paint, stop, destroy).
  - They are embedded inside a web page and executed in browsers.
  - They cannot read from or write to the files on local computer.
  - They cannot communicate with other servers on the network.
  - They cannot run any programs from the local computer.
  - They are restricted from using libraries from other languages.
- The above restrictions ensures that an Applet cannot do any damage to the local system.

Applet Programming

Building Applet Code: *An Example*

```
//HelloWorldApplet.java
import java.applet.Applet;
import java.awt.*;

public class HelloWorldApplet extends Applet {
        public void paint(Graphics g) {
                g.drawString ("Hello World of Java!",25, 25);
        }
}
```

Applet Programming

Embedding Applet in Web Page

```
<HTML>
        <HEAD>
            <TITLE>
                Hello World Applet
            </TITLE>
        </HEAD>
    <body>
        <h1>Hi, This is My First Java Applet on the Web!</h1>
        <APPLET CODE="HelloWorldApplet.class" width=500 height=400>
        </APPLET>
    </body>
</HTML>
```

Applet Programming

Accessing Web page (runs Applet)

Applet Programming

An Applet Skeleton / Applet Life Cycle

- All but the most trivial applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution.
- Four of these methods, **init( ), start( ), stop( ), and destroy( ),** apply to all applets and are defined by **Applet**.
- Default implementations for all of these methods are provided. Applets do not need to override those methods they do not use. However, only very simple applets will not need to define all of them.

Applet Programming

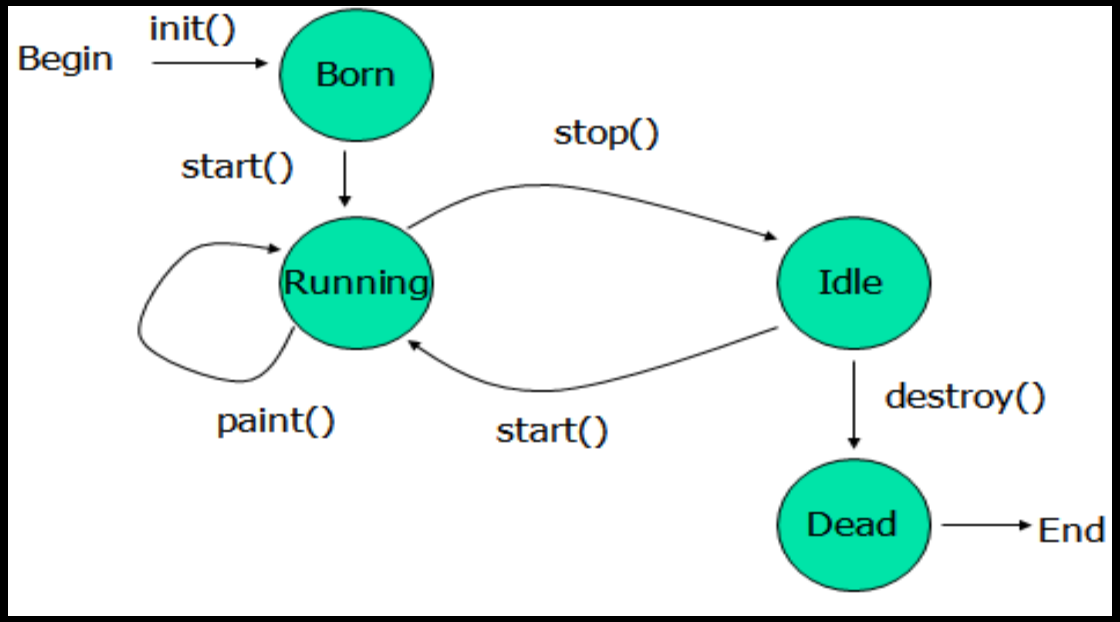
An Applet Skeleton / Applet Life Cycle ...

- Every applet inherits a set of default behaviours from the **Applet** class. As a result, when an applet is loaded, it undergoes a series of changes in its state. The applet states include:
  - **Initialisation – invokes init()**
  - **Running – invokes start()**
  - **Display – invokes paint()**
  - **Idle – invokes stop()**
  - **Dead/Destroyed State – invokes destroy()**

```
// An Applet skeleton.
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/

public class AppletSkel extends Applet {
  // Called first.
  public void init() {
    // initialization
  }

  /* Called second, after init().  Also called whenever
     the applet is restarted. */
  public void start() {
    // start or resume execution
  }

  // Called when the applet is stopped.
  public void stop() {
    // suspends execution
  }

  /* Called when applet is terminated.  This is the last
     method executed. */
  public void destroy() {
    // perform shutdown activities
  }

  // Called when an applet's window must be restored.
  public void paint(Graphics g) {
    // redisplay contents of window
  }
}
```

Applet Programming

An Applet Skeleton / Applet Life Cycle ...



- **public void init():** is used to initialized the Applet. It is invoked only once.
- **public void start():** is invoked after the init() method or browser is maximized. It is used to start the Applet.
- **public void stop():** is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.
- **public void destroy():** is used to destroy the Applet. It is invoked only once.

Applet Programming

Passing Parameters to Applet

```
<HTML>

    <HEAD>
        <TITLE>
                Hello World Applet
        </TITLE>
    </HEAD>

    <body>
    <h1>Hi, This is My First Communicating Applet on the Web!</h1>
        <APPLET CODE="HelloAppletMsg.class" width=500 height=400>
            <PARAM NAME="Greetings" VALUE="Hello Friend, How are you?">
        </APPLET>

    </body>

</HTML>
```

Applet Programming

Applet Program Accepting Parameters

```
//HelloAppletMsg.java
import java.applet.Applet;
import java.awt.*;

public class HelloAppletMsg extends Applet {

        String msg;

        public void init()
        {
                msg = getParameter("Greetings");
                if( msg == null)
                        msg = "Hello";
        }
        public void paint(Graphics g) {
                g.drawString (msg,10, 100);
        }
}
```

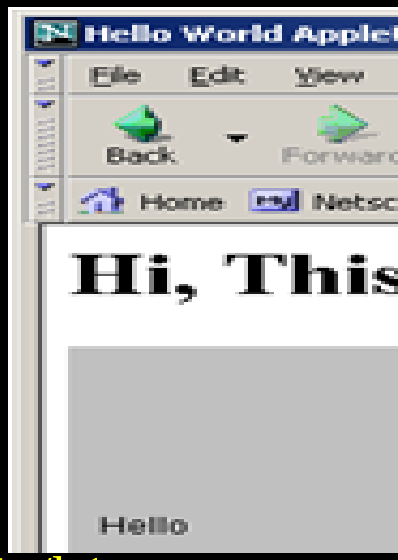
This is name of parameter specified in PARAM tag; This method returns the value of paramter.

Applet Programming

What happen if we don't pass parameter?

See HelloAppletMsg1.html

getParameter() returns *null*. Some default value may be used.




```
<HTML>

    <HEAD>
        <TITLE>
                Hello World Applet
        </TITLE>
    </HEAD>

    <body>
        <h1>Hi, This is My First Communicating Applet on the Web!</h1>
        <APPLET CODE="HelloAppletMsg.class" width=500 height=400>
        </APPLET>

    </body>

</HTML>
```

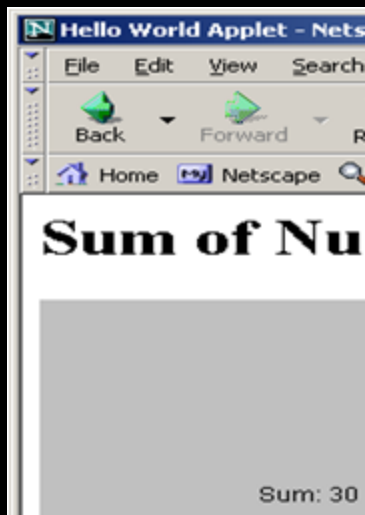
Applet Programming

Displaying Numeric Values

```
//SumNums.java
import java.applet.Applet;
import java.awt.*;

public class SumNums extends Applet {
   public void paint(Graphics g) {
          int num1 = 10;
          int num2 = 20;
          int sum = num1 + num2;
          String str = "Sum: "+String.valueOf(sum);
          g.drawString (str,100, 125);
   }
}
```

SumNums.html

```
<HTML>
<HEAD>
    <TITLE>
        Hello World Applet
    </TITLE>
</HEAD>

<body>
<h1>Sum of Numbers</h1>
<APPLET CODE="SumNums.class" width=500 height=400>
</APPLET>
</body>

</HTML>
```



Applet Programming

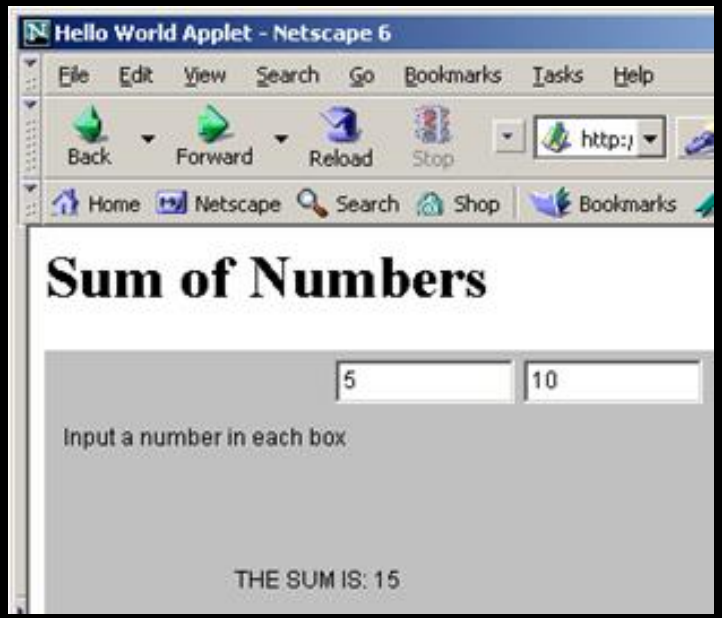
Interactive Applet

- Applets work in a graphical environment. Therefore, applets treats inputs as text strings.
- We need to create an area on the screen in which use can type and edit input items.
- We can do this using TextField class of the applet package.
- When data is entered, an event is generated. This can be used to refresh the applet output based on input values.

Applet Programming

Interactive Applet ...

```
//SumNumsInteractive..java
import java.applet.Applet;
import java.awt.*;

public class SumNumsInteractive  extends Applet {
                TextField text1, text2;
                public void init()
                {
                                text1 = new TextField(10);
                                text2 = new TextField(10);
                                text1.setText("0");
                                text2.setText("0");
                                add(text1);
                                add(text2);
                }
                public void paint(Graphics g) {
                                int num1 = 0;
                                int num2 = 0;
                                int sum;
                                String s1, s2, s3;

                                g.drawString("Input a number in each box ", 10, 50);
                                try {
                                                s1 = text1.getText();
                                                num1 = Integer.parseInt(s1);
                                                s2 = text2.getText();
                                                num2 = Integer.parseInt(s2);
                                }
                                catch(Exception e1)
                                {}
                                sum = num1 + num2;
                                String str = "THE SUM IS: "+String.valueOf(sum);
                                g.drawString (str,100, 125);
                }
                public boolean action(Event ev, Object obj)
                {
                                repaint();
                                return true;
                }
}
```



Applet Programming

Applet and Security

- An applet can be a program, written by someone else, that runs on your computer.
- Whenever someone else's program runs on your computer, there are security questions you should ask:
  - Will it read information from your files?
  - Will it corrupt your operating system?
- Applets are designed so that they cannot do any of these things (at least easily).

Applet Programming

Summary

- Applets are designed to operate in Internet and Web environment.
- They enable the delivery of applications via the Web.
- In this presentation we learned:
  - ✓ *How do applets differ from applications?*
  - ✓ *Life cycles of applets*
  - ✓ *How to design applets?*
  - ✓ *How to execute applets?*
  - ✓ *How to provide interactive inputs?*