LECTURE NOTES ON

Object Oriented Programming using JAVA

II B.TECH II SEMESTER

(JNTUA-R15)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NARAYANA ENGINEERING COLLEGE :: GUDUR

(AUTONOMOUS)

Approved by AICTE and Permanently Affiliated to JNTUA, Ananthapuramu, An ISO 9001:2015 Certified Institution, Recognized by UGC U/S 2(f) & 12(B)

Dhoorjati Nagar, Gudur, SPSR Nellore Dist.- 524101

Object Oriented Programming using JAVA

UNIT I

The History and Evolution of Java

What is Java

1. Java technology is both a *programming language* and *a platform*.
2. Java is a high level, robust, secured and object-oriented programming language.

Java programming language:

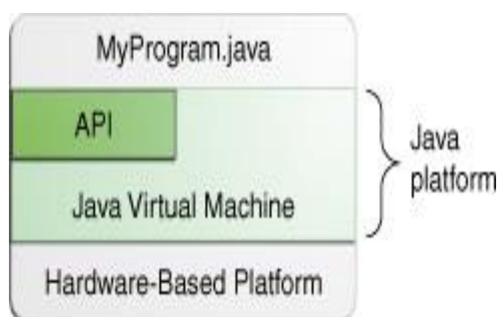
The Java programming language is a **high-level** and **object-oriented language** that can be characterized by all of the following ***buzzwords / features***:

- **i)** Simple
- **ii)** Object oriented
- **iii)** Distributed
- **iv)** Multithreaded
- **v)** Dynamic
- **vi)** Architecture Neutral
- **vii)** Portable
- **viii)** Interpreted
- **ix)** High Performance
- **x)** Robust
- **xi)** Secure

Java Platform:

Any hardware or software environment in which a program runs is known as a platform. Since Java has its own runtime environment (**JRE-Java Runtime Environment**) and API, it is called platform.

The Java platform has two components – **API** and **Java Virtual Machine**



The *Java Virtual Machine :*

It is an abstract machine and it is the base for the Java platform and is ported onto various hardware-based platforms.

The *Java Application Programming Interface* (API) :

The API is a large collection of ready-made software components that provide many useful capabilities. It is grouped into libraries of related classes and interfaces; these libraries are known as *packages*.

The Creation of Java

Java was originally developed to develop a language for digital devices such as set-top boxes and consumer electronic devices such as microwave ovens and remote controls.

Java was developed by **James Gosling**, Patrick Naughton, Mike Sheridan at *Sun Microsystems Inc.* in 1991. It took 18 months to develop the first working version.

The initial name was **Oak** but it was renamed to **Java** in 1995.

Primary Motivation(s) for Java creation:

1. The primary motivation was the need for a ***platform-independent (architecture-neutral/portable)*** language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls.

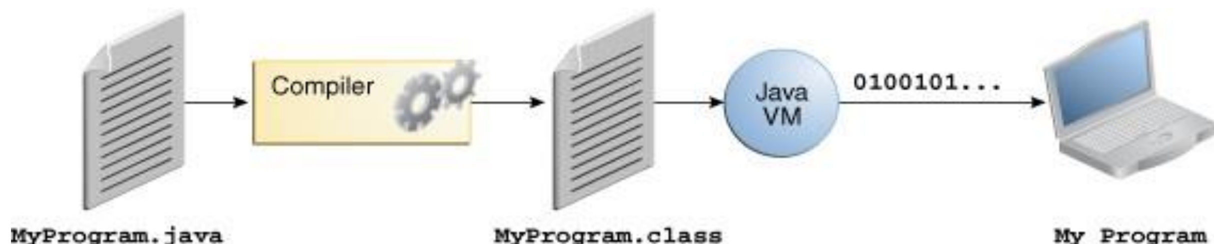
2. The second important factor is the ***World Wide Web / Internet***.

   a) Java is used to develop web based programs / Internet programming.
   b) The Internet consists of a diverse, distributed universe populated with various types of computers, operating systems, and CPUs.

Platform Independence or Architecture Neutral or Portable:

(Write Once, Run Anywhere)

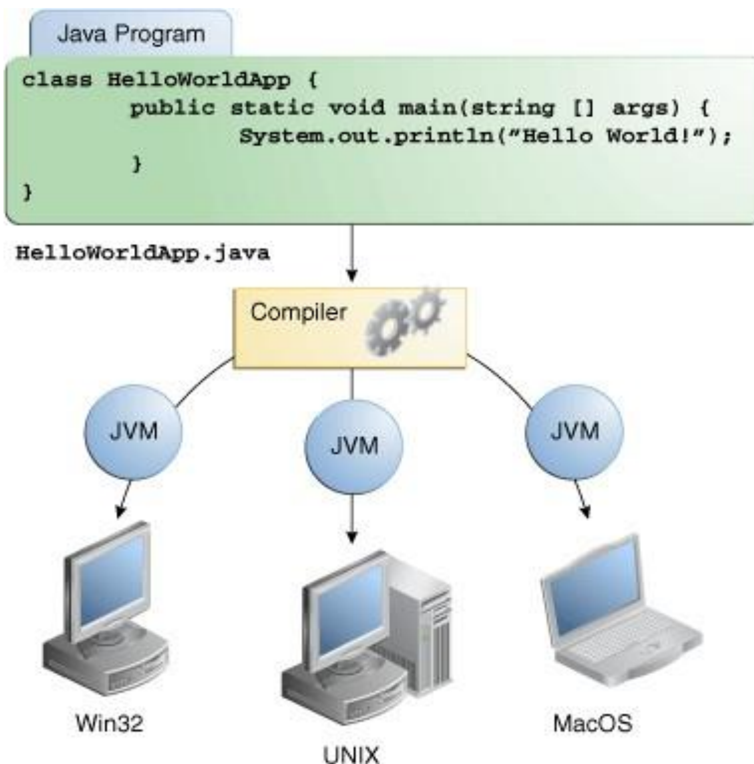
a) Java is a Platform Independent language. Java sometimes called as ***"Write Once, Run Anywhere (WORA)"*** language.
b) Java achieves platform independence through ***bytecode*** and ***JVM***.



Java Program Execution:

1. In the Java programming language, all source code is first written in **plain text files** ending with the **.java** extension.

2. Those source files are then **compiled** into **.class files** by the **javac** compiler.
3. A ***.class*** file does not contain code that is native to your processor; it instead contains ***bytecodes*** — the machine language of the **Java Virtual Machine (Java VM).**
4. The **java** launcher tool then runs your application with an instance of the Java Virtual Machine.
5. **Through the JVM**, the same application is capable of running on multiple platforms.



Bytecode:

Bytecode *is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM).*

Java is platform independent but JVM is platform dependent:

JVM is platform independent because JVM is designed to each platform separately.

Java's magic: bytecode

1) Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform.
2) Although the details of the JVM will differ from platform to platform, all understand the same Java *bytecode*.
3) A Java program is executed by the JVM also helps to make it secure. Because the JVM is in control, it can contain the program and prevent it from generating side effects outside of the system.

Java's Lineage (Ancestry or pedigree)

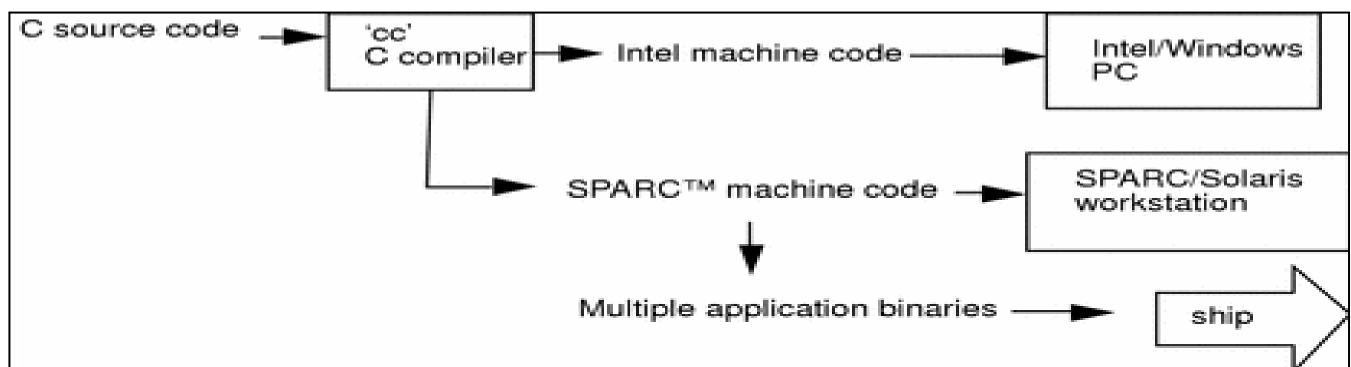
Java is related to C++, which is a direct descendant of C.

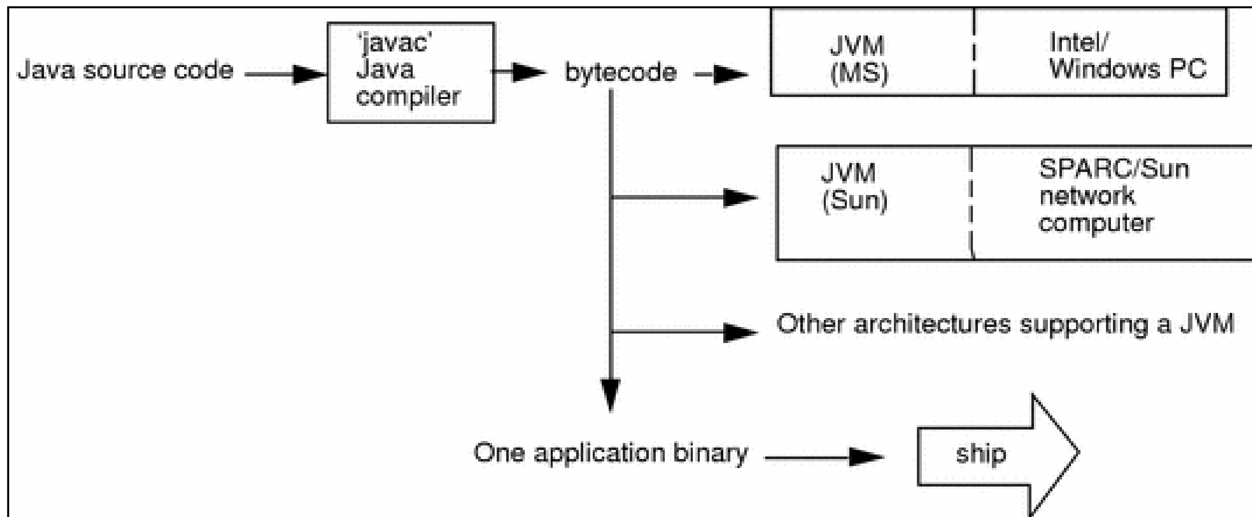
- Much of the character of Java is inherited from these two languages.
- From C, Java derives its syntax.
- Many of Java's object oriented features were influenced by C++.

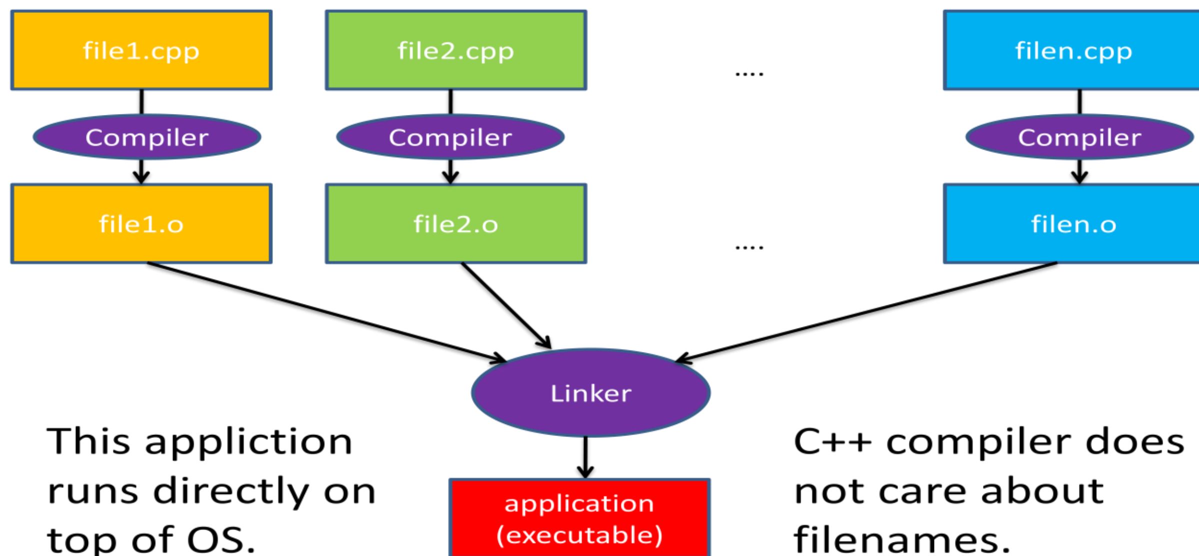
Java vs C vs C++

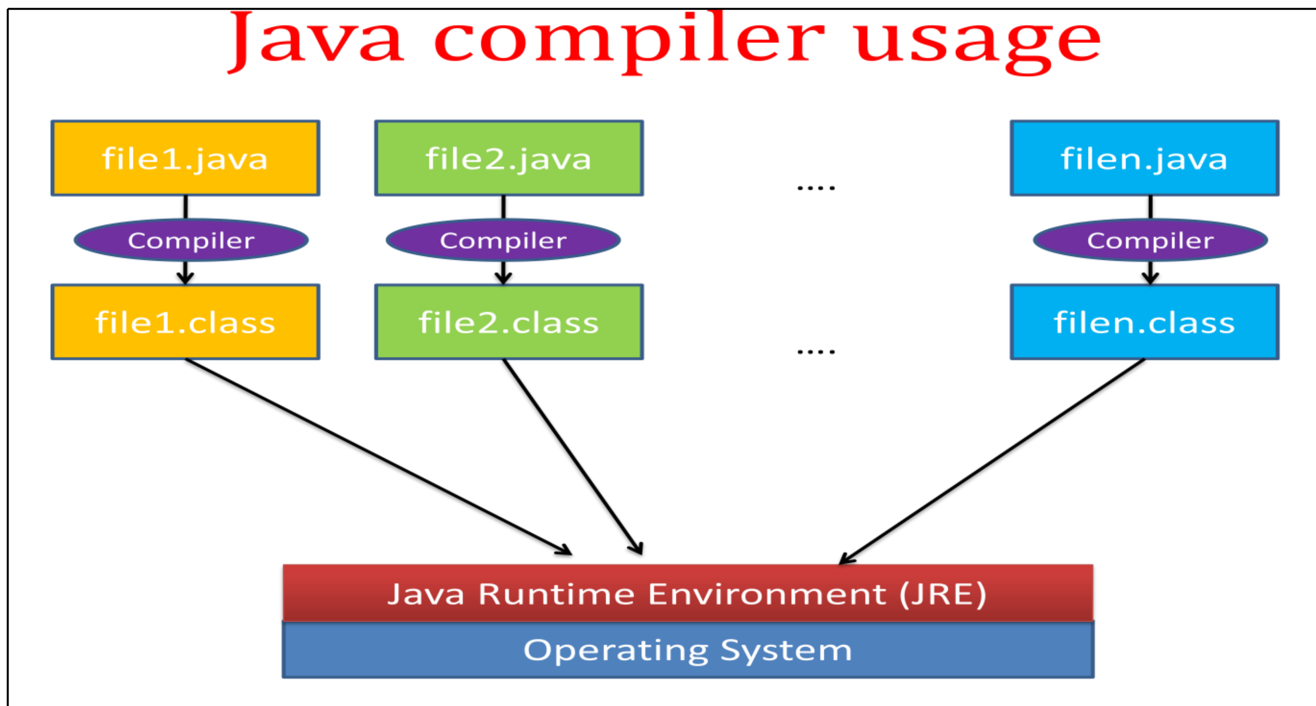
Aspects	C	C++	Java
Developed Year	1972	1979	1991
Developed By	Dennis Ritchie	Bjarne Stroustrup	James Gosling
Paradigms	Procedural	Object Oriented	Object Oriented
Platform Dependency	Dependent	Dependent	Independent
Header files	Supported	Supported	Use Packages (import)
Pointers	Supported	Supported	No Pointers
Storage Allocation	Uses malloc, calloc	Uses new , delete	uses garbage collector
Multi-threading	Not Supported	Not Supported	Supported
Exception Handling	No Exception handling	Supported	Supported
Destructors	No Constructor or Destructor	Supported	Not Supported
Inheritance	No Inheritance	Supported	Supported except Multiple Inheritance
Overloading	No Overloading	Supported	Operator Overloading not Supported

C compile time vs Java Compile Time




Java source code
'javac' Java compiler
bytecode
JVM (MS)
Intel/ Windows PC
JVM (Sun)
SPARC/Sun network computer
Other architectures supporting a JVM
One application binary
ship


C++ compiler & Linker usage
file1.cpp
file2.cpp
....
filen.cpp
Compiler
Compiler
Compiler
file1.o
file2.o
....
filen.o
Linker
application (executable)
This appliction runs directly on top of OS.
C++ compiler does not care about filenames.



How Java changed the Internet /

Java's contribution to the Internet

Servlets : Java on the Server Side

The Internet helped Java to the forefront of programming and Java, in turn, had a profound effect on the Internet.

Java helped in the following features:

a) Simplified web programming
b) Applet development
c) Addressed the issues with the Internet : **portability** and **security**

a) Simplified Web programming

i. Java is used in client / server programming. **Servlet** is used in this programming.
ii. A **servlet** is a small program that executes on the server.
iii. Servlets are used to create dynamically generated content that is then served to the client.

iv. Because servlets (like all Java programs) are compiled into bytecode and executed by the JVM, they are highly portable.
v. Example: **Online Store**

b) Applets

i. An **applet** is a special kind of Java program that is designed to be transmitted over the Internet and automatically executed by a Java-compatible web browser.
ii. The creation of the applet changed Internet programming because it expanded the universe of objects that can move about freely in cyberspace.
iii. The applet is a dynamic, self-executing program.

c) Portability and Security issues

i. **Portability** is a major aspect of the Internet because there are many different types of computers and operating systems connected to it.
   a. The Bytecode and JVM made Java code to be portable.
ii. **Security**
   a. Java achieved the security by confining internet based programs (applet) to the Java execution environment (JRE) and not allowing it access to other parts of the computer.
   b. Java programs will be executed within JVM so, security is preserved from virus attacks.

JAVA Buzzwords / Features

The prime reason behind creation of Java was to bring ***portability*** and ***security*** feature into a computer language. Beside these two major features, there were many other features that played an important role in Java language creation. Those features are:

1. Simple:

a) Java is easy to learn and its syntax is quite simple, clean and easy to understand.
b) It is simple because, Syntax is based on C and C++.
c) Java removed pointers and operator overloading concepts.

2. Object oriented:

a) In Java, everything is an object which has some data and behavior.
b) The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance non-objects.
c) Java supports all the OOP features:

i. Class
ii. Object
iii. Encapsulation
iv. Inheritance
v. Polymorphism
vi. Abstraction

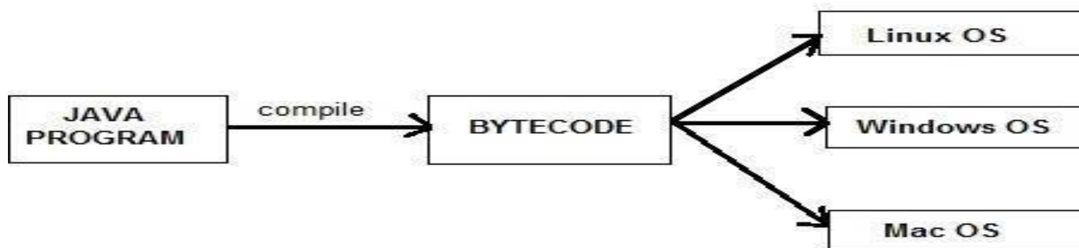
3. Distributed: Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. Java has features that enables a program to invoke methods across a network.

4. Multithreaded: With java's multi-threaded feature it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications. Benefit of multithreading is that it utilizes same memory and other resources to execute multiple threads at the same time.

5. Dynamic: Java programs carry with them substantial amounts of *run-time type information* that is used to verify and resolve accesses to objects at *run time*. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

6. Architecture Neutral (Platform Independent):

i. C and C++ are platform dependency languages, But Java is platform independent language.
ii. Java designers goal was **"Write once; run anywhere, any time, forever"(WORA)**. Java programs written in one operating system can able to run on any operating system.
iii. Java compiler generates an architecture-neutral bytecode which makes the compiled code to be executable on many processors with the presence of Java run time system (JVM).



7. Portable: Java is a portable language means, we can carry the same Java bytecode to any platform.

8. Interpreted: Java's bytecode is interpreted by Java Virtual Machine (JVM). Java's bytecode was carefully designed so that it will be easily translated into Native code.

9. High Performance: Java is faster than traditional interpretation since byte code 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 compiler (JIT).

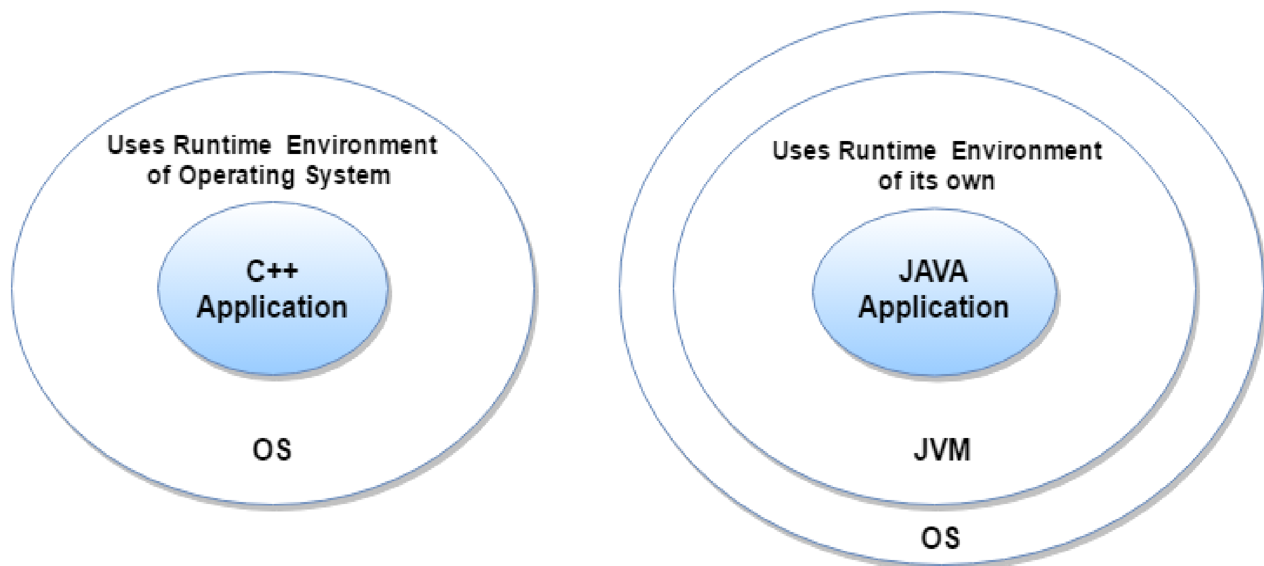
10. Robust: The ability to create robust programs was given a high priority in the design of Java.

Some of the reasons for Java is called as a **robust** language:

i. **Strongly Typed Language**
   a. Java checks the code at compile time.
ii. **Automatic Memory Management (Garbage collection)**
   a. In C/C++, the programmer will manually allocate and free all dynamic memory. This sometimes leads to problems, because programmer will forget to free the memory.
   b. In Java, memory de-allocation is done automatically through ***Garbage Collection***.
iii. **Error Handling**
   a. Java provides object-oriented exception handling to handle the errors in java program.
   b. Using exception handling, all run time errors should be managed by the program.

11. Secure: Java is secured because:

i. No explicit pointers.
ii. Java Programs run inside virtual machine environment.



Evolution of Java

Java continued to evolve at an explosive pace.

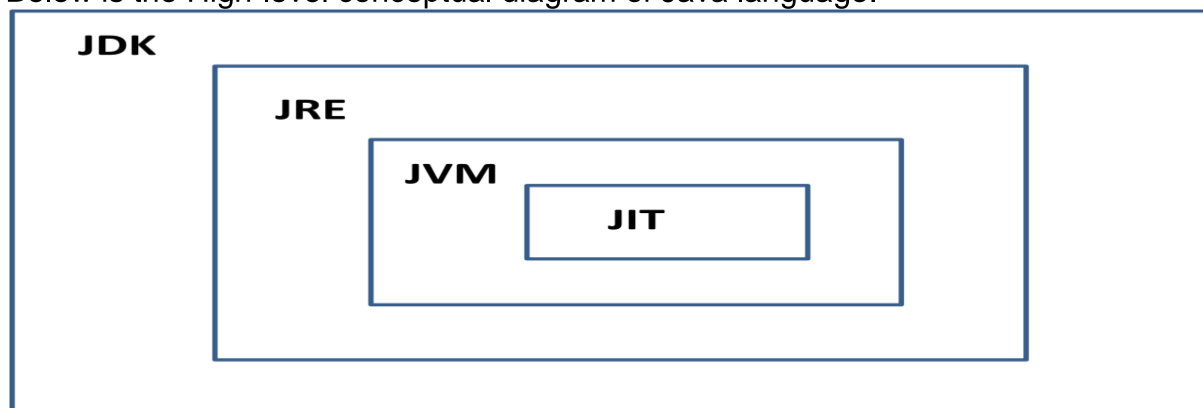
It has dynamic history. Since the beginning, Java has been at the center of a culture of innovation.

Java Version Release History

Version	Description / Features / Enhancements
Java 1.0 / JDK1.0	Initial Version Release
Java1.1/ JDK 1.1	Added AWT,JDBC,RMI,Javabeans etc
Java 2 / J2SE 1.2 (Java 2 Standard Edition)	Swings,Collection framework, JIT compiler in JVM
J2SE 1.3	Smaller set of changes added to the existing functionality
J2SE 1.4	ASSERT keyword,Chained exceptions, Regular Expressions,Image I/O API for reading and writing images
J2SE 5	Generics, Annotations,Autoboxing and auto-unboxing, Enumerations, Enhanced for loop, Variable-length arguments(varargs),static import, Formatted I/O,Concurrency utilities.
Java SE 6 (Java Platform Standard Edition)	Synchroization and compiler performance optimizations, Garbage collection algorithms
Java SE 7 / JDK 7	Try-with-resources,Type interface, Underscores in numeric literals,Binary integer literals, A String can control a **Switch** statement, Parallel programming, Multi Catch Exception
Java SE 8 / JDK 8	New feature – lambda expression, java.util.stream, Enhanced Security

Java Conceptual Diagram

Below is the High-level conceptual diagram of Java language.

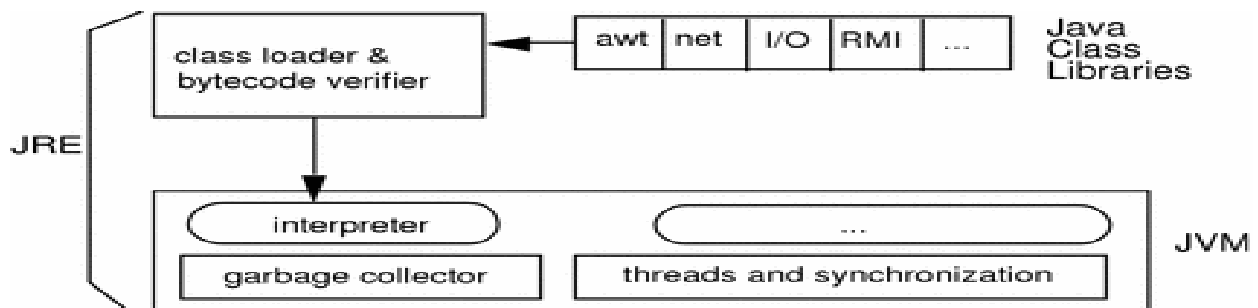


JRE (Java Runtime Environment)

The Java Runtime Environment (JRE) provides

1) the libraries,
2) the Java Virtual Machine,
3) and other components **to run** applets and applications written in the Java programming language.
4) In addition, two key deployment technologies are part of the JRE:
   - **Java Plug-in :** It enables applets to run in popular browsers;
   - **Java Web Start :** It deploys standalone applications over a network.
5) The JRE does not contain compilers or debuggers for developing applets and applications.

JRE = The Libraries + JVM + Java Plug-in + Java Web Start + Other components



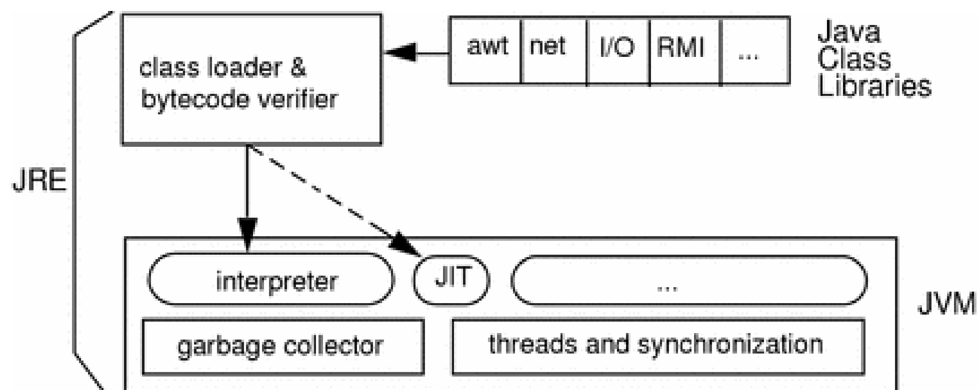
JDK (Java Development Kit)

1) The JDK is a superset of the JRE,
2) and contains everything that is in the JRE, plus tools such as the compilers and debuggers necessary for developing applets and applications.

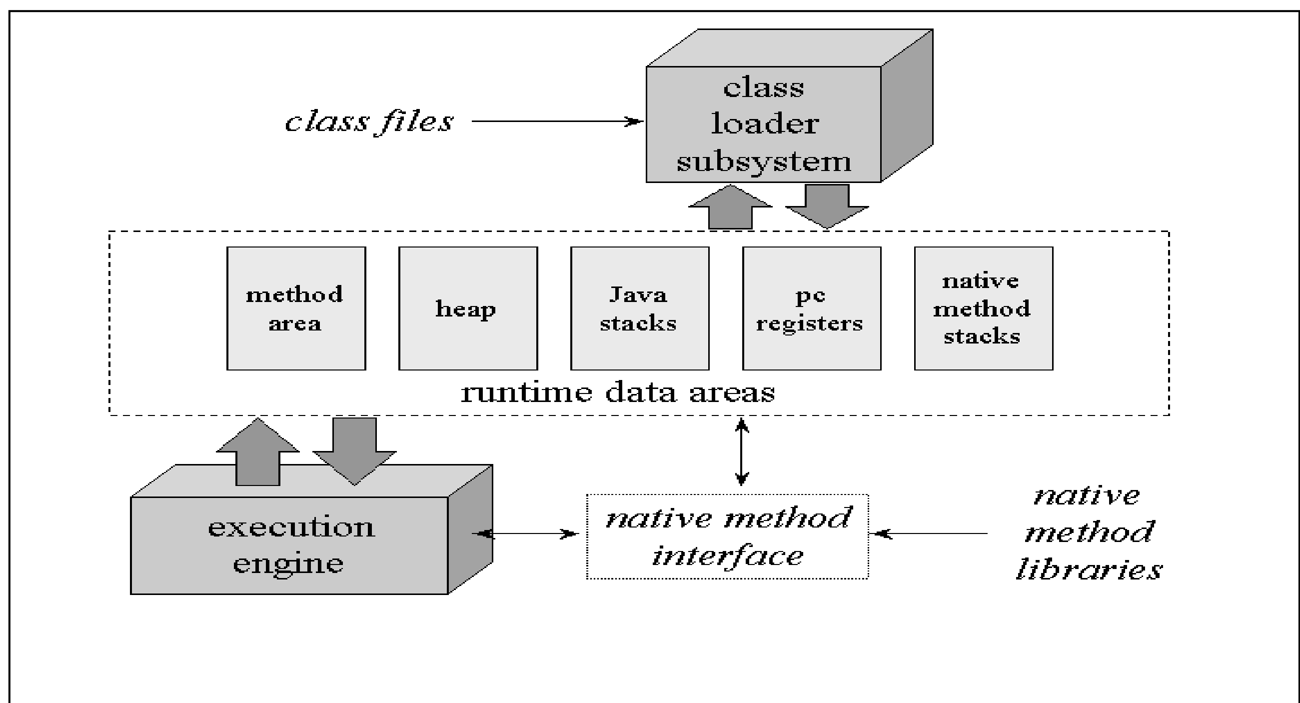
JDK = JRE + compilers + debuggers + etc.

JVM (Java Virtual Machine)

1) JVM is an Interpreter for bytecode.
2) The Java virtual machine is an abstract computing machine that has an instruction set and manipulates memory at run time.
3) The Java Virtual Machine is responsible for the hardware- and operating system-independence of the Java SE platform, the small size of compiled code (bytecodes), and platform security.
4) JVM includes dynamic compilers – **Just In Time Compilers (JIT)** that adaptively compile Java bytecodes into optimized machine instructions.



Internal Architecture of the JVM



1. ***class loader subsystem:*** a mechanism for loading types (classes and interfaces)

2. ***Execution engine:*** a mechanism responsible for executing the instructions contained in the methods of loaded classes.

3. **Run Time Data areas:** The Virtual machine organizes the memory it needs to execute a program into several **runtime data areas**.
   - **i. Method area:** All the class data has been placed on to the method area.
   - **ii. Heap:** All objects the program instantiates is placed onto the heap.
   - **iii. Java Stacks:** A Java stack stores the state of Java (not native) method invocations for the thread.
   - **iv. PC registers:** The value of the pc register indicates the next instruction to execute.
   - **v. Native Method stacks:** Native methods are the methods written in a language other than the Java programming language. These are the stacks created for other languages functions or operating system functions.

JIT (Just-In-Time Compiler)

1) Java is an interpreted language, so it is considered as a slow language when compared with compiled languages.
2) But Hot-spot technology of Java was introduced to boost performance of the Java language.
3) HotSpot provides a **Just-In-Time (JIT) compiler** for bytecode.
4) When a JIT compiler is part of the JVM, selected portions of bytecode are compiled into executable code in real time, on a piece-by-piece, demand basis.
5) Not all sequences of bytecode are compiled—only those that will benefit from compilation.
6) The remaining code is simply interpreted.
7) The just-in-time approach still yields a significant performance boost.

Object Oriented Programming (OOP)

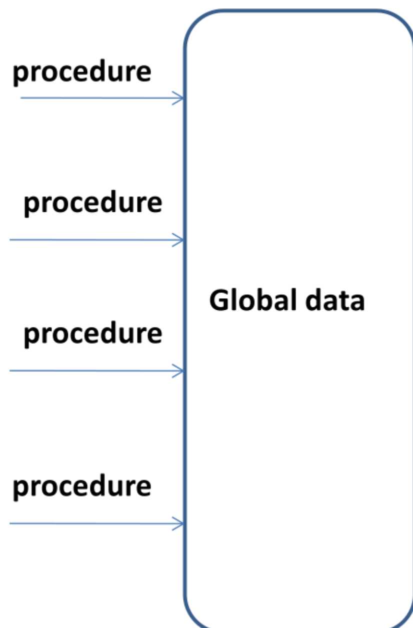
Programming Paradigms

There are two ways of writing programs:

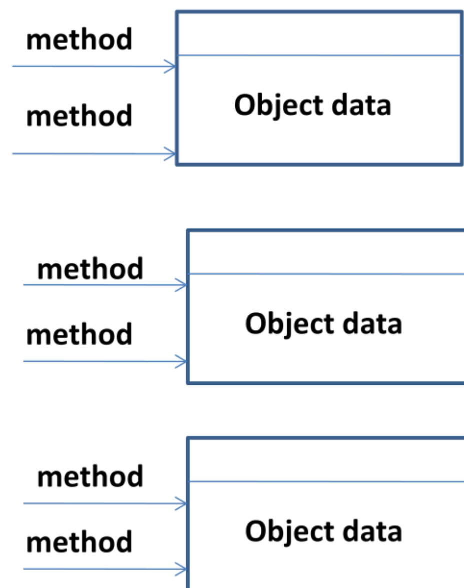
1) Process oriented model (Procedure oriented programming)
2) Object oriented programming

Process oriented model	OOP
Organizes a program around its code.	Organizes a program around its data (that is, objects).
A program is written as a series of linear steps.	A program is written as a collection of classes and objects
Program contains procedures.	Program contains classes, objects.
Ex: **C** uses this approach	Ex: **Java** uses this approach.

Procedure oriented model



Object oriented model



OOP principles / components

Definition: OOP is a programming methodology that helps organize complex programs through the use of inheritance, encapsulation, and polymorphism.

Components:

1. Classes
2. Objects
3. Abstraction

4. Encapsulation (Data Encapsulation)
5. Inheritance
6. Polymorphism
7. Access Levels
8. Interfaces
9. Packages

Three OOP Principles

1. Encapsulation
2. Inheritance
3. Polymorphism

Class: A class defines the structure and behavior (data and code) that will be shared by a set of objects. A class is a structure that defines the data and the methods (functions in other languages) to work on that data.

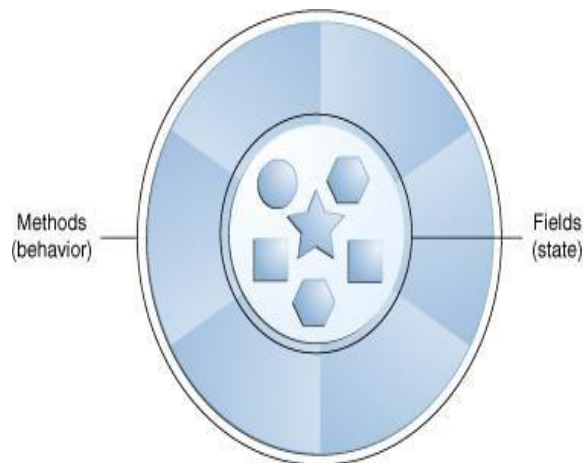
In Java, all program data is wrapped in a class.

```
import java.lang.System;
class ExampleProgram
{
        public static void main(String[] args)
        {
                System.out.println("Hello World");
        }
}
```

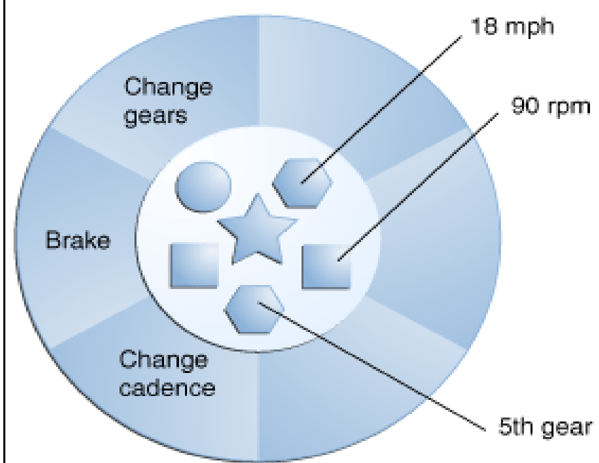
Object: An instance of a class is called as an Object.

1) There can be any number of objects of a given class in memory at any one time.
2) Software objects are conceptually similar to real-world objects.
3) **Objects consist of state and behavior**.
4) An object stores its state in *fields* (variables in some programming languages) and exposes its behavior through *methods* (functions in some programming languages).

A software object



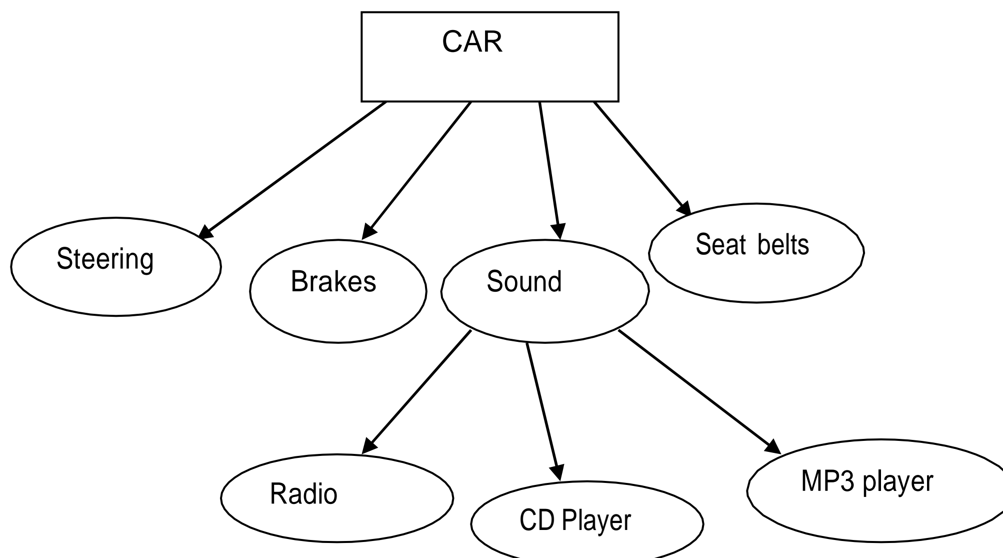
A bicycle modeled as a software object



Abstraction:

a) An essential element of object-oriented programming is abstraction.
b) **Abstraction** is defined as hiding the complexities of the system.
c) A powerful way to manage abstraction is through the use of **hierarchical abstractions** (classifications).
d) Humans manage complexity through abstraction.
e) **For example**, people do not think of a car as a set of tens of thousands of individual parts. They think of it as a well-defined object with its own unique behavior.

Hierarchical classifications



Three OOP Principles

1. Encapsulation:

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.

- ***i.*** Encapsulation is sometimes called as ***Information hiding.***
- *ii. In Java, the basis of encapsulation is the class.*
- iii. Using ***Access Modifiers (private, public,protected,default),*** the members of the class can be protected from misuse by the outside classes.
- iv. ***private*** members can only accessed by the members of the class.
- v. ***public*** members can be accessed by any members of the class or outside members of the class.
- vi. Programs should interact with object data only through the object's methods.
- vii. Benefits of Encapsulation are: **reuse** and **reliability**.

Example:

```
class Bicycle
{
        int cadence = 0;
        int speed = 0;
        int gear = 1;

        void changeGear(int newValue)
        {
                gear = newValue;
        }
}
class BicycleDemo
{
        public static void main(String[] args)
        {
                // Create two different  Bicycle objects
                Bicycle bike1 = new Bicycle();
                Bicycle bike2 = new Bicycle();

                bike1.changeGear(2);
                bike2.changeGear(5);
        }
}
```

Member variables (OR) Instance variables

Member methods (OR) methods

members

2. Inheritance:

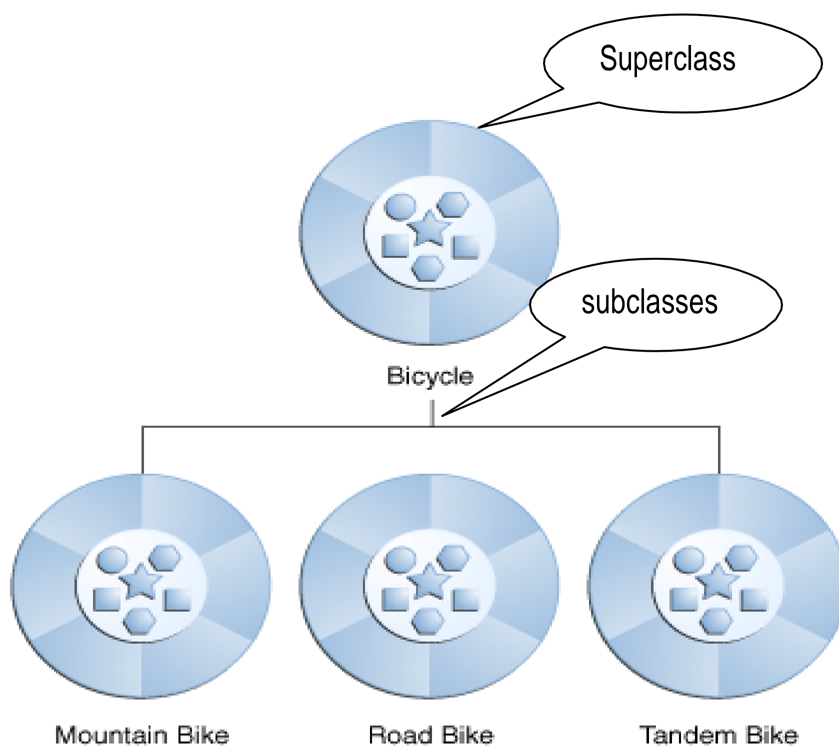
Inheritance is the process by which one object acquires the properties of another object.

i. Inheritance defines relationships among classes in an object-oriented language.
ii. Inheritance allows classes to *inherit* commonly used state and behavior from other classes.
iii. *It supports the concept of hierarchical classification*.
iv. Benefits are : **Code reuse and Decrease of Program size**
v. In the Java programming language, all classes descend from **java.lang.Object, that is, Object is the superclass for all classes.**

Example:

a) Different kinds of objects often have a certain amount in common with each other. **Mountain bikes, road bikes, and tandem bikes**, for example, all share the characteristics of bicycles (current speed, current pedal cadence, current gear).

b) Yet each also defines additional features that make them different: **tandem** bicycles have two seats and two sets of handlebars; **road bikes** have drop handlebars; some **mountain bikes** have an additional chain ring, giving them a lower gear ratio.



Syntax for creating subclasses:

The syntax for creating a subclass is simple. At the beginning of your class declaration, use the ***extends*** keyword, followed by the name of the class to inherit from:

```
class MountainBike extends Bicycle {

    // new fields and methods defining
    // a mountain bike would go here

}
```

Also contains the same fields and methods as Superclass Bicycle, but that code will not appear.

Subclass = Same attributes of super class + any attributes of subclass

- *In the Java programming language, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of subclasses.*

3. Polymorphism: Polymorphism in Greek, meaning "many forms".

Definition:

Polymorphism in java is a concept by which we can perform a *single action by different ways*.

a) In JAVA, More generally, the concept of polymorphism is often expressed by the phrase "one interface, multiple methods."
b) We can perform polymorphism in java by method overloading and method overriding.

Types of Polymorphism:

There are two types of polymorphism in java:

1. Compile time polymorphism and
2. Runtime polymorphism.

Example

i. You might have a program that requires three types of stacks.
ii. One stack is used for integer values, one for floating point values, and one for characters.
iii. The algorithm that implements each stack is the same, even though the data being stored differs.
iv. ***In a non–object-oriented language***, you would be required to create three different sets of stack routines, with each set using different names.

v. However, ***because of polymorphism***, in Java you can specify a general set of stack routines that all share the same names.

FIRST JAVA Program - Description

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

This is called as Multi Line Comment

This is called as Single Line Comment

class : class keyword is used to declare classes in Java.

public : It is an access specifier. Public means this function is visible to all.

static : static is again a keyword used to make a function static. To execute a static function you do not have to create an Object of the class. The **main()** method here is called by JVM, without creating any object for class.

void : It is the return type, meaning this function will not return anything.

main : main() method is the most important method in a Java program. This is the method which is executed, hence all the logic must be inside the main() method. If a java class is not having a main() method, it causes compilation error.

System.out.println: This is used to print anything on the console like *printf* in C language.

Steps to Compile and Run our first Java program

Step 1: Open a text editor and write the code as above.

Step 2: Save the file as Hello.java

Step 3: Open command prompt and go to the directory where you saved your first java program assuming it is saved in D:\

Step 4: Type javac Hello.java and press Enter to compile your code.

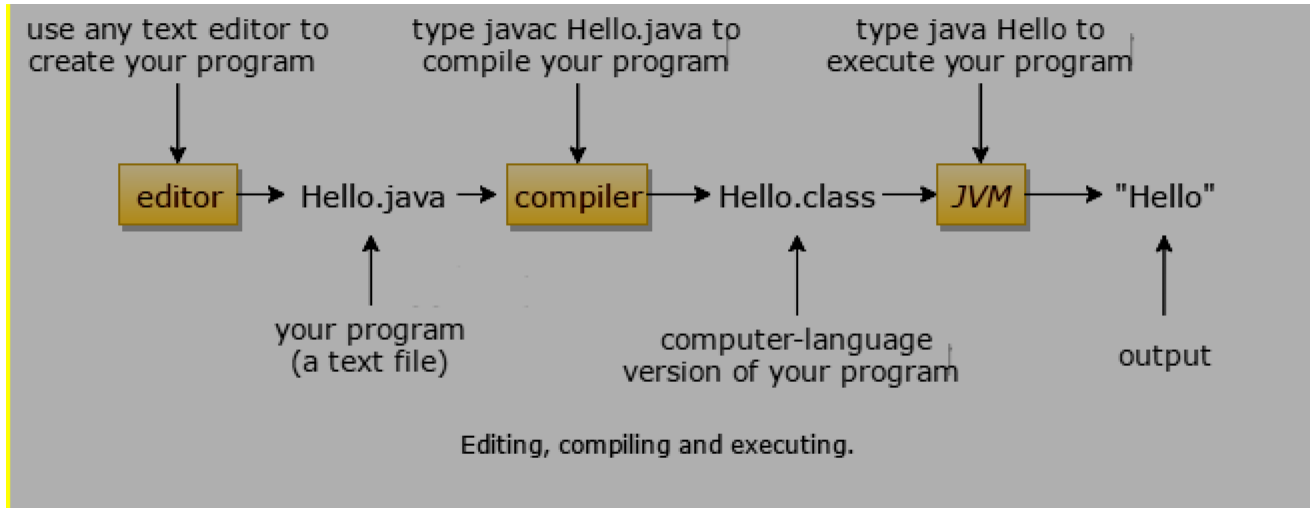
D:\>javac Hello.java

This command will call the Java Compiler asking it to compile the specified file. If there are no errors in the code the command prompt will take you to the next line.

Step 5: Now type java Hello on command prompt to run your program.

D:\>java Hello

Step 6: You will be able to see **Hello** printed on your command prompt.



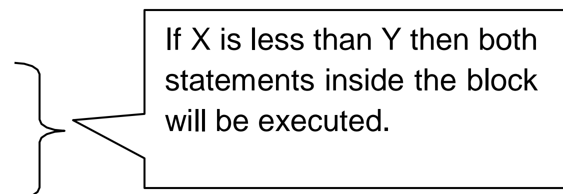
Editing, compiling and executing.

Code blocks (or) blocks of code

i. Java allows two or more statements to be grouped into ***blocks of code***, also called ***code blocks***.
ii. Code blocks can be created by ***enclosing the statements between opening and closing curly braces***.
iii. Once a block of code has been created, it becomes a logical unit that can be used any place that a single statement can.

Example:

```
/* Demonstrate a block of code.
   Call this file "BlockTest.java"
*/

class BlockTest {
   public static void main(String args[]) {
      if(x < y)
      {     // begin a block
        x = y;
        y = 0;
      }   // end of block
```



```
    }
}
```

Lexical Issues (or) Atomic Elements of Java

Java programs are a collection of –
1) whitespace
2) identifiers
3) literals
4) comments
5) operators
6) separators and
7) keywords

whitespace

a) Java is a free-form language – do not need to follow any indentation rules.
b) In Java, whitespace is a space, tab, or newline.

Identifiers

a) Identifiers are used to name classes, variables, and methods.
b) An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters.
c) Java is case-sensitive language.

Valid Identifiers:	**Invalid Identifiers**
AvgTemp	2count
Count	high-temp
A4	Not/ok
$test	
This_is_ok	

Literals

A constant value in Java is created by using a literal representation of it.

Example:

integer literal ⟶ 100
floating-point literal ⟶ 98.6
character literal ⟶ ‘X’
String literal ⟶ “This is a test”

Integer Literals

3 bases are used to represent integer literals –

1) Decimal (base 10)
   Ex: 1, 2, 3 and 42

2) Octal (base 8) - denoted by a leading zero
   Ex: 05

3) Hexadecimal (base 16) - denoted by a leading zero-x (0x or 0X)
   Ex: ox7f

Floating-point Literals

1) Represented in –
   a) Standard notation -> 2.0, 3.14
   b) Scientific notation -> 6.022E23, 314159E-05
2) Floating point literals in Java default to **double** precision.
3) To specify a float literal, we must append an **F** or **f** to the constant.
4) Hexadecimal floating point literals are also supported.
   Ex: 0x12.2P2

Boolean Literals

It contains only two logical values - **true** or **false.** These values do not convert into any numerical representation.

Character Literal

1) A literal character is represented inside a pair of single quotes.
2) All of the visible characters can be directly entered inside the quotes, such as 'a', 'z', and '@'.
3) For characters that are impossible to enter directly, there are several ***escape sequences***.
   Ex: **'\''** for single-quote character
   '\n' for the new line character
4) Characters can be represented in **Octal** and **Hexadecimal notation**
   Octal
   backslash followed by the three-digit number. **For example**, ' \141' is the letter 'a'.
   Hexadecimal
   a backslash-u (\u), then exactly four hexadecimal digits. **For example**, ' \u0061'

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

String Literal

String literals in Java are specified by enclosing a sequence of characters between a pair of double quotes.

Example:

```
 "Hello World"
"two\nlines"
" \"This is in quotes\""
```

Comments

There are 3 types of comments defined by Java.

1) **Single-line comments** ⟶ starts with //

   Example: // Test program

2) **Multi-line comments** ⟶ begins with /* and ends with */

   Example :
   ```
   /* Program for Multiplication
   */
   ```

3) **Documentation comment** ⟶ begins with a /** and ends with a */

   Example :
   ```
   /**
           First Java Program
   */
   ```

Separators

In Java, there are a few characters that are used as separators.

Symbol	Name	Purpose
( )	Parentheses	Used in method parameters, expressions, casting
{ }	Braces	Array initializer, block of code, classes and methods
[ ]	Brackets	Declare array types and used when dereferencing array values
;	Semicolon	Terminates statements.
,	Comma	Variable declarations, 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.
:: ::	Colons	Used to create a method or constructor reference. (Added by JDK 8.)

The Java Keywords

Keywords are the reserved words which can not be used as identifiers in program.

i. There are ***50 keywords*** currently defined in the Java language.
ii. The keywords **const** and **goto** are reserved but not used.
iii. **true**, **false**, and **null** are also reserved. These are the values defined by Java. We may not use these words for the names of variables, classes, and so on.

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

The Java class Libraries (API)

1) Java class libraries provide much of the functionality that comes with Java.
2) Java language is a combination of the Java language itself, plus its standard classes.

3) Java environment relies on several built-in class libraries that contain many built-in methods that provide support for Input / Output, String handling, networking, graphics and Graphical User Interface (GUI).
4) Java has very rich and large **Application Programming Interface (API)** content.

Example:

println() and **print()** methods are built-in methods and available through **System.out**.

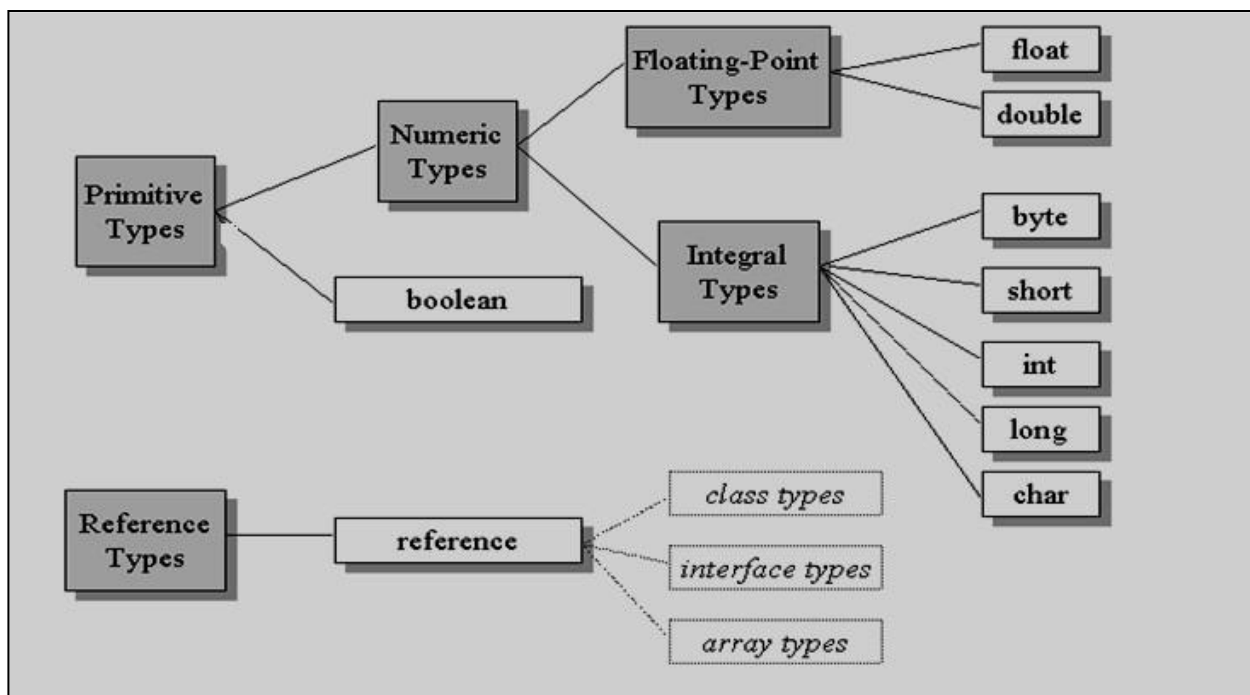
System is a predefined class and **out** is an output stream associated with system.

Data Types, Arrays and Variables

Data Types

Java is a Strongly Typed Language -

a) Every variable has a type, every expression has a type, and every type is strictly defined.
b) All assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.
c) There are no automatic conversions of conflicting types as in some languages.
d) The Java compiler checks all expressions and parameters to ensure that the types are compatible.



Primitive Data Types and Ranges

1) Primitive types are also called as Simple types. The primitive types represent single values.

2) A primitive can be one of ***eight types***:

char, boolean, byte, short, int, long, double, or float.

2) Primitive variables can be declared as class variables (static), instance variables, method parameters, or local variables.

```
byte b;

boolean myBooleanPrimitive;

int x, y, z;            // declare three int primitives
```

3) The number types (both integer and floating point types) are all signed, meaning they can be negative or positive.

4) The leftmost bit (the most significant digit) is used to represent the sign, where a 1 means negative and 0 means positive. The rest of the bits represent the value, using **two's complement notation.**

Type	Bits	Bytes	Minimum Range	Maximum Range
byte	8	1	-128 (-2^7)	127 (2^7-1)
short	16	2	-32768 (-2^{15})	32767 (2^{15}-1)
int	32	4	-2^{31}	$2^{31}-1$
long	64	8	-2^{63}	$2^{63}-1$
float	32	4	4.9e–324	1.8e+308
double	64	8	1.4e–045	3.4e+038
char	16	2	0	65535

boolean: The boolean data type has only two possible values: ***true*** and ***false***.

Characters (char)

a) In Java, the data type used to store characters is char.
b) **char** in Java is not the same as **char** in C or C++. In C/C++, **char** is 8 bits wide. In **Java, char is 16 bit**.
c) 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.
d) At the time of Java's creation, Unicode required 16 bits. Thus, in Java char is a 16-bit type. **The range of a char is 0 to 65,536**. **There are no negative chars**.

Example

Example for *char* data type

// Demonstrate char data type.

```
class CharDemo
 {
     public static void main(String args[]) {
          char ch1, ch2;
           ch1 = 88; // code for X
           ch2 = 'Y';
          System.out.print("ch1 and ch2: ");
          System.out.println(ch1 + " " + ch2);
      }
}
```

Output:

ch1 and ch2 : X Y

// 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);
       }
}
```

Output:

ch1 contains X

ch1 is now Y

Example for *double* data type

// Compute the area of a circle.

```
class Area {
  public static void main(String args[]) {
    double pi, r, a;
    r = 10.8;       // radius of circle
    pi = 3.1416; // pi, approximately
    a = pi * r * r; // compute area

   System.out.println ("Area of circle is ” + a);
  }
}
```

Output:

Area of circle is
366.436224

Example for *boolean* data 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));
   }
}
```

Output:

b is false

b is true

This is executed.

10 > 9 is true

Reference Types

reference

1) A reference variable is used to refer to (or access) an object.
2) Class types, array types and interface types are called as references.

3) Reference variables can be declared as static variables, instance variables, method parameters, or local variables.

```
Object o;
Pen p1;
String s1, s2, s3;            // declare three String vars.
```

Example

```
class A
{
    int i;
    void m1() {
        System.out.println("Hello");
}

class Test
{
        public static void main(String args[])
        {
                A obj;      // reference
                obj = new A(); // object
        }
}
```

Variables

Variables are the identifiers of the memory location, which used to store the data temporary for later use.

Declaring a Variable:

a) In Java, all variables must be declared before they can be used.
b) The basic form of a variable declaration is shown here:

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

*Here, type is the **data type** or **class name** or **interface name**.*

c) To declare more than one variable of the specified type, use a comma-separated list.

Example:

```
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.

```
class DynInit {
   public static void main(String args []) {
          double a = 3.0, b = 4.0;

         // c is dynamically initialized
         double c = a + b;

         System.out.println ("C value is ” + c);
    }
}
```

The Scope(Visibility) and Lifetime of Variables

Scope:

a) A scope determines what objects are visible to other parts of your program.
b) It also determines the lifetime of those objects.

In Java, there are two major scopes:

i. Scope defined by a Class
ii. Scope defined by a Method

Scope defined by a Method (or) block scope

a) The scope defined by a method begins with its opening curly brace and ends with closing curly brace.
b) Variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope.
c) Every block defines a new scope.
d) Scopes can be nested:
   i. objects declared in the **outer scope** will be visible to the code within the **inner scope.**
   ii. Objects declared within the inner scope will not be visible outside it.
   iii. We cannot declare a variable to have the same name as one in an outer scope.

// Demonstrate block scope and Nested scope

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

             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);
     }
}
```

// Nested scope – Outer scope and inner scope with same variable

```
class ScopeErr {
            public static void main(String args[]) {
                        int bar = 1;
                        {                 // creates a new scope
                             int bar = 2; // Compile-time error – bar already defined!
                        }
             }
}
```

Lifetime of a variable

a) The lifetime of a variable is confined to its scope.
b) Variables are created when their scope is entered, and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope.
c) If a variable declaration includes an initializer, then that variable will be reinitialized each time the block in which it is declared is entered.

// 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);
        }
    }
}
```

Type Conversion and Casting

1) Assigning a value of one type to a variable of another type is known as **Type conversion**.
2) In Java, Type conversions will occur in two places :
    a. In Assignments
    b. In Expressions

3) In Java, **Type conversion** is classified **into two types**:

    a. Automatic / Widening conversion / Implicit conversion (Between compatible types)

    b. Casting / Narrowing conversion / Explicit conversion (Between Incompatible types)

Type Conversion in Assignments

1. Automatic Conversions / Widening conversion / Implicit conversion

1) 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:
    - **i. The two types are compatible.**
    - **ii. The destination type is larger than the source type.**
2) For *widening conversions*, the numeric types, including integer and floating-point types, are compatible with each other.
3) There are no automatic conversions from the numeric types to char or boolean.
4) Java also performs an *automatic type conversion* when storing a ***literal integer constant*** into variables of type *byte*, *short*, *long*, or *char*.

```
public class Test
{
    public static void main(String[] args)
    {
        int i = 100;
        long L = i;     //no explicit type casting required
        float f = L;    //no explicit type casting required
        char ch = 100; //no explicit type casting required

        System.out.println("Int value "+i);
        System.out.println("Long value "+L);
        System.out.println("Float value "+f);
        System.out.println("Char value "+ch);
    }
}
```

Ouput:

```
Int value 100
Long value 100
Float value 100.0
Char value  d
```

2. Explicit conversion or Casting or Narrowing conversion

1) To create a conversion between two **incompatible types**, we must use a ***cast***.
2) A *cast is* simply an explicit type conversion.
3) The general form of Casting is –

(*target-type) value*

Note:

- When a floating-point value is assigned to an integer type then ***truncation*** will occur.

- If the size of the value is too large to fit into the target integer type, then that value will be **reduced modulo (the remainder of an integer division by the) target type's range.**

```
// Demonstrate casting
class Casting {
    public static void main(String args[]) {
        byte b;
        int I = 257;
        double d = 323.142;

        System.out.println("\n Conversion of int to byte.");
        b = (byte) I;
        System.out.println("I and b: " + I + " " + b);

        System.out.println("\n Conversion of double to int.");
        I = (int) d;
        System.out.println("d and I :" + d + " " + I);

        System.out.println("\n Conversion 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 in Expressions

Automatic Type Promotion in Expressions

Java's Type Promotion Rules

1. All *byte*, *short*, and *char* values are promoted to *int*.
2. If one operand is a *long*, the whole expression is promoted to *long*.
3. If one operand is a float, the entire expression is promoted to float.
4. If any of the operands are double, the result is double.

Example:

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

Output:

Compilation Error (because, in expression, byte is promoted to int so, the result also int)

Example

```
class Test {
        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);
        }
}
```

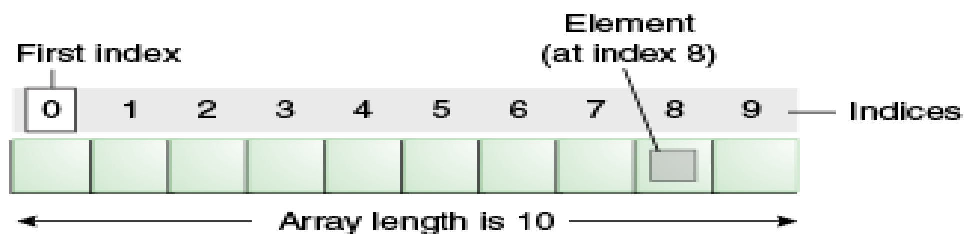
Output:

235.2 + 515 - 126.3616

result = 623.8384122070313

Arrays

1. In Java, arrays are objects.
2. An array is a container object that holds a fixed number of values of a single type.
3. Arrays can hold either **primitives** or **object references**, but the array itself will always be an object on the heap memory.
4. Arrays offer a convenient means of grouping related information.
5. In Java all arrays are dynamically allocated.
6. We can access a specific element in the array by specifying its index within square brackets. **All array indexes start at zero**.



Types of Arrays

2 types of arrays exist:

a) One-Dimensional arrays
b) Multidimensional arrays

One-Dimensional arrays

Syntax:

```
type var-name [ ];
var-name = new type [size];
```

Declaring an Array of Primitives

```
int [ ] key; // Square brackets before name (recommended)
int key [ ]; // Square brackets after name (legal but less readable)
```

Declaring an Array of Object References

```
String [ ] s1;      // Recommended
String s1 [ ];      // Legal but less readable
```

Note:

It is never legal to include the size of the array in your declaration.

Ex:

```
int[5] scores;    // this code won't compile.
```

Creating, Initializing, and Accessing an Array

1) Arrays are created or constructed by using **new** on the array type.
2) To create an array object, Java must know how much space to allocate on the heap, so we must specify the size of the array at creation time.
3) ***The size of the array* is the number of elements the array will hold.**

Creating One-Dimensional Arrays :

```
int[ ]  months;
months = new int[12];
```

Initializing an array:

```
months[0] = 1;    // initialize first element
months[1] = 2;    // initialize second element
months[1] = 2;    // and so forth
```

Accessing an array:

Each array element is accessed by its numerical index:

```
System.out.println("Element 1 at index 0: " + months[0]);
System.out.println("Element 2 at index 1: " + months[1]);
System.out.println("Element 3 at index 2: " + months[2]);
```

Default values for Array elements

The elements in the array allocated by ***new*** will automatically be initialized to

i. ***Zero*** for **numeric** types
ii. ***false*** for **boolean**
iii. ***null*** for **reference** types
iv. *'**\u0000'** for **char** types*

// Demonstrate a one-dimensional array.

```
class Array
{
    public static void main(String args[])
    {
        int month_days[];              // declares an array of integers
        month_days = new int[12];   // allocates memory for 12 integers
        month_days[0] = 31;
        month_days[1] = 28;
        month_days[2] = 31;
        month_days[3] = 30;
        month_days[4] = 31;
        month_days[5] = 30;
        month_days[6] = 31;
        month_days[7] = 31;
        month_days[8] = 30;
        month_days[9] = 31;
        month_days[10] = 30;
        month_days[11] = 31;
    System.out.println("April has " + month_days[3] + " days.");
    }
}
```

array initializer

- Arrays can be initialized when they are declared.
- An array initializer is a list of ***comma-separated*** *expressions* surrounded by curly braces.
- There is no need to use new.
- *The Java **run-time system** will check to be sure that all array indexes are in the correct range.*

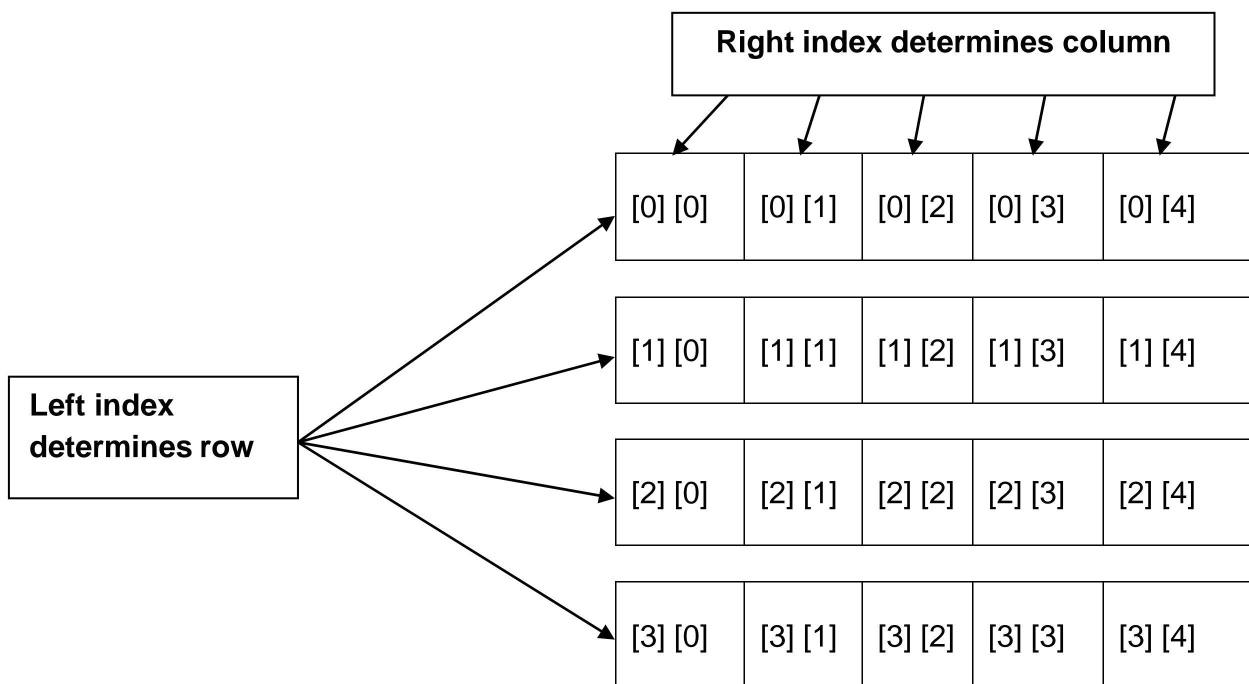
- *If we try to access elements **outside the range of the array** (negative numbers or numbers greater than the length of the array), we will get a run-time error.*

```
class Array
{
    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

1. **In Java, multidimensional arrays are actually arrays of arrays.**

    int twoD[ ][ ] = new int[4][5];   -> an array of arrays of int.



2. **In Java, When we allocate memory for a multidimensional array, we need only specify the memory for the first (leftmost) dimension. We can allocate the remaining dimensions separately.**

```
int twoD[ ][ ] = new int[4][ ];
        twoD[0] = new int[5];
        twoD[1] = new int[5];
        twoD[2] = new int[5];
        twoD[3] = new int[5];
```

// Demonstrate a two-dimensional array.

```
class TwoDArray {  p
        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.println(twoD[i][j] + " ");
            System.out.println();
         }
      }
}
```

Output:

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

<u>Example:</u>

Write a Java Program to print the following output

```
0
1   2
3   4   5
6   7   8   9
```

```
class TwoD
{
       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];

              int i, j, k = 0;

             for(i=0; i<4; i++)
```

```
                for(j=0; j<i+1; j++)
                {
                        twoD[i][j] = k;
                         k++;
                }

            for(i=0; i<4; i++)
              {
                    for(j=0; j<i+1; j++)
                            System.out.print(twoD[i][j] + " ");
                      System.out.println();
              }

        }
}
```

Three dimensional arrays

It requires three dimensions.

int threeD[ ][ ][ ] = new int[3][4][5];

Strings

1) String is not a primitive type.
2) String defines an object.
3) The String type is used to declare string variables.
4) We can also declare arrays of strings.
5) A quoted string constant can be assigned to a String variable.
6) A variable of type String can be assigned to another variable of type String.
7) We can use an object of type String as an argument to println( ).

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

str is an object reference of type

Pointers

1) C and C++ supports pointers.
2) But, Java does not support or allow pointers
3) Java is designed in such a way that as long as we stay within the confines of the execution environment, we never need to use a pointer.
4) There would not be any benefit using pointers.
5) In Java, memory management is handled by **Garbage Collector**.

PART-A

1. Define code block or blocks ofcode.
2. Define Literal? Explain different literal types in Java.
3. Define API.
4. What are the Java's Type promotion rules in expressions?
5. Define bytecode
6. Define JRE,JDK,JIT and JVM
7. Whatis scope and lifetime of a variable?
8. Java is strongly typed language. Explain.

PART-B

1. What is Java'smagic? Explain why Javais called as Platform independentlanguage / Architecture Neutral language.
   (OR)
   Whatisaplatformindependentlanguage?HowJavaachieves platformindependence.
   (OR)
   What is portability? Javaisaportable language. Justify.
   (OR)
   Explain Java's conceptual diagram in-line with platform independence.
   (OR)
   Explain bytecode, JDK, JRE, JVM and JIT with neat diagrams.
2. What is Java? Explain **java Buzzwords or features**.
3. Explain why Javaismore secured than otherlanguages
   (OR)
   How Javaisimportant to the Internet / What is Java's contribution to the internet?
   (OR)
   Explain Java's applets, servlets, portability and security.
4. Explain Java'sEvolution?
   (OR)
   Write different versions of Java.
5. What are the Object Oriented Programming (OOP) features? Explain them.
   (OR)
   Explain Data abstraction, Encapsulation, Inheritance and Polymorphism.
6. Discuss Lexical issues or atomic elements in Java.
7. Whatisa data type? Explain different data types in Java
   (OR)
   Explain 8 primitive data types and non-primitive types in Java.
   (OR)
   What are Java's data types? Why character data type is 16 bits in Java.
8. Define a variable? Explain the scope and lifetime of a variable.
9. Write the differences between procedure orientedprogramming and Object oriented programming.
10. What is a Typeconversion and Casting/Type casting?
    (OR)
    What is widening conversion and Narrowing conversion?
11. Define an array? What are the different types of arrays.
    (Or)
    Explain one-dimensional array and multi-dimensional array with an example.

UNIT – I END

Object Oriented Programming using JAVA

UNIT II

Operators

Java's operators can be divided into **following** groups:

1. Arithmetic operators
2. The Bitwise operators
3. Relational operators
4. Boolean Logical operators
5. The assignment operator
6. The **? (Ternary)** operator

1. Arithmetic Operators:

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra.

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

- The operands of the arithmetic operators must be of a **numeric type.**
- We cannot use them on **boolean** types, but we can use them on **char** types, since the **char** type in Java is, essentially, a subset of **int.**
- The **unary minus** operator negates its single operand.
- The **unary plus** operator just returns the operand value.
- When the **division operator** is applied to an integer type, there will be no fractional component attached to the result.

// Demonstrate the basic arithmetic operators.

```
class BasicMath {
    public static void main(String args[]) {
    // arithmetic using integers
    System.out.println("Integer Arithmetic");
    int a = 1 + 1;
    int b = a * 3;
    int c = b / 4;
    int d = c - a;
    int e = -d;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
    System.out.println("d = " + d);
    System.out.println("e = " + e);

    // arithmetic using doubles
    System.out.println("\nFloating Point Arithmetic");
    double da = 1 + 1;
    double db = da * 3;
    double dc = db / 4;
    double dd = dc - a;
    double de = -dd;
    System.out.println("da = " + da);
    System.out.println("db = " + db);
    System.out.println("dc = " + dc);
    System.out.println("dd = " + dd);
    System.out.println("de = " + de);
    }
}
```

Output:

```
Integer Arithmetic
a = 2
b = 6
c = 1
d = -1
e = 1

Floating Point Arithmetic
da = 2.0
db = 6.0
dc = 1.5
dd = -0.5
de = 0.5
```

The Modulus Operator (%) : The modulus operator, **%**, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types.

// **Demonstrate the % operator.**

```
class Modulus {
    public static void main(String args[]) {
        int x = 42;
        double y = 42.25;
        System.out.println("x mod 10 = " + x % 10);
        System.out.println("y mod 10 = " + y % 10);
    }
}
```

Output:

```
x mod 10 = 2
y mod 10 = 2.25
```

Arithmetic Compound Assignment Operators (+=, – =, *=, /=, %=)

Any statement of the form **var = var op expression;** can be written as **var op= expression;**

// Demonstrate several assignment operators.

```
class OpEquals {
        public static void main(String args[]) {
                int a = 1;
                int b = 2;
                int c = 3;
                a += 5;
                b *= 4;
                c += a * b;
                c %= 6;
                System.out.println("a = " + a);
                System.out.println("b = " + b);
                System.out.println("c = " + c);
        }
}
```

Output:

a = 6
b = 8
c = 3

Benefits of compound assignment operators

1. They save a bit of typing.
2. In some cases they are more efficient than are their equivalent long forms.

Increment and Decrement (++ and --)

The increment operator (++)

The increment operator increases its operand by one.

x = x + 1; can be rewritten like this by use of the increment operator: **x++;**

The decrement operator (--)

The decrement operator decreases its operand by one.

x = x - 1; can be rewritten like this by use of the increment operator: **x--;**

// Demonstrate ++.

```
class IncDec {
  public static void main(String args[]) {
      int a = 1;
      int b = 2;
      int c;
      int d;
      c = ++b;
      d = a++;
      c++;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
    System.out.println("d = " + d);
  }
}
```

Output:

a = 2
b = 3
c = 4
d = 1

- These operators can appear both in **prefix** or **postfix** form.

Prefix form: The operand is incremented or decremented before the value is obtained for use in the expression.

x = 42; y = ++x; equal to x = x + 1; y = x; → Y = 43, X = 43

Postfix form: The previous value is obtained for use in the expression, and then the operand is modified.

x = 42; y = x++; equal to Y=x; X = x + 1; → Y = 42 , X = 43

2. 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.**

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

2's complement

1) Java uses an **encoding** known as **two's complement** to represent negative integers.
   i. Write the binary number for +ve value
   ii. Invert the bits (changing 1's to 0's and vice versa)
   iii. Add 1 to the result obtained in step ii.

   Example:

   –42 is represented by inverting all of the bits in 42, or 00101010, which yields 11010101, then adding 1, which results in 11010110, or –42.

2) **Decoding**

To decode a negative number, first invert all of the bits, then add 1.

For example, –42, or 11010110 inverted, yields 00101001,or 41,so when you add 1, you get 42.

3) The leftmost bit (high-order bit) determines sign of an integer. If the bit is 1 then it is a negative number. If the bit is 0 then it is a positive number.

The Bitwise Logical Operators

The bitwise logical operators are &, |, ^, and ~.

The Bitwise NOT

Also called the **bitwise complement**, the **unary NOT operator**, **~**, inverts all of the bits of its operand.

The Bitwise AND

The AND operator, **&**, produces a **1** bit if both operands are also **1**.

A **zero** is produced in all other cases.

The Bitwise OR

The OR operator, |, combines bits such that if either of the bits in the operands is a 1, then the resultant bit is a 1.

The Bitwise XOR

The XOR operator, ^, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is **zero**.

A	B	A	B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

```
// Demonstrate the bitwise logical operators.
class BitLogic
{
    public static void main(String args[]) {
        int a = 3;  //     0011 in binary
        int b = 6; // 0110 in binary
```

Output

a|b   = 7

a & b = 2

a ^ b = 3

```
            int c = a | b;
            int d = a & b;
            int e = a ^ b;

             System.out.println("      a|b = " + a | b);
             System.out.println("      a&b = " + a & b);
             System.out.println("      a^b = " + a ^ b);
        }

}
```

The Left Shift

The left shift operator, <<, shifts all of the bits in a value to the left a specified number of times. It has this general form:

```
value << num
```

a) Here, **num** specifies the number of positions to left-shift the value in value.
b) For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right.

```
// Left shifting a byte value.
class ByteShift
{
  public static void main(String args[]) {
        byte a = 64, b;
        int i;

       i = a << 2;
      b = (byte) (a << 2);
      System.out.println("Original value of a: " + a);
      System.out.println("i and b: " + i + " " + b);
   }
}
```

Output:

Originalvalueof a: 64

i and b: 256 0

The Right Shift

1) The right shift operator, >>, shifts all of the bits in a value to the right a specified number of times. It has this general form:

```
value >> num
```

2) Here, ***num*** specifies the number of positions to right-shift the value in ***value***.

3) Each time you shift a value to the right, it **divides that value by two—and discards any remainder.**

```
int a = 35;
a = a >> 2; // a contains 8
```

```
00100011  35
    >> 2
00001000   8
```

4) When we are shifting right, the top (leftmost) bits exposed by the right shift are filled in with the previous contents of the top bit. This is called **sign extension** and serves to preserve the sign of negative numbers.

```
11111000   –8
   >>  1
11111100   –4
```

The Unsigned Right Shift (>>>)

Java's unsigned, shift right operator, >>>, which always **shifts zeros into the high-order bit.**

```
int a = -1;

a = a >>> 24;
```

11111111 11111111 11111111 11111111 **–1 in binary as an int**

    >>>24

00000000 00000000 00000000 11111111 **255 in binary as an int**

Bitwise Operator Compound Assignments

a = a >> 4; can be written as a >>= 4;

```
class OpBit {
   public static void main(String args[]) {
        int a = 1;
        int b = 2;
        int c = 3;
        a |= 4;
        b >>= 1;
        c <<= 1;
        a ^= c;
      System.out.println("a = " + a);
      System.out.println("b = " + b);
      System.out.println("c = " + c);
   }
}
```

Output

a=3

b=1

c=6

3. Relational Operators

1) The relational operators determine the relationship that one operand has to the other. Specifically, they determine equality and ordering.
2) The outcome of these operations is a boolean value.

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

3) The relational operators are most frequently used in the expressions that control the **if** statement and the various **loop statements**.

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

Output:

false

4. Boolean Logical Operators

The Boolean logical operators shown here 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

Example

A	B	A | B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

// Demonstrate the boolean logical operators.

```
class BoolLogic {

public static void main(String args[]) {
        boolean a = true;
        boolean b = false;
        boolean c = a | b;
        boolean d = a & b;
        System.out.println( a);
        System.out.println(b);
        System.out.println( c);
        System.out.println(d);
    }
}
```

Output:

```
true
false
true
false
```

Short-Circuit Logical Operators(&& , ||)

The secondary versions of the **Boolean AND (&)** and **OR (|)** operators are known as Short-circuit operators.

If we use short-circuit operators || and &&, Java will not bother to evaluate the right hand operand when the outcome of the expression can be determined by the left operand alone.

Short-circuit AND - && (Conditional-and)

```
class ShortCktAnd
{
	public static void main(String args[])
	{
		int num = 20;
		int denom = 0;

		if (denom != 0 && num / denom > 10)
		{
			System.out.println("Inside If");
		}
		System.out.println("After if");
	}
}
```

Output:

After if

Short-circuit OR - || (Conditional-or)

```
class ShortCktOr
{
	public static void main(String args[])
	{
		int num = 20;
		int denom = 0;

		if (denom == 0 || num / denom > 10)
		{
			System.out.println("Inside If");
		}
		System.out.println("After if");
	}
}
```

Ouput:

Inside if

After if

5. The Assignment Operator

The assignment operator is the single equal sign, **=.**

It's general form is:

```
var = expression;
```

Here, the type of ***var*** must be compatible with the type of ***expression***.

Example:

int x, y, z;

x = y = z = 100;

The ternary / three-way operator (?)

The ? has this general form:

expression1 ? expression2 : expression3

- expression1 can be any expression that evaluates to a boolean value.
- If expression1 is true, then expression2 is evaluated; otherwise, expression3 is evaluated.
- Both expression2 and expression3 are required to return the same (or compatible) type, which can't be void.
- It will be used to replace certain types of if-then-else statements.

Example

// Demonstrate ?.

```
class Ternary {
	public static void main(String args[])
	{
		int num = 20;
		int denom = 0;

		int ratio = denom == 0 ? 0 : num / denom;
		System.out.println(ratio);
	}
}
```

Ouput:

0

Operator Precedence

1) In the below table, Operators with higher precedence are evaluated before operators with relatively lower precedence. Operators on the same line have equal precedence.
2) When operators of equal precedence appear in the same expression, All binary operators except for the assignment operators are evaluated from left to right; assignment operators are evaluated right to left.

Highest						
postfix ++	postfix --					
prefix ++	prefix --	~	!	unary +	unary -	(type-cast)

*	/	%				
+	-					
>>	>>>	<<				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
|						
&&						
||						
?:						
->						
=	op=					
Lowest						

Using Parentheses

1) **Parentheses** raise the precedence of the operations that are inside them.
2) **Parentheses** can sometimes be used to help clarify the meaning of an expression.

Example

a >> b + 3 → This expression first adds 3 to b and then shifts a right by that result.

If we want to first shift a right by b positions and then add 3 to that result, we will need to parenthesize the expression like this:

(a >> b) + 3

Control Statements

Programming languages uses control statements to cause the flow of execution to advance and branch based on changes to the state of a program.

Categories of Java's Control statements

Java's program control statements can be put into **three categories:**

1) selection,
2) iteration, and
3) jump.

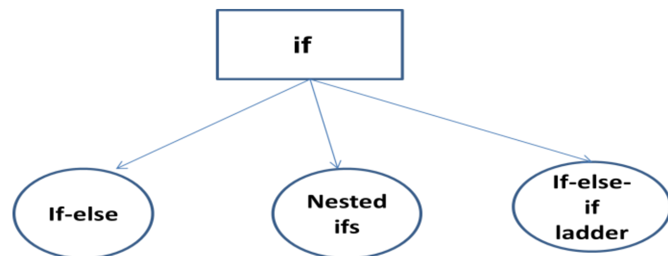
1) Selection statements (if and switch)

Selection statements allow the program to choose different paths of execution based upon the outcome of an expression or the state of a variable.

Java supports two selection statements: **if** and **switch.**

if

- The **if** statement is Java's conditional branch statement.
- It can be used to route program execution through two different paths.



General form

if (*condition*) *statement1*;
else *statement2*;

- 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.
- **else** clause is optional.

Working of if

- If the *condition* is true, then *statement1* is executed.
- Otherwise, *statement2* (if it exists) is executed.

Example:

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

Explanation:

If a is less than b, then a is set to zero. Otherwise, b is set to zero.

Nested ifs

A nested if is an if statement that is the target of another if or else.

Example:

```
if(i == 10) {
   if(j < 20) a = b;
   if(k > 100) c = d;
```

```
        else a = c;
 }
else a = d;
```

The if-else-if Ladder

A sequence of nested ifs is the if-else-if ladder.

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

Working of if-else-if ladder

1) The if statements are executed from the top down.
2) As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed.
3) If none of the conditions is true, then the final else statement will be executed.
4) If there is no final else and all other conditions are false, then no action will take place.

```
// Demonstration 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 = "Invalid Month";

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

Q: This program is to determine which season a particular month is in.

Output:

April is in the Spring.

switch

```
switch (expression) {
        case value1:
             // statement sequence
             break;
        case value2:
              // statement sequence
```

- The **switch** statement is Java’s multiway branch statement.
- It provides an easy way to dispatch execution to different parts of your code based on the value of an expression.
- It often provides a better alternative than a large series of if-else-if statements.
- *expression* must be of type **byte,short,int,char,enum,or String**
- Duplicate **case** values are not allowed. **case** constants must be unique.

```
            break;
         .
         .
         .
      case valueN :
            // statement sequence
            break;
      default:
            // default statement sequence
}
```

Working of switch

- The value of the **expression** is compared with each of the values in the case statements.
- If a match is found, the code sequence following that case statement is executed.
- If none of the constants matches the value of the expression, then the **default** statement is executed.
- However, the **default** statement is optional.
- If no case matches and no default is present, then no further action is taken.
- The **break** statement is used inside the switch to terminate a statement sequence.
- When a **break** statement is encountered, execution branches to the first line of codethat follows the entire switch statement.
- **break** is also optional. If **break** is omitted, the program just keeps executing the remaining case blocks until either a break is found or the switch statement ends.

//Demonstration of switch statement

```
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:
```

Q: This program is to determine which season a particular month is in.

Output:

April is in the Spring.

```
            case 10:
            case 11:
                  season = "Autumn";
                  break;
            default:
                  season = "Invalid Month";
        }
        System.out.println("April is in the " + season + ".");
    }
}
```

Switch Example without break:

```
int x = 1;
switch(x) {
        case 1: System.out.println("x is one");
        case 2: System.out.println("x is two");
        case 3: System.out.println("x is three");
      }
 System.out.println("out of the switch");
```

Output:

x is one
x is two
x is three
out of the switch

Nested switch Statements

We can write 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 1: // no conflicts with outer switch
            System.out.println("target is one");
            break;
       }
   break;
   case 2: // ...
```

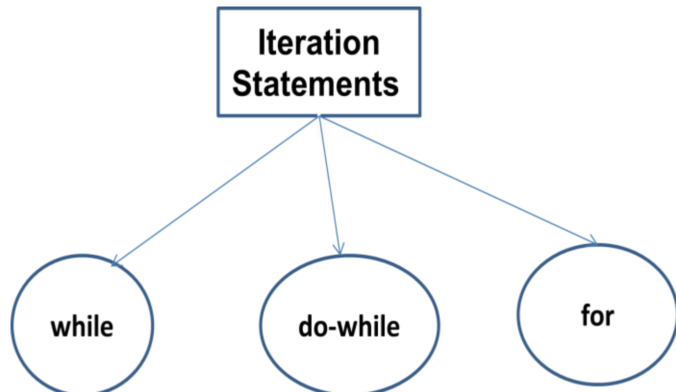
Here, outer switch and inner switch contains the same case constant 1, but it is perfectly valid.

Difference between if and switch

- The **switch** differs from the **if** in that switch can only test for equality, whereas if can evaluate any type of Boolean expression.
- A **switch** statement is usually more efficient than a set of **nested ifs**.
- If we need to select among a large group of values, a **switch** statement will run much faster than the equivalent logic coded using a sequence of **if-elses.**

Iteration statements (while,do-while and for)

- Java's **iteration** statements are **for**, **while**, and **do-while**
- Iteration statements enable program execution to repeat one or more statements, which are called as **loops.**
- A **loop** repeatedly executes the same set of instructions until a termination condition is met.



while

It repeats a statement or block while its controlling expression is true.

syntax

```
while(condition)
{
    // body of loop
}
```

Working of while

- 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.

Example

```
class While {
    public static void main(String args[]) {
        int n = 1;
         while(n < 5) {
            System.out.println(n);
            n++;
         }
    }
 }
```

Output

```
1
2
3
4
```

Null statement

null statement is a statement which consists only of a semicolon.

The body of the while can be empty, that is, it can be a **null statement**.

Example

```
int n = 1;

while(n++ < 5);
```

Output:

No output will be printed but the loop stops execution after 4 iterations.

do-while

```
do {
        // body of loop
} while (condition);
```

Working of do-while

- In **do-while**, conditional expression is at the bottom of the loop.
- Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression.
- If this expression is true, the loop will repeat. Otherwise, the loop terminates.
- The do-while loop always executes its body at least once.

Example

```
class While {
    public static void main(String args[]) {
        int n = 1;
          do{
              System.out.println(n);
               n++;
          }while(n < 5);
      }
 }
```

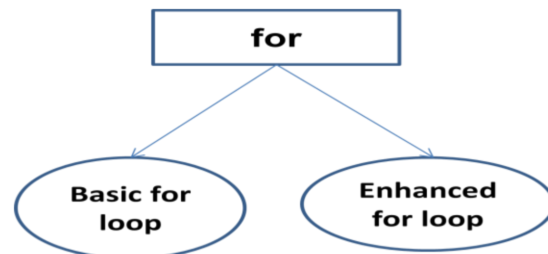
Output

```
1
2
3
4
```

for

Beginning with JDK 5, there are two forms of the for loop.

1. **Basic for loop**
2. **For-each (Enhanced for loop)**



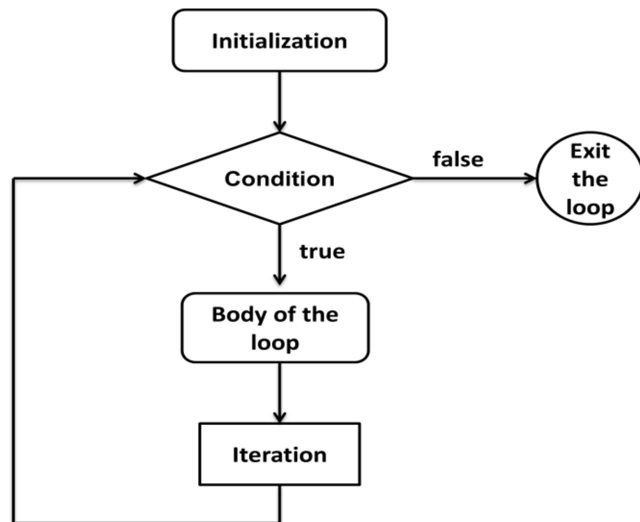
Basic for loop

```
for(initialization; condition;
iteration)
{
        // body
 }
```

Working of for loop

1. When the loop first starts, the **initialization** portion of the loop is executed.
2. Next, **condition** is evaluated. This must be a Boolean expression.
3. If this expression is true, then the body of the loop is executed.
   If it is false, the loop terminates.
4. Next, the **iteration** portion of the loop is executed.
5. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass.

Flow Diagram



Example

```
class For {
    public static void main(String args[]) {
          for(i=1; i<5;i++) {
                    System.out.println(n);
          }
     }
 }
```

Output

```
1
2
3
4
```

1. **Declaring Loop Control Variables Inside the for Loop**
   It is possible to declare the variable inside the initialization portion of the for.

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

2. **Using the Comma**
   i. Using **comma**, we can include more than one statement in the **initialization** and **iteration** portions of the **for** loop.
   ii. We can not use more than one statement in increment section.

```
int a, b;

  for(a=1, b=4; a<b; a++, b--) {
      System.out.println("a = " + a);
      System.out.println("b = " + b);
}
```

Output:

a=1
b=4
a=2
b=3

For loop variations

1. The condition can be any Boolean expression.
2. Either the **initialization** or the **iteration** expression or both may be absent.

```
public static void main(String args[]) {
        int i;
        boolean done = false;

         i = 0;
        for( ; !done; ) {
             System.out.println( i);
                   if(i == 4) done = true;
                   i++;
             }
        }
```

Output:

1
2
3
4

3. We can leave all three parts of the **for** empty.

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

Output:
Infinite loop

The For-Each Version of the for Loop

i. Beginning with JDK 5, a second form of **for** was defined that implements a “for-each” style loop.
ii. A **for-each** style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish.

form of for-each

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

Here,

- **type** specifies the type

- **itr-var** specifies the name of an iteration variable that will receive the elements from a collection, one at a time, from beginning to end.
- With each iteration of the loop, the next element in the collection is retrieved and stored in itr-var.
- The loop repeats until all elements in the collection have been obtained.

Example 1:

```
int nums[] = { 1, 2, 3, 4, 5 };

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

Output:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
```

Example 2: Iterating over multidimensional arrays

```
int sum = 0;
   int nums[][] = { {1,2},{3,4}};

  // 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("Sum: " + sum);
```

Output:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Sum: 10
```

Nested For Loops

Like all other programming languages, Java allows loops to be nested. That is, one loop may be inside another.

```
// 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();
         }
     }
}
```

Output:

```
* * * * * * * * * *
* * * * * * * * *
* * * * * * * *
* * * * * * *
* * * * * *
* * * * *
* * * *
* **
* *
*
```

Jump Statements (break, continue, return)

Jump statements allow the program to execute in a nonlinear fashion.

Java supports three jump statements: **break, continue, and return**.

break

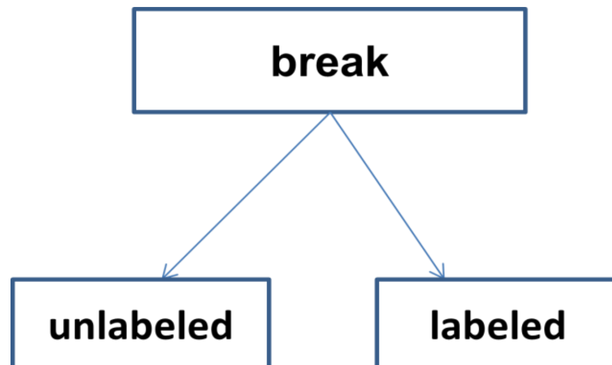
In Java, the break statement has **three** uses.

Unlabeled break

1) It terminates a statement sequence in a **switch** statement.
2) It can be used to **exit a loop**.

Labeled break:

3) It can be used as a "civilized" form of goto.



Unlabeled break : break to Exit a loop

- By using **break**, we can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.
- When a **break** statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.
- The **break** statement can be used with any of Java's loops, including intentionally infinite loops.

Example

```
class BrkEx
    public static void main(String args[]) {
            for(int i=1; i<5;i++) {
                    if(i == 4)
                        break;
                    System.out.println(n);
            }
      }
 }
```

Output

```
1
2
3
```

Labeled break : Form of Goto

- In Java, **break** statement can also be used as **form of Goto**.
- Java does not have a goto statement. **break** gives you the benefits of a **goto** without its problems.

General form

break label;

- ✓ **label** is the name of a label that identifies a block of code.

- A **label** is any valid Java identifier followed by a colon.
- To name a block, put a label at the start of it.
- **Labeled break** causes execution to resume at the end of the **labeled block**.

Example:

```
class Break {
public static void main(String args[])
{
   boolean t = true;

   first:
  {
    second:
  {
     third:
  {
       System.out.println("Before the break.");
       if(t) break second; // break second block
       System.out.println("This won't execute");
   }
   System.out.println("This won't execute");
   }
   System.out.println("This is after second block.");
   }
  }
}
```

Output:

Before the break.
This is after second block.

continue

1) **continue** is used to force an early iteration of the loop.
2) In any iteration, **continue** will stop processing the code that is present after the **continue** and continues the execution from the next iteration.
3) In **while** and **do-while** loops, a **continue** statement causes control to be transferred directly to the conditional expression that controls the loop.
4) In a **for** loop, control goes first to the iteration portion of the for statement and then to the conditional expression.

Example:

```
class Continue
    public static void main(String args[]) {
            for(int i=1; i<5;i++) {
                    if( i % 2 == 0) continue;
                    System.out.print(i);
            }
    }
 }
```

Output

1
3

return

- The **return** statement is used to explicitly return from a method.
- It causes program control to transfer back to the caller of the method.
- The **return** statement immediately terminates the method in which it is executed.

Example

```
// 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.");
    }
}
```

class

class

- **class** is a logical construct which defines shape and nature of an object.
- **class** is a **template** which defines **data** and **code** that acts on that data.
- **class** defines a new data type.
- A **class** is declared by use of the **class** keyword.

The General Form of a Class

```
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
  }
}
```

Instance variables or member variables

- The **data**, or **variables**, defined within a class are called **instance variables**.
- **Instance variables** are called so, because each instance of the class (that is, each object of the class) contains its own copy of these variables.

Members of a class

The **variables** and **methods** defined within a class are called members of the class.

Methods / Instance methods / member methods

The code is contained within methods.

Class Example

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

Declaring Objects (new operator)

- Objects are created with **new** keyword.
- The **new** operator dynamically allocates (at runtime) memory for an object and returns a reference to it. This reference is memory address of object. This reference is stored in a variable.

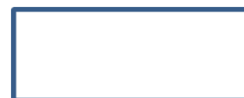
Form

classname object_reference = new classname ( );

```
Box mybox;              // declare reference to object
mybox = new Box();      // allocate a Box object
        OR
Box mybox = new Box();
```

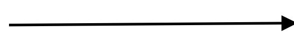
Statement **Effect**

Box mybox;

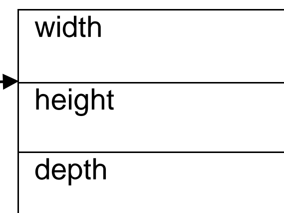


mybox

mybox = new Box();



mybox



Box object

Class vs Object

class	object
Class is a template.	Object is an instance of a class
Class is declared with class keyword.	Object is created with **new** keyword.
Class does not create an actual object.	Object will create an actual object.
Ex: class Test { }	Ex: Test t = new Test();

Methods

This is the general form of a method:

```
type name(parameter-list) {
    // body of method
}
```

- **type** specifies the type of data returned by the method.
- If the method does not return a value, its return type must be **void**.
- Methods that have a return type other than **void** return a value to the calling method using the following form:
- return *value*;

Points about methods

- Method can return values (primitives and objects)
- Method can take parameters (primitives and objects)

```
class Box {
        double width, height, depth;
        // compute and return volume
        double volume() {
                return width * height * depth;
        }
 class BoxDemo
 {
        public static void main(String args[]) {
                Box b1 = new Box();
                Box b2 = new Box();
                double vol;

                // assign values to mybox1's instance variables
                b1.width = 10;
                b1.height = 20;
                b1.depth = 15;
```

```
        /* assign different values to mybox2's instance variables */
        b2.width = 3;
        b2.height = 6;
        b2.depth = 9;

        // get volume of first box
        vol = b1.volume();
        System.out.println("Volume is " + vol);

        // get volume of second box
        vol = b2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

Constructors

1. Constructors are used to initialize objects.
2. A constructor initializes an object immediately upon creation.
3. Constructor has the same name as the class name.
4. Constructor is syntactically similar to a method but it does not contain any return type, including **void.**
5. Constructor is called when the object is created, before the **new** operator completed.

Types of Constructors

1. Default or no-argument constructor
2. Parameterized constructor

1. Default or no-argument constructor

1. The constructor which does not take any parameters is called as **default** or **no-arg constructor**.
2. When we do not explicitly define a constructor for a class, then Java creates a default constructor for the class.
3. The default constructor automatically initializes all instance variables to their default values, which are zero, null, and false, for numeric types, reference types, and boolean, respectively.

/* 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;
 }
}
```

2. Parameterized constructor

1. The constructor which takes parameters is called as **parameterized constructor**.
2. This is useful to initialize different objects with different values.

```
/* 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)
 {
     System.out.println("Constructing Box");
     width = w;
     height = h;
     depth = d;
  }

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

this keyword

this always refers to the currently executing object.

There are two uses with the **this** keyword.

1. To resolve the name collisions between instance variables and local variables
2. To explicitly call a constructor from another constructor

1. Instance Variable Hiding
To resolve the name collisions between instance variables and local variables

a) when a **local variable** has the same name as an **instance variable**, the local variable **hides** the instance variable.
b) To resolve this name conflicts, we use **this.**

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

2. To explicitly call a constructor from another constructor

a) We can use **this** to call a constructor from another constructor. This is called a Explicit constructor invocation.
b) **this()** should be the first statement inside a constructor.

```
class Box {
    double width;
    double height;
    double depth;
 Box()
 {
    System.out.println("Defualt constructor");
 }
  // This is the constructor for Box.
  Box(double w,double h, double d) {
```

```
        // calling default constructor
        this();
        System.out.println("Constructing Box");
        width = 10;
        height = 10;
        depth = 10;
    }
}
```

Garbage Collection

- In Java, objects are allocated memory dynamically using new operator. Butmemory de-allocation happens automatically.
- In Java, Garbage collection is the technique through which the memory de-allocation will happen automatically.

Working of Garbage Collection

a) The job of the Garbage collector is to identify the unused objects and delete themfrom memory.

b) **Garbage collector** checks the objects which do not have any reference anddeletes them from memory.

c) **Garbage collector** runs periodically during the program execution. But there is no guarantee when a Garbage collector runs.

d) The garbage collector runs periodically, checking for objects that are nolonger referenced by any running state or indirectly through other referencedobjects.

e) Garbage collector calls **finalize( )** method just before it runs.

gc() method

1. The gc() method is used to invoke the garbage collector to perform cleanup processing.
2. The gc() is found in **System** and **Runtime** classes.

```
public static void gc(){ }
```

Example

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

```
                Runtime r=Runtime.getRuntime();
                System.out.println("Total Memory: "+r.totalMemory());
                System.out.println("Free Memory: "+r.freeMemory());

                 for(int i=0;i<100;i++){
                        new MemoryTest();
                 }
        System.out.println("After creating 100 instance, Free Memory: "+r.freeMemory());

         System.gc();
         System.out.println("After gc(), Free Memory: "+r.freeMemory());
        }
}
```

finalize( ) method / *finalization*

a) By using **finalize()** method or **finalization**, we can define specific actions that will occur when an object is just about reclaimed by the Garbage collector.

b) To add a **finalizer** to a class, we simply define the **finalize( ) method**. The Java run time calls that method whenever it is about to recycle an object of that class.

c) Inside the **finalize( ) method**, you will specify those actions that must be performed before an object is destroyed.

The finalize( ) method has this general form:

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

Overloading Methods

1. Overloading is the process through which we can define two or more methods within the same class that share the same name but with different parameters (type and/or number).
2. The Overloaded methods must differ in the type and / or number of their parameters.
3. When an overloaded method is invoked, Java uses the type and/or number of arguments as itsguide to determine which version of the overloaded method to actually call.

4. In some cases, Java's **automatic type conversions** can play a role in overload resolution.

```
// Demonstrate method overloading.
class OverloadDemo {
        void test() {
                System.out.println("No parameters");
        }

   // Overload test for one integer parameter.
   void test(int a) {
           System.out.println("a: " + a);
   }

    // Overload test for two integer parameters.
    void test(int a, int b) {
                System.out.println("a and b: " + a + " " + b);
        }

    // Overload test for a double parameter
    double test(double a) {
           System.out.println("double a: " + a);
            return a*a;
    }
}
class Overload {
        public static void main(String args[]) {
                OverloadDemo ob = new OverloadDemo();
                double result;

    // call all versions of test()
    ob.test();
    ob.test(10);
    ob.test(10, 20);
    result = ob.test(123.25);
    System.out.println("Result of ob.test(123.25): " + result);
    }
 }
```

Output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25):
15190.5625
```

Constructor Overloading

We can define two or more constructors with the same name but with different parameters. This process is called **Constructor overloading**.

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

  // This is the constructor for Box.
  Box() {
```

```
  width = -1;
  height = -1;
  depth = -1;  }

  // 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 BoxDemo {
   public static void main(String args[]) {
        // declare, allocate, and initialize Box objects
        Box mybox1 = new Box(10, 20, 15);
        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);
   }
}
```

Argument Passing Techniques

There are two argument passing techniques

1. **Call-by-value**

   Java uses call-by-value when we pass primitive types to a method

2. **Call-by-reference**

   Java uses call-by-reference when we pass object references to a method.

1. Call-by-value

- This approach copies the value of an argument into the formal parameter of the method. Therefore, changes made to the parameter of the method have no effect on the argument.

```
// 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);
   }
}
```

Output:

a and b before call: 15 20
a and b after call: 15 20

2. Call-by-reference

1. In this approach, a reference to an argument (not the value of the argument) is passed to the parameter.
2. 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 method.
3. Although Java uses **call-by-value** to pass all arguments, the precise effect differs between whether a primitive type or a reference type is passed.

```
// Objects are passed through their references.
class Test {
        int a, b;

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

Output:

a and b before call: 15 20
a and b after call: 30 10

```
        /= 2;
        }
}

class PassObjRef {
        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);
        }
}
```

Recursion

- Java supports recursion.
- Recursion is the process of defining something in terms of itself.
- A method that calls itself is said to be recursive.
- When a method calls itself, **new local variables** and **parameters are allocated storage on the stack**, and the method code is executed with these new variables from the start.
- As each recursive call returns, the old local variables and parameters are removed from the stack.
- The **main advantage** to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives.

  Ex: QuickSort and AI related problems

- When writing recursive methods, we must have an **if** statement somewhere to force the method to return without the recursive call being executed.

- **Recursive versions** of many routines may execute a bit more slowly than the iterative equivalent because of the added overhead of the additional method calls.

Example:

Factorial of a Number:

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

                if(n==1) return 1;
                else
                {
                        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));
          }
        }
```

Encapsulation (Access Control)

Encapsulation provides two features:

1. It links data with the code it manipulates.
2. Access control

Access Control (private, public, protected and default access)

- Through encapsulation, we can control what parts of a program can access the members of a class.
- By controlling access, you can prevent misuse.
- Java uses ***access modifier*** to determine how a member can be accessed.
- Java provides three **access modifiers – private, public and protected.**
- Java also provides **default access level.**

private

When a member of a class is specified as **private**, then that member can only be accessed by other members of its class.

public

When a member of a class is specified as **public**, then that member be accessed by any other code.

Protected

protected applies only when **inheritance** is involved.

default access

When **no access modifier** is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.

/* 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());
  }
}
```

Static

Static is used to access a member by itself, without reference to a specific instance. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.

Things can be declared as static

1. Variables
2. Methods
3. Static blocks
4. Nested classes

Things can not be declared as static

1. Local variables
2. classes
3. constructor

static variables (class variables)

- 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.

static methods

Methods declared as static have several restrictions:

- They can only directly call other static methods.
- They can only directly access static data.
- They cannot refer to **this** or **super** in any way.

Static blocks (OR) static initialization block

- Static blocks are used to initialize the static variables.
- Static block gets executed exactly once, when the class is first loaded.

```
// 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);
  }
}
```

Order:

Static variables
Static blocks
Static methods (first main() then other methods)

Output:

Inside the same class, we can access static members directly

Outside the class, we can access the static members,using the format:

Classname.variable
Classname.method( )

Nested and Inner classes

Defining a class inside another class is known as *nested class.*

Nested classes are 4 types in Java :

1. Inner class (Non-static Nested class)
2. Method-Local Inner class
3. Static Nested class
4. Anonymous Inner class

1. 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, including private members
- We can create an instance of an Inner class only through the object of the outer class.

```
// Demonstrate Inner class.
class Outer {
        private int x = 7;

  class Inner{
    void meth() {
          System.out.println("x = " + x);
    }
  }

 public static void main(String args[]) {
          Outer ou = new Outer();
          Outer.Inner in = ou.new Inner();
          in.meth();
 }
}
```

2. Method-Local Inner class

- We can define a nested class inside a method or any block scope (loops).
- We must create the instance of the inner class somewhere inside the method but after the inner class definition.

```
// Demonstrate method-local nested class.
class Outer {

  void display()
  {
              class Nest{
                     void meth() {
```

```
                        System.out.println("Hello");
                    }
                }
                Nest in = new Nest();
            in.meth();
        }

    public static void main(String args[]) {
        Outer ou = new Outer();
        ou.display();

    }
}
```

3. Static Nested class

A static nested class is one that has the **static modifier** applied.

// Demonstrate Static Nested class.

```
class Outer {
    private int x = 7;

  static class Inner{
   void meth() {
        System.out.println("x = " + x);
   }
  }

 public static void main(String args[]) {
    Inner in = new Inner();
    in.meth();
 }
}
```

String class

Java API defines String as a class

Two important things about Strings

1. For every string, Java creates an object of type **String.** Even string constants are actually **String** objects.
2. String objects are **immutable.**

Creation of Strings / Construction of Strings

String are created many ways, but the easiest are two ways :

1) Without **new** operator

   String s = "Java";

2) With **new** operator

   String s = new String("Java");

Difference Between the above two ways (OR) **Important facts about String and Memory**

- When a String is created using without **new** operator, then the string object will be created inside the separate memory called **"String Constant pool".**
- When a String is created using new operator, then the string object will be created both heap memory and inside "String Constant pool". But the heap memory copy will be referred by the reference variable.
- When creating a string in **String constant pool**, JVM first checks the pool to see if there is a similar string exists or not. If exists, no new string will be created, existing one will be reused. **If string does not exist, then JVM will create a new one.**

Immutability:

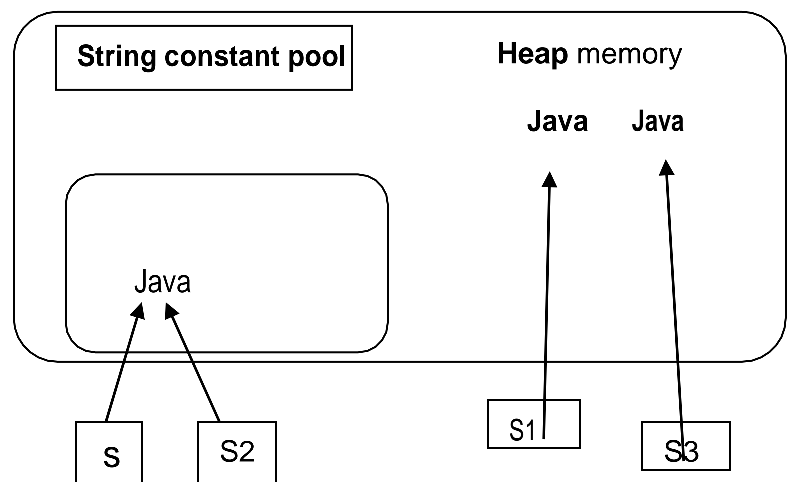
Once a string is created, its contents cannot be altered.

String s = "Java";

String s1 = new String("Java");

String s2 = "Java";

String s3 = new String("Java");



String Methods

Important methods in the String class

1. **charAt()** Returns the character located at the specified index
2. **concat()** Appends one String to the end of another ("+" also works)
3. **equalsIgnoreCase()** Determines the equality of two Strings, ignoring case
4. **length()** Returns the number of characters in a String
5. **replace()** Replaces occurrences of a character with a new character
6. **substring()** Returns a part of a String

7. **toLowerCase()** Returns a String with uppercase characters converted
8. **toString()** Returns the value of a String
9. **toUpperCase()** Returns a String with lowercase characters converted
10. **trim()** Removes whitespace from the ends of a String

1. public char charAt(int index)

String x = “CARGO";
System.out.println(x.charAt(2)); **Output is : R**

2. public String concat(String s)

String x = "taxi";
System.out.println(x.concat(" cab")); **Output is : taxi cab**

+ and += operators

String x = "library";
System.out.println(x + " card"); **Output is : library card**

String x = "Atlantic";
x+= " ocean";
System.out.println(x); **Output is : Atlantic ocean**

3. public boolean equals(Object anObject)

Compares this string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.

String x = "Exit";
System.out.println(x.equals("EXIT")); **Output is: false**

String x = "Exit";
System.out.println(x.equals("Exit")); **Output is: true**

4. public boolean equalsIgnoreCase(String s)

This is similar to equals() method and it will return true even when characters in the String objects being compared have differing cases.

String x = "Exit";
System.out.println(x.equalsIgnoreCase("EXIT")); **Output is: true**

5. public int length()

```
String x = "01234567";
System.out.println( x.length() );
```
Output is: 8

6. public String replace(char old, char new)

```
String x = "oxoxoxox";
System.out.println( x.replace('x', 'X') );
```
Output is: oXoXoXoX

7. public String substring(int begin)

```
String x = "0123456789";
System.out.println( x.substring(5) );
```
Output is: 56789

8. public String substring(int begin, int end)

```
String x = "0123456789";

System.out.println( x.substring(5, 8));
```
Output is: 567

9. public String toLowerCase()

```
String x = "A New Moon";
System.out.println( x.toLowerCase() );
```
Output is: a new moon

10. public String toUpperCase()

```
String x = "A New Moon";
System.out.println( x.toUpperCase() );
```
Output is: A NEW MOON

11. public String trim()

```
String x = "      hi        ";
System.out.println( x.trim());
```
Output is: hi

12. public String toString()

All objects in Java must have a toString() method, which typically returns a String that in some meaningful way describes the object in question.

```
String x = "big surprise";
System.out.println( x.toString() );
```
Output is: big surprise

Write a Java program to demonstrate String class and its methods

```
// Demonstrating some String methods.
class StringDemo {
	public static void main(String args[]) {
		String strOb1 = "First String";
		String strOb2 = "Second String";
		String strOb3 = strOb1;

		System.out.println("Length of strOb1: " +strOb1.length());

		System.out.println("Char at index 3 in strOb1: " +strOb1.charAt(3));

		if(strOb1.equals(strOb2))
			System.out.println("strOb1 == strOb2");
		else
			System.out.println("strOb1 != strOb2");


		if(strOb1.equals(strOb3))
			System.out.println("strOb1 == strOb3");
		else
			System.out.println("strOb1 != strOb3");
	}
}
```

String arrays

We can have arrays of strings, just like we can have arrays of any other type of object.

```
// Demonstrate String arrays.
class StringDemo {
	public static void main(String args[]) {
		String str[] = { "one", "two", "three" };

		for(int i=0; i<str.length; i++)
			System.out.println(str[i]);
	}
}
```

Output:

one
two
three

Command-Line Arguments

1) By using, command-line arguments, we can pass information into a program when we run it.

2) A command-line argument is the information that directly follows the program's name on the command line **when it is executed.**

3) A Java application can accept any number of arguments from the command line.

4) Command Line arguments are stored as strings in a **String array passed to the args parameter of main( ).**

5) The first command-line argument is stored at args[0], the second at args[1], and so on.

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 10 -20
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

Varargs (OR) Variable-Length arguments

1) In Java, as of JDK 5.0, It is possible to create methods that can take a variable number of arguments.
2) A method that takes a variable number of arguments is called a **variable-aritymethod**, or simply a **varargs method**.
3) A variable-length argument is specified by **three periods (...)**.

Example:

```
void display(int ... v) {  }
```

This syntax tells the compiler that display( ) can be called with **zero** or **more arguments**. As a result, v is **implicitly declared as an array of type** int[ ].

The declaration rules for var-args

a) We can write normal parameters in a method that uses a var-arg parameter, but the var-arg parameter must come at the last in the list.

b) **Var-args limits:** There must be only one var-args in a method

Valid var-args	**Invalid var-args**
void display(int... x) { }	void display(int x...) { }
void display(char c, int... x) { }	void display(int... x, char... y) { }
void display(Box... b) { }	void display(String... s, byte b) { }

// Demonstrate variable-length arguments.

```
class VarArgs {

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

    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
        }
}
```

Overloading Vararg Methods

- We can overload a method that takes a variable-length argument.
- A **varargs method** can also be overloaded by **a non-varargs method.**

// Varargs and overloading.

```
class VarArgs {

  static void vaTest(int ... v) {
          System.out.print("Number of args: " + v.length);
  }

  static void vaTest(boolean ... v) {
```

```
        System.out.print("Number of args: " + v.length);

 }
 static void vaTest(String msg, int ... v) {
                System.out.print("Number of args: " + v.length);
 }

        public static void main(String args[]) {
                        vaTest(1, 2, 3);
                        vaTest("Testing: ", 10, 20);
                        vaTest(true, false, false);
        }
}
```

Varargs and Ambiguity

- Somewhat unexpected errors can result when overloading a method that takes a variable length argument.
- These errors involve ambiguity because it is possible to create an ambiguous call to an overloaded var-args method.

Ambiguity Error 1

```
// Varargs, overloading, and ambiguity.
class VarArgs {

  static void vaTest(int ... v) {
      System.out.print("Number of args: " + v.length);
  }

  static void vaTest(boolean ... v) {
      System.out.print("Number of args: " + v.length);
  }

  public static void main(String args[]) {
          vaTest(1, 2, 3); // OK
          vaTest(true, false, false); // OK

          vaTest(); // Error: Ambiguous!
  }
}
```

Ambiguity Error 2

```
// Varargs, overloading, and ambiguity.
class VarArgs {

  static void vaTest(int ... v) {
       System.out.print("Number of args: " + v.length);
  }

  static void vaTest(int n, int ... v1) {
       System.out.print("Number of args: " + v1.length);
  }

  public static void main(String args[])  {

                vaTest(10); // Error: Ambiguous!
  }
}
```

final keyword

final has 3 uses in Java.

1. To define the constants
2. To prevent Inheritance
3. TO prevent Method overriding

To define constants:

Once a constant is defined, it's value can not be modified.

Ex:

```
final int PI = 3.14;
```

Object Oriented Programming using JAVA

UNIT III

Inheritance (IS-A)

Definition:

Inheritance is the process by which one object acquires the properties of another object.

- Using inheritance, we can create **a general class** that defines characteristics 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.

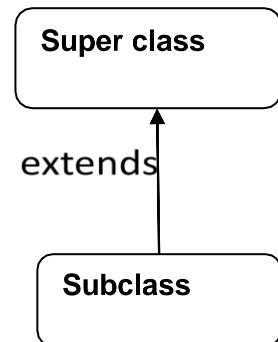
Super class / Base class / Parent class

The class from which a subclass is derived is called the *superclass*.

Subclass / Derived class / Extended class / Child class

A class that is derived from another class is called a subclass.

A ***subclass*** inherits state(variables) and behavior (method) from all of its ancestors.



Note:

In the Java programming language, each class is allowed to have one direct *superclass*, and each superclass has the potential for an unlimited number of *subclasses*.

Creating subclasses / extends keyword

To create a subclass of another class use the **extends** clause in the class declaration.

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

Benefits of Inheritance

1. To promote code reuse
2. To use polymorphism

IS-A relationship

Inheritance represents **IS-A** relationship. *Inheritance* defines **is-a** relationship between a **Super class** and its **Sub class**.

Example

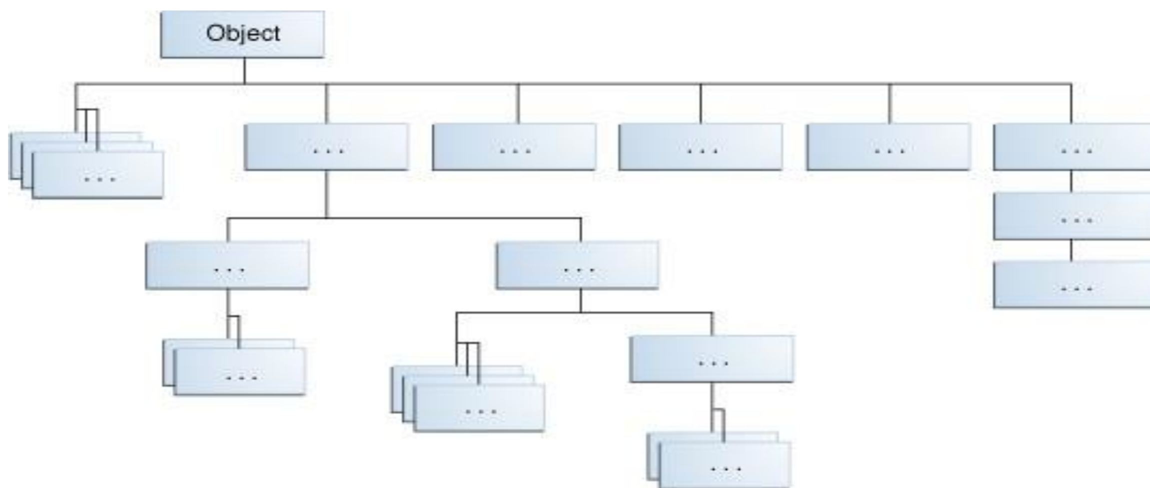
```
class Vehicle
{
    // fields of Vehicle
}
class Car extends Vehicle
{
    // new fields and methods of Car.
}
```

In OOPs term we can say that,

- **Vehicle** is super class of **Car**.
- **Car** is sub class of **Vehicle**.
- Car **IS-A** Vehicle.

The Object class

- The Object class is the superclass to all the classes in Java platform and every class that we create.
- The Object class, defined in the **java.lang** package, defines and implements behavior common to all classes.



Methods in Object class

clone() -> Creates and returns a copy of this object.
equals(Object obj) -> Indicates whether some other object is "equal to" this one.
finalize() -> Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
getClass() -> Returns the runtime class of this Object.
toString() -> Returns a string representation of the object.
hashCode() -> Returns a hash code value for the object.

A Simple Example of Single Inheritance

```
class Parent
{
        int i;
        void showi()
        {
                System.out.println("i="+i);
        }
}

class Child extends Parent
{
        int k;
        void showk()
        {
                System.out.println("k="+k);
        }
        void sum()
        {
                System.out.println("i+k="+(i+k));
        }
}
class InheritanceDemo
{
        public static void main(String a[]) {
                Parent p = new Parent();
                System.out.println("Parent class");
                p.i = 10;
                p.showi();

                Child ch = new Child();
                System.out.println("Child class");
                ch.i = 20;
                ch.k = 40;
                ch.showi();
                ch.showk();
                ch.sum();
        }
}
```

Output:

```
Parent class
i=10
Child class
i=20
k=40
i+k=60
```

Note:

- A subclass can inherits all the members of the super class except **private members.**
- We can use the inherited members as is, replace them, hide them, or supplement them with new members.

What We Can Do in a Subclass

1) The inherited fields can be used directly, just like any other fields.
2) we can declare a field in the subclass with the same name as the one in the superclass, - ***hiding***
3) We can declare new fields in the subclass that are not in the superclass.
4) The inherited methods can be used directly as they are.
5) We can write a new *instance* method in the subclass that has the same signature as the one in the superclass – ***Method overriding***.

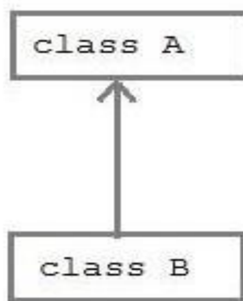
6) We can declare new methods in the subclass that are not in the superclass.
7) We can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword ***super***.

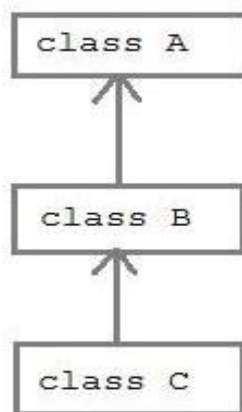
Types of Inheritance

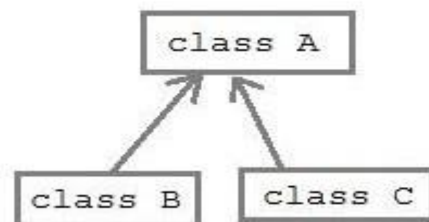
1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance

NOTE :

Multiple inheritance is not supported in java







Hierarchical inheritance

Simple Inheritance Example

```
class A
{

}
class B extends A
{

}
```

Multilevel Inheritance Example

```
class A
{
}
class B extends A
{
}
class C extends B
{
}
```

Hierarchical Inheritance Example

```
class A
{
}
class B extends A
{
}
class C extends A
{
}
```

MultiLevel Inheritance

In a class hierarchy, constructors are executed in the order of derivation, from superclass to subclass.

Write a Java program to demonstrate multilevel inheritance.

```
class A
{
        A()
        {
                System.out.println("A constructor");
        }
        void printA()
        {
                System.out.println("A");
        }
}
class B extends A
{
        B()
        {
                System.out.println("B constructor");
        }
        void printB()
        {
                System.out.println("B");
        }
}

class C extends B
{
        C()
        {
                System.out.println("C constructor");
```

```
        }
        void printC()
        {
                System.out.println("C");
        }
}

class MLInheritance
{
        public static void main(String a[])
        {
                C c = new C();
                c.printA();
                c.printB();
                c.printC();
        }
}
```

Output:

```
A constructor
B constructor
C constructor
A
B
C
```

Method Overriding

Method Overriding

Declaring an instance method in **subclass** which is already present in **parent class** is known as method overriding.

If a subclass defines an instance method with the same signature as an instance method in the superclass, then the method in the subclass ***overrides*** the one in the superclass.

The method in the subclass is called as ***overriding method.*** The method in the superclass is called as ***overridden method.***

Method overriding is also referred to as runtime polymorphism.

Rules of method overriding in Java

a) **Argument list:** The argument list must exactly match that of the overridden method.
b) **Return Type:** The return type must be the same as, or a subtype (***covariant return type)*** of, the return type declared in the original overridden method in the superclass.
c) Instance methods can be overridden only if they are inherited by the subclass.

Method Overridng Example

```
class Animal {
   void eat() {
      System.out.println("Animal eating");
   }
}
class Lion extends Animal {
```

```
    void eat() {
       System.out.println("Lion eating");
    }
}
public class Override
{
    public static void main(String[] args)
    {
                                Animal t = new Animal();
                                t.eat();

                                Lion l = new Lion();
                                l.eat();
    }
}
```

Output:

Animal eating

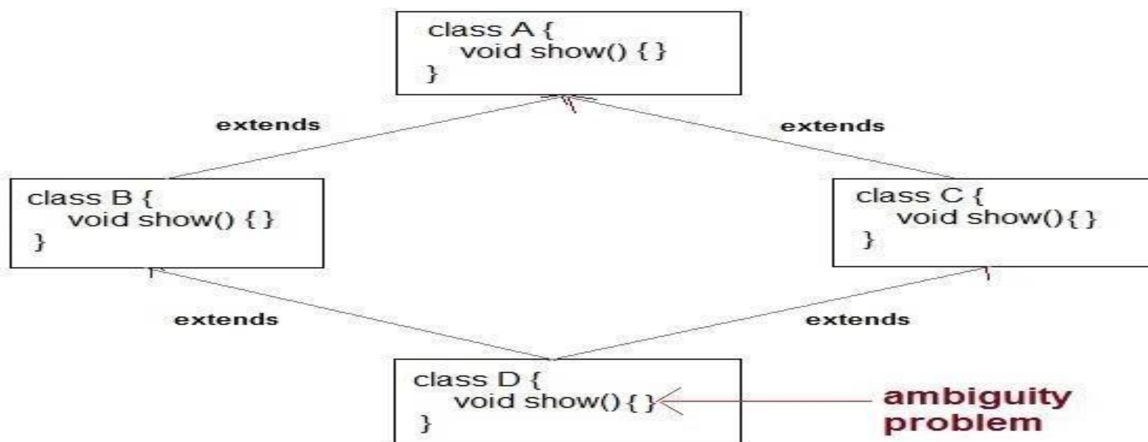
Lion eating

Why multiple inheritance is not supported in Java

- To remove ambiguity (OR) Deadly Diamond of Death problem.

Deadly Diamond of Death problem or Diamond problem

- The problem is, if a class extended two other classes, and both super classes have a show() method, then which version of show() method would the subclass inherit?. This is an ambiguity.
- This issue can lead to a scenario known as the "Deadly Diamond of Death," because of the shape of the class diagram that can be created in a multiple inheritance design.
- The diamond is formed when classes B and C both extend A, and both B and C inherit a method from A. If class D extends both B and C, and both B and C have the show() the method from A.
- Drawn as a class diagram, the shape of the four classes looks like a **diamond**.



“super” keyword

Super keyword is used for two purposes:

a) To call a superclass constructor from a subclass
b) To access members of a superclass that has been hidden by the members of a subclass (to overcome name hiding)

a) To call a superclass constructor from a subclass

i. A superclass constructor can be called by a subclass constructor by using the following form of super:

super(arg-list);

➔ Here, **arg-list** specifies any arguments needed by the constructor in the superclass.

ii. **super()** must be the first statement to be executed inside the subclass constructor.

Example

```
class A
{
	A(int p)
	{
		System.out.println("Super class constructor");
	}
}
class B extends A
{
	B(int q)
	{
		super(10);
		System.out.println("Sub class constructor");
	}
}
class SuperDemo1
{
	public static void main(String ar[])
	{
		B b = new B(20);
	}
}
```

Output:

```
Super class constructor
Sub class constructor
```

b) To overcome name hiding

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.

super.*member*

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

Example

```
class A   {

        int i=10;

        void print()
        {
                System.out.println("Super class print");
        }
}
class B extends A
{
        int i=20; // this hides the i in A
        void print()
        {
                super.print();
                System.out.println("Sub class print");
                System.out.println("Super class i="+super.i);
                System.out.println("Sub class i="+i);
        }
}
class SuperDemo
{
        public static void main(String ar[])
        {
                B b = new B();
                b.print();
        }
}
```

Output:

```
Super class print
Sub class print
Super class i=10
Sub class i=20
```

Polymorphism

Polymorphism is two types:

1. Compile time polymorphism (static polymorphism)
2. Run time polymorphism (dynamic polymorphism / dynamic method dispatch / virtual method invocation)

Java’s principle in Polymorphism - A Superclass Variable Can Reference a Subclass Object.

Java uses the following principle in achieving polymorphism:

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.

1. Compile time polymorphism (static polymorphism)

- ***a)*** Java achieves compile time polymorphism using ***Method Overloading.***
- ***b)*** In this, method resolution will happen at compile-time.
- ***c)*** **In this, compiler looks only at the *reference type (not at the object type)* and determines which Overloaded method to call.**

Example

```
// Compile time polymorphism or Static polymorphism
class Animal {
        void eat() {
                System.out.println("Animal eating");
        }
}
class Lion extends Animal {

         void eat(int b) {
                System.out.println("Overloading - Lion eating");
        }
}
class RTPoly {
   public static void main(String args[]) {
        Animal a = new Animal(); // object of type Animal
        Lion l = new Lion(); // object of type Lion

        Animal r; // obtain a reference of type Animal

        r = a; // r refers to an Animal object
        r.eat(); // calls Animal's version of eat

        r = l; // r refers to a Lion object
        r.eat();
        //r.eat(10); // calls Animal's version of eat(int) – compiler error
   }
}
```

Output:

```
Animal eating
Animal eating
```

2. Run time polymorphism

(Dynamic polymorphism / Dynamic Method dispatch / Virtual Method invocation)

1) Java achieves run time polymorphism using ***Method Overriding.***
2) In this,method resolution will happen at run time based on actual object.
3) **Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.**
4) When a method is overridden, the child version of the method always executes at runtime.

Example

```
// Dynamic Method Dispatch or Run time polymorphism
// or Virtual Method Invocation
class Animal {
        void eat() {
                System.out.println("Animal eating");
        }
}
class Lion extends Animal {
        // override eat()
        void eat() {
                System.out.println("Lion eating");
        }
}
class RTPDemo {
  public static void main(String args[]) {
        Animal a = new Animal(); // object of type Animal
        Lion l = new Lion(); // object of type Lion

        Animal r; // obtain a reference of type Animal

        r = a; // r refers to an Animal object
        r.eat(); // calls Animal's version of eat

        r = l; // r refers to a Lion object
        r.eat(); // calls Lionss version of eat
  }
}
```

Output:

Animal eating
Lion eating

final keyword: Three uses with final keyword:

1. To define constants
2. To prevent Method Overriding
3. To prevent Inheritance

2 & 3 are used with Inheritance

To Prevent Method Overriding

By using **final**, we can prevent a method from being overridden. 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!");
	}
 }
```

Because meth( ) is declared as final, it cannot be overridden in B. If we attempt to do so, a compile-time error will result.

To Prevent Inheritance

To prevent a class from being inherited, declare the class as **final.** Declaring a class as **final** implicitly declares all of its methods as **final**, too.

```
final class A
{
	//...
}

// The following class is illegal.
class B extends A
{
	// ERROR! Can't subclass A
	//...
}
```

Abstract class

An abstract class is a class that cannot be instantiated.

An abstract class contains one or more abstract methods.

Abstract method

Abstract method is a method which do not contain it's body. In other words, abstract method is a method without any implementation.

Abstract method syntax:

abstract type name(parameter-list);

- Abstract methods are sometimes referred to as subclasser responsibility, because they have no implementation specified in the superclass. So, subclass must override them.
- It is not possible to 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 declared abstract itself.

Example:

Write a Java program to demonstrate abstract methods and abstract 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());
	}
}
```

Differences Between Method Overloading and Method Overriding

	Overloaded Methods	**Overridden Methods**
Argument(s)	Must change.	Must not change.
Return type	Can change.	Can't change except for covariant returns.
Exceptions	Can change.	Can reduce or eliminate. Must not throw new or broader checked exceptions
Access	Can change.	Must not make more restrictive (can be less restrictive)
Invocation	Reference type determines which overloaded version (based on declared argument types) is selected. Happens at compile time.	Object type (in other words, the type of the actual instance on the heap) determines which method is selected. Happens at runtime.

Packages

A *package* is a grouping of related classes and interfaces.

Benefits of packages:

Packages provides -

a) **access protection**
   - ✓ We can allow types within the package to have unrestricted access to one another yet still restrict access for types outside the package.

b) **Name space management**.
   - ✓ The names of one package types won't conflict with the types in other packages.

c) Two different programmers can easily determine the types defined in the package are related.

Defining / Creating a Package

To create a package, we choose a name for the package and put a package statement with that name at the top of *every source file*.

General form of a package statement:

package pkg; -> Here, pkg is the name of the package.

- ✓ The package statement (for example, **package graphics;**) must be the first line in the source file.
- ✓ There can be only one package statement in each source file, and it applies to all types in the file.
- ✓ Java uses **file system directories** to store packages.
- ✓ Ex: **package MyPackage;**
  - ➢ the **.class** files for any classes that are part of **MyPackage** must be stored in a directory called **MyPackage**.

Multilevel Packages

We 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 is shown here:

package pkg1[.pkg2[.pkg3]];

Ex: package java.lang.system;

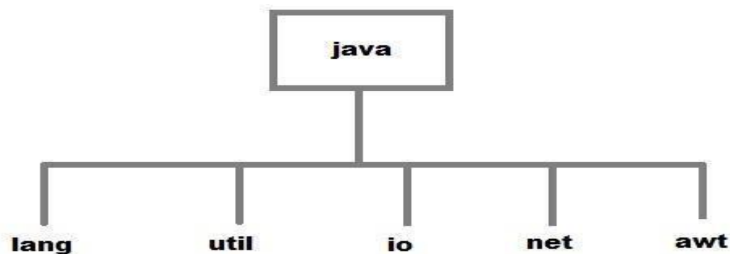
This needs to be stored in java\lang\system directory in windows environment.

Package categories

Two forms:

Built-in package:

- 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.**



User-defined package:

User defined packages are the packages defined by the programmers specific to their application.

Ex: package edu.cbit.example;

Access Protection in Java using Packages

In addition to the access modifiers (private,public,protected and default), **packages** will add another dimension to access control.

Java addresses four categories of visibility for class members:

1. Subclasses in the same package
2. Non-subclasses in the same package
3. Subclasses in different packages
4. Classes that are neither in the same package nor subclasses

	private	No Modifier (default)	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

Example: Write a Java program to demonstrate Access Protection in java using Packages.

```
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);
  }
}

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);
        }
```

```
}

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);
	}
}


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);
	}
}

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);
        }
}
```

CLASSPATH

The **CLASSPATH** refers to the path on your file system where your **.class** files are saved, and the **classpath** is defined by the **CLASSPATH environment variable**.

The **CLASSPATH** environment variable specifies the directories where you want the compiler and the JVM to search for bytecode.

Finding Packages

How does the Java run-time system know where to look for packages that we create?

1. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if our package is in a subdirectory of the current directory, it will be found.
2. We can specify a directory path or paths by setting the **CLASSPATH** environmental variable.
3. we can use the **-classpath** option with java and javac to specify the path to your classes.

Using the - classpath flag overrides the CLASSPATH environment variable.

Importing Packages

The types that comprise a package are known as the ***package members***.

To use a public package member from outside its package, we must do one of the following:

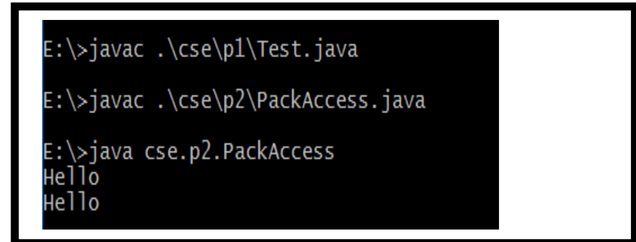
1. Refer to the member by its **fully qualified name**
2. **Import** the package member
3. **Import** the member's entire package

1. Referring to a Package Member by Its Qualified Name

In this, package name will be included to access the member of a package outside its package.

Test.java

```
package cse.p1;
public class Test
{
        public void print()
        {
                System.out.println("Hello");
        }
}
```



PackAccess.java

```
package cse.p2;
class B extends cse.p1.Test
{

}
class PackAccess
{
        public static void main(String a[])
        {
                cse.p1.Test t = new cse.p1.Test();
                t.print();

                B b = new B();
                b.print();
        }
}
```

2. Import statement

In a Java source file, **import** statements occur immediately following the package statement (if it exists) and before any class definitions.

import pkg1[.pkg2].(classname|*);

Ex:

import java.util.Date;   import java.io.*

importing package member

Test.java

```
package cse.p1;
public class Test
{
        public void print()
        {
```

```
		System.out.println("Hello");
	}
}
```

PackAccess.java (Importing package member)

```
package cse.p2;
import cse.p1.Test;
class B extends Test
{

}
class PackAccess
{
	public static void main(String a[])
	{
		Test t = new Test();
		t.print();

		B b = new B();
		b.print();
	}
}
```

```
E:\>javac .\cse\p1\Test.java

E:\>javac .\cse\p2\PackAccess.java

E:\>java cse.p2.PackAccess
Hello
Hello
```

importing Entire package

```
package cse.p2;

import cse.p1.*;
```

Interfaces

Interfaces are like a 100-percent abstract superclass. An interface is a contract.

Definition: An *interface* is a reference type, similar to a class, that can contain *only* constants, method signatures, default methods, static methods.

- Using interface, we can specify what a class must do, but not how it does it.
- Method bodies exist only for default methods and static methods.
- Interfaces cannot be instantiated—they can only be *implemented* by classes or *extended* by other interfaces.
- Using interfaces, Java achieves the **run time polymorphism** - “one interface, multiple methods” aspect of polymorphism.

Defining an Interface

```
access interface name {
	// constants
	type final-varname = value;
```

```
        //abstract methods
        return-type method-name(parameter-list);
        //default methods
        default type method-name(parameter-list)
        {
                //body
        }
        //static methods
        static type method-name(parameter-list)
        {
                //body
        }
}
```

1. access can be only default or public.

2. Variables defines inside an interface are implicitly public, final and static, they must also be initialized.

3. All methods and variables are implicitly public.

Example:

```
public interface IntStack
{
        void push(int n);
        int pop();
}
```

Implementing Interfaces

Once an interface has been defined, one or more classes can implement that interface.

implements

```
class classname [extends superclass] [implements interface [,interface...]]
{
        // class-body
}
```

Note: When we implement an interface method, it must be declared as public.

Example:

```
public interface IntStack
{
        void push(int n);
        int pop();
}

class FixedStack implements IntStack
{
        public void push(int n) { }
        public int pop() { }
}
```

Interfaces properties

1) Interfaces are implicitly abstract.
2) All interface methods are implicitly public and abstract.
3) All variables defined in an interface must be public, static, and final
4) Because interface methods are abstract, they cannot be marked final.
5) An interface can extend one or more other interfaces.
6) An interface can't extend anything except another interface.
7) An interface cannot implement another interface or class.
8) An interface must be declared with the keyword interface.
9) Interface types can be used polymorphically .

Runtime polymorphism with interfaces

Interface is another mechanism Java achieves polymorphism using the below **principle**. **An interface reference variable can access object of any class that implements the interface**.

Ex: Java program to demonstrate runtime polymorphism using interfaces through Stack example.

```
// Define an integer stack interface.
interface IntStack {
        void push(int item); // store an item
        int pop(); // retrieve an item
}
// 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--];
}
}

// 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 RTPInterfaces {
            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());
    }
```

```
}
```

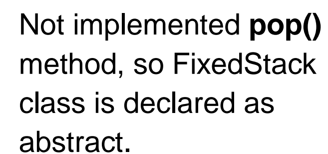
Variables in Interfaces

```
interface SharedConstants {
        int NO = 0;
        int YES = 1;
        int MAYBE = 2;
        int LATER = 3;
        int SOON = 4;
        int NEVER = 5;
}
```

Partial Implementations

If a class implements an interface but does not fully implement the methods required by that interface, then that class must be declared as **abstract.**

```
abstract class FixedStack implements IntStack {
    public void push(int n) {
    }
    //…
}
```



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 default, public, private, or protected.

```
// 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");
    }
}
```

Extending Interfaces

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 required by the interface inheritance chain.

// 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();
        }
}
```

Default Interface Methods (extension method)

This is the new feature added in JDK8.

Reason for adding Default method

1. To expand the capabilities of an interface
2. To define optional methods

```
interface IntStack {
        void push(int item); // store an item
        int pop(); // retrieve an item

         // Because clear( ) has a default, it need not be implemented by a preexisting class that
        // uses IntStack.
        default void clear() {
                System.out.println("clear() not implemented.");
        }
}

Class FixedStack implements IntStack {
        public int pop()
        {
        }
        public void push(int n)
        {
        }
}
```

It is optional to implement **default method – clear()**

Multiple Inheritance Issues with Interfaces

Assume, we have a class that implements two interfaces. If each of these interfaces provides default methods, then some behavior is inherited from both. Thus, to a limited extent, **default methods do support multiple inheritance of behavior**.

For example, assume that two interfaces called **Alpha** and **Beta** are implemented by a class called MyClass.

1. What happens if both **Alpha** and **Beta** provide a method called **reset( )** for which both declare a default implementation?

2. Beta extends Alpha. Then Which version of the default method is used?

3. If MyClass provides its own implementation of the method?

To handle the above situations, Java defines a set of rules that resolves such conflicts.

Rule 1: In all cases, a class implementation takes priority over an interface default implementation.

Rule 2: In cases in which a class implements two interfaces that both have the same default method, but the class does not override that method, then an error will result.

Rule 3: In cases in which one interface inherits another, with both defining a common default method, the inheriting interface's version of the method takes precedence.

If Beta extends Alpha, then Beta's version of reset( ) will be used.

super keyword in interfaces

It is possible to explicitly refer to a default implementation in an inherited interface by using a new form of super.

Its general form is shown here:

```
interfaceName.super.methodName( )
```

For example, if Beta wants to refer to Alpha's default for reset( ), it can use this statement:

```
Alpha.super.reset();
```

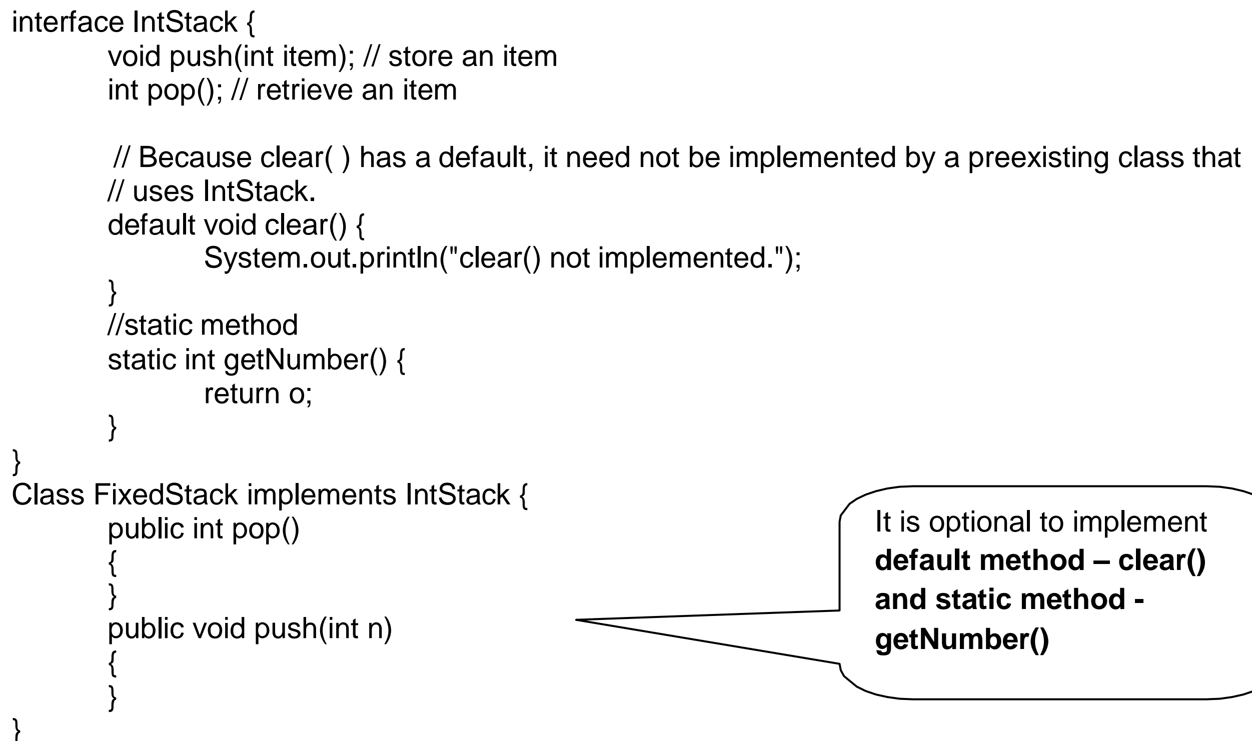
Static methods in an Interface

JDK 8 added another new capability to interface: **the ability to define one or more static methods.**

Like static methods in a class, a static method defined by an interface can be called independently of any object.

A static method is called by specifying the interface name, followed by a period, followed by the method name.

InterfaceName.staticMethodName



```
interface IntStack {
	void push(int item); // store an item
	int pop(); // retrieve an item

	 // Because clear( ) has a default, it need not be implemented by a preexisting class that
	// uses IntStack.
	default void clear() {
		System.out.println("clear() not implemented.");
	}
	//static method
	static int getNumber() {
		return o;
	}
}
Class FixedStack implements IntStack {
	public int pop()
	{
	}
	public void push(int n)
	{
	}
}
```


static int getNumber() method can be called as below:

int defNum = IntStack.getNumber();

Note: static interface methods are not inherited by either an implementing class or a sub-interface.

Abstract class vs interface

Abstract class contains abstract and non-abstract methods.	**Interfaces do not contain non-abstract methods but it contains only abstract methods**
Abstract methods contains instance variables and constants.	Interfaces do not contain instance variables, it contains only constants.
Abstract methods needs to be declared as abstract explicitly.	All methods are implicitly public and abstract.
Class needs to be declared abstract explicitly.	Interface is abstract implicitly.
Abstract class can extends another class, not interfaces	An interface can extends another interfaces only, not classes
An abstract class can implement an interface	An interface cannot implement another interface or class.
Ex: abstract class A{ }	Ex: interface Bounceable

Exception Handling

1. The Java programming language uses *exceptions* to handle errors and other exceptional events.
2. An exception is an abnormal condition that arises in a code sequence at run time.
3. In other words, an exception is a **run-time error.**
4. In Java, an exception is an object.
5. Exceptions can be generated by
   - the Java run-time system, or
   - they can be manually generated by our code.

6. Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.

General form of an exception-handling block

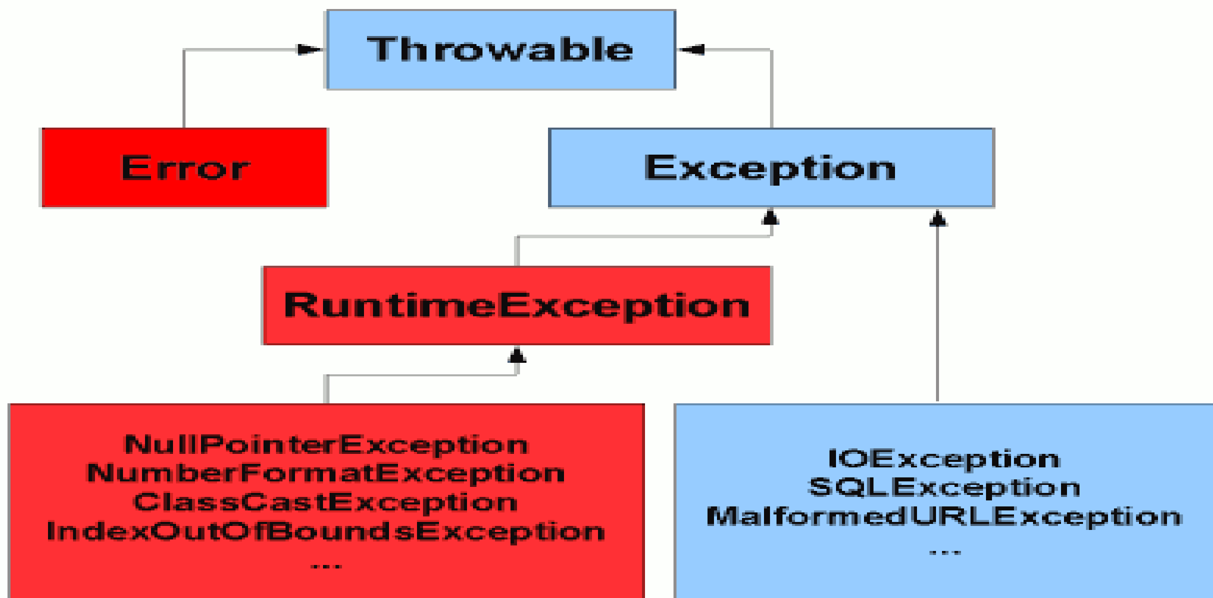
```
try {
   // block of code to monitor for errors
 }
 catch (ExceptionType1 exOb) {
   // exception handler for ExceptionType1
 }
  catch (ExceptionType2 exOb) {
    // exception handler for ExceptionType2
  }
 // ...
finally {
    // block of code to be executed after try block ends
    // cleanup code
}
```

Points:

1) Each try statement requires at least one catch or a finally clause..
2) Multiple catch statements can be associated with a single try block
3) finally block is optional. But, finally block always executes after try/catch block completes.

Exception Types / Exception Hierarchy

- All exceptions are subclasses of **Throwable** class.
- **Throwable** class has two sub-types: 1) **Error** and 2) **Exception**
- **Exception** class has a subclass called – **RuntimeException**
- **User-defined exceptions** can be created by **extending Exception class.**



Exception Types (Built-in Exceptions)

There are two types of exceptions:

1) Checked exceptions
2) Unchecked exceptions

1) Checked Exceptions

- These are exceptional conditions that a well-written application should anticipate and recover from.
- Checked exceptions *must follow the requirement* - **Catch** or **declare the exception using throws**.

Ex: java.io.FileNotFoundException.

- All exceptions are checked exceptions, except for those indicated by Error, RuntimeException, and their subclasses.

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.
ReflectiveOperationException	Superclass of reflection-related exceptions

2) Unchecked Exceptions

- **Errors** and **runtime exceptions** are collectively known as *unchecked exceptions*.
- Unchecked exceptions need not *follow the requirement* - **Catch** or **declare the exception using throws**.

Error:

- These are exceptional conditions that are **external to the application**, and that the application usually cannot anticipate or recover from.

Ex: An application successfully opens a file for input, but is unable to read the file because of a hardware or system malfunction.

Runtime Exception:

- These are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from.
- These usually indicate programming bugs, such as logic errors or improper use of an API.

Ex: NullPointerException

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.
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.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.

Methods in Throwable

Throwable fillInStackTrace( )	Returns a Throwable object that contains a completed stack trace. This object can be rethrown.
String getMessage( )	Returns a description of the exception.
void printStackTrace( )	Displays the stack trace.
String toString( )	Returns a String object containing a description of the exception. This method is called by println( ) when outputting a Throwable object.

Propagating Uncaught Exceptions – call stack

```
class ExceptionTest1 {
    public static void main (String [] args) {
            doStuff();
    }
    static void doStuff()
    {
            doMoreStuff();
    }
    static void doMoreStuff()
```

```
	{
		int x = 5/0;
	}
}
```

> ➔ Unhandled or uncaught exceptions will be handled by default handler provided by JVM.
>
> **Call stack:**
>
> Call stack is the sequence of methods which led to the error

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at ExceptionTest1.doMoreStuff(ExceptionTest1.java:11)
	at ExceptionTest1.doStuff(ExceptionTest1.java:7)
	at ExceptionTest1.main(ExceptionTest1.java:3)
```

CALL STACK

Example for Exception Handling using try/catch/finally

```
class ExcExample{
	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);
		}
		finally
		{
			System.out.println("In finally block.");
		}

		System.out.println("After try/catch/finally blocks.");
	}
}
```

Nested Try

- A try statement can be inside the block of another try. This is called as Nested Try statement.

- If an inner try statement does not have a catch handler for a particular exception, 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.

// 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);
            }
    }
}
```

throw

- It is possible for the program to throw an exception explicitly, using the **throw** statement.
- The general form of throw is shown here:

throw ThrowableInstance;

- **Here, ThrowableInstance** must be an object of type Throwable or a subclass of Throwable.
- The flow of execution stops immediately after the throw statement; any subsequent statements are not executed.

```
// Demonstrate throw.
class ThrowDemo {
    static void demoproc() {
        try {
            throw new NullPointerException("demo");
        } catch(NullPointerException e) {
            System.out.println("Caught inside demoproc.");
        }
    }
    public static void main(String args[]) {
        try {
            demoproc();
        } catch(NullPointerException e) {
            System.out.println("Recaught: " + e);
        }
    }
}
```

throws clause

If a method is causing an exception but it does not handle then it must specify it's behavior so that callers of the method can guard themselves against that exception.

throws clause is used to specify it's behavior.

```
type method-name(parameter-list) throws exception-list
{
        // body of method
}
```

//Example for throws demo

```
class ThrowsDemo {
        static void demoproc() throws IllegalAccessException{
                        throw new IllegalAccessException("demo");
        }
  public static void main(String args[]) throws IllegalAccessException
  {
                demoproc();
  }
}
```

throw vs throws

throw	throws
Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception
Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
Throw is followed by an instance.	Throws is followed by class.
Throw is used within the method.	Throws is used with the method signature
You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException

finally

1) **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.
2) The **finally** block will execute whether or not an exception is thrown.
3) If an exception is thrown, the **finally** block will execute even if no catch statement matches the exception.

4) 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.

// Demonstrate finally

```
class FinallyDemo {
    // Throw 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();
    }
}
```

Output:

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

Custom / User-defined Exception (Creating our Own Exception Subclasses)

To create a custom exception, program just need to define a class that extends **Exception** class.

```
// This program creates a custom exception type.
class MyException extends Exception {
        public String toString() {
                return "MyException - Custom Exception";
        }
}
class CustomExcDemo {
        static void compute(int a) throws MyException {
                System.out.println("Called compute(" + a + ")");
                        if(a > 10)
                                throw new MyException();
                System.out.println("Normal exit");
        }

  public static void main(String args[]) {
                try {
                        compute(1);
                        compute(20);
                }
                catch (MyException e) {
                        System.out.println("Caught " + e);
                }
        }
}
```

Output:

```
Called compute(1)
Normal exit
Called compute(20)
Caught MyException - Custom
Exception
```

Chained Exceptions

- The chained exception feature allows us to associate another exception with an exception.
- To allow chained exceptions, two constructors and two methods were added to **Throwable**.

The constructors are shown here:

Throwable(Throwable causeExc)

Throwable(String msg, Throwable causeExc):

The chained exception methods supported by Throwable are:

getCause( ) and

initCause( ).

// Example for Chained Exceptions

```
import java.io.*;
class ChianedExceptionDemo {
        static void compute()
        {
                ArithmeticException ae = new ArithmeticException();
                 //add a cause
                 ae.initCause(new IOException("Original cause"));
                 throw ae;
        }
        public static void main(String args[])
        {
                try
                {
                        compute();
                }
                catch(Exception e)
                {
                        System.out.println("Caught:"+ e);
                        //display cause exception
                        System.out.println("Original clause:"+ e.getCause());
                }
        }
}
```

Output:

```
Caught:java.lang.ArithmeticException
Original clause:java.io.IOException: Original cause
```

Three recently added Exception features

Following are the three new features added to Exceptions.

1. **try-with-resources** statement
2. **multi-catch** statement
3. **precise rethrow** statement

1.try-with-resources

In Java, resources can be closed automatically by using **try-with-resources** statement feature. This process is called as **Automatic Resource management (ARM).**

Syntax:

```
try(resource-specification)
{
}
```

Generally, when we use any resources like streams, connections, etc. we have to close them explicitly using finally block.

But, by using **try-with-resources,** resources will be closed automatically.

Example

```
import java.io.*;

class CopyFile {
        public static void main(String args[]) throws IOException {
        int i;

   // Open and manage two files via the try statement.
   try (FileInputStream fin = new FileInputStream("D:\\A.java");
       FileOutputStream fout = new FileOutputStream("E:\\B.java"))
   {
                do {
                        i = fin.read();
                        if(i != -1)
                                fout.write(i);
                        } while(i != -1);

        } catch(IOException e) {
                System.out.println("I/O Error: " + e);
        }
  }
}
```

2. multi-catch statement

- The multi-catch feature allows two or more exceptions to be caught by the same catch clause.
- To use a multi-catch, separate each exception type in the catch clause with the OR operator.
- Each multi-catch parameter is implicitly final.
- Here is a catch statement that uses the multi-catch feature to catch both **ArithmeticException** and **ArrayIndexOutOfBoundsException**:

  catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {

// Demonstrate the multi-catch feature.

```
class MultiCatch {
        public static void main(String args[]) {
                int a=10, b=0;
                int vals[] = { 1, 2, 3 };
```

```
		try {
			int result = a / b; // generate an ArithmeticException

		       vals[10] = 19; // generate an ArrayIndexOutOfBoundsException
	.
		}
		// This catch clause catches both exceptions
		catch(ArithmeticException | ArrayIndexOutOfBoundsException e)
		 {
			System.out.println("Exception caught: " + e);
		}
		System.out.println("After multi-catch.");
	}
}
```

3. Precise rethrow statement

In JDK 7, Java allows to throw precise exception rather than generic exception.

```
class OpenException extends Exception { }
class CloseException extends Exception { }

public class PreciseRethrows {
	public static void main(String args[]) throws OpenException, CloseException {
		boolean flag = true;
		try {
			if (flag){
				throw new OpenException();
			}
			else {
				throw new CloseException();
			}
		} catch(OpenException oe) {
			System.out.println(oe.getMessage());
			throw oe;
		}
		catch (CloseException ce) {
			System.out.println(ce.getMessage());
			throw ce;
		}
	}
}
```

2 marks

1. What is Dynamic method dispatch?
2. What is the use of **super** keyword?
3. What is an abstract class and abstract method
4. Explain the use of **final** keyword.
5. What is Object class and write its methods.
6. Define **finally** block in Exceptions
7. Write differences between **throw vs throws**
8. Write the differences between checked and unchecked exceptions.
9. Explain try-with-resources feature.
10. How to prevent method overriding and Inheritance in Java
11. Define package and CLASSPATH
12. What is import statement.
13. Write the differences between **interface** and **abstract class**
14. What are the exception types.

PART – B

1. What is Inheritance? Explain different Types of Inheritance with an example.
   (OR)
   Explain Single inheritance, Multi-level inheritance and Hierarchical inheritance with an example.

2. What is Multiple Inheritance? Why Java does not support multiple inheritance?
   (OR)
   What is **Deadly diamond of death** problem? Explain with an example.

3. What is Method Overriding? Write the differences between **Method Overloading** and **Method Overriding**.

4. What is **Polymorphism**? Explain its types with an example.
   (OR)
   What is Compile Time polymorphism / static polymorphism
   and Run Time Polymorphism / Dynamic Method dispatch / Virtual Method Invocation?
   (OR)
   What is the use of the Method Overriding? Explain the principle used in Polymorphism

5. What is an Abstract class and abstract method? Explain with an example.

6. What is **final?** Explain its uses.
   (OR)
   How to prevent method overriding and Inheritance in Java

7. Define a Package? Explain how package members can be accessed.
   (OR)
   Define a Package? Explain the concept of importing packages using **import** statement.

8. Explain the Access Protection mechanism with Packages.
   (OR)

Explain **private**, **public**, **protected** and **default** access privileges in Java with Packages.

9. What is an Interface and Nested Interface? Explain how the interfaces will be implemented.

10. Explain how the multiple inheritance is achieved with interfaces.
    (OR)
    Explain how one interface can extend another interface and implemented by a class.

11. Explain how the polymorphism is achieved with interfaces.
    (OR)
    How Java achieves run time polymorphism with interfaces.
    (OR)
    Write a Java program to demonstrate Run time polymorphism using interfaces using **Fixed stack** and **Dynamic stack** classes.

12. What are the new features to interfaces in JDK 8?
    (OR)
    Explain interface default methods and static methods in interfaces.

13. What is an Exception? Explain Exception handling mechanism in Java.
    (OR)
    Define an Exception? Explain with an example to demonstrate Exception handling using try, Nested try, catch and finally statements.

14. Explain **throw** and **throws** statements. Write the differences between throw and throws.

15. What is checked and unchecked Exceptions? Write the differences between checked and unchecked exceptions.

16. What are the Java's Built-in exceptions? How to create custom / user-defined Exception class in Java.

17. What is chained Exception? Explain with an example.

18. What are the three new features added to Exception in JDK 7?
    (OR)
    Explain new features of Exceptions – **try-with-resources,multicatch** and **Precise rethrow**.

Object Oriented Programming using JAVA

UNIT IV

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.
- A thread introduces **Asynchronous** behaviour.

Multitasking

Two types:

1. Process based multitasking
2. Thread based multitasking

Process based multitasking	Thread based multitasking
Process is a program under execution	Thread is one part of a program.
Two or more programs run concurrently	Two or more parts of a single program run concurrently.
Heavyweight process.	Lightweight process.
Programs requires separate address spaces	Same address space is shared by threads.
Interprocess communication is expensive and limited.	Interthread communication is inexpensive.
Context switching from one process to another is also costly.	Context switching from one thread to the next is lower in cost.
May create more idle time.	Reduces the idle time.
Ex: Running a Java compiler and downloading a file from a web site at the same time	**Ex:** We can format a text using a Text editor and printing the data at the same time.

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.

a) 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 class methods

Method	Meaning
getName()	Obtain a thread's name.
getPriority()	Obtain a thread's priority.
setName()	Give a name to a thread
setPriority()	Set the priority to a thread
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.
currentThread()	returns a reference to the thread in which it is called

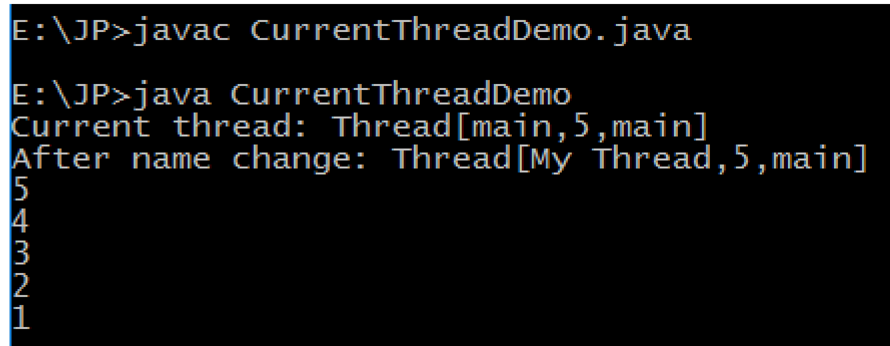
b) The Main thread

- When a Java program starts up, one thread begins running immediately. This is usually called the **main thread of the program.**
- It is the thread from which other "child" threads will be spawned.

```
// 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");
                }
        }
}
```

Output:



```
E:\JP>javac CurrentThreadDemo.java

E:\JP>java CurrentThreadDemo
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

Thread Group:

A ***thread group*** is a data structure that controls the state of a collection of threads as a whole.

Creating and Starting a Thread

Two ways to create a thread –

1. **By extending Thread class**
2. **By implementing Runnable interface**

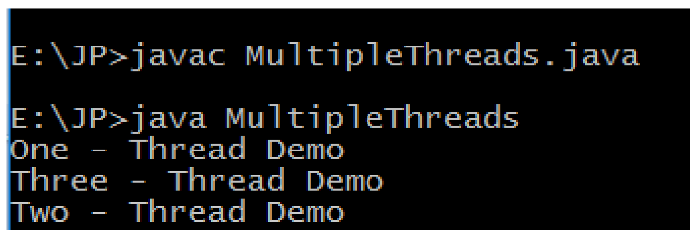
➔ **run()** method is the entry point for another concurrent thread of execution in the program.

1. By extending Thread class

- The first 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.

Example

```
class MultipleThreads extends Thread
{
        MultipleThreads(String name)
        {
          super(name);
           start();
        }
        public void run()
        {
                System.out.print(Thread.currentThread().getName());
                System.out.println(" - Thread Demo");
        }
        public static void main(String ar[])
        {
                MultipleThreads t1 = new MultipleThreads("One");
                MultipleThreads t2 = new MultipleThreads("Two");
                MultipleThreads t3 = new MultipleThreads("Three");
        }
}
```



2. By implementing Runnable interface

- The easiest way to create a thread is to create a class that implements the Runnable interface.
- To implement Runnable, a class need only implement a single method called **run( ),** which is declared like this:

  public void run( )

- Inside run( ), we 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.

Example

```
class RMultipleThreads implements Runnable
{
        String tname;
        Thread t;
        RMultipleThreads(String name)
        {
                tname = name;
                t = new Thread(this,tname);
                t.start();
        }
        public void run()
        {
         System.out.println(Thread.currentThread().getName()+"- Thread Demo");
        }
        public static void main(String ar[])
        {
                RMultipleThreads t1 = new RMultipleThreads("One");
                RMultipleThreads t2 = new RMultipleThreads("Two");
                RMultipleThreads t3 = new RMultipleThreads("Three");
        }
}
```

```
E:\JP>javac RMultipleThreads.java

E:\JP>java RMultipleThreads
Three- Thread Demo
One- Thread Demo
Two- Thread Demo
```

Which of the above two approaches is better?

- The first approach is easier to use in simple applications, but is limited by the fact that our class must be a subclass of Thread. In this approach, our class can't extend any other class.
- The second approach, which employs a Runnable object, is more general, because the Runnable object can subclass a class other than Thread.

Thread Priorities, isAlive( ) and join( ) methods

Thread Priorities

1) Every thread has a priority that helps the operating system to determine the order in which threads are scheduled for execution.
2) Thread priorities are integers that ranges between, 1 to 10.

   MIN-PRIORITY (a constant of 1)

   MAX-PRIORITY (a constant of 10)

3) By default every thread is given a **NORM-PRIORITY(5).** The **main** thread always have **NORM-PRIORITY**.
4) Thread's priority is used in ***Context Switch.***

Context Switch

Switching from one running thread to the next thread is called as ***Context Switch***.

Rules for Context Switch

1) ***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.

2) ***A thread can be preempted by a higher-priority thread***: In this case, a lower-priority thread preempted by a higher-priority thread. This is called **preemptive multitasking**.

Note:

- In *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.

final void setPriority(int level)

To set a thread's priority, use the setPriority( ) method, which is a member of Thread.

final int getPriority( )

By using getPriority(), we can obtain the current priority.

isAlive( ) & join( )

isAlive( ) -> The isAlive( ) method returns true if the thread upon which it is called is still running. It returns false otherwise.

final boolean isAlive( )

join( ) -> This method waits until the thread on which it is called terminates.

final void join( ) throws InterruptedException

Example:

```
class ThreadDemo implements Runnable
{
        public void run()
        {
                try
                {
                        for(int i=0;i<3;i++)
                        {
                        System.out.println("Thread
Demo:"+Thread.currentThread().getName());
                        Thread.currentThread().sleep(1000);
                        }
                }
                catch(InterruptedException ie)
                {
                        System.out.println("Thread interrupted");
                }
        }
}
class MultiThreadDemo{
        public static void main(String ar[])
        {
                ThreadDemo r = new ThreadDemo();
                Thread t1 = new Thread(r);
                t1.setName("First Thread");
                t1.setPriority(2);
                t1.start();

                Thread t2 = new Thread(r);
                t2.setName("Second Thread");
                t2.setPriority(7);
```

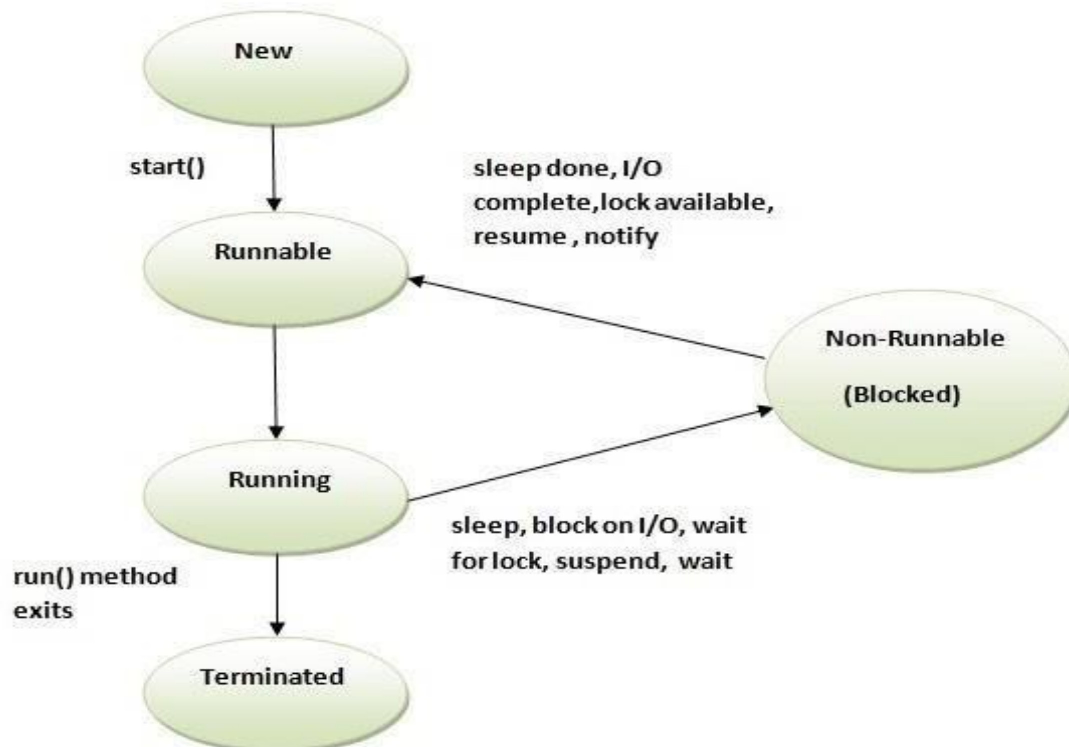
```
            t2.start();

            Thread t3 = new Thread(r);
            t3.setName("Third Thread");
            t3.setPriority(9);
            t3.start();

            System.out.println("Thread One is alive: "+ t1.isAlive());
            System.out.println("Thread Two is alive: "+ t2.isAlive());
            System.out.println("Thread Three is alive: "+ t3.isAlive());
            // wait for threads to finish
            try {
                    System.out.println("Waiting for threads to finish.");
                    t1.join();
                    t2.join();
                    t3.join();
            }
            catch (InterruptedException e) {
                    System.out.println("Main thread Interrupted");
            }
            System.out.println("Thread One is alive: "+ t1.isAlive());
            System.out.println("Thread Two is alive: "+ t2.isAlive());
            System.out.println("Thread Three is alive: "+ t3.isAlive());

            System.out.println("Main thread exiting.");
        }
}
```

```
E:\JP>java MultiThreadDemo
Thread One is alive: true
Thread Demo:First Thread
Thread Demo:Third Thread
Thread Demo:Second Thread
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
Thread Demo:Third Thread
Thread Demo:First Thread
Thread Demo:Second Thread
Thread Demo:Third Thread
Thread Demo:First Thread
Thread Demo:Second Thread
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```

Thread States / Life cycle of a thread

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated

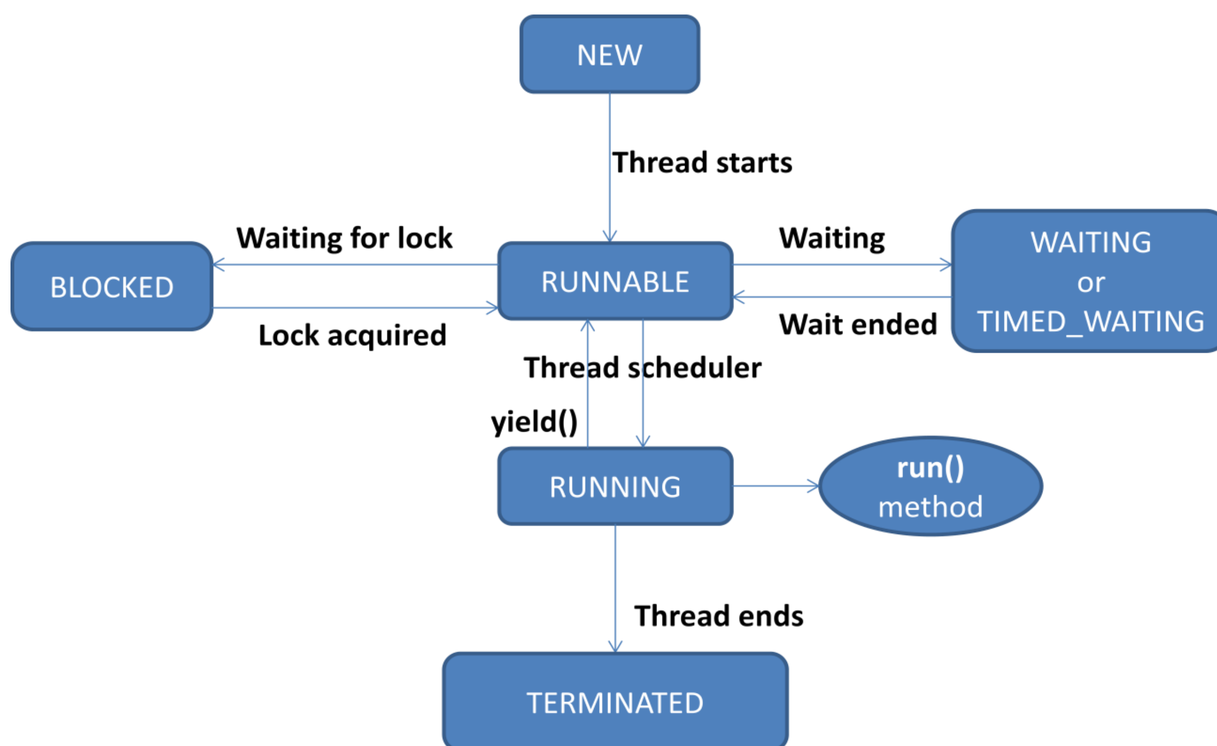


Obtaining A Thread's State

We can obtain the current state of a thread by calling the getState( ) method defined by Thread.

Thread.State getState( )

Value	State
BLOCKED	A thread that has suspended execution because it is waiting to acquire a lock
NEW	A thread that has not begun execution.
RUNNABLE	A thread that either is currently executing or will execute when it gains access to the CPU.
TERMINATED	A thread that has completed execution.
TIMED_WAITING	A thread that has suspended execution for a specified period of time, such as when it has called sleep( ). This state is also entered when a timeout version of wait( ) or join( ) is called.
WAITING	A thread that has suspended execution because it is waiting for some action to occur. For example, it is waiting because of a call to a non-timeout version of wait( ) or join( ).



Synchronization

Definition:

- 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**.

Process behind Synchronization

- Key to synchronization is the concept of the monitor.
- A monitor is an object that is used as a mutually exclusive lock.
- Only one thread can own a monitor at a given time.
- When a thread acquires a lock, it is said to have entered the monitor.
- All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.
- These other threads are said to be waiting for the monitor.

Synchronizing code

We can synchronize the code in two ways:

1. **Using synchronized methods**

```
synchronized void test( )
{
}
```

2.Using Synchronized statements (synchronized blocks)

synchronized statement

```
synchronized(objRef) {
        // statements to be synchronized
}
```

Here, objRef is a reference to the object being synchronized

Example for Synchronization

```
class Account {
        private int balance = 50;
        public int getBalance()
```

```
        {
                return balance;
        }
        public void withdraw(int amount)
        {
                balance = balance - amount;
        }
}
class AccountDanger implements Runnable
{
        private Account acct = new Account();

        public void run() {
                for (int x = 0; x < 5; x++) {
                        makeWithdrawal(10);
                        if (acct.getBalance() < 0) {
                                System.out.println("account is overdrawn!");
                        }
                }
        }
        private synchronized void makeWithdrawal(int amt) {
                if (acct.getBalance() >= amt) {
                        System.out.println(Thread.currentThread().getName()+ " is going
                        to withdraw");
                        try {
                                Thread.sleep(500);
                        }
                        catch(InterruptedException ex) { }
                        acct.withdraw(amt);

                        System.out.println(Thread.currentThread().getName()+ "
                        completes the withdrawal");
                }
                else {
```

```
                System.out.println("Not enough in account for " +
            Thread.currentThread().getName()+ " to withdraw " + acct.getBalance());
             }
        }
    }
    public class SyncExample
    {
        public static void main (String [] args)
        {
            AccountDanger r = new AccountDanger();
            Thread t1 = new Thread(r);
            t1.setName("A");
            Thread t2 = new Thread(r);
            t2.setName("B");
            t1.start();
            t2.start();
        }
    }
```

Output

```
E:\JP>java SyncExample
A is going to withdraw
A completes the withdrawal
A is going to withdraw
A completes the withdrawal
A is going to withdraw
A completes the withdrawal
B is going to withdraw
B completes the withdrawal
B is going to withdraw
B completes the withdrawal
Not enough in account for A to withdraw 0
Not enough in account for B to withdraw 0
Not enough in account for A to withdraw 0
Not enough in account for B to withdraw 0
Not enough in account for B to withdraw 0
```

Interthread Communication

- Java includes an elegant interprocess communication mechanism via the **wait( )**, **notify ( )**, and **notifyAll( )** methods.
- These methods are implemented as final methods in Object, so all classes have them.
- All three methods can be called only from within a **synchronized context**.

wait( ) - wait( ) tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )** or **notifyAll( ).**
notify( ) - notify( ) wakes up a thread that called wait( ) on the same object.
notifyAll( ) - notifyAll( ) wakes up all the threads that called wait( ) on the same object. One of the threads will be granted access.

Example: Producer and Consumer Problem

Producer produces an item and consumer consumes an item produced by the producer immediately. Producer should not produce any item until the consumer consumes the item. Consumer should wait until producer produces a new item.

```
class Q
{
        int n;
        boolean valueSet = false;
        synchronized int get() {
                while(!valueSet)
                        try {
                                        wait();
                                }
                                catch(InterruptedException e) {
                                        System.out.println("InterruptedException caught");
                                }
                System.out.println("Got: " + n);
                valueSet = false;
```

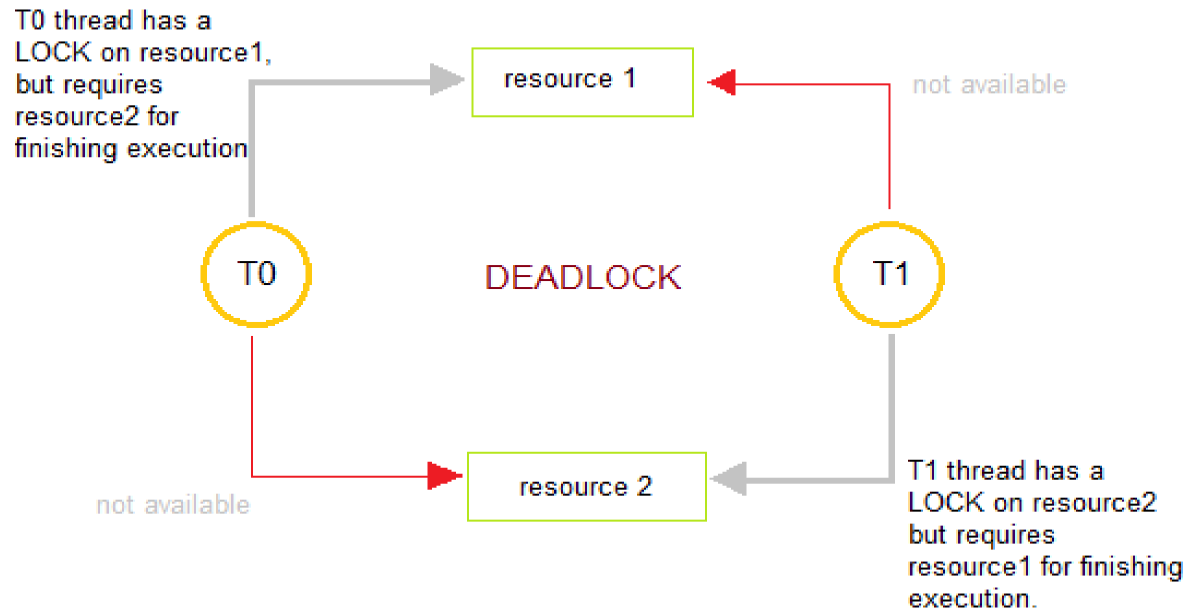
```
                notify();
                return n;
        }
        synchronized void put(int n) {
                 while(valueSet)
                        try {
                                wait();
                        }
                        catch(InterruptedException e) {
                                System.out.println("InterruptedException caught");
                        }
                         this.n = n;
                         valueSet = true;
                         System.out.println("Put: " + n);
                         notify();
        }
}
class Producer implements Runnable
{
        Q q;
        Producer(Q q)
        {
                this.q = q;
                Thread t = new Thread(this, "Producer");
                t.start();
        }
        public void run()
        {
                int i = 0;
                while(i<10)
                {
                        q.put(i++);
                }
        }
```

```
}
class Consumer implements Runnable
{
        Q q;
        Consumer(Q q)
        {
                this.q = q;
                Thread t = new Thread(this, "Consumer");
                t.start();
        }
   public void run()
   {
                int i = 0;
                while(i < 10)
                {
                        q.get();
                }
        }
}

class PC
{
        public static void main(String args[])
        {
                Q q = new Q();
                Producer p = new Producer(q);
                Consumer c = new Consumer(q);
        }
}
```

```
E:\JP>javac PC.java

E:\JP>java PC
Put:  0
Got:  0
Put:  1
Got:  1
Put:  2
Got:  2
Put:  3
Got:  3
Put:  4
Got:  4
Put:  5
Got:  5
Put:  6
Got:  6
Put:  7
Got:  7
Put:  8
Got:  8
Put:  9
Got:  9
```

Deadlock

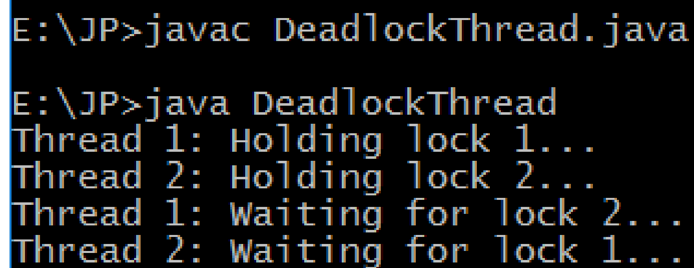
➢ *Deadlock* describes a situation where two or more threads are blocked forever, waiting for each other.



Example:

```
public class DeadlockThread {
  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...");
            }
         }
      }
   }
}
```



Suspending, Resuming, and Stopping Threads

Sometimes, suspending execution of a thread is useful.

- ✓ Java 1.0 has methods for suspending,resuming and stopping threads.
  - **Suspend()** -> to pause a thread
  - **Resume()** -> to restart a thread
  - **Stop()** -> To stop the execution of a thread
- ✓ But these methods are inherently unsafe.
- ✓ Due to this, from Java 2.0, these methods are deprecated (available, but not recommended to use them).

Example

```
// Suspending and resuming a thread the modern way.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;
    boolean suspendFlag;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        suspendFlag = false;
        t.start(); // Start the thread
    }
    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 3; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(200);
                synchronized(this) {
```

```
                    while(suspendFlag) {
                        wait();
                    }
                }
            }
        } catch (InterruptedException e) {
            System.out.println(name + " interrupted.");
        }
        System.out.println(name + " exiting.");
    }
     synchronized void mysuspend() {
        suspendFlag = true;
    }

    synchronized void myresume() {
        suspendFlag = false;
        notify();
    }
}
class SuspendResume {
    public static void main(String args[]) {
        NewThread ob1 = new NewThread("One");
        NewThread ob2 = new NewThread("Two");

        try {
            Thread.sleep(1000);
            ob1.mysuspend();
            System.out.println("Suspending thread One");

            Thread.sleep(1000);
            ob1.myresume();
            System.out.println("Resuming thread One");

            ob2.mysuspend();
```

```
                    System.out.println("Suspending thread Two");
                    Thread.sleep(1000);
                    ob2.myresume();
                    System.out.println("Resuming thread Two");
            }
            catch (InterruptedException e) {
                    System.out.println("Main thread Interrupted");
            }

            // wait for threads to finish
            try {
                    System.out.println("Waiting for threads to finish.");
                    ob1.t.join();
                    ob2.t.join();
            }
            catch (InterruptedException e) {
                    System.out.println("Main thread Interrupted");
            }
            System.out.println("Main thread exiting.");
        }
}
```

```
E:\JP>java SuspendResume
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
One: 3
Two: 3
One: 2
Two: 2
One: 1
Two: 1
Two exiting.
One exiting.
Suspending thread One
Resuming thread One
Suspending thread Two
Resuming thread Two
Waiting for threads to finish.
Main thread exiting.
```

Input and Output (I/O)

stream

- ➔ Java programs perform I/O through **streams**.
- ➔ A **stream** is an abstraction that either produces or consumes information.
- ➔ A **stream** is linked to a physical device by the Java I/O system.
- ➔ Java implements **streams** within class hierarchies defined in the **java.io** package.
- ➔ Streams are two types:
  - Byte streams
  - Character Streams

Byte Streams

Byte streams provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data.

The Byte Stream classes

InputStream -> **abstract class**

OutputStream -> **abstract class**

BufferedInputStream

BufferedOutputStream

ByteArrayInputStream

ByteArrayOutputStream

DataInputStream

DataOutputStream

PrintStream

RandomAccessFile

Character Streams

Character streams provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized. Also, in some cases, character streams are more efficient than byte streams.

Character Stream classes

Character streams are defined by using two class hierarchies. At the top are two abstract classes: **Reader** and **Writer**. These abstract classes handle Unicode character streams.

Reader -> **abstract class**
Writer -> **abstract class**
BufferedReader
BufferedWriter
CharArrayReader
CharArrayWriter
FileReader
FileWriter
InputStreamReader -> translates bytes to characters
OutputStreamWriter -> translates characters to bytes
PrintWriter -> output stream contains print() and println() methods

The Predefined Streams

System class defines three predefined streams – **in,out, err**

1) **System.in** is an object of type **InputStream**
2) **System.out** and **System.err** are objects of type **PrintStream**.
3) **System.in** refers to standard input, which is the keyboard by default.
4) **System.out** refers to the standard output stream.
5) **System.err** refers to the standard error stream, which also is the console by default.

Reading Console Input – reading characters & Strings

- ➔ In Java, console input is accomplished by reading from **System.in**.
- ➔ To obtain a characterbased stream that is attached to the console, wrap **System.in** in a **BufferedReader** object.

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

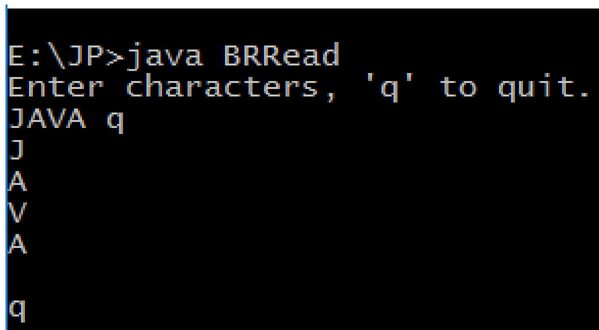
➔ To read a character from a BufferedReader, use **read()** method.

int read( ) throws IOException

➔ Each time that read( ) is called, it reads a character from the input stream and returns it as an integer value. It returns –1 when the end of the stream is encountered.

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');
        }
}
```



Applet Fundamentals

An applet is a GUI based program. Applets are event –driven programs. **applets do not contain main() method**

Two types:

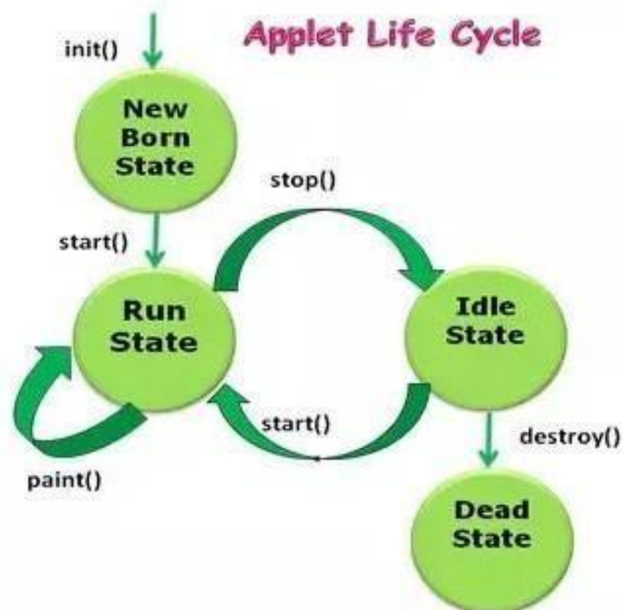
AWT based

SWING based

- applets are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and executed by Java compatible **web browser** or **appletviewer**.

Applet Lifecycle / Applet Skeleton

- When an applet begins, the following methods are called, in this sequence:
  1. init( )
  2. start( )
  3. paint( )
- When an applet is terminated, the following sequence of method calls takes place:
  1. stop( )
  2. destroy( )



init() : The init( ) method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.

start() : The start( ) method is called after init( ). It is also called to restart an applet after it has been stopped. start( ) is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at start( ).

stop(): The stop( ) method is called when a web browser leaves the HTML document containing the applet—when it goes to another page.

destroy(): The destroy( ) method is called when the environment determines that your applet needs to be removed completely from memory. The stop( ) method is always called before destroy( ).

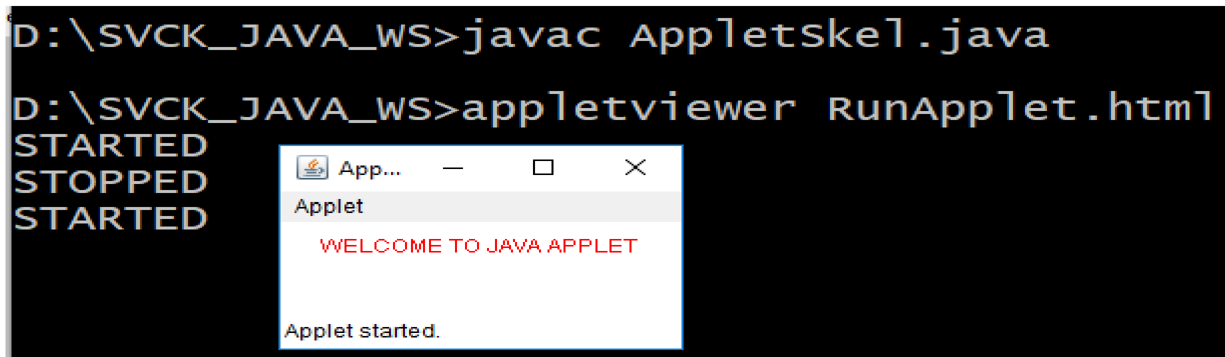
AppletSkel.java

```
// An Applet skeleton.
import java.awt.*;
import java.applet.*;

public class AppletSkel extends Applet {
        String s;
        // Called first.
        public void init() {
                // initialization
                s = "WELCOME TO JAVA APPLET";
        }

        /* Called second, after init(). Also called whenever the applet is restarted. */
        public void start() {
                // start or resume execution
                 System.out.println("START");
        }

        // Called when the applet is stopped.
        public void stop() {
                // suspends execution
                System.out.println("STOPPED");
        }

        /* Called when applet is terminated. This is the last method executed. */
        public void destroy() {
                // perform shutdown activities
                System.out.println("DESTROY");
        }

        // Called when an applet's window must be restored.
        public void paint(Graphics g) {
                // redisplay contents of window
```

```
            g.setColor(Color.red);
            g.drawString(s,20,20);
      }
}
```

RunApplet.html

```
<applet code="AppletSkel.class" width=200 height=60> </applet>
```



Applet Program with Parameters

```
import java.awt.*;
import java.applet.*;

public class SimpleApplet extends Applet {
      public void paint(Graphics g) {
           String myFont = getParameter("font");
           String myString = getParameter("string");
           int mySize     = Integer.parseInt(getParameter("size"));
          Font f = new Font(myFont, Font.BOLD, mySize);
          g.setFont(f);
          g.setColor(Color.red);
          g.drawString(myString, 20, 20);
    }
}
```

RunApplet.html

```
<applet code="SimpleApplet.class" width=200 height=60>
  <PARAM NAME="font"    VALUE="Dialog">
```

<PARAM NAME="size" VALUE="24">
<PARAM NAME="string" VALUE="Hello, world...It's Java Applet">
</applet>

Enumerations

a) An enumeration is a list of named constants.
b) Java enumerations are class types.
c) Each enumeration constant is an object of its enumeration type.
d) Each enumeration constant has its own copy of any instance variables defined by the enumeration.
e) All enumerations automatically inherit one: **java.lang.Enum**.

```
// An enumeration of apple varieties.
    enum Apple {
            A, B, C, D, E
    }
```

- The identifiers Jonathan, GoldenDel, and so on, are called enumeration constants.
- Each is implicitly declared as a public, static, final member of Apple.
- In the language of Java, these constants are called **self-typed.**

Built-in Methods of ENUM

2 methods: values() and valueOf()

The values( ) and valueOf( ) Methods

All enumerations automatically contain two predefined methods: **values( )** and **valueOf( ).** Their general forms are shown here:

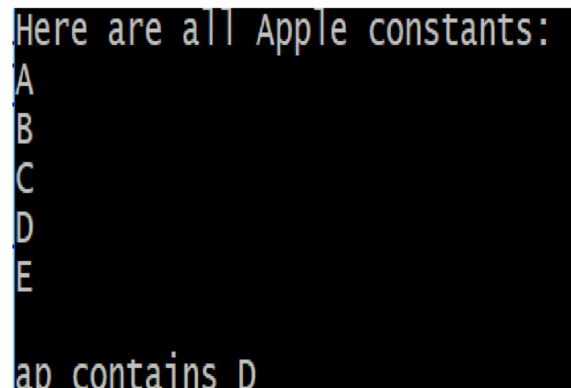
public static enum-type [ ] values( )

public static enum-type valueOf(String str)

- ➔ The **values( )** method returns an array that contains a list of the enumeration constants.
- ➔ The **valueOf( )** method returns the enumeration constant whose value corresponds to the string passed in str.

Enum Example:

```
// An enumeration of apple varieties.
enum Apple {
        A, B, C, D, E
}

class EnumDemo {
   public static void main(String args[]) {
        Apple ap;
        System.out.println("Here are all Apple constants:");
        // use values()
        Apple allapples[] = Apple.values();
        for(Apple a : allapples)
            System.out.println(a);
            System.out.println();
          // use valueOf()
         ap = Apple.valueOf("Winesap");
        System.out.println("ap contains " + ap);
   }
}
```



Java enumerations with the constructors, instance variables and method

```
// Use an enum constructor, instance variable, and method.

enum Apple {
        A(10), B(9), C(12), D(15), E(8);

        private int price; // price of each apple
```

```
        // Constructor
        Apple(int p) {
                price = p;
        }
        int getPrice() {
                return price;
        }
}


class EnumConsDemo {
        public static void main(String args[]) {
                Apple ap;

                // Display price of B.
                System.out.println("B costs " + Apple.B.getPrice() + " cents.\n");

                // Display all apples and prices.
                System.out.println("All apple prices:");
                for(Apple a : Apple.values())
                        System.out.println(a + " costs " + a.getPrice() +" cents.");
                }
}
```

```
B costs 9 Rupees.

All apple prices:
A costs 10 Rupees.
B costs 9 Rupees.
C costs 12 Rupees.
D costs 15 Rupees.
E costs 8 Rupees.
```

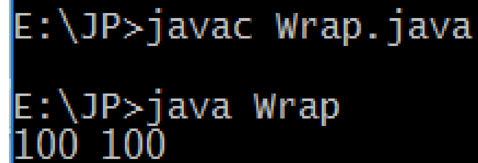
Type Wrappers (Autoboxing and autounboxing)

- ➔ Type wrappers are classes that encapsulate a primitive type within an object.
- ➔ The type wrappers are **Double, Float, Long, Integer, Short, Byte, Character, and Boolean**.
- ➔ Type wrappers are related directly to **auto-boxing / auto-unboxing** feature.

Type Wrapper	**Constructor**	**Method name**
Character	Character(char ch)	char charValue( )
Boolean	Boolean(boolean boolValue) Boolean(String boolString)	boolean booleanValue( )
	Numeric Type Wrappers	
Byte	Byte(int num)	byte byteValue( )
Short	Short(int num)	short shortValue( )
Long	Long(int num),Long(String str)	long longValue( )
Float	Float(float num)	float floatValue( )

Double	Double(double num)	double doubleValue( )

```
// Demonstrate a type wrapper.
class Wrap {
        public static void main(String args[]) {
          // boxing
           Integer iOb = new Integer(100);
          //unboxing
           int i = iOb.intValue();
           System.out.println(i + " " + iOb);
        }
}
```



boxing: The process of encapsulating a value within an object is called **boxing**.

Unboxing: The process of extracting a value from a type wrapper is called **unboxing**.

Autoboxing and Autounboxing

Beginning with JDK 5, Java does this **boxing** and **unboxing** automatically through auto-boxing and auto-unboxing features.

Autoboxing

Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper.

Autounboxing

Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper.

Autoboxing/Autounboxing- benefits

1. No manual boxing and unboxing values.
2. It prevents errors
3. It is very important to generics, which operate only on objects.
4. autoboxing makes working with the Collections Framework.

Where it Works

1) In assignments
2) In Method arguments and return types
3) In Expressions
4) In switch statement and any loop statements

1) Autoboxing in assignments

```
Integer iOb = 100; // autobox an int

int i = iOb; // auto-unbox
```

2) Autoboxing in Methods

```
int m(Integer v) {
        return v ; // auto-unbox to int
}
```

3) Autoboxing in expressions

```
Integer iOb, iOb2;
 int i;
iOb = 100;
++iOb;
System.out.println("After ++iOb: " + iOb);     // output: 101

iOb2 = iOb + (iOb / 3);
System.out.println("iOb2 after expression: " + iOb2); // output: 134
```

4) Autoboxing in switch and loops

```
Integer iOb = 2;

switch(iOb) {
        case 1: System.out.println("one");
        break;
        case 2: System.out.println("two");
        break;
        default: System.out.println("error");
}
```

Annotations

Definition:

Java Annotation is a tag that represents the *metadata* i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

In Java, Annotations can be **Built-in annotations** or **Custom annotations / user-defined annotations**.

Java's Built-in Annotations / Predefined Annotation Types

➔ **Built-In Java Annotations used in java code**

@Override
@SuppressWarnings
@Deprecated

1) **@Override**

@Override annotation assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.

2) **Deprecated**

@Deprecated annoation marks that this method is deprecated so compiler prints warning. It informs user that it may be removed in the future versions. So, it is better not to use such methods.

3) **SuppressWarnings**

@SuppressWarnings annotation is used to suppress warnings issued by the compiler.

➔ **Built-In Java Annotations used in other annotations are called as *meta-annotations.*** There are several **meta-annotation types** defined in **java.lang.annotation**.

@Target
@Retention
@Inherited
@Documented

1) **@Target**

a) @Target annotation marks another annotation to restrict what kind of Java elements the annotation can be applied to.

b) A target annotation specifies one of the following element types as its value:
ElementType.FIELD can be applied to a field or property.
ElementType.LOCAL_VARIABLE can be applied to a local variable.
ElementType.METHOD can be applied to a method-level annotation.
ElementType.PACKAGE can be applied to a package declaration.
ElementType.PARAMETER can be applied to the parameters of a method.
ElementType.TYPE can be applied to any element of a class.

2) @Retention

@Retention annotation specifies how the marked annotation is stored:

- ➔ **RetentionPolicy.SOURCE** – The marked annotation is retained only in the source level and is ignored by the compiler.
- ➔ **RetentionPolicy.CLASS** – The marked annotation is retained by the compiler at compile time, but is ignored by the Java Virtual Machine (JVM).
- ➔ **RetentionPolicy.RUNTIME** – The marked annotation is retained by the JVM so it can be used by the runtime environment.

3) @Inherited

@Inherited annotation indicates that the annotation type can be inherited from the super class.

4) @Documented

@Documented annotation indicates that whenever the specified annotation is used those elements should be documented using the Javadoc tool.

Built-in Annotations Example

```
class Animal{
        void eat(){
                System.out.println("eating something");
        }
}

class Dog extends Animal{
        @Override
        void Eat(){
```

```
            System.out.println("eating foods");
      }//should be eatSomething
}
class TestAnnotation1{
      public static void main(String args[]){
            Animal a=new Dog();
            a.eat();

      }
}
```

```
E:\JP>javac TestAnnotation1.java
TestAnnotation1.java:9: error: method does not override or implement a method from a supertype
@Override
^
1 error
```

Custom Annotations / User defined annotations

Java Custom annotations or Java User-defined annotations are easy to create and use. The *@interface* element is used to declare an annotation. For example:

```
@interface MyAnnotation
{
}
```

Types of Annotation

There are three types of annotations.

1. Marker Annotation
2. Single-Value Annotation
3. Multi-Value Annotation

1) Marker Annotation

An annotation that has no method, is called marker annotation. For example:

```
@interface MyAnnotation{ }
```

The @Override and @Deprecated are marker annotations.

2) Single-value annotation

An annotation that has one method, is called single-value annotation. We can provide the default value also.

For example:

```
@interface MyAnnotation{
    int value() default 0;
}
```

3) Multi-value Annotation

An annotation that has more than one method, is called Multi-Value annotation. For example:

```
@interface MyAnnotation{
        int value1();
        String value2();
        String value3();
    }
}
```

Example of Custom Annotation (creating, applying and accessing annotation)

```
//Creating custom annotation
import  java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface MyAnnotation{
    int value();
}
//Applying annotation
class Hello{
    @MyAnnotation(value=10)
    public void sayHello(){
        System.out.println("hello annotation");
    }
}
//Accessing annotation
class TestCustomAnnotation{
    public static void main(String args[])throws Exception{

        Hello h=new Hello();
```

```
                Method m=h.getClass().getMethod("sayHello");

                MyAnnotation manno=m.getAnnotation(MyAnnotation.class);
                System.out.println("value is: "+manno.value());
        }
}
```

```
E:\JP>java TestCustomAnnotation
value is: 10
```

Generics

- ➔ Generics are called as parameterized types.
- ➔ The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects.
- ➔ Generics can be applied to a class, interface or a method.
- ➔ Generics Work Only with Reference Types.

Advantages of Java Generics

1) Type-safety:

We can hold only a single type of objects in generics. It doesn't allow to store other objects.

2) Compile-Time Checking:

It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

3) Enabling programmers to implement generic algorithms

Generic class

The General Form of a Generic Class

class class-name<type-param-list > { // …

A class that can refer to any type is known as generic class. Here, we are using **T** type parameter to create the generic class of specific type.

```
class MyGen<T>{
        T obj;
        void add(T obj){
                this.obj=obj;
```

```
		}
		T get(){
			return obj;
		}
	}
	class TestGenerics{
		public static void main(String args[]){
			MyGen<Integer> m=new MyGen<Integer>();
			m.add(2);
			//m.add("vivek");//Compile time error
			System.out.println(m.get());
		}
	}
```

```
E:\JP>java TestGenerics
2
```

Generic Method

Like generic class, we can create generic method that can accept any type of argument.

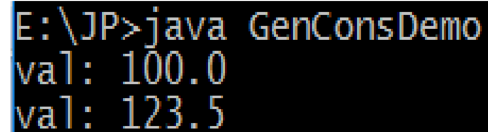
```
public class GenericMethod{
	public static < E > void printArray(E[] elements) {
	    for ( E element : elements){
	        System.out.println(element );
	     }
	     System.out.println();
	 }
	 public static void main( String args[] ) {
	    Integer[] intArray = { 10, 20, 30, 40, 50 };
	    Character[] charArray = { 'J', 'A', 'V', 'A'};

	    System.out.println( "Printing Integer Array" );
	    printArray( intArray  );

	   System.out.println( "Printing Character Array" );
	    printArray( charArray );
	 }
}
```

```
E:\JP>java GenericMethod
Printing Integer Array
10
20
30
40
50

Printing Character Array
J
A
V
A
```

Generic Constructor

It is possible for constructors to be generic, even if their class is not.

```
// Use a generic constructor.
class GenCons {
        private double val;

        <T extends Number> GenCons(T arg) {
                val = arg.doubleValue();
        }
        void showval() {
                System.out.println("val: " + val);
        }
}
class GenConsDemo {
        public static void main(String args[]) {

                GenCons test = new GenCons(100);
                GenCons test2 = new GenCons(123.5F);

                test.showval();
                test2.showval();
        }
}
```



Generic Interfaces

In addition to generic classes and methods, you can also have generic interfaces.

interface interface-name<type-param-list> { // …

class class-name<type-param-list> implements interface-name<type-arg-list> {

```
        interface GenInterface<T> {
                void test(T a);
                T get();
        }

      class MyGen<T> implements GenInterface<T> {
     }
```

Important Questions

Part-A

1. What is main thread?
2. Explain thread priorities.

3. **What is context switch**
4. Define thread group and thread priority.
5. What are the benefits of Java's multithreaded programming.
6. Define Generics? Explain its advantages.
7. Define autoboxing andunboxing.
8. Define Annotation.
9. Explain PrintWriter class
10. Define Stream? Explain it's types.

Part-B

1. Explain Java Thread Modeland Threadstates(transitions)
   (OR)
   What is Multithreaded programming? Explain thread states and how to obtain the thread state.
2. What is Thread class and explain its methods: getName( ),getPriority( ), isAlive( ), join( ),run( ),sleep( ),start( ),currentThread( ).
3. What is multitasking? What are the types of multitasking? Write the differences between **process based multitasking** and **thread based multitasking**.
4. Explain how to create a thread or multiple threads with an example.
5. What is synchronization? What is the need for synchronization? Explain with a suitable example how to synchronize the code. (Account example)
6. In Java, how interthread communication happens. Explain with an example.
   [OR]
   Explain producer and consumer problem using Interthread communication with an example.
   [OR]
   Explain the interthread communication methods - **wait( ), notify( )** and **notifyAll( ).**
7. What is Deadlock? Explain with an example.
8. Explain how to perform Suspending, Resuming and stopping threads in Java.
9. Define Stream? Explain the different types of streams.
   [OR]
   What is Stream? Explain **byte stream** and **character stream** classes.
10. Explain withanexample howtoperform the reading the console inputandwriting the console output.
11. Explain with an example how to read, write and closing of files (automatically).
12. WhatisanApplet?Explainthelifecycleof anapplet.
    [OR]
    Explain a simple Applet program with an example. Also explain how to compile and execute an applet program
13. Define Generics? Explain the general form of a generic class and write its advantages.
14. Explain Generic Methodand Generic Interfaces.
15. Define Enumerations? Explainwithanexample.
16. What is a Type wrapper? Explain Autoboxing and unboxing with an example.
17. Write short notes on Annotations.

UNIT – IV – END

Object Oriented Programming using JAVA

UNIT V

AWT

awt was the first GUI framework in JAVA since 1.0. It defines numerous classes and methods to create windows and simple control such as Labels, Buttons, and TextFields etc.

Delegation Event Model

The modern approach to handle events is based on *delegation event model*.
Delegation event model consists of only two parts – **Source** and **Listeners**.

Concept

A source generates an event and sends it to one or more listeners. In this scheme, the listener simply waits until it receives an event.

Once an event is received, the listener processes the event and then returns. In this model, Listeners must register with a source in order to receive an event notification.

Events

An event is an object that describes a state change in a source.
Ex: Pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.

Event Sorces

A source is an object that generates an event. Sources may generate more than one type of event.
Ex: mouse, keyboard

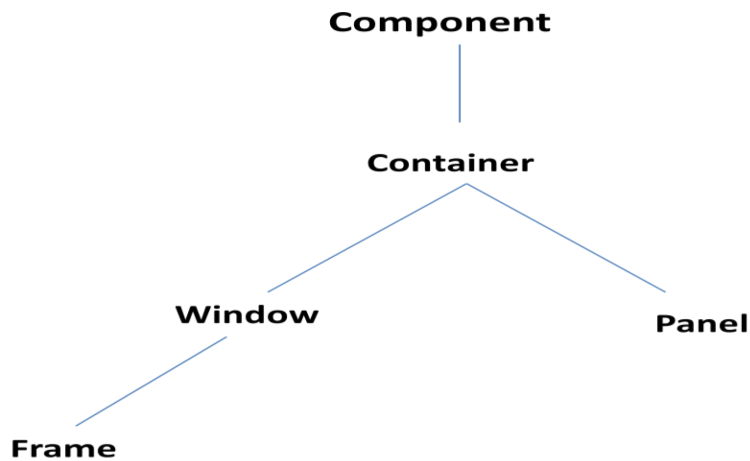
Event Listeners

A listener is an object that is notified when an event occurs. It has two major requirements.

1) It must have been registered with one or more sources to receive notifications about specific types of events.
2) It must implement methods to receive and process these notifications.

The methods that receive and process events are defined in a set of interfaces, such as those found in **java.awt.event.**

AWT classes & Hierarchy



Component

Component is an abstract class that encapsulates all of the attributes of a visual component.

Except for menus, all user interface elements that are displayed on the screen and that interact with the user are subclasses of Component.

Container

The Container class is a subclass of Component. Other Container objects can be stored inside of a Container

A container is responsible for laying out (that is, positioning) any components that it contains. It does this through the use of various layout managers.

Panel

The Panel class is a concrete subclass of Container. Panel is the superclass for Applet. a Panel is a window that does not contain a title bar, menu bar, or border.

Window

The Window class creates a top-level window. A top-level window is not contained within any other object; it sits directly on the desktop.

Frame

It is a subclass of Window and has a title bar, menu bar, borders, and resizing corners.

Creating Windows (Frame based and Applet based)

Windows are created by using applet and by using Frame.

Windows can be created **:**

1. By extending Applet class
2. By extending Frame class
    a. Frame class is used to create child windows within applets
    b. Top-level or child-windows of stand-alone applications

Frame constructor

Frame( ) throws HeadlessException
Frame(String title) throws HeadlessException

A **HeadlessException** is thrown if an attempt is made to create a Frame instance in an environment that does not support user interaction.

Key methods in Frame windows

1. Setting the Window's Dimensions

void setSize(int newWidth, int newHeight)
void setSize(Dimension newSize)

2. Get the window's size

Dimension getSize()

3. Hiding and Showing a Window

After a frame window has been created, it will not be visible until you call setVisible( ).

void setVisible(boolean visibleFlag)

The component is visible if the argument to this method is true. Otherwise, it is hidden.

4. Setting a Window's Title

void setTitle(String newTitle)

5. Closing a Frame Window

To close the window, We must implement the **windowClosing( )** method of the WindowListener interface. Inside windowClosing( ), we must remove the window from the screen by calling setVisible(false).

Creating a Frame window in an AWT-based Applet

Creating a new frame window from within an AWT-based applet is actually quite easy.

a) First, create a subclass of Frame.
b) Next, override any of the standard applet methods, such as init( ), start( ), and stop( ), to show or hide the frame as needed.
c) Finally, implement the windowClosing( ) method of the WindowListener interface, calling setVisible(false) when the window is closed.

AppletFrame.java

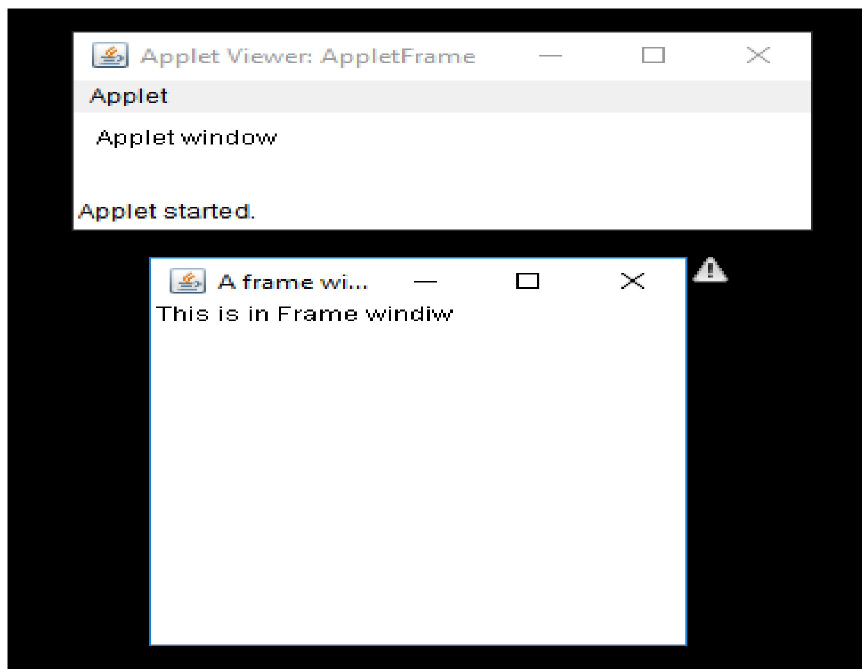
```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class AppletFrame extends Applet
{
	Frame f;

	public void init(){
		f = new SampleFrame("A frame window");
		f.setSize(250,250);
	}
	public void start()
	{
		f.setVisible(true);
	}
	public void stop()
	{
		f.setVisible(false);
	}
	public void paint(Graphics g)
	{
		g.drawString("Applet window",10,20);
	}
}
class SampleFrame extends Frame
{
	SampleFrame(String title)
	{
		setTitle(title);

		// remove the window when closed
		addWindowListener(new WindowAdapter() {
			public void windowClosing(WindowEvent we) {

				setVisible(false);
			}
		});
```

```
        }
        public void paint(Graphics g)
        {
                g.drawString("This is in Frame windiw",10,40);
        }
}
```

RunAppletFrame.html

```
<applet code ="AppletFrame" width=300 height=50>
</applet>
```

Output:



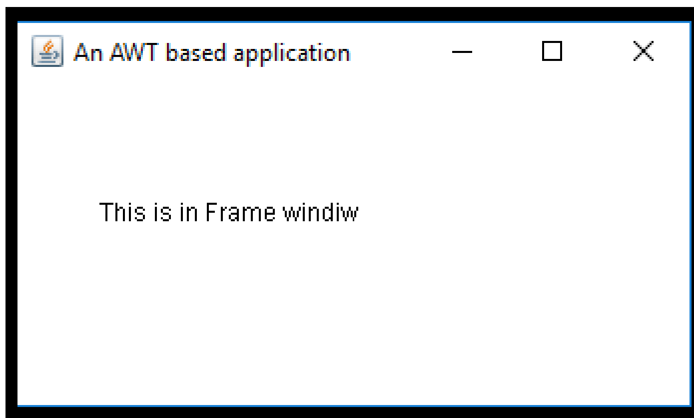
Creating a window using Frame class – AWT-window

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class FrameWindow extends Frame
{
        FrameWindow(String title)
        {
                setTitle(title);

                // remove the window when closed
                addWindowListener(new WindowAdapter() {
```

```
                public void windowClosing(WindowEvent we) {

                        //setVisible(false);
                        System.exit(0);
                }
        });
    }
    public void paint(Graphics g)
    {
        g.drawString("This is in Frame windiw",50,100);
    }

    public static void main(String ar[])
    {
        FrameWindow fw = new FrameWindow("An AWT based application");

        fw.setSize(100,200);
        fw.setVisible(true);
    }
}
```

<u>Output:</u>

```
javac FrameWindow.java
java FrameWindow
```



<u>Handling Events in a Frame Window using AWT based Applet</u>

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class AppletFrameEvents extends Applet
{
    Frame f;
```

```
        public void init(){
                f = new SampleFrame("A frame window");
                f.setSize(250,250);
        }
        public void start()
        {
                f.setVisible(true);
        }
        public void stop()
        {
                f.setVisible(false);
        }
        public void paint(Graphics g)
        {
                g.drawString("Applet window",10,20);
        }
}
class SampleFrame extends Frame implements MouseMotionListener
{
        String msg = " ";
        int mouseX=10,mouseY=40;
        int movX=0,movY=0;
        SampleFrame(String title)
        {
                setTitle(title);

                // remove the window when closed
                addWindowListener(new WindowAdapter() {
                        public void windowClosing(WindowEvent we) {

                                setVisible(false);
                        }
                });

                addMouseMotionListener(this);
        }

        public void mouseDragged(MouseEvent me) {
                mouseX = me.getX();
                mouseY = me.getY();

                movX = me.getX();
                movY = me.getY();
                msg = "Mouse Dragged";
                repaint();
        }

        public void mouseMoved(MouseEvent me) {
                mouseX = me.getX();
                mouseY = me.getY();
```

```
            movX = me.getX();
            movY = me.getY();
            msg = "Mouse Moved";
            repaint();
        }

        public void paint(Graphics g)
        {
            g.drawString(msg,mouseX,mouseY);
            g.drawString("Mouse at "+ movX +", "+movY,10,40);
        }

}
```

Handling Events in a Frame Window using AWT

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class FrameWindowEvents extends Frame implements MouseMotionListener
{

        String msg = " ";
        int mouseX=10,mouseY=40;
        int movX=0,movY=0;
        FrameWindowEvents (String title)
        {
            setTitle(title);

            // remove the window when closed
            addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent we) {

                    System.exit(0);
                }
            });

            addMouseMotionListener(this);
        }

        public void mouseDragged(MouseEvent me) {
            mouseX = me.getX();
            mouseY = me.getY();

            movX = me.getX();
            movY = me.getY();
            msg = "Mouse Dragged";
```

```
                repaint();
        }

        public void mouseMoved(MouseEvent me) {
                mouseX = me.getX();
                mouseY = me.getY();

                movX = me.getX();
                movY = me.getY();
                msg = "Mouse Moved";
                repaint();
        }

        public void paint(Graphics g)
        {
                g.drawString(msg,mouseX,mouseY);
                g.drawString("Mouse at "+ movX +", "+movY,10,40);
        }

        public static void main(String ar[])
        {
                FrameWindowEvents fw = new FrameWindowEvents("An AWT based
application");

                fw.setSize(100,200);
                fw.setVisible(true);
        }
}
```

Graphics Class

Graphics class has methods which are used to display the information within a window.

Methods in Graphics class

1. **Drawing Strings**

    drawstring(String msg,int x, int y)

2. **Drawing Lines**

    void drawLine(int startX, int startY, int endX, int endY)

3. **Drawing Rectangles**

    The drawRect( ) and fillRect( ) methods display an outlined and filled rectangle, respectively.

void drawRect(int left, int top, int width, int height)
void fillRect(int left, int top, int width, int height)

void drawRoundRect(int left, int top, int width, int height, int xDiam, int yDiam)
void fillRoundRect(int left, int top, int width, int height, int xDiam, int yDiam)

4. Drawing Ellipses and Circles

To draw an ellipse, use drawOval( ). To fill an ellipse, use fillOval( ).

void drawOval(int left, int top, int width, int height)
void fillOval(int left, int top, int width, int height)

5. Drawing Arcs

void drawArc(int left, int top, int width, int height, int startAngle,int sweepAngle)
void fillArc(int left, int top, int width, int height, int startAngle,int sweepAngle)

The arc is drawn counterclockwise if sweepAngle is positive, and clockwise if sweepAngle is negative.

6. Drawing Polygons

- It is possible to draw arbitrarily shaped figures using drawPolygon( ) and fillPolygon( ):

void drawPolygon(int x[ ], int y[ ], int numPoints) void fillPolygon(int x[ ], int y[ ], int numPoints)

- The polygon’s endpoints are specified by the coordinate pairs contained within the x and y arrays. The number of points defined by these arrays is specified by numPoints.

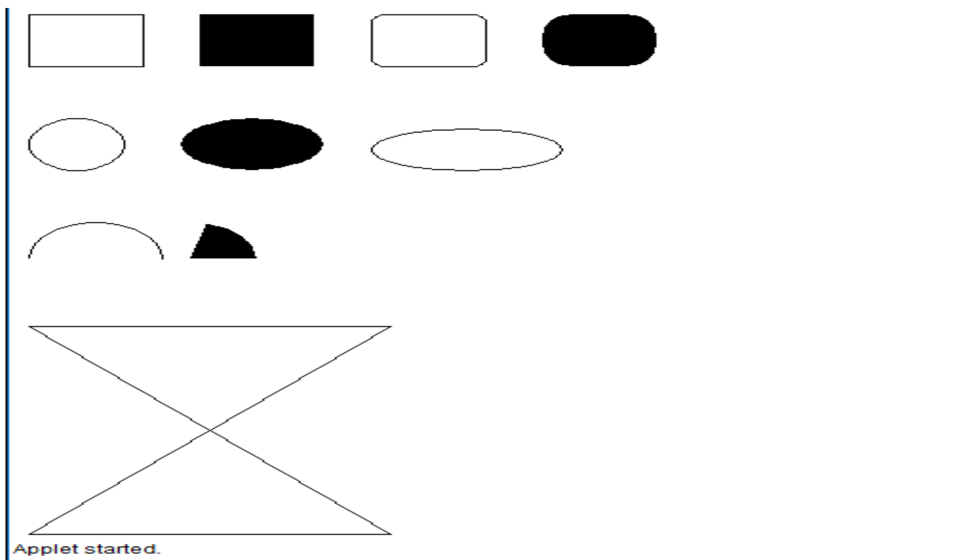
Program to demonstrate Graphics class methods – Drawing methods

// Draw graphics elements.

```
import java.awt.*;
import java.applet.*;

public class GraphicsDemo extends Applet {
	public void paint(Graphics g)
	{
		// Draw lines.
		g.drawLine(0, 0, 100, 90);
		g.drawLine(0, 90, 100, 10);
		g.drawLine(40, 25, 250, 80);

		// Draw rectangles.
```

```
        g.drawRect(10, 150, 60, 50);
        g.fillRect(100, 150, 60, 50);
        g.drawRoundRect(190, 150, 60, 50, 15, 15);
        g.fillRoundRect(280, 150, 60, 50, 30, 40);

        // Draw Ellipses and Circles
        g.drawOval(10, 250, 50, 50);
        g.fillOval(90, 250, 75, 50);
        g.drawOval(190, 260, 100, 40);

        // Draw Arcs
        g.drawArc(10, 350, 70, 70, 0, 180);
        g.fillArc(60, 350, 70, 70, 0, 75);

        // Draw a polygon
        int xpoints[] = {10, 200, 10, 200, 10};
        int ypoints[] = {450, 450, 650, 650, 450};
        int num = 6;

        g.drawPolygon(xpoints, ypoints, num);
    }
}
```

Output:



Color class

Java supports color in a portable, device-independent fashion. The AWT color system allows us to specify any color we want. It then finds the best match for that color, given the limits of the display hardware currently executing your program or applet.

We can create our own color by using the below constructor:

```
Color(int red, int green, int blue)
```

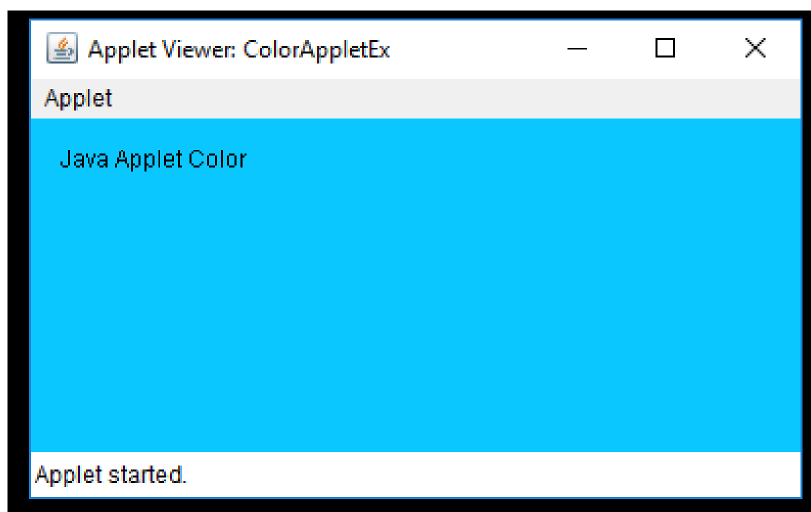
It takes three integers that specify the color as a mix of red, green, and blue. These values must be between 0 and 255, as in this example:

```
new Color(255, 100, 100); // light red
```

Once we have created a color, we can use it to set the foreground and/or background color by using the **setForeground( )** and **setBackground( )** methods.

Program on Color class

```
import java.awt.*;
import java.applet.*;

public class ColorAppletEx extends Applet
{
	Color c = new Color(10,200,255);

	public void init()
	{
		//setColor(c);
		setBackground(c);
		//setBackground(Color.red); // another way to set the background color

	}
	public void paint(Graphics g)
	{
		g.drawString("Java Applet Color", 15,25);
	}
}
```



Font class

The AWT supports multiple type fonts. Fonts have a family name, font style and font size.

Methods in Font class

Method	Description
String getFamily( )	Returns the name of the font family
String getName()	Returns the logical name of the invoking font.
int getSize( )	Returns the size
int getStyle( )	Returns the style of the font

Determining the Available Fonts

To obtain the available fonts on our machine, use the **getAvailableFontFamilyNames( )** method defined by the **GraphicsEnvironment** class.

String[ ] getAvailableFontFamilyNames( )

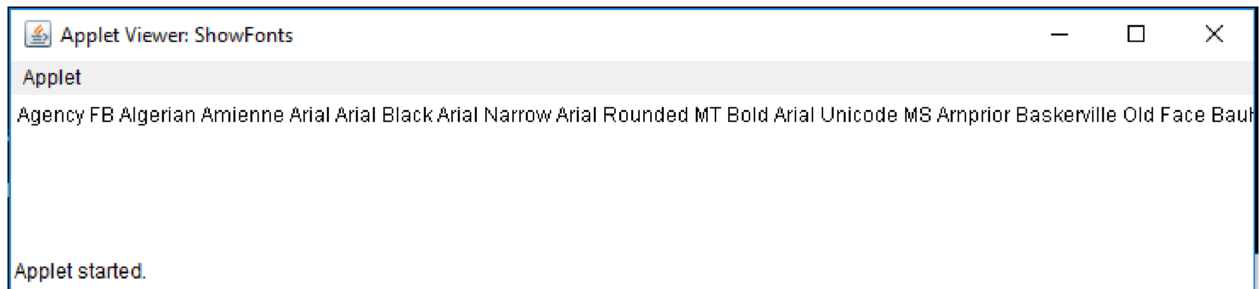
In addition, the **getAllFonts( )** method is defined by the GraphicsEnvironment class.

Font[ ] getAllFonts( )

Ex: Program to obtain the names of the available font families.

```
// Display Fonts
import java.applet.*;
import java.awt.*;

public class ShowFonts extends Applet {
        public void paint(Graphics g) {
                String msg = "";
                String FontList[];

                GraphicsEnvironment ge =
GraphicsEnvironment.getLocalGraphicsEnvironment();
                FontList = ge.getAvailableFontFamilyNames();
                for(int i = 0; i < FontList.length; i++)
                        msg += FontList[i] + " ";

                        g.drawString(msg, 4, 16);
        }
}
```

Output:



Creating and Selecting Font

To create a new font, construct a Font object that describes that font.

Font(String fontName, int fontStyle, int pointSize)

- ***fontName*** specifies the name of the desired font.
- The ***style*** of the font is specified by fontStyle. It may consist of one or more of these three constants: **Font.PLAIN, Font.BOLD**, and **Font.ITALIC**.
- To combine styles, OR them together. For example, **Font.BOLD | Font.ITALIC** specifies a bold, italics style.
- To use a font that you have created, you must select it using **setFont( ),** which is defined by Component. It has this general form:

  void setFont(Font fontObj)

```
// Creating ans Selecting Fonts
import java.applet.*;
import java.awt.*;

public class CreateFont extends Applet {
	Font f;
	String msg=" ";
	public void init()
	{
		f = new Font("Times New Roman",Font.BOLD, 12);
		msg = "JAVA FONT APPLET";
		setFont(f);
	}
	public void paint(Graphics g) {
		g.drawString(msg, 4, 16);
	}
}
```

Displaying a Font information

To obtain the information about the currently selected font, we must get the current font by calling **getFont()** method.

```
public void paint(Graphics g) {
        Font f = g.getFont();
        String fontName = f.getName();
        String fontFamily = f.getFamily();
        int fontSize = f.getSize();
        int fontStyle = f.getStyle();
}
```

FontMetrics Class

FontMetrics class is used to get the information about a font.

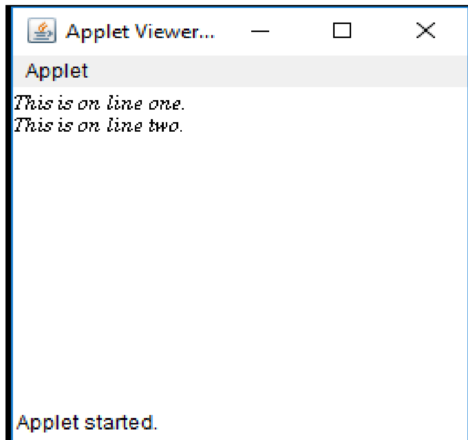
Methods in FontMetrics class

Method	Description
Font getFont( )	Returns the font.
int getHeight( )	Returns the height of a line of text. This value can be used to output multiple lines of text in a window.
int stringWidth(String str)	Returns the width of the string specified by str.
int getLeading( )	Returns the space between lines of text.

// Demonstrate multiline output using FontMetrics class

```
import java.awt.*;
import java.applet.*;

public class FontMetricsEx extends Applet {
        int X=0, Y=0; // current position
        String s1 = "This is on line one.";
        String s2 = "This is on line two.";

        public void init() {
                Font f = new Font("Times New Roman", Font.PLAIN, 12);
                setFont(f);
        }
        public void paint(Graphics g) {
                FontMetrics fm = g.getFontMetrics();

                Y += fm.getHeight(); // advance to next line
```

```
                X = 0;
                g.drawString(s1, X, Y);
                X = fm.stringWidth(s1); // advance to end of line

                Y += fm.getHeight(); // advance to next line
                X = 0;
                g.drawString(s2, X, Y);
                X = fm.stringWidth(s2); // advance to end of line
        }
}
```



AWT Controls

AWT supports the following controls:

1. Labels
2. Push buttons
3. Check boxes
4. Check box group
5. Choice lists
6. Lists
7. Scroll bars
8. Text Editing (TextField, TextArea)

1. Labels

A label is an object of type Label, and it contains a string, which it displays. Labels are passive controls that do not support any interaction with the user.

Constructors

➔ Label( ) throws HeadlessException
-> This creates a blank label.

➔ Label(String str) throws HeadlessException
-> This creates a label that contains the string specified by str.

➔ Label(String str, int how) throws HeadlessException

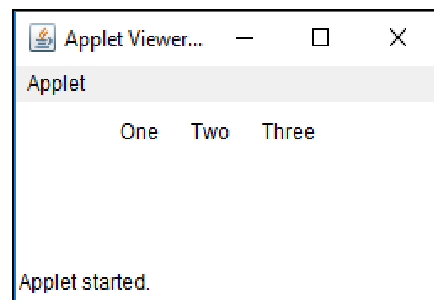
The value of ***how*** must be one of these three constants: **Label.LEFT**, **Label.RIGHT**, or **Label.CENTER**.

➔ **setText(String str) ->** we can set or change the text in a label by using the setText( ) method.

➔ **getText( )** -> We can obtain the current label by calling getText( ).

Example

```
public class LabelDemo extends Applet {
    public void init() {
      Label one = new Label("One");
      Label two = new Label("Two");
      Label three = new Label("Three");

      // add labels to applet window
      add(one);
      add(two);
      add(three);
    }
}
```



2. Buttons

➔ A push button is a component that contains a label and that generates an event when it is pressed.

➔ Push buttons are objects of type Button.

constructors

Button( ) throws HeadlessException
Button(String str) throws HeadlessException

➔ **setLabel(String str)** -> this is used to set the label.

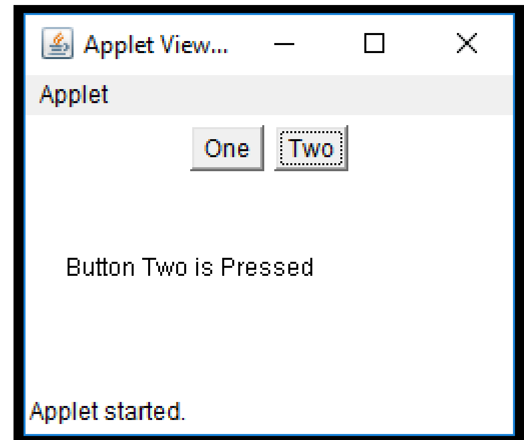
➔ **getLabel( )** -> this is used to retrieve the label

Event Handling with Buttons

a) Each time a button is pressed, an action event is generated.
b) The listener implements the **ActionListener** interface.
c) **ActionListener** defines the **actionPerformed( )** method, which is called when an event occurs.

// Program to demonstrate Button Event handling

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class ButtonDemo extends Applet implements ActionListener{
        String actionName,msg="";
        public void init() {
                Button b1 = new Button("One");
                Button b2 = new Button("Two");

                // add buttons to applet window
                add(b1);
                add(b2);

                b1.addActionListener(this);
                b2.addActionListener(this);
        }

        public void actionPerformed(ActionEvent ae)
        {
                actionName=ae.getActionCommand();
                if(actionName.equals("One"))
                    msg = "Button One is Pressed";
                else
                    msg = "Button Two is Pressed";

                 repaint();
        }
        public void paint(Graphics g)
        {
                g.drawString(msg,20,80);
        }
}
```



3. Check Boxes

- A check box is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not.
- Check boxes can be used individually or as part of a group. Check boxes are objects of the Checkbox class.

constructors:

Checkbox( ) throws HeadlessException
Checkbox(String str) throws HeadlessException
Checkbox(String str, boolean on) throws HeadlessException
Checkbox(String str, boolean on, CheckboxGroup cbGroup) throws HeadlessException
Checkbox(String str, CheckboxGroup cbGroup, boolean on) throws HeadlessException

- ➔ If on is true, the check box is initially checked; otherwise, it is cleared.
- ➔ If thw check box is not part of a group, then cbGroup must be null.

- **getState() ->** Used to retrieve the current state of a check box
- **setState(boolean on) ->** We can set the state.
- **getLabel() ->** obtain the current label associated with a check box
- **setLabel(String str) ->** Used to set the label.

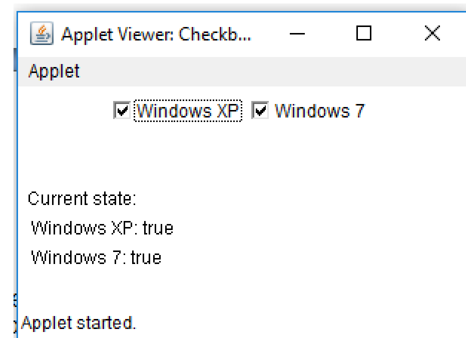
Event Handling with Checkboxes

1) Each time a check box is selected or deselected, an **item event** is generated.
2) Each listener implements the **ItemListener** interface.
3) That interface defines the **itemStateChanged( )** method.

// Demonstrate check boxes.

```
import java.awt.*;

import java.awt.event.*;

import java.applet.*;


public class CheckboxDemo extends Applet implements ItemListener {
        String msg = "";
        Checkbox winXP, win7;
        public void init() {
                winXP = new Checkbox("Windows XP", null, true);
                win7 = new Checkbox("Windows 7");

                add(winXP);
                add(win7);

                winXP.addItemListener(this);
                win7.addItemListener(this);
        }
        public void itemStateChanged(ItemEvent ie) {
```

```
            repaint();
        }
        // Display current state of the check boxes.
        public void paint(Graphics g) {
           msg = "Current state: ";
                g.drawString(msg, 6, 80);
                msg = " Windows XP: " + winXP.getState();
                g.drawString(msg, 6, 100);
                msg = " Windows 7: " + win7.getState();
                g.drawString(msg, 6, 120);
        }
}
```



4. Checkbox Group

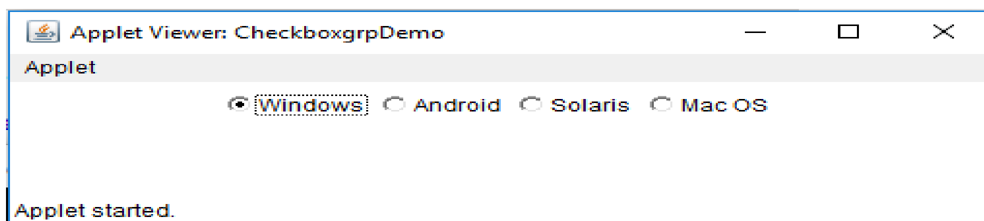
Check box group is used to create **radio buttons**. It is used to create a set of mutually exclusive check boxes.

Check box groups are objects of type **CheckboxGroup**.

- **getSelectedCheckbox( )** -> used to determine which check box in a group is currently selected.
- **setSelectedCheckbox(Checkbox ch)** -> We can set a check box to be selected.

Example:

```
Checkbox windows, android, solaris, mac;
CheckboxGroup cbg;

cbg = new CheckboxGroup();
windows = new Checkbox("Windows", cbg, true);
android = new Checkbox("Android", cbg, false);
solaris = new Checkbox("Solaris", cbg, false);
mac = new Checkbox("Mac OS", cbg, false);

add(windows);
add(android);
add(solaris);
add(mac);
```



Event Handling with Checkbox Groups

1) Each time a check box is selected or deselected, an **item event** is generated.
2) Each listener implements the **ItemListener** interface.
3) That interface defines the **itemStateChanged( )** method.

5. Choice Cntrols

- The Choice class is used to create a *pop-up list of items from which the user may choose.*
- Thus, a Choice control is a form of menu.

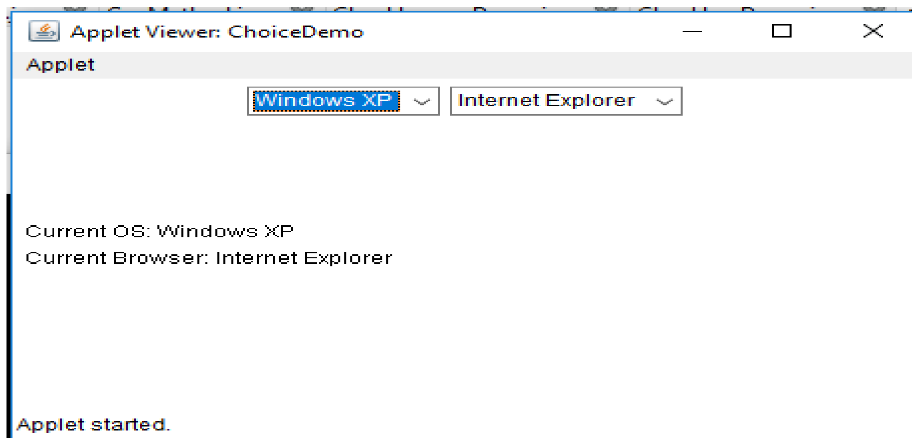
- **getSelectedItem( ) ->** returns a string containing the name of the item
- **int getSelectedIndex( ) ->** returns the index of the item, The first item is at index 0. By default, the first item added to the list is selected.
- **getItemCount( ) ->** obtain the number of items in the list.

Event Handling with Choice control

1) Each time a choice is selected, an item event is generated.
2) Each listener implements the **ItemListener** interface.
3) That interface defines the **itemStateChanged( )** method.

```
// Demonstrate choice control
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class ChoiceDemo extends Applet implements ItemListener {
        Choice os, browser;
        String msg = "";
        public void init() {
                os = new Choice();
                browser = new Choice();
                // add items to os list
                os.add("Windows XP");
                os.add("Windows 7");
                os.add("Solaris");
                os.add("Mac OS");
                // add items to browser list
                browser.add("Internet Explorer");
                browser.add("Firefox");
```

```
                browser.add("Opera");
                // add choice lists to window
                add(os);
                add(browser);
                // register to receive item events
                os.addItemListener(this);
                browser.addItemListener(this);
        }
        public void itemStateChanged(ItemEvent ie) {
                repaint();
        }
// Display current
// Display current selections.
        public void paint(Graphics g) {
                msg = "Current OS: ";
                msg += os.getSelectedItem();
                g.drawString(msg, 6, 120);
                msg = "Current Browser: ";
                msg += browser.getSelectedItem();
                g.drawString(msg, 6, 140);
        }
}
```



6. Lists

The **List class provides a compact, multiple-choice, scrolling selection list.**

Choice	List
Shows only the selected item	Shows any number of choices in the visible window
Allows single selection	Allows multiple selections
Single click generates ItemEvent	Single click generates ItemEvent and doubleclick generates ActionEvent

<u>Constructors</u>

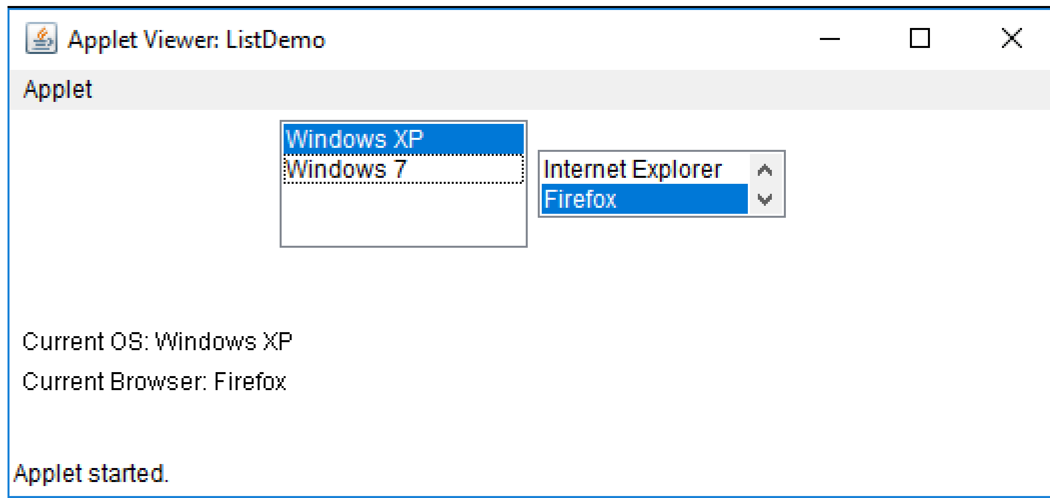
- **List( ) throws HeadlessException ->** The first version creates a List control that allows only one item to be selected at any one time.

- **List(int *numRows) throws HeadlessException* ->** the value of *numRows specifies the number of entries in the list* that will always be visible

- **List(int *numRows, boolean multipleSelect) throws HeadlessException* ->** if *multipleSelect is true, then the user may select two or more items at a time.*

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class ListDemo extends Applet implements ActionListener {
	List os, browser;
	String msg = "";
	public void init() {
		os = new List(4, true);
		browser = new List(2, false);
		// add items to os list
		os.add("Windows XP");
		os.add("Windows 7");
		// add items to browser list
		browser.add("Internet Explorer");
		browser.add("Firefox");
		browser.add("Opera");
		browser.select(1);
		// add lists to window
		add(os);
		add(browser);
		// register to receive action events
		os.addActionListener(this);
		browser.addActionListener(this);
	}

	public void actionPerformed(ActionEvent ae) {
		repaint();
	}
	// Display current selections.
	public void paint(Graphics g) {
		int idx[];
		msg = "Current OS: ";
		idx = os.getSelectedIndexes();
		for(int i=0; i<idx.length; i++)
			msg += os.getItem(idx[i]) + " ";
		g.drawString(msg, 6, 120);
		msg = "Current Browser: ";
```

```
                msg += browser.getSelectedItem();
                        g.drawString(msg, 6, 140);
        }
}
```



7. Scroll bars

- *Scroll bars are used to select continuous values between a specified minimum and maximum*
- Scroll bars may be oriented horizontally or vertically.
- Each end has an arrow that you can click to move the current value of the scroll bar one unit in the direction of the arrow.

Constructors

a) Scrollbar( ) throws HeadlessException -> creates a vertical scroll bar.

b) Scrollbar(int *style) throws HeadlessException*

- If *style is* ***Scrollbar.VERTICAL, a vertical scroll bar is created.***
- If *style is* ***Scrollbar.HORIZONTAL, the scroll bar is horizontal.***

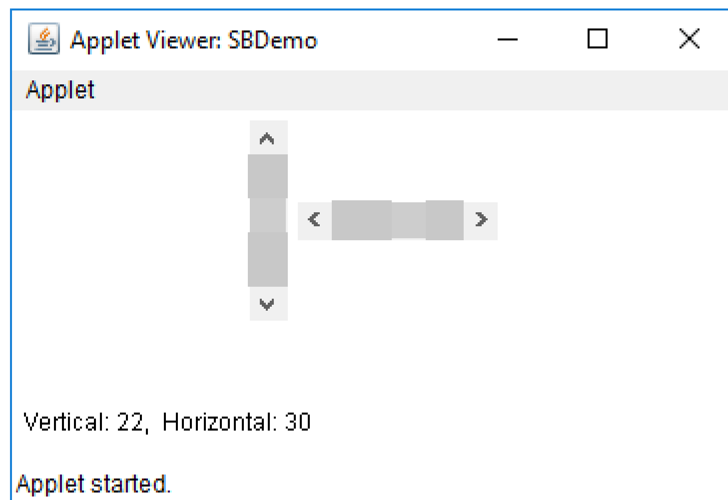
c) Scrollbar(int style, int initialValue, int thumbSize, int min, int max) throws HeadlessException

getValue( ) -> To obtain the current value of the scroll bar.

Event Handling in Scrollbars

- Each time a user interacts with a scroll bar, an **AdjustmentEvent** object is generated.
- To process scroll bar events, you need to implement the **AdjustmentListener** interface.
- **getAdjustmentType( )** method can be used to determine the type of the adjustment.

```
// Demonstrate scroll bars.
import java.awt.*;
import java.awt.event.*;
import java.applet.*; /*     </applet> */

public class SBDemo extends Applet implements AdjustmentListener {
        String msg = "";
        Scrollbar vertSB, horzSB;

        public void init() {

                vertSB = new Scrollbar(Scrollbar.VERTICAL,0, 1, 0, 50);
                vertSB.setPreferredSize(new Dimension(20, 100));

                horzSB = new Scrollbar(Scrollbar.HORIZONTAL,0, 1, 0, 50);
                horzSB.setPreferredSize(new Dimension(100, 20));

                add(vertSB);
                add(horzSB);

                // register to receive adjustment events
                 vertSB.addAdjustmentListener(this);
                 horzSB.addAdjustmentListener(this);
        }

                public void adjustmentValueChanged(AdjustmentEvent ae) {
                        repaint();
                }

                 // Display current value of scroll bars.
                 public void paint(Graphics g) {
                        msg = "Vertical: " + vertSB.getValue();
                        msg += ", Horizontal: " + horzSB.getValue();

                g.drawString(msg, 6, 160);
   }
}
```



8. Text Editing (TextField and TextArea classes)

The TextField class implements a single-line text-entry area, usually called an edit control.

TextField is a subclass of TextComponent.

Constructors

```
TextField( ) throws HeadlessException
TextField(int numChars) throws HeadlessException
TextField(String str) throws HeadlessException
TextField(String str, int numChars) throws HeadlessException
```

Methods

getText() -> Obtain the string currently contained in the text field

getSelectedText() -> returns the selected text.

setEchoChar(char ch) -> We can disable the echoing of the characters as they are typed.

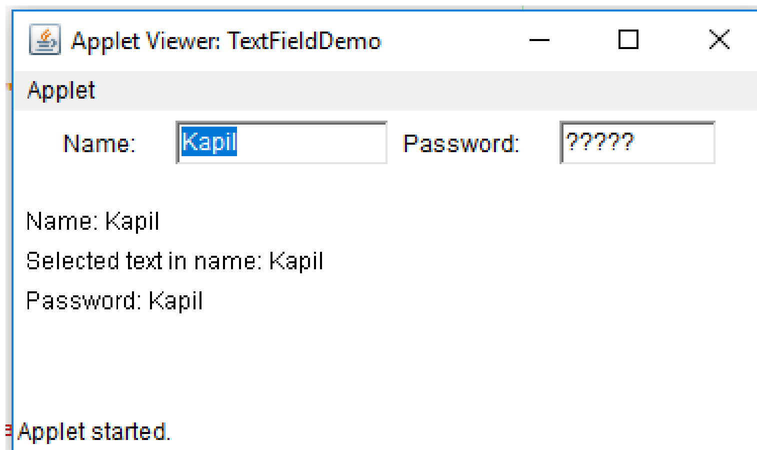
Event Handling TextField

When user presses ENTER, an action event is generated.

Example

```
// Demonstrate text field.
import java.awt.*;
import java.awt.event.*;
```

```
import java.applet.*;

public class TextFieldDemo extends Applet implements ActionListener {

  TextField name, pass;

  public void init() {
          Label namep = new Label("Name: ");
          Label passp = new Label("Password: ");
          name = new TextField(12);
          pass = new TextField(8);
          pass.setEchoChar('?');

    add(namep);
    add(name);
    add(passp);
    add(pass);

    // register to receive action events
    name.addActionListener(this);
    pass.addActionListener(this);  }

          // User pressed Enter.
          public void actionPerformed(ActionEvent ae) {
                    repaint();
          }

          public void paint(Graphics g) {
                    g.drawString("Name: " + name.getText(), 6, 60);
                    g.drawString("Selected text in name: "+ name.getSelectedText(), 6, 80);
                    g.drawString("Password: " + pass.getText(), 6, 100);
          }
}
```



TextArea

It is a multiline editor. TextArea is a subclass of TextComponent.

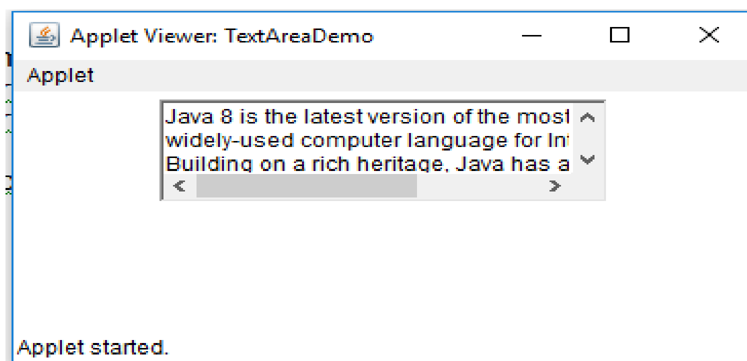
Constructors:

TextArea( ) throws HeadlessException
TextArea(int numLines, int numChars) throws HeadlessException
TextArea(String str) throws HeadlessException
TextArea(String str, int numLines, int numChars) throws HeadlessException
TextArea(String str, int numLines, int numChars, int sBars) throws HeadlessException

Methods

- **append(String str)** -> appends the string specified by str to the end of the current text.
- **insert(String str, int index)** -> Inserts the string passed in str at the specified index.
- **void replaceRange(String str, int startIndex, int endIndex)** -> It replaces the characters from startIndex to endIndex–1, with the replacement text passed in str.

```
// Demonstrate TextArea.
import java.awt.*;
import java.applet.*;
public class TextAreaDemo extends Applet {
        public void init() {
                String val =    "Java 8 is the latest version of the most \n" +
                        "widely-used computer language for Internet programming.\n" +
                        "Building on a rich heritage, Java has advanced both \n" +
                        "the art and science of computer language design. \n\n";

                TextArea text = new TextArea(val, 3, 30);
                add(text);
        }
}
```



Layout Managers

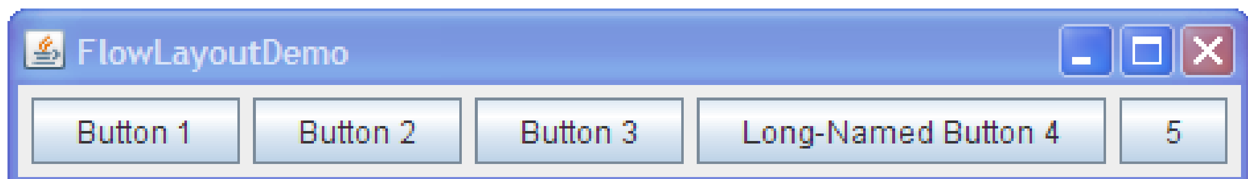
- A layout manager automatically arranges your controls within a window by using some type of algorithm.
- Each **Container object has a layout manager associated with it.**
- A layout manager is an instance of any class that implements the **LayoutManager interface.**
- The layout manager is set by the **setLayout(LayoutManager lmg) method.**
- Layout managers basically do two things:
  - Calculate the minimum/preferred/maximum sizes for a container.
  - Lay out the container's children.

There are 5 layouts supported by AWT:

1. FlowLayout
2. BorderLayout
3. GridLayout
4. GridbagLayout
5. CardLayout

Flow Layout

- The FlowLayout is used to arrange the components in a line, one after another.
- It simply lays out components in a single row, starting a new row if its container is not sufficiently wide.



Constructors

FlowLayout( )

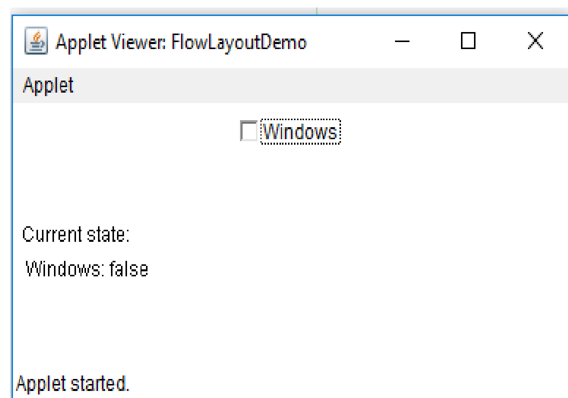
FlowLayout(int how)

FlowLayout(int how, int horz, int vert)

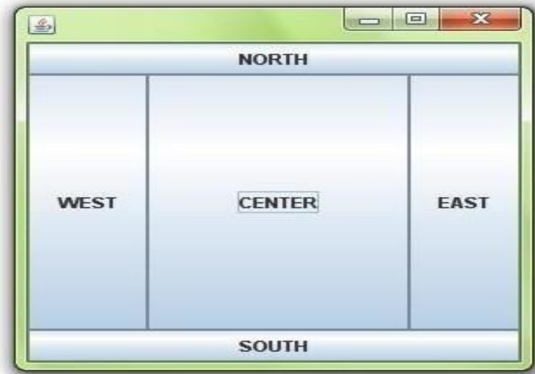
<u>Values for how :</u> FlowLayout.LEFT, FlowLayout.CENTER, FlowLayout.RIGHT
FlowLayout.LEADING ,FlowLayout.TRAILING

<u>FlowLayout Example:</u>

```
// Use left-aligned flow layout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class FlowLayoutDemo extends Applet implements ItemListener {
    String msg = "";
    Checkbox windows, android, solaris, mac;
     public void init() {
         // set left-aligned flow layout
        setLayout(new FlowLayout(FlowLayout.CENTER));
        windows = new Checkbox("Windows");
        add(windows);
          // register to receive item events
         windows.addItemListener(this);
 }
// Repaint when status of a check box changes.
 public void itemStateChanged(ItemEvent ie) {
          repaint();
 }
 // Display current state of the check boxes.
 public void paint(Graphics g) {
          msg = "Current state: ";
          g.drawString(msg, 6, 80);
          msg = " Windows: " + windows.getState();
          g.drawString(msg, 6, 100);
 }
}
```



BorderLayout



➢ The **BorderLayout** is used to arrange the components in five regions: north, south, east, west and center.

BorderLayout – constructors

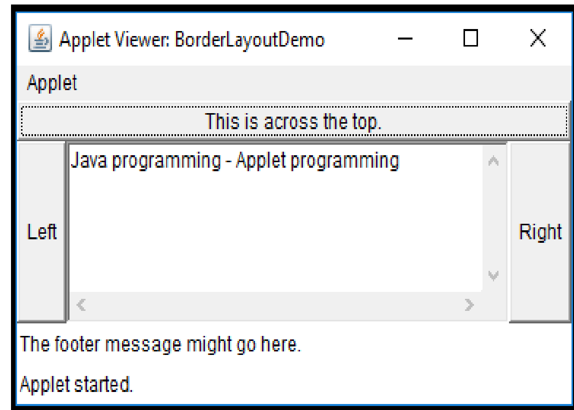
1. **BorderLayout( )** -> creates a default border layout
2. **BorderLayout(int horz, int vert)** -> allows to specify the horizontal and vertical space left between components in horz and vert, respectively.
3. BorderLayout defines the following constants that specify the regions:
   - BorderLayout.CENTER
   - BorderLayout.SOUTH
   - BorderLayout.EAST
   - BorderLayout.WEST
   - BorderLayout.NORTH

Adding the components to a BorderLauout

void add(Component compRef, Object region) -> region specifies where the component will be added.

```
// Demonstrate BorderLayout.
import java.awt.*;
import java.applet.*;
import java.util.*;

public class BorderLayoutDemo extends Applet {
 public void init() {
        setLayout(new BorderLayout());
        add(new Button("This is across the top."),BorderLayout.NORTH);
        add(new Label("The footer message might go here."),BorderLayout.SOUTH);
        add(new Button("Right"), BorderLayout.EAST);
```

```
  add(new Button("Left"), BorderLayout.WEST);

  String msg = "Java programming - " +
                    "Applet programming" +
                    "\n";
 add(new TextArea(msg), BorderLayout.CENTER);
 }
}
```



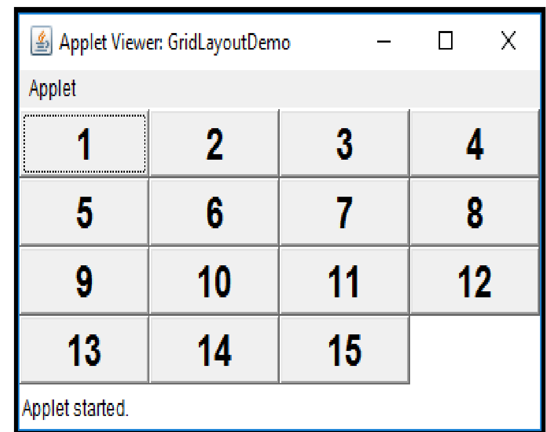
GridLayout

- GridLayout simply makes a bunch of components equal in size and displays them in the requested number of rows and columns.
- GridLayout lays out components in a two-dimensional grid.
- When we instantiate a GridLayout, we define the number of rows and columns.

GridLayout Constructors

1. **GridLayout( )** -> creates a single-column grid layout.
2. **GridLayout(int numRows, int numColumns)** -> creates a grid layout with the specified number of rows and columns.
3. **GridLayout(int numRows, int numColumns, int horz, int vert)** -> specify the horizontal and vertical space left between components in horz and vert, respectively.

Example: A sample program that creates a 4×4 grid and fills it in with 15 buttons, each labeled with its index:




```
import java.awt.*;
import java.applet.*;
public class GridLayoutDemo extends Applet {
    static final int n = 4;
    public void init() {
        setLayout(new GridLayout(n, n));
        setFont(new Font("SansSerif", Font.BOLD, 24));
        for(int i = 0; i < n; i++) {
          for(int j = 0; j < n; j++) {
              int k = i * n + j;
              if(k > 0)
                  add(new Button("" + k));
        }
      }
    }
}
```

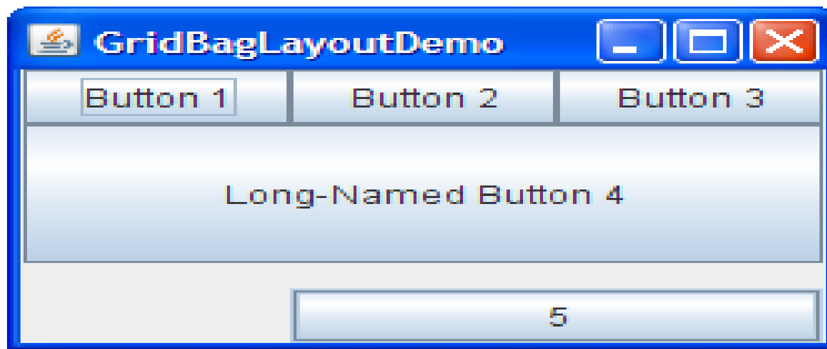
GridBagLayout

- GridBagLayout is a sophisticated, flexible layout manager.
- The rows in the grid can have different heights, and grid columns can have different widths.
- The key to the grid bag is that each component can be a different size, and each row in the grid can have a different number of columns.
- The location and size of each component in a grid bag are determined by a set of constraints linked to it.
- The constraints are contained in an object of type **GridBagConstraints**

GridBagLayout – constructor

GridBagLayout defines only one constructor,

GridBagLayout( )



GridBagLayout – constraints

GridBagConstraints.CENTER

GridBagConstraints.SOUTH

GridBagConstraints.EAST

GridBagConstraints.SOUTHEAST

GridBagConstraints.NORTH

GridBagConstraints.SOUTHWEST

GridBagConstraints.NORTHEAST

GridBagConstraints.WEST

GridBagConstraints.NORTHWEST

int anchor -> Specifies the location of a component within a cell. The default is GridBagConstraints.CENTER.

int gridheight -> Specifies the height of component in terms of cells. The default is 1.

int gridwidth -> Specifies the width of component in terms of cells. The default is 1

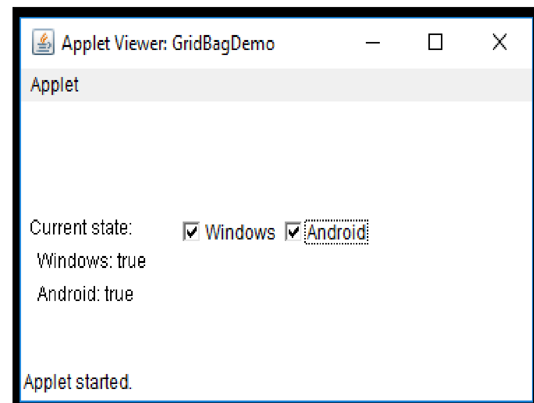
int gridx -> Specifies the X coordinate of the cell to which the component will be added

int gridy -> Specifies the Y coordinate of the cell to which the component will be added.

Example:

```
// Use GridBagLayout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class GridBagDemo extends Applet implements ItemListener {

        String msg = "";
        Checkbox windows, android, solaris, mac;
```

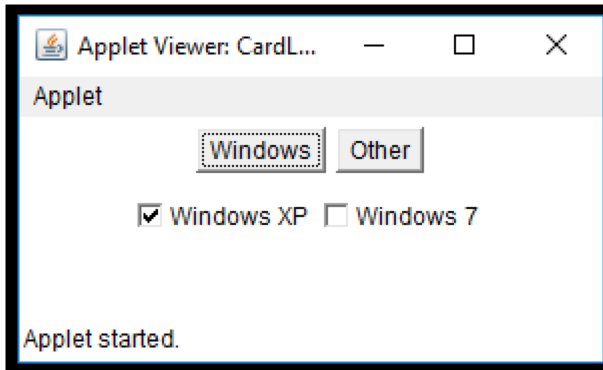
```
        public void init() {
                GridBagLayout gbag = new GridBagLayout();
                GridBagConstraints gbc = new GridBagConstraints();
                setLayout(gbag);

                // Define check boxes.
                windows = new Checkbox("Windows ", null, true);
                android = new Checkbox("Android");

                gbc.anchor = GridBagConstraints.NORTHEAST;
                gbc.gridwidth = GridBagConstraints.RELATIVE;
                gbag.setConstraints(windows, gbc);

                 gbc.gridwidth = GridBagConstraints.REMAINDER;
                 gbag.setConstraints(android, gbc);

                  // Add the components.
                  add(windows);
                  add(android);

                   // Register to receive item events.
                   windows.addItemListener(this);
                   android.addItemListener(this);
        }

        // Repaint when status of a check box changes.
        public void itemStateChanged(ItemEvent ie) {
                        repaint();
        }

         // Display current state of the check boxes.
         public void paint(Graphics g) {
                 msg = "Current state: ";
                 g.drawString(msg, 6, 80);
                 msg = " Windows: " + windows.getState();
                 g.drawString(msg, 6, 100);
                 msg = " Android: " + android.getState();
                 g.drawString(msg, 6, 120);
        }
}
```

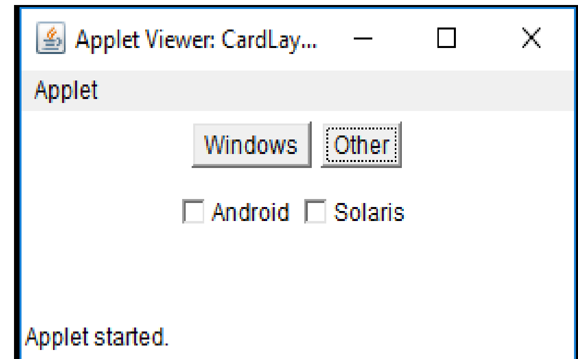


CardLayout

The CardLayout class is used to implement an area that contains different components at different times.

It is similar to deck of cards.





Example:

// Demonstrate CardLayout with the output shown above

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class CardLayoutDemo extends Applet implements ActionListener{
        Checkbox windowsXP, windows7, windows8, android, solaris, mac;
        Panel osCards;
        CardLayout cardLO;
        Button Win, Other;
        public void init() {
                Win = new Button("Windows");
                Other = new Button("Other");
                add(Win);
                add(Other);
                cardLO = new CardLayout();
                osCards = new Panel();
                osCards.setLayout(cardLO); // set panel layout to card layout
                windowsXP = new Checkbox("Windows XP", null, true);
                windows7 = new Checkbox("Windows 7", null, false);
                android = new Checkbox("Android");
                solaris = new Checkbox("Solaris");
                // add Windows check boxes to a panel
                Panel winPan = new Panel();
                winPan.add(windowsXP);
```

```
                winPan.add(windows7);
                // Add other OS check boxes to a panel
                Panel otherPan = new Panel();
                otherPan.add(android);
                otherPan.add(solaris);
                // add panels to card deck panel
                osCards.add(winPan, "Windows");
                osCards.add(otherPan, "Other");
            // add cards to main applet panel
            add(osCards);
            // register to receive action events
            Win.addActionListener(this);
            Other.addActionListener(this);
    }
     public void actionPerformed(ActionEvent ae) {
             if(ae.getSource() == Win) {
                     cardLO.show(osCards, "Windows");
            }
            else {
                     cardLO.show(osCards, "Other");
            }
    }
}
```

CardLayout – Methods

1) **void first(Container deck)** -> the first card in the deck will be shown
2) **void last(Container deck)** -> the last card in the deck will be shown
3) **void next(Container deck)** -> the next card in the deck will be shown
4) **void previous(Container deck)** -> the previous card in the deck will be shown
5) **void show(Container deck, String cardName)** -> displays the card whose name is passed in cardName

Menu Bars and Menus

A top-level window can have a menu bar associated with it. A menu bar displays a list of top-level menu choices. Each choice is associated with a drop-down menu. This concept is implemented in the AWT by the following classes: **MenuBar, Menu**, and **MenuItem.**

- To create a menu bar, first create an instance of **MenuBar.**

  // create menu bar and add it to frame

  MenuBar mbar = new MenuBar();

  setMenuBar(mbar)

- Next, create instances of **Menu** that will define the selections displayed on the bar.

  // create the menu items

  Menu file = new Menu("File");

- Next, create individual menu items are of type **MenuItem**

  MenuItem item1, item2, item3, item4, item5;

  file.add(item1 = new MenuItem("New"));

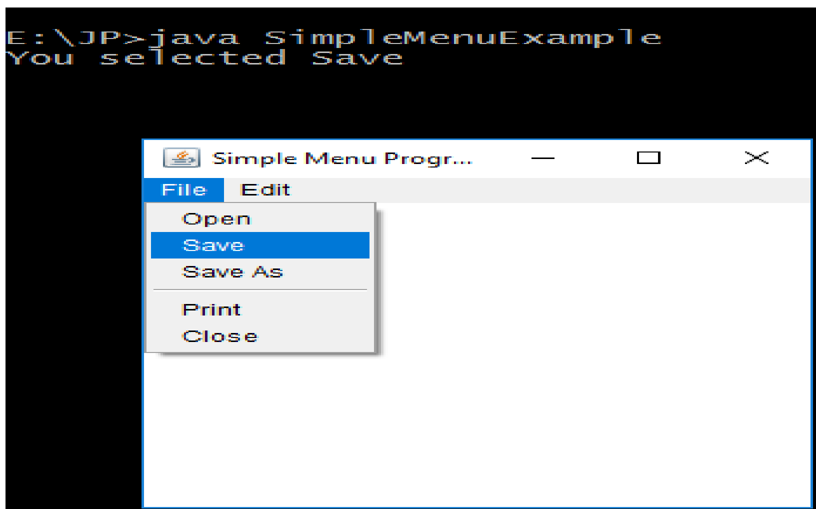
  file.add(item2 = new MenuItem("Open"));

Example

Create a Sample Menu Program

```
import java.awt.*;
import java.awt.event.*;

public class SimpleMenuExample extends Frame implements ActionListener
{
  Menu file, edit;
  public SimpleMenuExample()
  {
    MenuBar mb = new MenuBar();                    // begin with creating menu bar
    setMenuBar(mb);                              // add menu bar to frame

    file = new Menu("File");               // create menus
    edit = new Menu("Edit");

    mb.add(file);                                // add menus to menu bar
    mb.add(edit);

    file.addActionListener(this);                // link with ActionListener for event handling
    edit.addActionListener(this);
    file.add(new MenuItem("Open"));
```

```
    file.add(new MenuItem("Save"));
    file.add(new MenuItem("Save As"));
    file.addSeparator();
    file.add(new MenuItem("Print"));
    file.add(new MenuItem("Close"));

    edit.add(new MenuItem("Cut"));
    edit.add(new MenuItem("Copy"));
    edit.add(new MenuItem("Paste"));
    edit.addSeparator();
    edit.add(new MenuItem("Special"));
    edit.add(new MenuItem("Debug"));

    setTitle("Simple Menu Program");           // frame creation methods
    setSize(300, 300);
    setVisible(true);

    //closing the window
          addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e) {
           dispose();
        }
      });
  }
  public void actionPerformed(ActionEvent e)
  {
    String str = e.getActionCommand();              // know the menu item selected by the user
    System.out.println("You selected " + str);
  }
  public static void main(String args[])
  {
    new SimpleMenuExample();
  }
}
```



Dialog Boxes

- We use a dialog box to hold a set of related controls.
- Dialog boxes are primarily used to obtain user input and are often child windows of a top-level window.
- Dialog boxes don’t have menu bars.
- Dialog boxes are two types : **modal** or **modeless**.
- **Modal dialog box :** In this, other parts of the program can not be accessible while the dialog box is active
- **Modeless dialog box:** In this, other parts of the program can be accessible while the dialog box is active.

Constructors:

Dialog(Frame parentWindow, boolean mode)

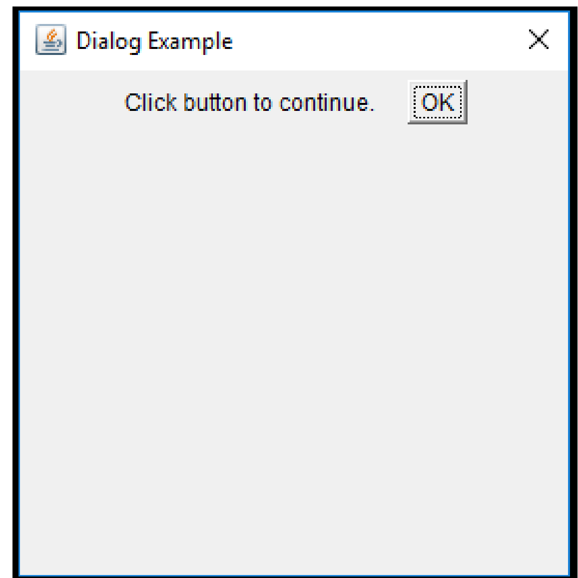
Dialog(Frame parentWindow, String title, boolean mode)

- If mode is true, the dialog box is modal
- If mode is false, the dialog box is modeless

Example:

```
import java.awt.*;
import java.awt.event.*;
public class DialogExample {
    private static Dialog d;
    DialogExample() {
        Frame f= new Frame();
        d = new Dialog(f , "Dialog Example", true);
        d.setLayout( new FlowLayout() );
        Button b = new Button ("OK");
        b.addActionListener ( new ActionListener()
        {
            public void actionPerformed( ActionEvent e )
            {
                DialogExample.d.setVisible(false);
            }
```

```
        });
        d.add( new Label ("Click button to continue."));
        d.add(b);
        d.setSize(300,300);
        d.setVisible(true);
    }
    public static void main(String args[])
    {
        new DialogExample();
    }
}
```



FileDialog

- Java provides a built-in dialog box that lets the user specify a file.
- To create a file dialog box, instantiate an object of type FileDialog. This causes a file dialog box to be displayed.

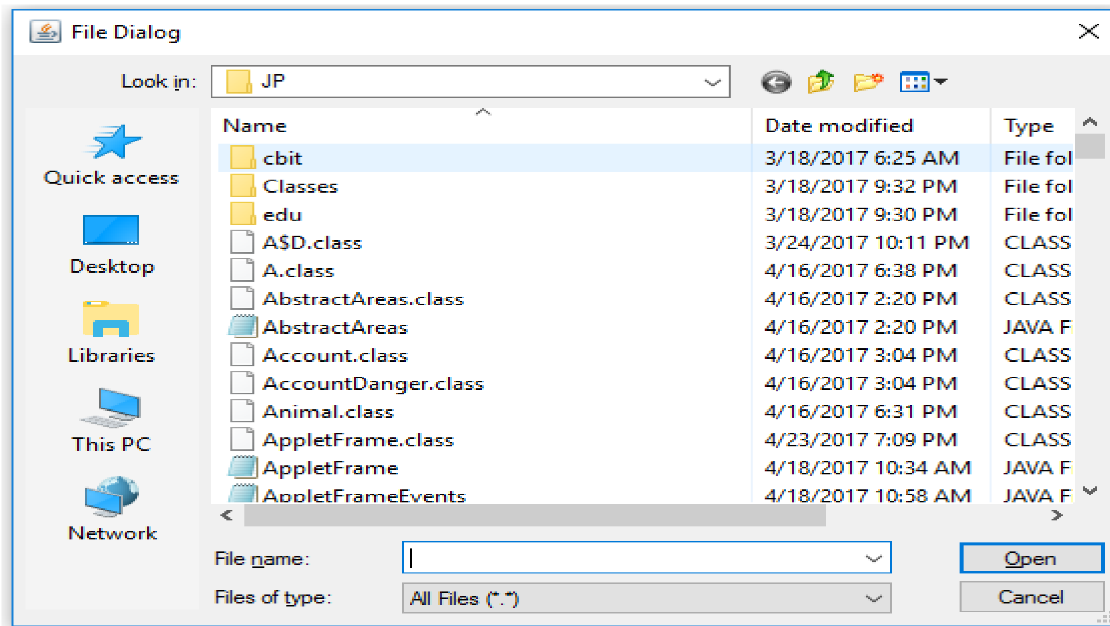
Constructors:

FileDialog(Frame parent)

FileDialog(Frame parent, String boxName)

FileDialog(Frame parent, String boxName, int ***how***)

- If ***how*** is **FileDialog.LOAD**, then the box is selecting a file for reading.
- If ***how*** is **FileDialog.SAVE**, the box is selecting a file for writing.
- If ***how*** is **omitted**, the box is selecting a file for reading.



EVENTS – EVENTS SOURCES – EVENTS LISTENERS

SOURCE	Event Class	Listener	Listener Methods
MOUSE	MouseEvent	MouseListener, MouseMotionListener	mouseClicked(),mouseEntered(), mouseExited(),mousePressed() mouseReleased(), mouseDragged(),mouseMoved()
KEYBOARD	KeyEvent	KeyListener	keyPressed(),keyTyped()
BUTTON	ActionEvent	ActionListener	actionPerformed()
CHECKBOX	ItemEvent	ItemListener	itemStateChanged()
LIST	ItemEvent	ItemListener	itemStateChanged()
CHOICE	ItemEvent	ItemListener	itemStateChanged()
MENUITEM	ActionEvent	ActionListener	actionPerformed()
TEXTFIELD or TEXTAREA	ActionEvent	ActionListener	actionPerformed()
SCROLLBAR	AdjustmentEvent	AdjustmentListener	adjustmentValueChanged()