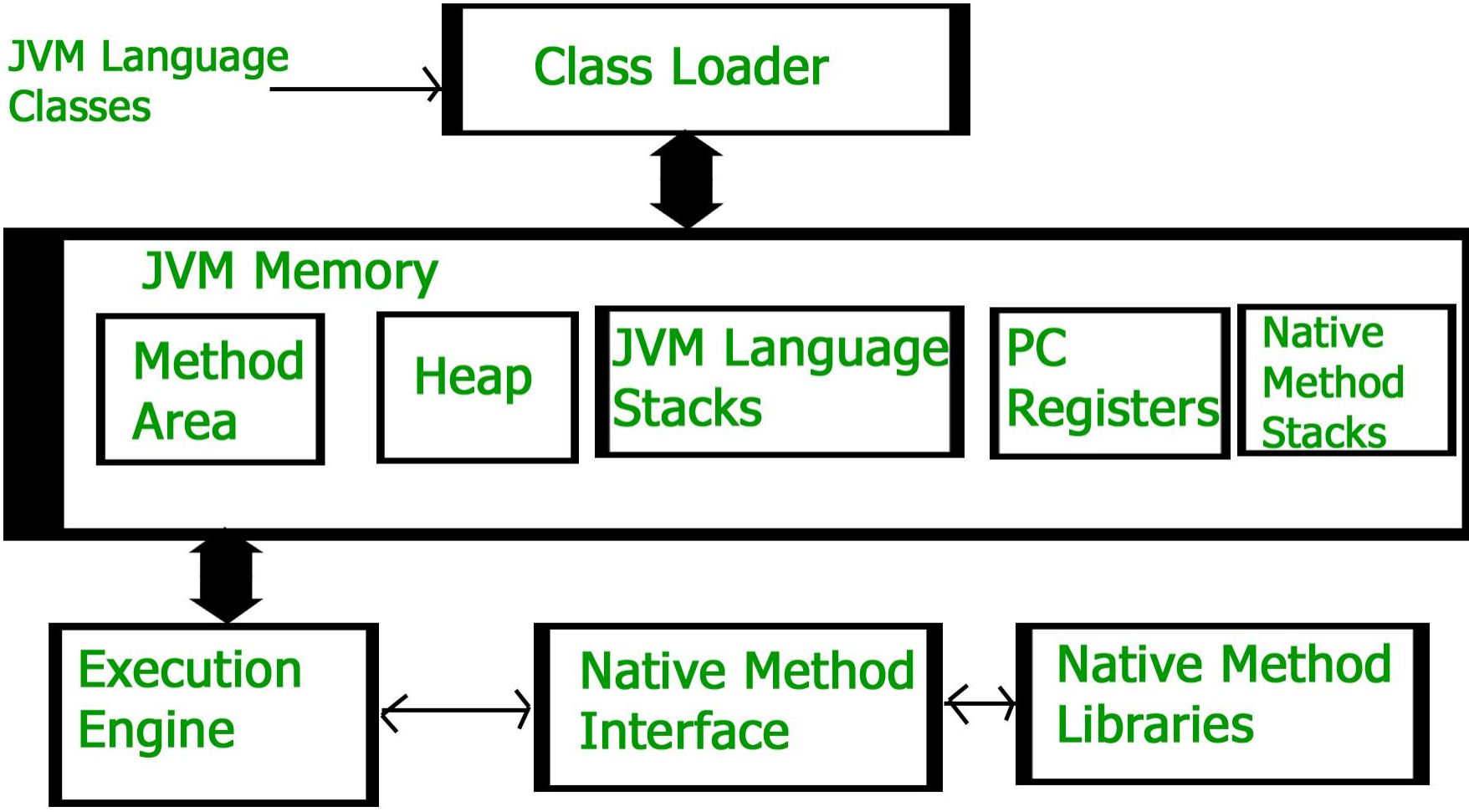
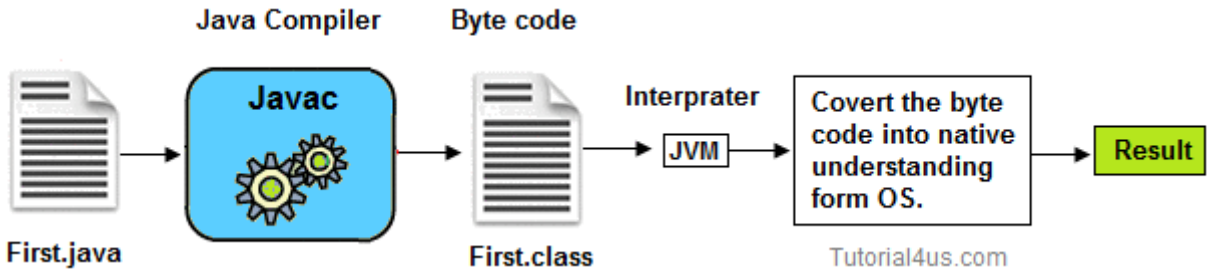
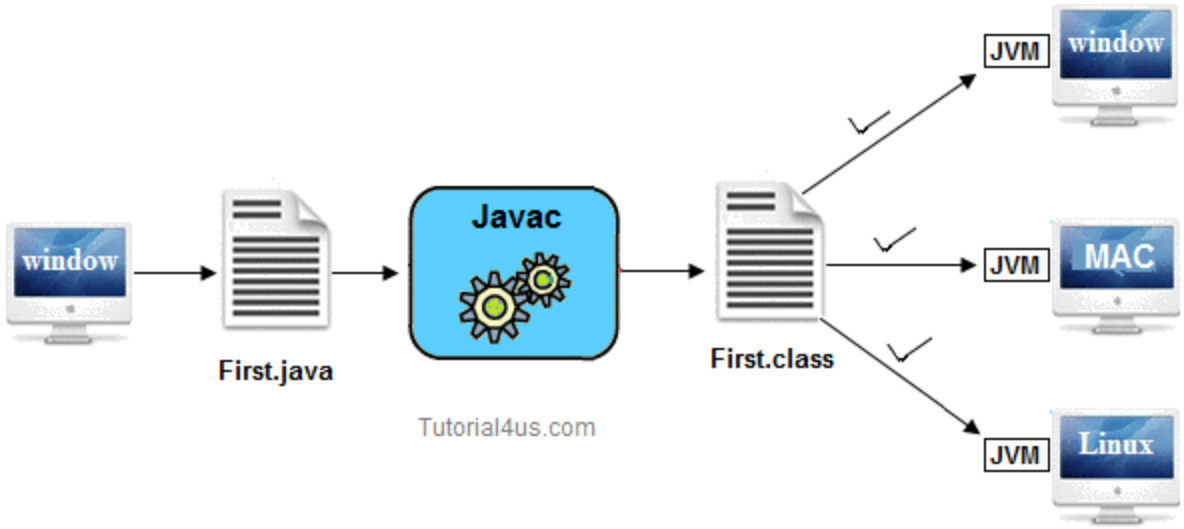
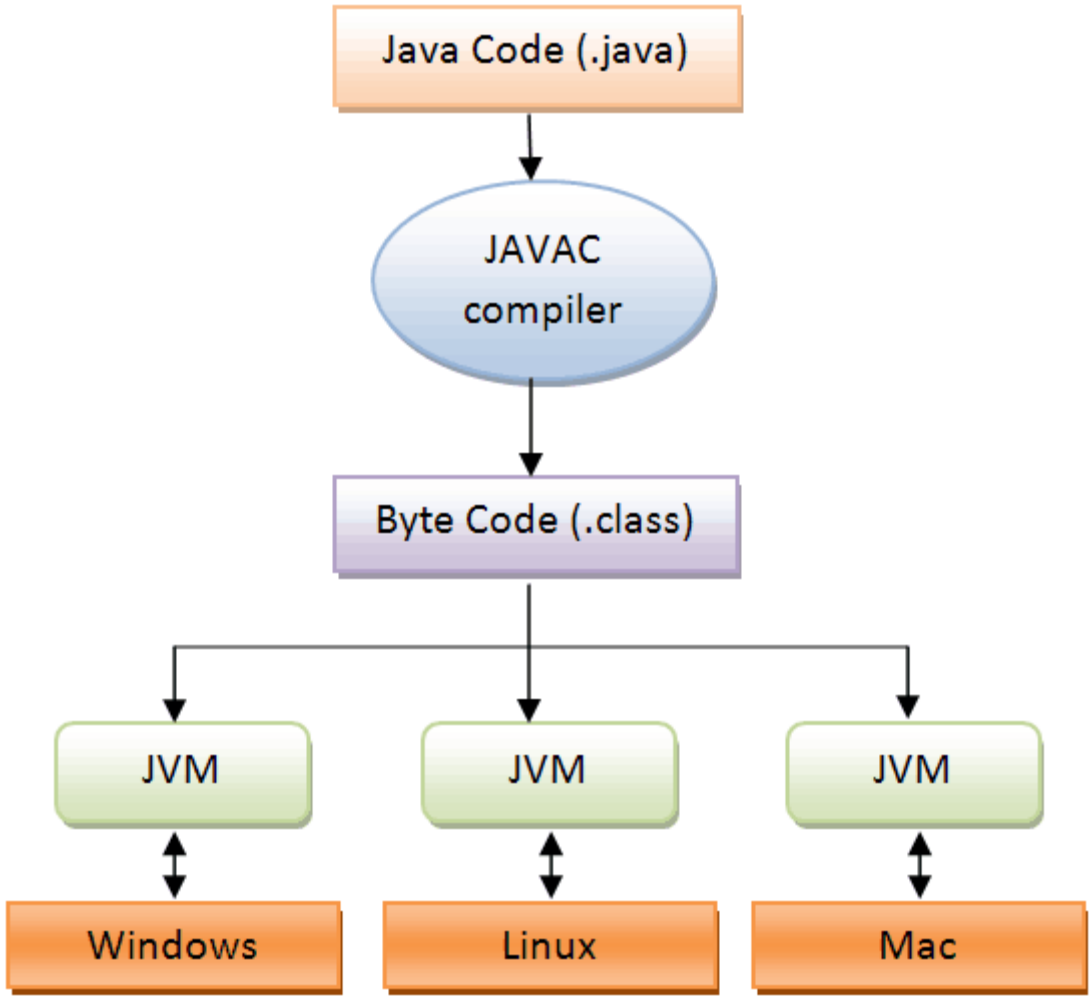
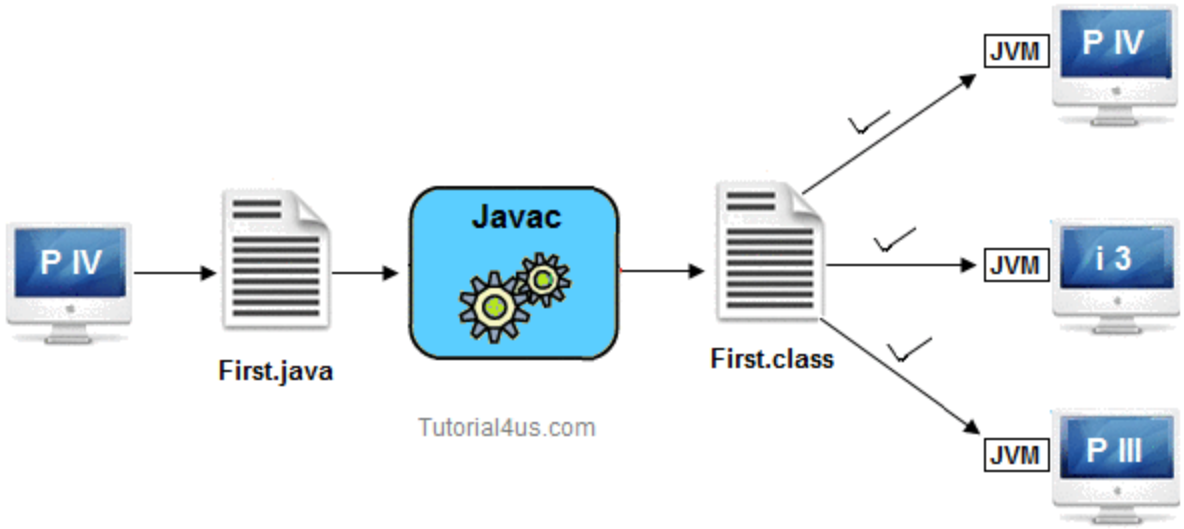
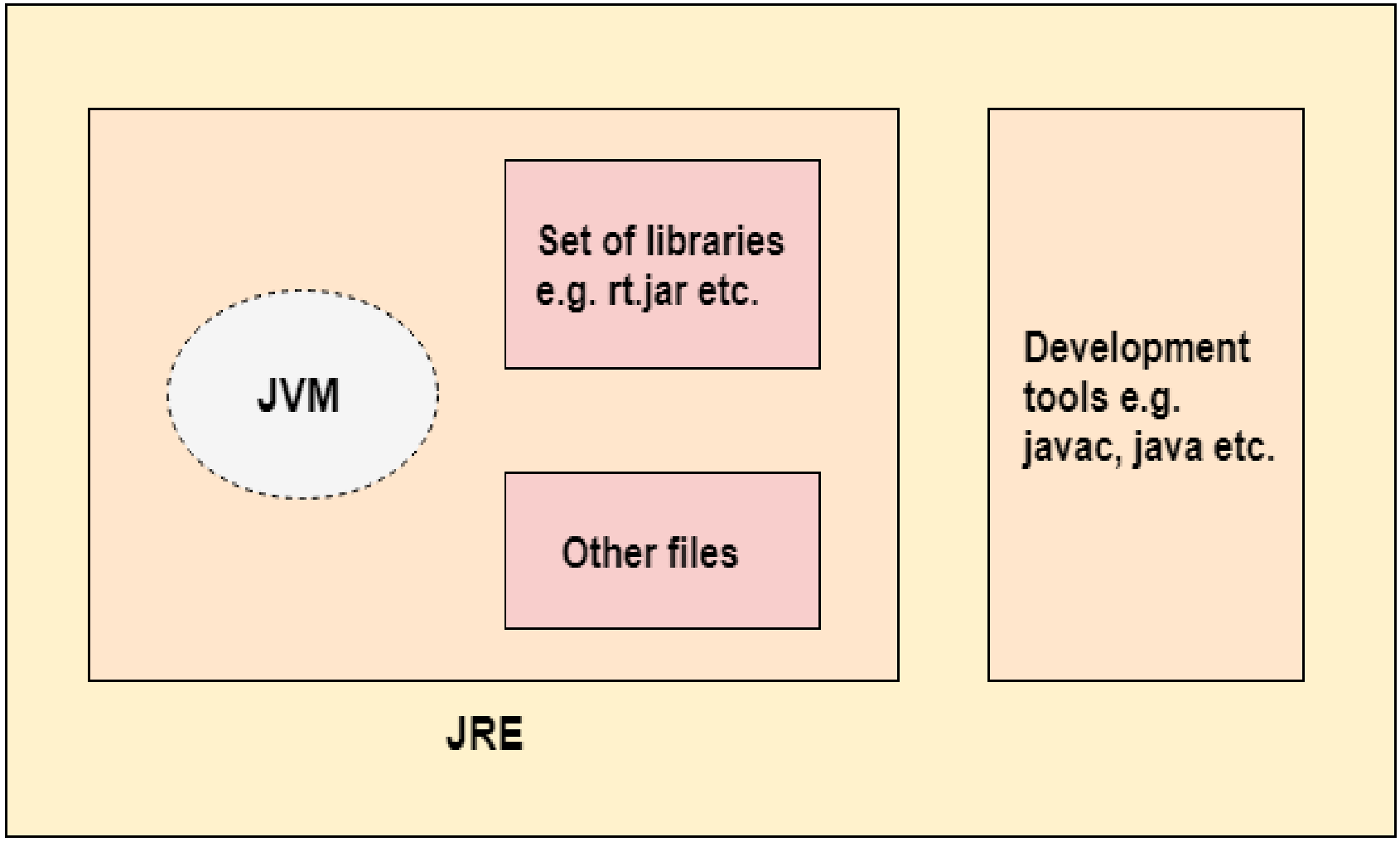

JVM Language Classes
Class Loader
JVM Memory
Method Area
Heap
JVM Language Stacks
PC Registers
Native Method Stacks
Execution Engine
Native Method Interface
Native Method Libraries


Java Compiler
Byte code
Javac
Interprater
JVM
Covert the byte code into native understanding form OS.
Result
First.java
First.class
Tutorial4us.com


window
First.java
Javac
First.class
JVM
window
JVM
MAC
JVM
Linux
Tutorial4us.com


Java Code (.java)
JAVAC compiler
Byte Code (.class)
JVM
JVM
JVM
Windows
Linux
Mac


P IV
First.java
Javac
First.class
JVM
P IV
JVM
i 3
JVM
P III
Tutorial4us.com


Set of libraries
e.g. rt.jar etc.
JVM
Other files
Development
tools e.g.
javac, java etc.
JRE
JDK

Core Java

Introduction

- Java is a high level programming language and a platform.
- JAVA is a object oriented programming language with a built-in application programming interface (API) and also have support to handle graphics user interfaces.
- Syntax of Java is inspired by C and C++
- It’s more Platform Independent
- It has a vast library of predefined objects and operations
- Java manages the memory allocation and de-allocation for creating new objects with the garbage collector in it.

Types of java applications

- Standalone application

  desktop application or window based application
- Web application

  application which runs on server side and create dynamic pages
- Enterprise application

  application which is distributed widely
- Mobile application

  which is created for mobile devices.

Creation of java

- **James Gosling**, Mike Sheridan, and Patrick Naughton - Sun Microsystems(1991)- group is called greentalk
- Initially it was called- oak
- Oak to Java on May 20, 1995, Sun World
- Why the name Java?
- **JDK Evolutions**
- JDK 1.0 (January 21, 1996)
- JDK 1.1 (February 19, 1997)
- J2SE 1.2 (December 8, 1998)
- J2SE 1.3 (May 8, 2000)
- J2SE 1.4 (February 6, 2002)
- J2SE 5.0 (September 30, 2004)
- Java SE 6 (December 11, 2006)
- Java SE 7 (July 28, 2011)
- Java SE 8 (March 18, 2014)

How is Java different from C++…

Features removed in java:

- Java doesn't support pointers to avoid unauthorized access of memory locations.
- Java does not include structures, unions data types.
- Java does not support operator over loading.
- Preprocessor plays less important role in C++ and so eliminated entirely in java.
- Java does not perform automatic type conversions that result in loss of precision.

How is Java different from C++…

- Java does not support global variables. Every method and variable is declared within a class and forms part of that class.
- Java does not support inheritance of multiple super classes by a sub class (i.e., multiple inheritance). This is accomplished by using 'interface' concept.
- In java objects are passed by reference only. In C++ objects may be passed by value or reference.

How is Java different from C++…

New features added in Java:

- Multithreading, that allows two or more pieces of the same program to execute concurrently.
- C++ has a set of library functions that use a common header file. But java replaces it with its own set of API classes.
- It adds packages and interfaces.
- Java supports automatic garbage collection.
- The use of unicode characters ensures portability.

How is Java different from C++…

Features that differ:

- Though C++ and java supports Boolean data type, C++ takes any nonzero value as true and zero as false. **True** and **false** in java are predefined literals that are values for a boolean expression.

- C++ supports exception handling that is similar to java's. However, in C++ there is no requirement that a thrown exception be caught.

Characteristics(features)of Java

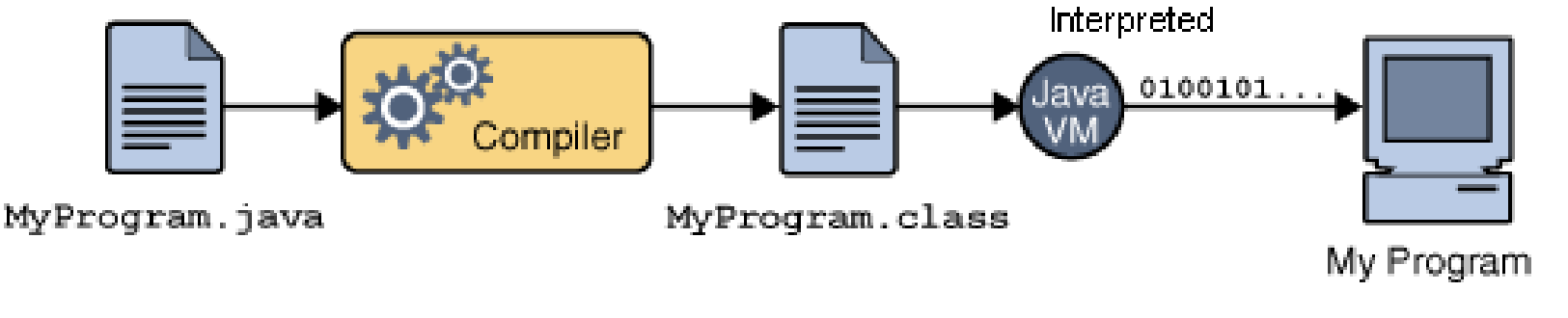
- Java is simple
- Java is object-oriented
- Java is platform independent ,architectural neutral and so portable
- Java is multithreaded
- Java is robust , secure
- Java is dynamic, distributed

Simple , Object oriented

- Java is a simple . Because java has the more library function than c/c++ languages.
- Java syntax is almost similar to c/c++ syntaxes.
- Java developers omitted the concepts of pointers (because it has to be work on electronic devices, where less memory is available)
- The Java is purely object oriented language(almost). (even integers, files etc)
- Java comes with an extensive set of classes arranged in packages

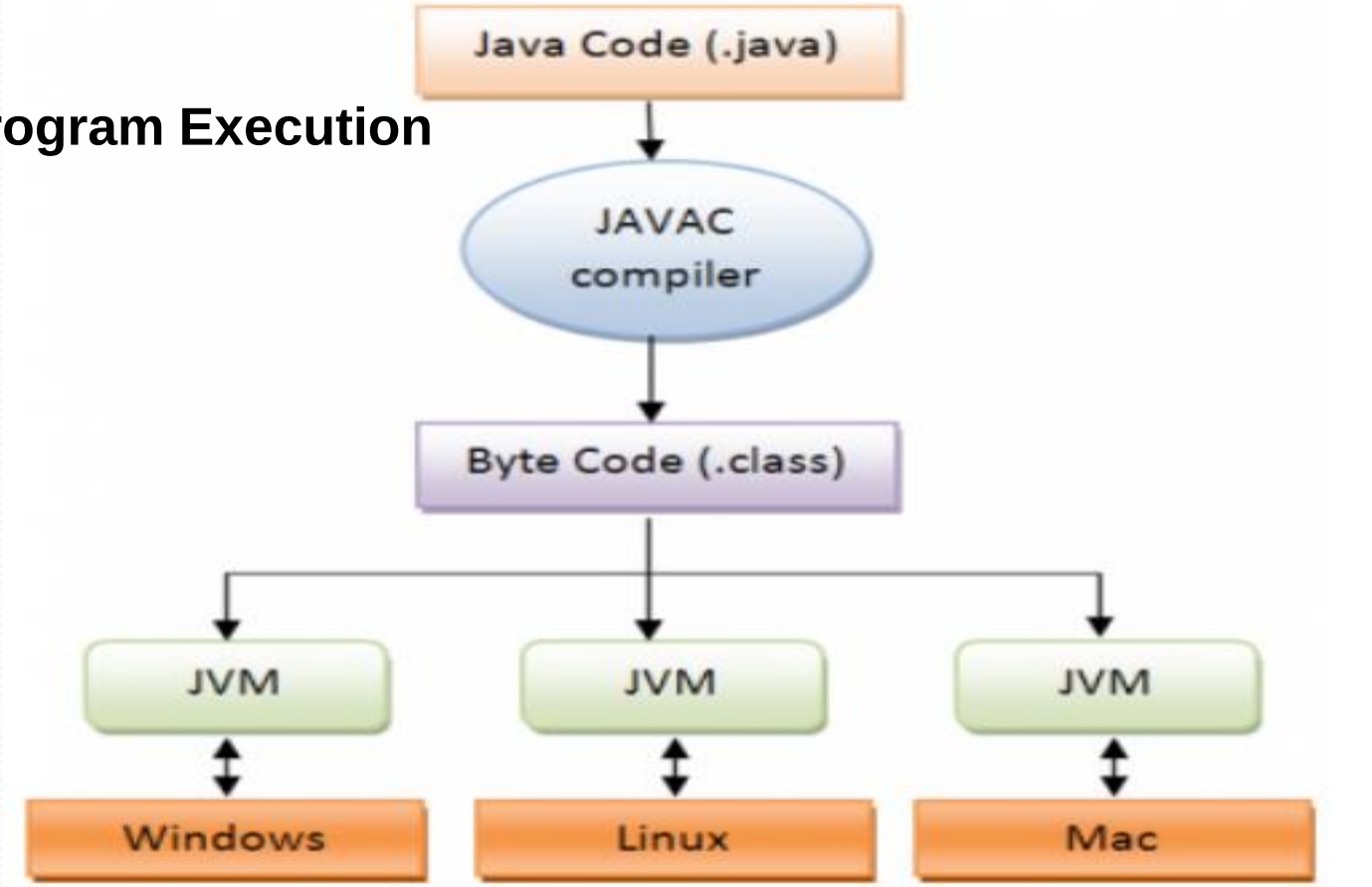
Platform independent

- Java Program will be undergoing two steps:
- In First stage java compiler translates the source code into byte code instructions
- Byte codes are not machine instructions
- In second stage java interpreter generates the machine codes.
- That can be directly executed by the machine
- WORA(Write Once Run Anywhere)



Java is architecture-neutral

JAVA Program Execution



Robust ,Secure

- Absence of pointers ensures that **No pointing** to invalid memory location
- It provides many safeguards to ensure reliable code
- Java has no undefined states
- In java we have **exception handling** to catch and handle the errors at run time.
- It has strict compile time and runtime check for data types
- It runs in its own runtime environment, so null interaction with OS.

Dynamic and Extensible

- Java is a dynamic language
- Java is capable of dynamically linking new
  - Class libraries
  - Methods
  - Objects
- Java programs supports functions written in other languages such as C and C++
  - These functions are known as “native methods”
  - Native methods are dynamically linked at runtime

Multi threaded and Distributed

- Multi threaded: it lets to reduce the CPU time
- Allows Paralelly executing the 2 parts of the execution which are functionally independent.
- Java is designed as a distributed language for creating applications on networks
- It Enables multiple programmers at multiple locations work together on a single project
- It has the ability to share both data and programs

Important Features

- **Packages-** Directories in file system
- Collection of set of related class files
- **Interfaces-** Pure abstract class types
- Used for dynamic binding
- Nothing defined for interfaces
- **Inheritance-** supports single multi level inheritance, but not multiple inheritance.
- **Encapsulation-** putting together all variables and methods in the class.

Important Features

- **Polymorphism-** many forms

  2 types

  Static binding- checked at compile time

  EX: method overloading

  Dynamic binding- checked during run time.

  EX: Particular objects acts differently

  Depending on the reference passed to it.

- **Exception** handling

- Some runtime errors can be handled by java

- **Threads**

- Applications and applets:
- Applications- are stand alone programs, which starts at the particular point- **main()** function
- Applets- web client which runs on the browser.
- Starts at **init()** function.

Editions

- Everyone who want to work with JAVA should need to know about all Editions provided by JAVA

- Mainly , JAVA provide these editions

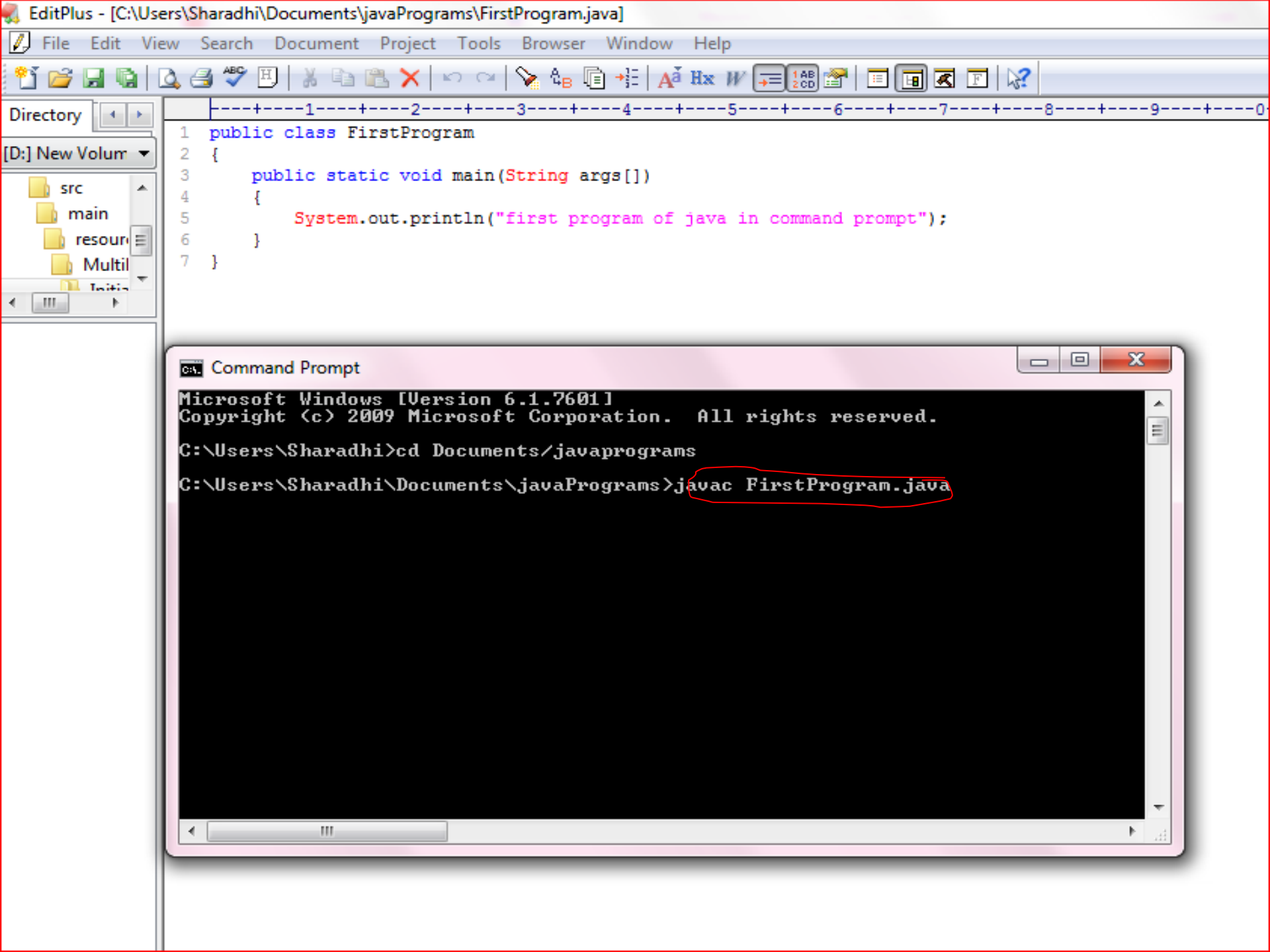
    - JAVA Standard Edition a.k.a J2SE

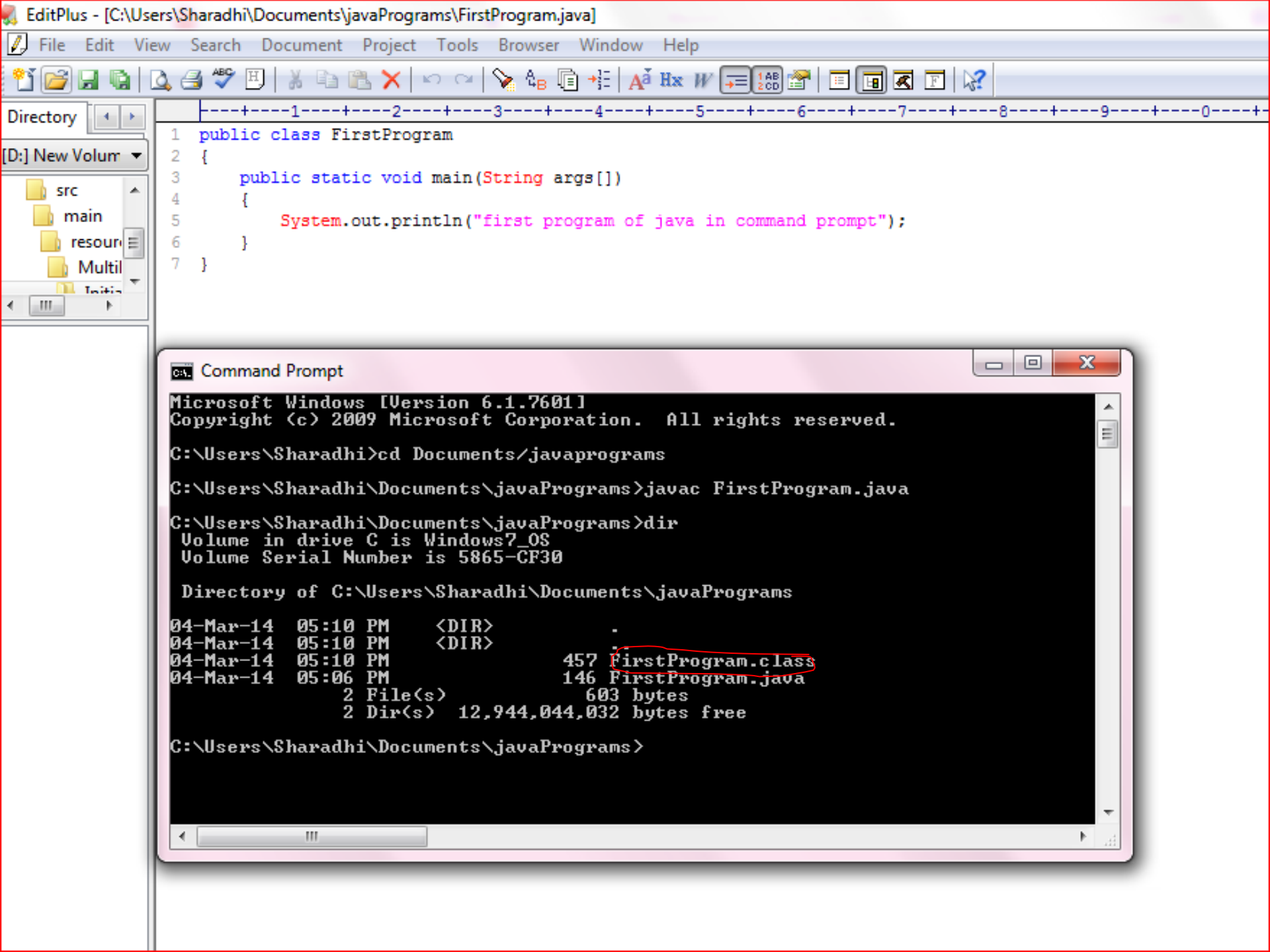
    - JAVA Enterprise Edition a.k.a J2EE

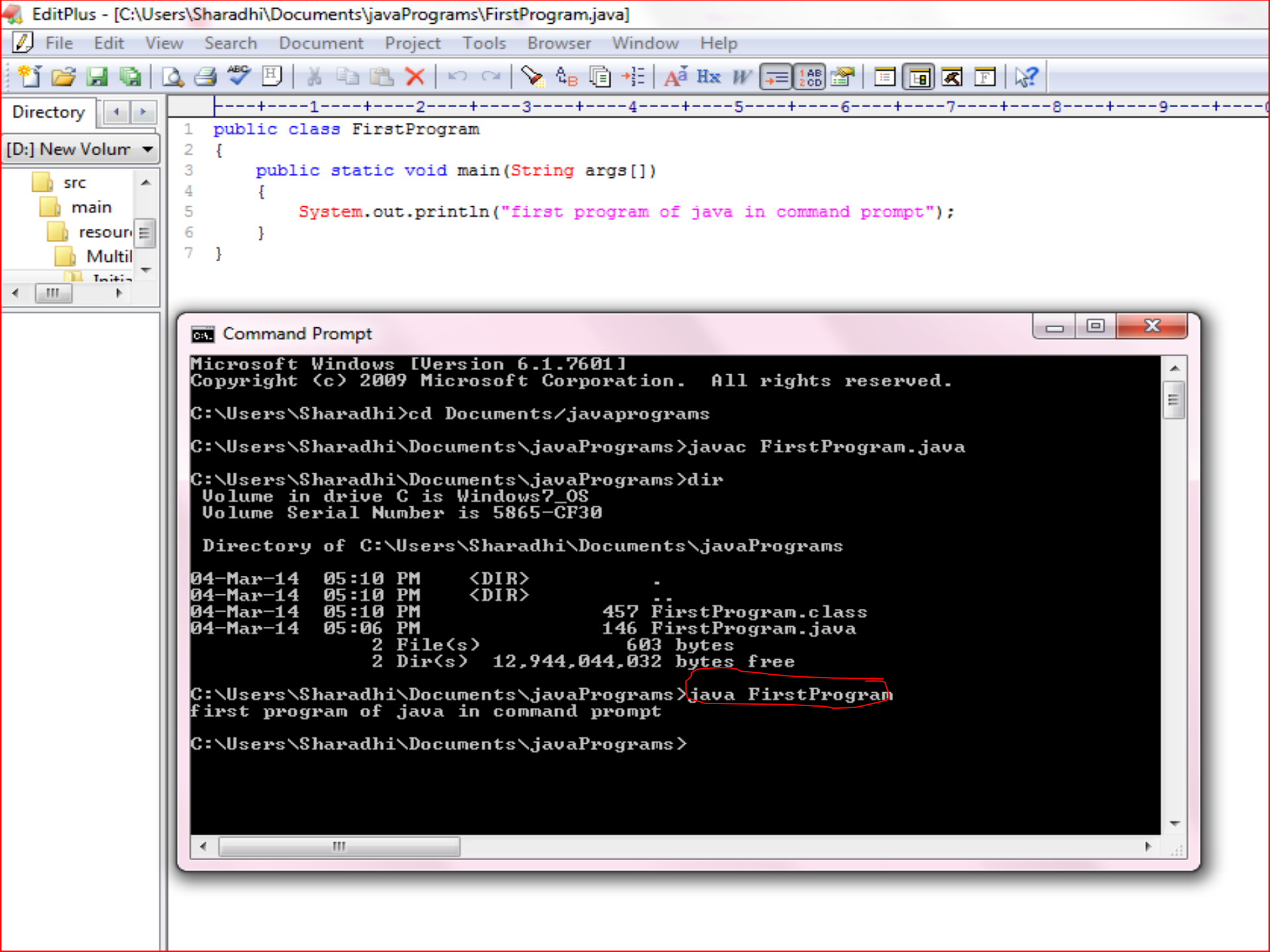
    - JAVA Micro Edition a.k.a J2ME

First java program

- Steps to be followed
- download and install java SE development Kit (Download JDK not JRE)
- Check the path – set the environment variable- point till jdk/bin.
- create source file – now using notepad or texteditor and name it as fileName.java
- Compile the source code to .class file (javac does it)
- Run the program (java fileName)


EditPlus - [C:\Users\Sharadhi\Documents\javaPrograms\FirstProgram.java]
File Edit View Search Document Project Tools Browser Window Help
Directory
[D:] New Volum
src
main
resour
Multil
1 public class FirstProgram
2 {
3 public static void main(String args[])
4 {
5 System.out.println("first program of java in command prompt");
6 }
7 }
Command Prompt
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
C:\Users\Sharadhi>cd Documents/javaprograms
C:\Users\Sharadhi\Documents\javaPrograms>javac FirstProgram.java


EditPlus - [C:\Users\Sharadhi\Documents\javaPrograms\FirstProgram.java]
File Edit View Search Document Project Tools Browser Window Help
Directory
[D:] New Volum
src
main
resour
Multil
1 public class FirstProgram
2 {
3 public static void main(String args[])
4 {
5 System.out.println("first program of java in command prompt");
6 }
7 }
Command Prompt
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
C:\Users\Sharadhi>cd Documents/javaprograms
C:\Users\Sharadhi\Documents\javaPrograms>javac FirstProgram.java
C:\Users\Sharadhi\Documents\javaPrograms>dir
Volume in drive C is Windows7_OS
Volume Serial Number is 5865-CF30
Directory of C:\Users\Sharadhi\Documents\javaPrograms
04-Mar-14 05:10 PM <DIR> .
04-Mar-14 05:10 PM <DIR> ..
04-Mar-14 05:10 PM 457 FirstProgram.class
04-Mar-14 05:06 PM 146 FirstProgram.java
2 File(s) 603 bytes
2 Dir(s) 12,944,044,032 bytes free
C:\Users\Sharadhi\Documents\javaPrograms>


EditPlus - [C:\Users\Sharadhi\Documents\javaPrograms\FirstProgram.java]
File Edit View Search Document Project Tools Browser Window Help
Directory
[D:] New Volum
src
main
resour
Multil
1 public class FirstProgram
2 {
3 public static void main(String args[])
4 {
5 System.out.println("first program of java in command prompt");
6 }
7 }
Command Prompt
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
C:\Users\Sharadhi>cd Documents/javaprograms
C:\Users\Sharadhi\Documents\javaPrograms>javac FirstProgram.java
C:\Users\Sharadhi\Documents\javaPrograms>dir
Volume in drive C is Windows7_OS
Volume Serial Number is 5865-CF30
Directory of C:\Users\Sharadhi\Documents\javaPrograms
04-Mar-14 05:10 PM <DIR> .
04-Mar-14 05:10 PM <DIR> ..
04-Mar-14 05:10 PM 457 FirstProgram.class
04-Mar-14 05:06 PM 146 FirstProgram.java
2 File(s) 603 bytes
2 Dir(s) 12,944,044,032 bytes free
C:\Users\Sharadhi\Documents\javaPrograms>java FirstProgram
first program of java in command prompt
C:\Users\Sharadhi\Documents\javaPrograms>

Java programming environment

- *Java compiler*: Transform Java programs into Java byte code
- *Java byte code*: Intermediate representation for Java programs
- *Java interpreter*: Read programs written in Java byte code and execute them
- *Java virtual machine*: Runtime system that provides various services to running programs

JVM

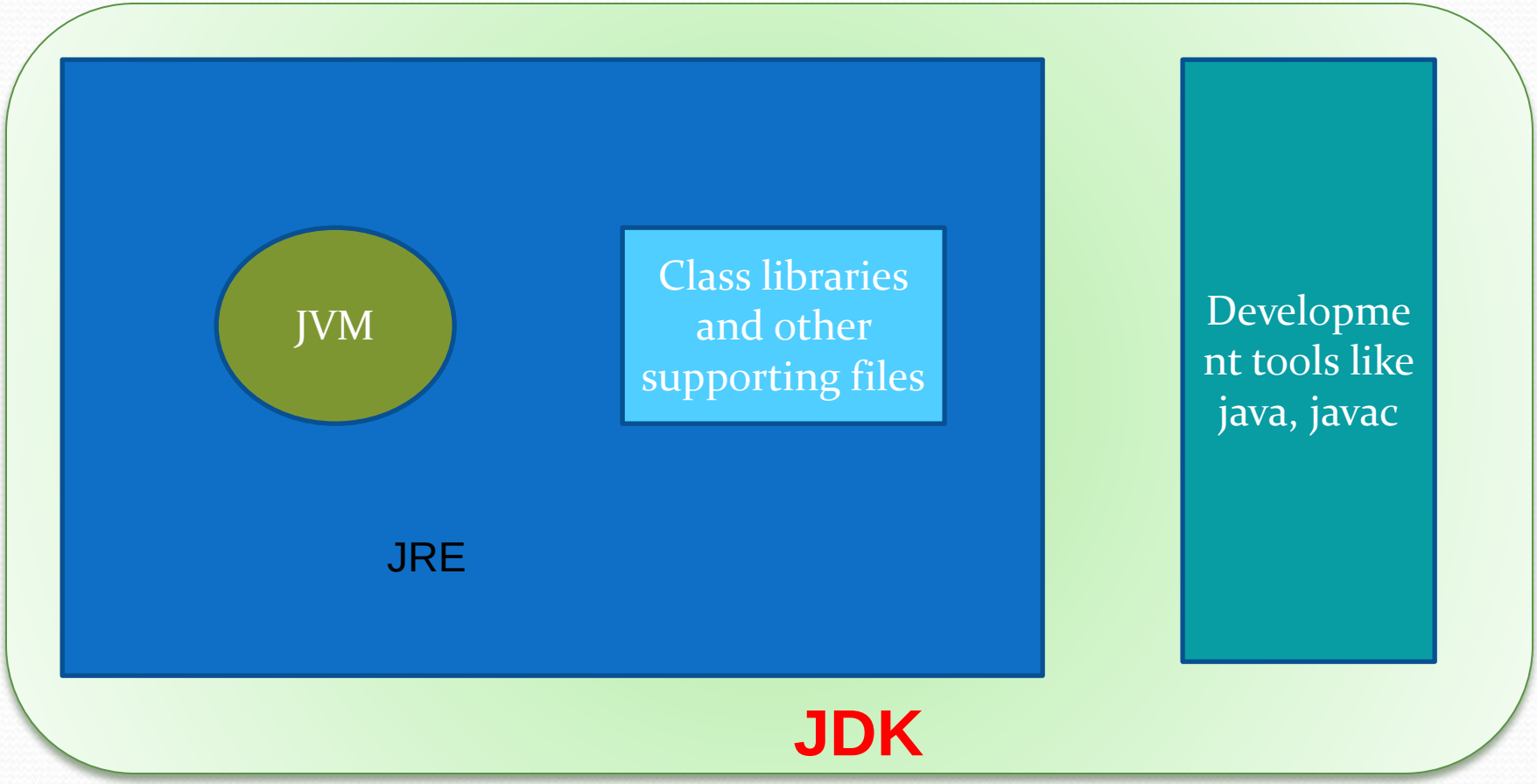
- JVM (Java Virtual Machine) is an abstract machine.
- Specification that provides runtime environment in which java bytecode can be executed.
- It is platform dependent.
- It does following tasks
  - Loads code
  - Verifies code
  - Executes code
  - Provides runtime environment

JRE

- The Java Runtime Environment (JRE) is an implementation of the JVM that actually executes our java programs.
- Java Runtime Environment contains JVM, class libraries, and other supporting files.
- It does not contain any development tools such as compiler, debugger, etc.
- Actually JVM runs the program, and it uses the class libraries, and other supporting files provided in JRE. If you want to run any java program, you need to have JRE in the system

JDK

- It is Java Development Kit.
- It contains JRE + development tools



First java program (briefly)

- **class** keyword is used to declare a class in java.
- **public** keyword it means it is visible to all.
- **static** is a keyword, if we declare any method as static, it is known as static method. The core advantage of static method is that there is **no need to create object** to invoke the static method.
- The main method is executed by the JVM, so it doesn't require to create object to invoke the main method. So it saves memory.
- **void** is the return type of the method
- **main** represents startup of the program.
- **System.out.println()** is used to print the statement.
- System → Built in class by java
- Out → instance
- Println → method

Command line arguments

- Commands that are passed to the program while running it.
- Arguments passed are stored in String array , passed in main() method.
- We can loop over the String array and access each element.

```
public static void main(String args[])
  {
        for(int i=0; i<args.length; i++ )
        System.out.println("good morning"+args[i]);


  }
```

Basics- data types

Type	Contains	Default	Size	Range
boolean	true or false	false	1 bit	NA
char	Unicode character unsigned	\u0000	16 bits or 2 bytes	0 to 2^{16}-1 or \u0000 to \uFFFF
byte	Signed integer	0	8 bit or 1 byte	-2^{7} to 2^{7}-1 or -128 to 127
short	Signed integer	0	16 bit or 2 bytes	-2^{15} to 2^{15}-1 or -32768 to 32767
int	Signed integer	0	32 bit or 4 bytes	-2^{31} to 2^{31}-1 or -2147483648 to 2147483647
long	Signed integer	0	64 bit or 8 bytes	-2^{63} to 2^{63}-1 or -9223372036854775808 to 9223372036854775807
float	IEEE 754 floating point single-precision	0.0f	32 bit or 4 bytes	1.4E-45 to 3.4028235E+38
double	IEEE 754 floating point double-precision	0.0	64 bit or 8 bytes	439E-324 to 1.7976931348623157E+308

Operators

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Misc Operators

Arithmetic

+	Addition - Adds values on either side of the operator
-	Subtraction - Subtracts right hand operand from left hand operand
*	Multiplication - Multiplies values on either side of the operator
/	Division - Divides left hand operand by right hand operand
%	Modulus - Divides left hand operand by right hand operand and returns remainder
++	Increment - Increases the value of operand by 1
--	Decrement - Decreases the value of operand by 1

Relational

==	Checks if the values of two operands are equal or not, if yes then condition becomes true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

Bitwise

&	Binary AND Operator copies a bit to the result if it exists in both operands.
|	Binary OR Operator copies a bit if it exists in either operand.
^	Binary XOR Operator copies the bit if it is set in one operand but not both.
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.(<< 1 =multiply by 2)
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.(>> 1= divide by 2)

Logical

&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.
||	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.

Misc operator

- **Conditional operator**: This operator consists of three operands and is used to evaluate Boolean expressions. Ex:
- variable x = (expression) ? value if true : value if false

- **instanceOf operator**: This operator checks whether the object is of a particular type(class type or interface type)

```
String s="hello";
if( s instanceof String)
        System.out.println("s is string");
```

Control statements

```
if(Boolean_expression) {
//Statements will execute if the Boolean expression is true
}
 else if(Boolean_expression) {
//Statements will execute if the Boolean expression is true
}
 else{
}
```

Switch

```
switch(expression){
 case value :
//Statements
 break; //optional
case value :
//Statements
break; //optional
default : //Optional
//Statements
}
```

Rules for switch

- The variable used in a switch statement can only be a byte, short, int, or char.(String also after java 7)
- Any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The statements are followed by a optional *break* statement
- When a *break* statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A *switch* statement can have an optional default case, which must appear at the end of the switch.
- The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Loop controlling

- while loops

```
while(Boolean_expression) {
//Statements
}
```

- do while loops

```
do {
//Statements
}while(Boolean_expression);
```

for loops

Normal :

```
for(initialization; Boolean_expression; update) {
//Statements
}
```

```
   int arr[]={1,2,3};
  for(int i=0;i<n;i++)
    SOP(arr[i]);
```

Enhanced for java: (java SE5)

```
for(declaration : expression) {
//Statements
}
```

```
     for(int i : arr)
            SOP(i);
```

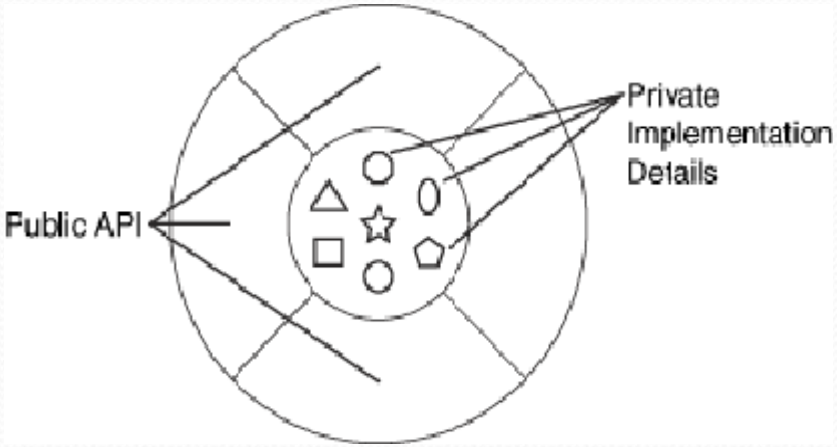
Important keywords

- **break**
- The *break* keyword is used to stop the entire loop.
- The break keyword can be used inside any loop or a switch statement.

- **Continue**
- It causes the loop to immediately jump to the next iteration of the loop.
- In a for loop, the continue keyword causes flow of control to immediately jump to the update statement.
- In a while loop or do/while loop, flow of control immediately jumps to the Boolean expression.

Classes in Java

- A class is a blueprint that defines the state(variables) and the behavior(methods) of particular entity.
- It can be shown like this:



```
class MyClass
{
//instance variables(fields)
//methods
//constructors
}
```

- Instance variables(fields) provides the state of the class and its objects
- Methods to implement the behavior of the class and its objects
- Constructors for initializing new objects, declarations for the fields.

Class Myclass extends MySuperClass, implements MyInterFaceClass

{

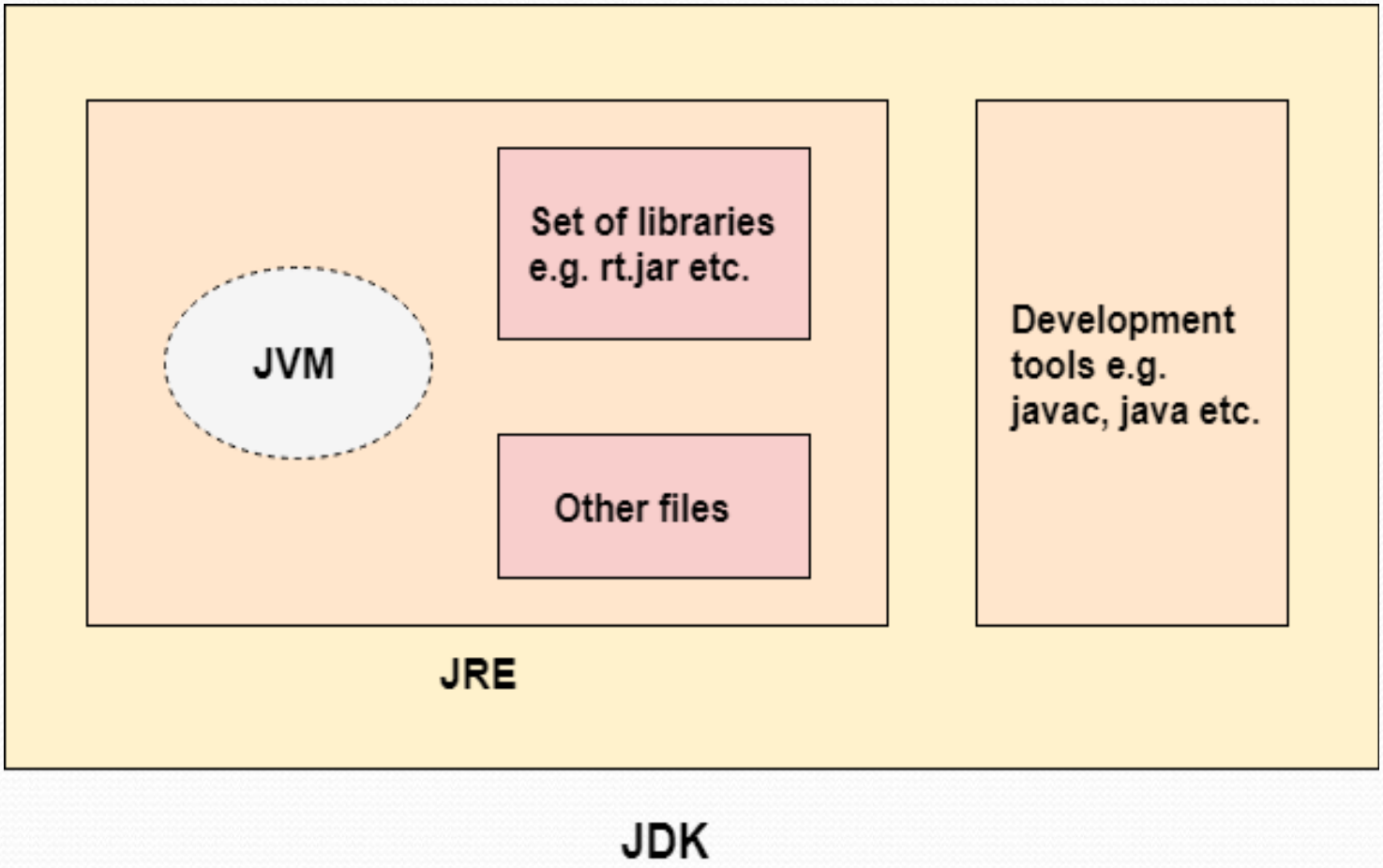
//instance variables(fields)

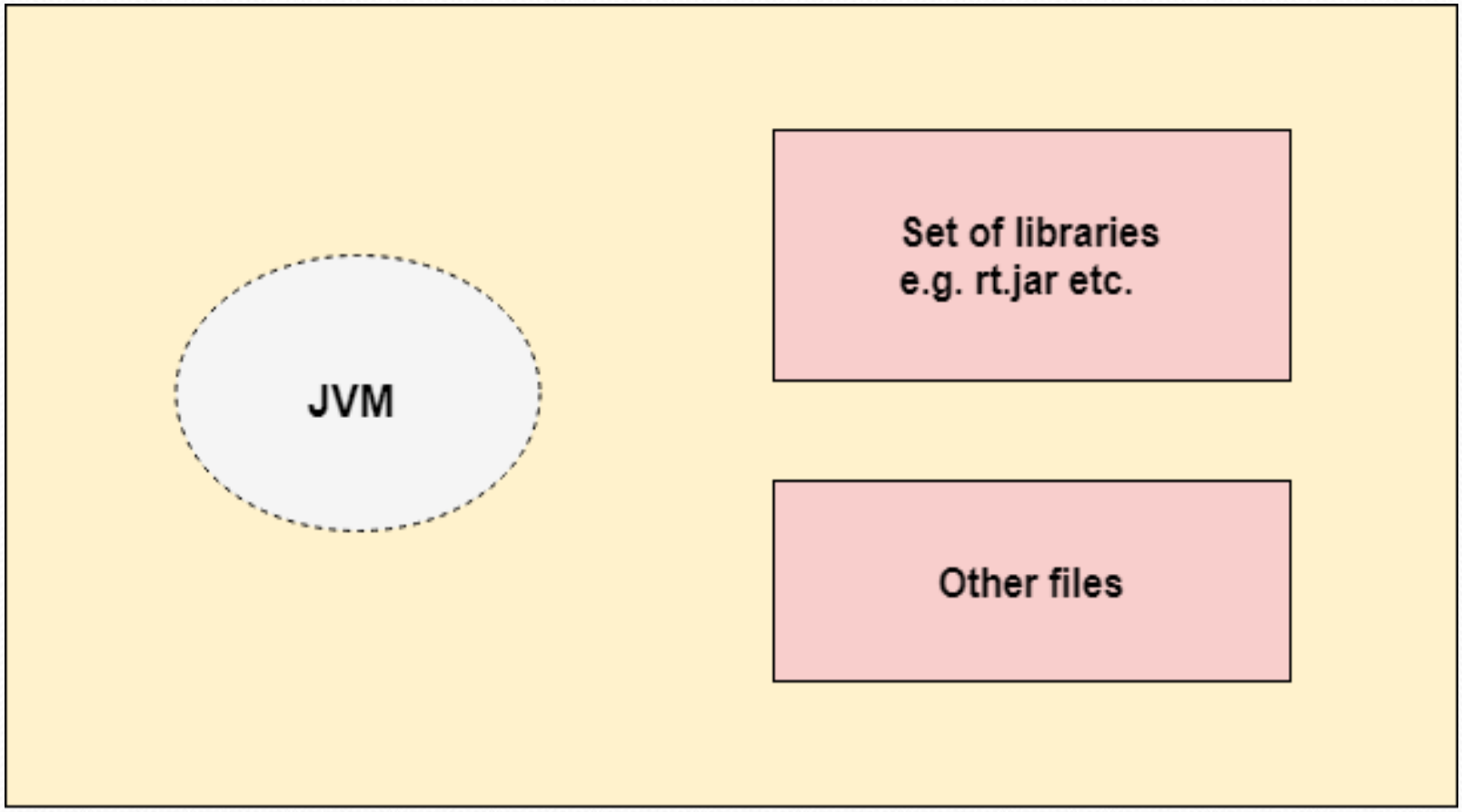
//methods

//constructors

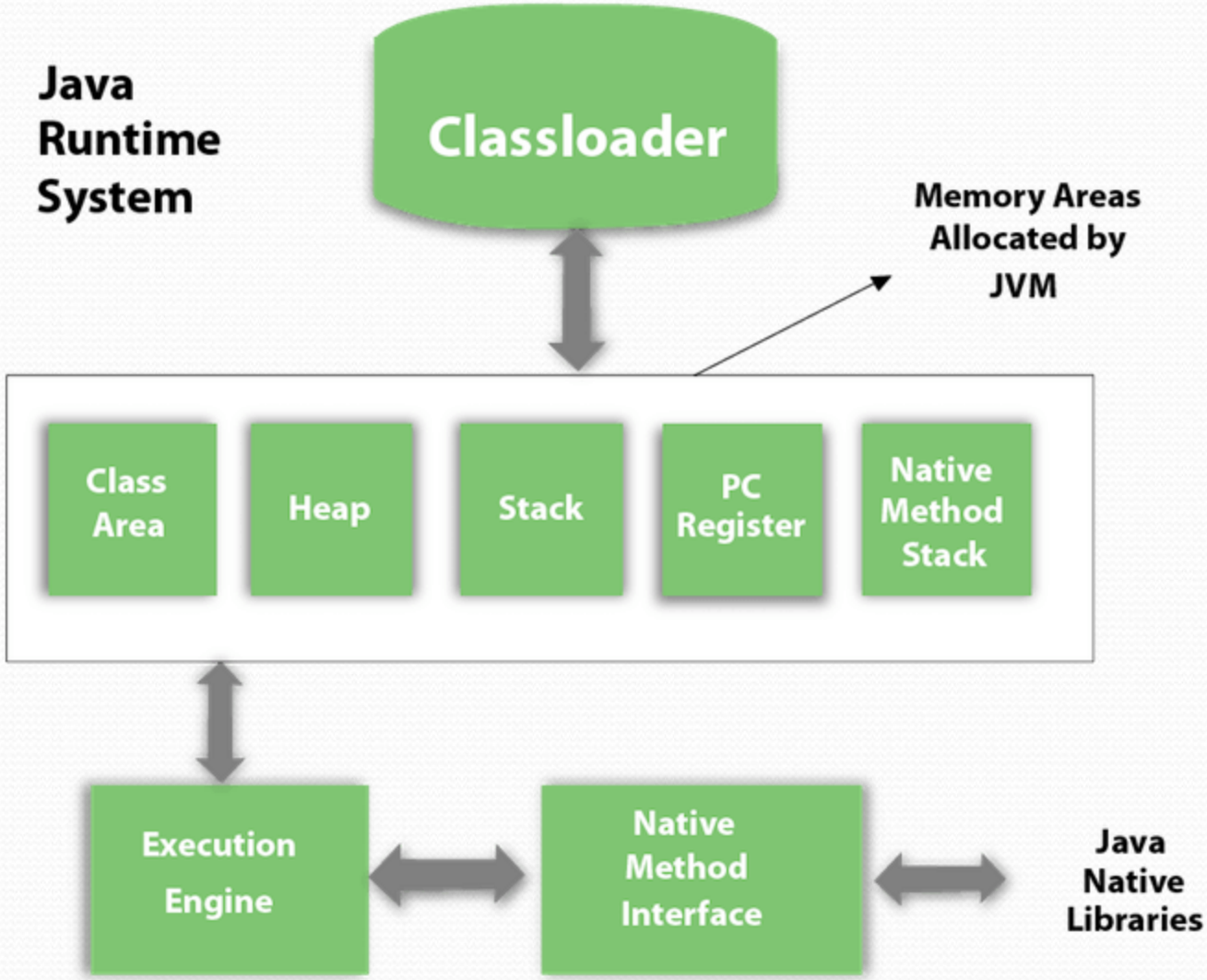
}

- Instance variables do have a default value.
- 0 or 0.0 for numeric primitives, false for booleans, and null for references.

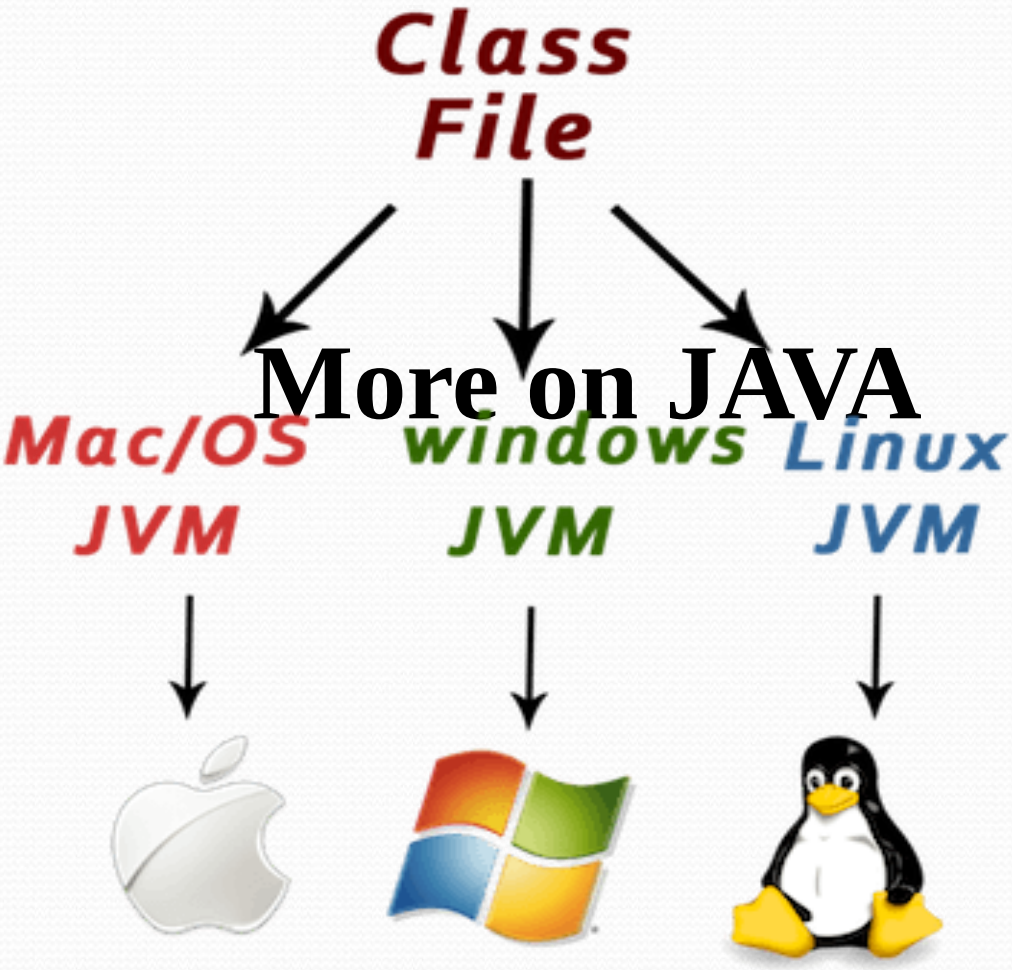

Set of libraries
e.g. rt.jar etc.
JVM
Other files
Development
tools e.g.
javac, java etc.
JRE
JDK

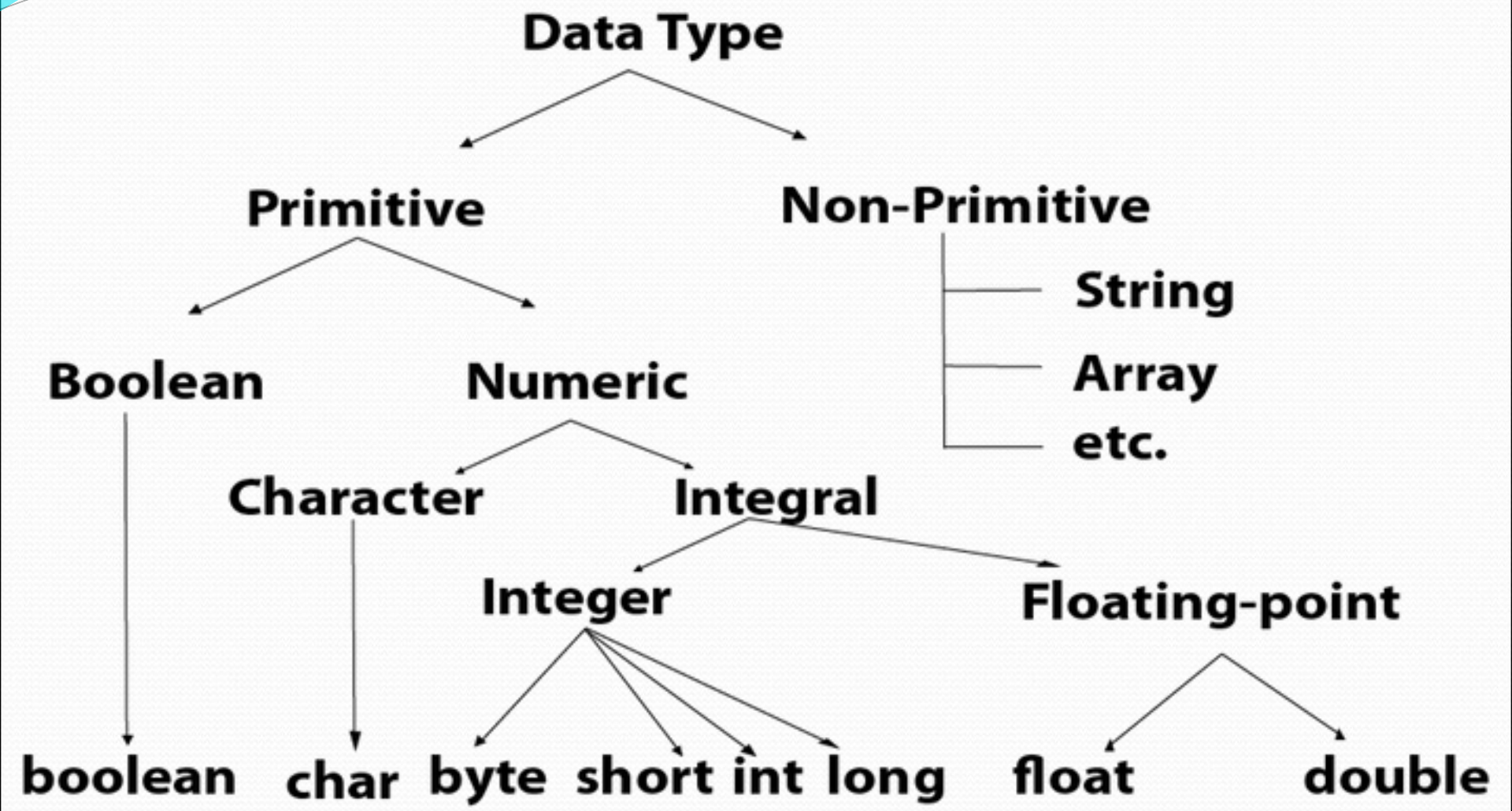


JRE


Java
Runtime
System
Classloader
Memory Areas
Allocated by
JVM
Class
Area
Heap
Stack
PC
Register
Native
Method
Stack
Execution
Engine
Native
Method
Interface
Java
Native
Libraries

More on JAVA

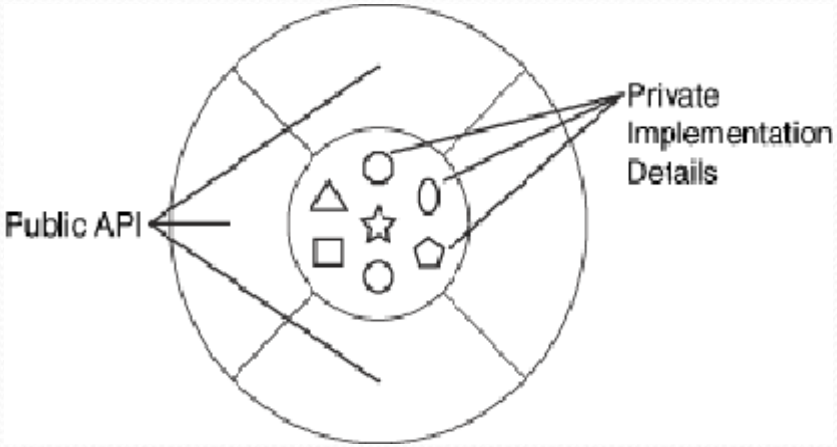



Data Type
Primitive
Non-Primitive
String
Array
etc.
Boolean
Numeric
Character
Integral
Integer
Floating-point
boolean
char
byte
short
int
long
float
double

More on JAVA

Classes in Java

- A class is a blueprint that defines the state(variables) and the behavior(methods) of particular entity.
- It can be shown like this:



```
class MyClass
{
//instance variables(fields)
//methods
//constructors
}
```

- Instance variables(fields) provides the state of the class and its objects
- Methods to implement the behavior of the class and its objects
- Constructors for initializing new objects, declarations for the fields.

Class Myclass extends MySuperClass, implements MyInterFaceClass

{

//instance variables(fields)

//methods

//constructors

}

- Instance variables do have a default value.
- 0 or 0.0 for numeric primitives, false for booleans, and null for references.

- In general, class declarations can include these components, in order:

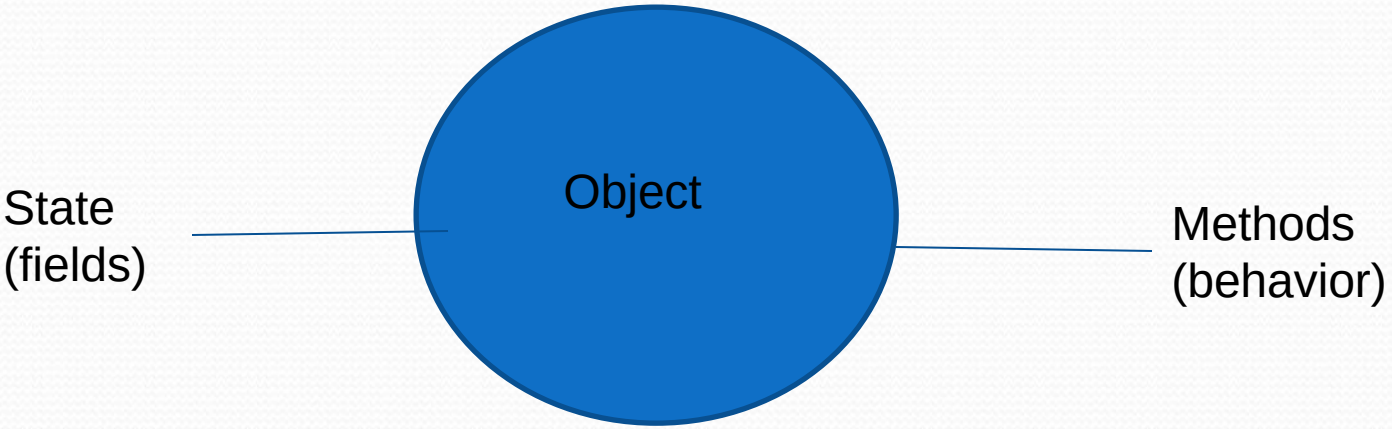
- **Modifiers** used should be *public* or *private*.
- It must have ‘class’ keyword, followed by the name with the initial letter capitalized by convention.
- it can extend one parent class- with keyword ‘extends’
- It can implement comma-separated list of interfaces with the keyword *implements*.
- The class body, surrounded by braces, {}.
- Each .java file can have only one public class
- source file name should match with public class name

Access modifiers

- public modifier—the field is accessible from all classes.
- private modifier—the field is accessible only within its own class.
- Default(package)– the field is accessible within the package
- Protected – its accessible within the pacakge only for subclass

Objects and creation of objects

- Java is OOL , So technology deals with the objects
- Instance of class is called as objects
- Object has states and behaviors



Creating the objects

It has 3 steps

- **Declaration**: Declaring a reference variable

  Cat c = new Cat();

- **Instantiation**: Creating an object The new keyword is used to create the object

  Cat c = new Cat();

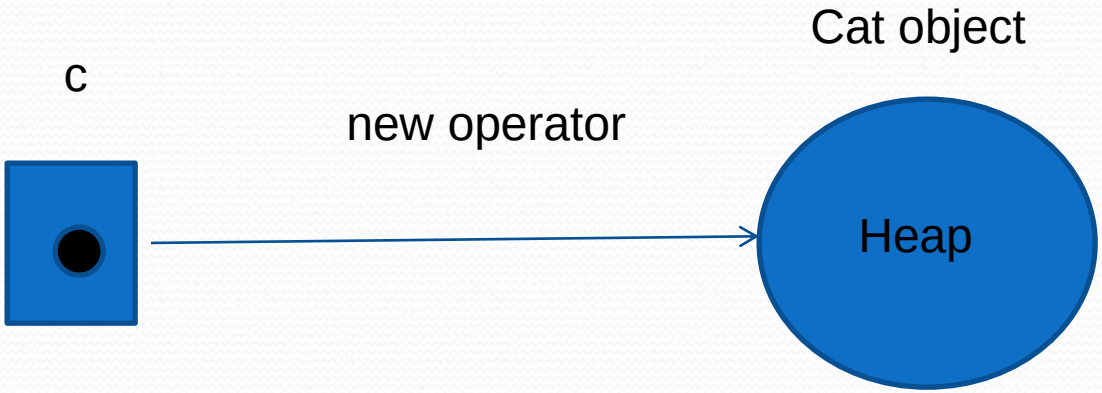
- **Initialization**: Link the object and the reference. Calling the constructor.

  Cat c = new Cat();

Where c – is the reference variable , which refers to the class- Cat

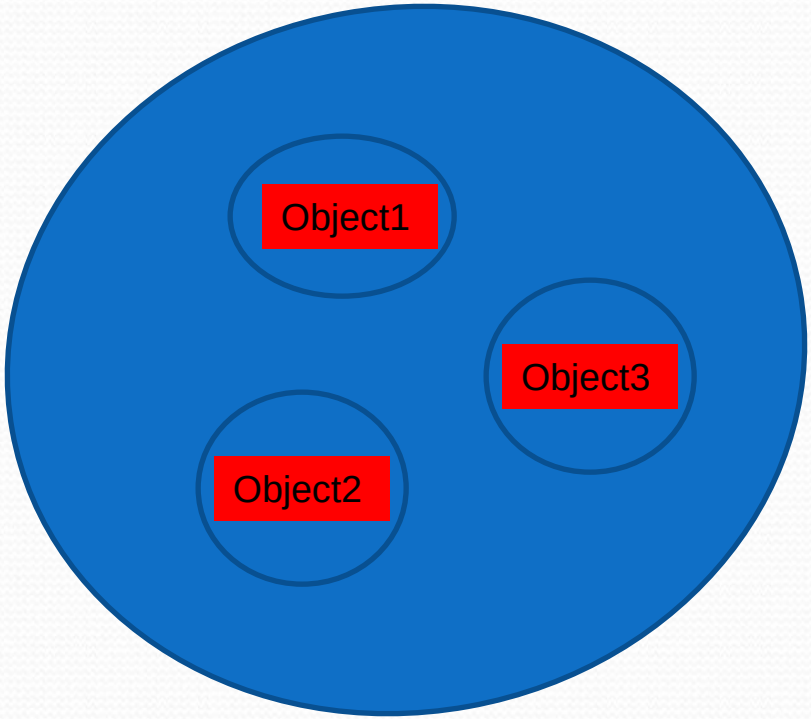
What happens when object is created

Cat c= new Cat();

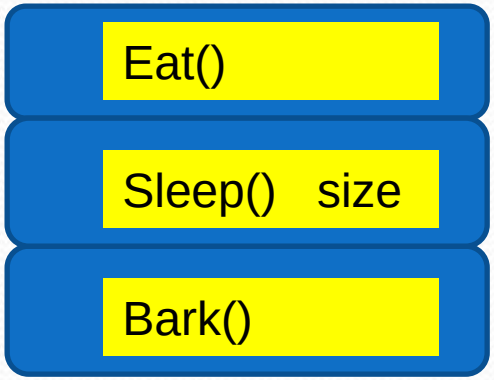


Its like Creating the remote control (c) to an object.

The heap where all objects live



The Stack where all local variables and method invocations live



- **Instance variables** are declared inside the class not inside the method so they live inside the object they belong to (heap)

```
Public class cat
{
   int size;  // inside the cat object in the heap
}
```

- **Local variables** live inside the method , including the method parameters.

```
Public void method1(int y)
  {
   int x=3+y;   // x and y are local variables which live
   //inside the stack with the method invoaction
  }
```

Constructors

- A class contains constructors that are invoked to create objects from the class blueprint.
- Constructor declarations look like method declarations—except that they use the name of the class and have no return type
- A constructor is called by the new operator, which automatically returns the newly created object.
- programmer uses constructor to initialize variables, instantiating objects and setting colors.

- If the class doesn’t have any constructor , java compiler provides the default constructor which is no- arg constructor .
- Cat c= new Cat();
- Constructor runs before the object is assigned to a reference.
- Its not compulsory to provide constructor to your class since java is providing no-arg constructor
- But be careful when doing it.

- If you write a constructor that takes arguments, and you *still* want a no-arg constructor, you'll have to build the no-arg constructor yourself!
- If you have more than one constructor in the class, then constructors should be overloaded.
- Java platform differentiates constructors on the basis of the number of arguments in the list and their types. (constructor overloading)

Access specifiers for constructors

- private

  Only this class can use this constructor.

- protected

  Subclasses of this class and classes in the same package can use this constructor.

- public

  Any class can use this constructor.

- no specifier

  Gives package access. Only classes within the same package and this class can use this constructor.

Initializing the state of an object

- Constructor or setter methods?

- If the developer knows what should be the state of the object, then constructor can be used by developer to initiate the object

- If programmer who uses the objects wants to initiate the instance variable then he has to use setter method.

Usage

```
public class Cat
 {
   int private size;
   public Cat(){
   }
public void setSize(int s){
   size=s;
 }
```

```
public class UseCat
{
Public static void
    main(String args[])
{
Cat c= new Cat();
c.setSize(15);
}
}
```

Reading values from keyboard

- Scanner is the best way to read input from key board
- Scanner class breaks the input into tokens using a delimiter which is **whitespace** by default.

```
//scanner
Scanner s= new Scanner(System.in);

System.out.println("enter the string");
String st=s.next();
System.out.println("string "+st);

System.out.println("enter the integer");
int i=s.nextInt();
System.out.println("string "+i);

Scanner s1= new Scanner(System.in);
System.out.println("enter the String again");
String str=s1.nextLine();
System.out.println("string "+str);
```

Arrays

- fixed-size sequential collection of similar elements
- It is static DS, since size of array must be specified at declaration time
- **Creation, initialization and accessing array**

```
dataType[] arrayRefVar; // preferred way. or

dataType arrayRefVar[]; // works but not  preferred way.
```

- **Initialization**:

```
dataType[] arrayRefVar = new dataType[arraySize];  or
dataType[] arrayRefVar = {value0, value1, ..., valuek};

int [] numberArray={1,2,3};
```

Multi- dimensional array :

```
int [][] numberArray = {
{10, 20, 30, 40, 50},{60,70,80}
};
```

```
System.out.println("size"+ numberArray.length);
System.out.println("size"+ numberArray [0].length);
```

- Each element in array is accessed using index value.
- In java, since array is a object, manipulation can be done using some predefined methods

Methods

copyOfRange():copies array elements, but destination array is returned by the method.

copyOfRange(Object src, int srcPos,int destPos);

copyOf (Object src, length)

binarySearch(Object [] src,object key);

equals(Object[] a1, object[] a2);

fill(Object[] a1,int val);

sort(Object[]);

Ex: Arrays.*sort(arr);*

Arrays.*binarySearch(arr, 2);*

System class method for arrays

System.arraycopy(): efficiently copy data from one array into another

public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)

srcPos: element position in the source array from which elements are to be copied

destPos: element position in the destination array to where elements to be copied

Length: no of elements to be copied

Ex: System.*arraycopy(arr1, 0, arr2, 1, 3);*

Date and time

- **Date** class is available in **java.util** package

- **Date constructors**
- Date();→ initializes the object with the current date and time.

- Most of the Date class is deprecated and instead we can use **java.util.Calendar**

Some methods

```
//Date
Date today = new Date();

System.out.println("Today Particulars: " + today);
System.out.println("Hours Part of today: " + today.getHours());
System.out.println("Minutes part of today: " + today.getMinutes());
System.out.println("Seconds part of today: " + today.getSeconds());
System.out.println("Month part of today: " + today.getMonth());
System.out.println("Date part of today: " + today.getDate());
System.out.println("Day part of today: " + today.getDay());
System.out.println("Year part of today: " + today.getYear());
System.out.println("Milliseconds representation of today " + today.getTime());
```

Note

- Add 1900 to the year returned by the Date class to get the correct year. That is, for 2011, the date returns 111.
- Months are numbered to 0 to 11. That is 0, returned by the Date class, should be learnt as January.
- Month dates are numbered 1 to 31.
- Hours are given as 0 to 23.
- Minutes are numbered as 0 to 59.
- Seconds are printed as 0 to

Some more methods

- int compareTo(Date date)→ Compares the value of the invoking object with that of date.
- 0→ if the values are equal.
- +ve→ if the invoking object is later than date.
- -ve→ if the invoking object is earlier than date.
- boolean equals(Date d)→ Returns true if the invoking Date object is same as the argument

Calendar

- **Calendar** class is available in **java.util.Calendar** package
- It comes with many variables and methods to manipulate and retrieve date and time
- Its object can be obtained from its static method **getInstance()**.

  Calendar todays= Calendar.*getInstance();*

Field name	Description
YEAR	Represents YEAR
MONTH	Represents Month
DATE	Represents the date of the month (1 to 31)
HOUR	Represents the hour (0 to 12) of either morning or evening
TIME	Displays the system time in milliseconds (after 01-01-1970)
MINUTE	Represents the minutes part of the hour
SECOND	Represents the seconds part of the minute(starts with 1)
WEEK_OF_YEAR	Represents the current week number in the current year
WEEK_OF_MONTH	Represents the current week number in the current month
DAY_OF_MONTH	Represents the current day number in the month
DAY_OF_YEAR	Represents the current day number in the year
DAY_OF_WEEK	Represents the current day number in the week(sun,mon)

```
    Calendar todays= Calendar.getInstance();

    System.out.println("Year info: " + todays.get(Calendar.YEAR));
    System.out.println("Month info: " + todays.get(Calendar.MONTH));
    System.out.println("Date info: " + todays.get(Calendar.DATE ));
    System.out.println("week info: " + todays.get(Calendar.WEEK_OF_MONTH));
    System.out.println("Hour info: " + todays.get(Calendar.HOUR));
    System.out.println("Minutes info: " + todays.get(Calendar.MINUTE ));
    System.out.println("Day info: " + todays.get(Calendar.DAY_OF_MONTH));

    Calendar cal1 = Calendar.getInstance(); //todays date
    Calendar cal2 = Calendar.getInstance();  //some other date

    cal2.set(2014, Calendar.SEPTEMBER, 11);   //setting the date

    System.out.println("checking for aftr "+cal1.after(cal2)); //checking
    System.out.println("checking for "+cal1.compareTo(cal2)); //comparing 2 dates
    cal1.add(Calendar.MONTH, -2);
    System.out.println(" afetr subtracting "+cal1.get(Calendar.MONTH));
    System.out.println(" afetr subtracting "+cal1.getTime());


}
```

SimpleDateFormat

- **SimpleDateFormat** from **java.text** package allows the programmer to choose the format of date.
- In this, the year is chosen in four digits, month in two digits and date in two digits.
- The parse() method of SimpleDateFormat converts the specified date in string format into an object of Date class.
- Format can be specified by creating the object of the same.

SimpleDateFormat sdf = new SimpleDateFormat(“yyyy-MM-dd”);

Example

```
//simple date format

SimpleDateFormat sd = new SimpleDateFormat("yyyy-MM-dd");
Date d1=sd.parse("2014-11-10");
Date d2=sd.parse("2014-11-12");
//SimpleDateFormat s= new SimpleDateFormat("MM-yyyy-dd");

System.out.println("checking for aftr "+d1.after(d2)); //checking
System.out.println("checking for "+d1.compareTo(d2)); //comparing 2 dates
```


Java™

Methods in java

- The procedures and functions are known as methods in Java.
- Methods are called with an object in java
- A Java method signature may comprises of optional access specifiers, return type, name, optional parameter list and also optional exceptions the method can throw.

Method overloading

- If a class have multiple methods by same name but different parameters, it is known as Method Overloading.
- Method overloading increases the readability of the program
- In java overloading can be done by changing the number of parameters or data type of the parameters.
- But method overloading is **not** possible just by changing the **return type** of the method
- Ex:

```
public void eat()
public int eat()
```

} not possible

Static binding and polymorphism

- Compiler decides at compile time itself which overloaded method is to be called.
- That is, method binding is done at compile time itself. This is known as **static binding**
- Polymorphism means same method called at different times and gives different types of output.
- Through **method overloading** it is done during compile time , so its is called **static polymorphism**.

- Main method can be overloaded?

Inheritance

- Inheritance can be defined as the process where one object acquires the properties of another.
- Allows the classes to inherit the properties of other class.
- It improves the reusability and lead to polymorphism
- Implements Is-A relationship
- In java Keyword used to inherit the class behavior is- "**extends**"
- Parent class is named as – superclass
- Child class as- subclass

example

```
public class Animal{
}
public class Dog extends Animal{
 }
 public class Cat extends Animal{
}
```

Dog is sub class of Animal

Dog IS- A animal

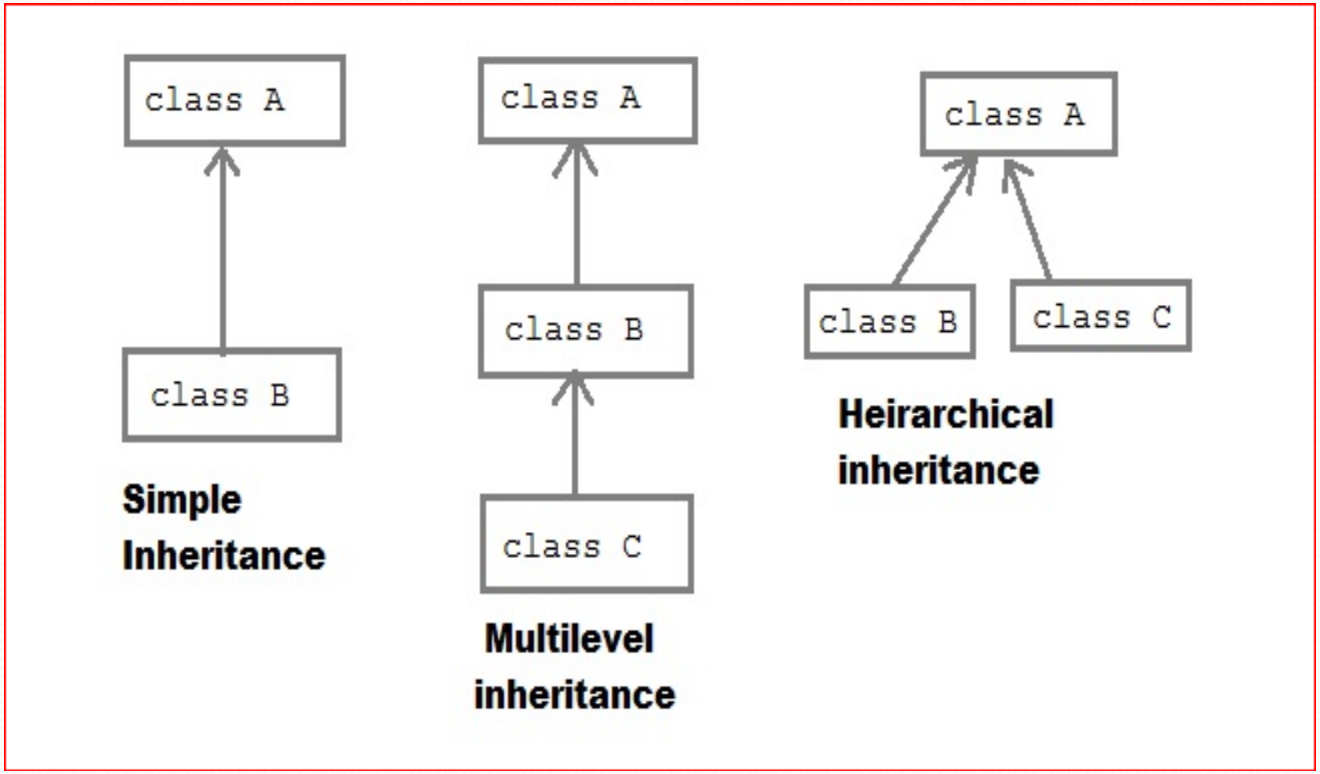
Types of Inheritance

- Java supports only 3 types of inheritance:

Single

Multilevel

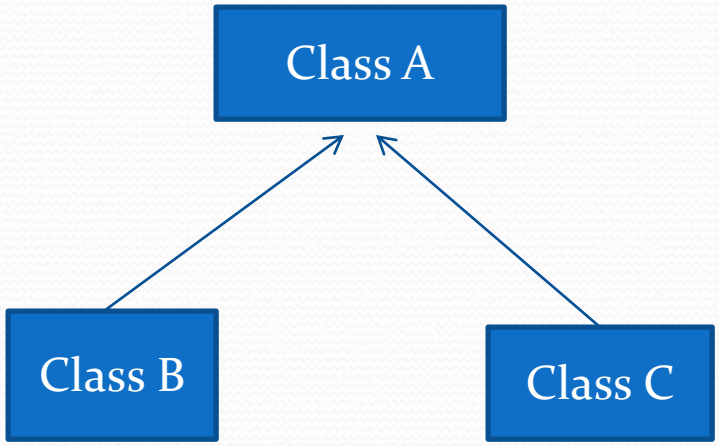
Hierarchical



Types

- Java doesn't support **multiple** inheritance.

  Why?????



Removing ambiguity and confusions

Purpose:

- Code reusability
- Method overriding → runtime polymorphism

Method overriding

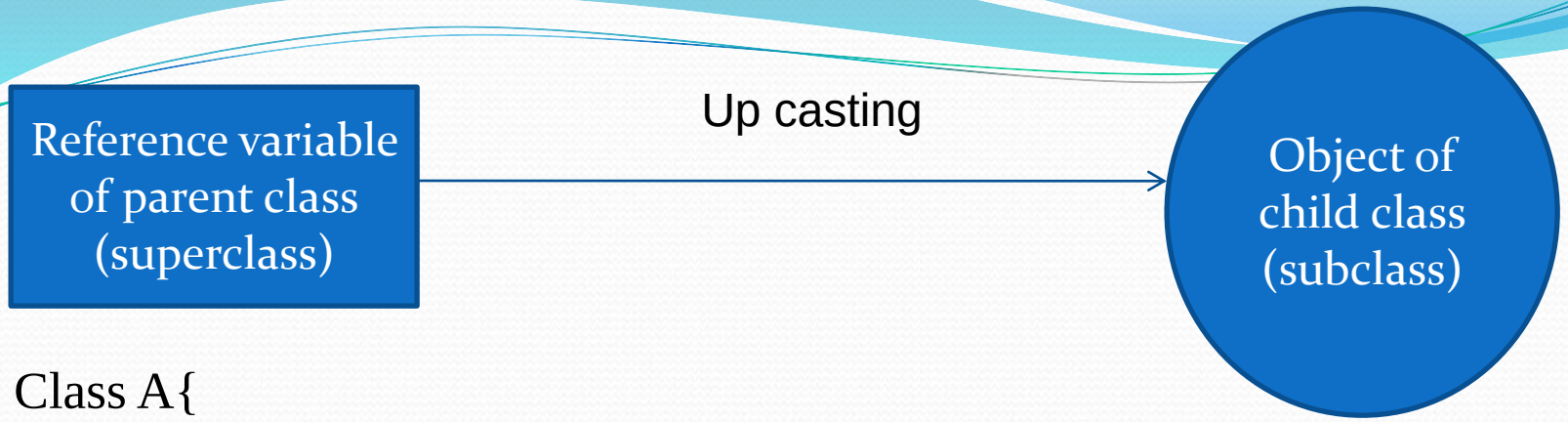
- Subclass can make use of super class methods straight away by virtue of inheritance.
- If the subclass does not satisfy about the functionality of the super class method, the subclass can **rewrite** with its own functionality with the same method name.
- This concept is known as "**method overriding**"
- In method overriding, super class and subclass have the method with the same signature

Differences

Method overloading	Method overriding
Happens in the same class	Happens among two classes – a super class and a subclass
No inheritance	Happens in inheritance only
Should have different parameters	Should have same parameters
Return type is not bothered	Should have the same return type
Leads to static binding	Leads to dynamic binding

Run time polymorphism

- Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile -time.
- Overridden method is called through the super class reference variable.
- Which method to be called→ is decided by the JVM at run time based on the object being referred.
- So it is called dynamic binding
- We get different types of outputs , due to binding methods during run time so its called RTP.




```
Class A{
}

Class B extends A{
}
A a= new B();  //upcasting
```

Rules: (summary)

- when a subclass object is assigned to a super class object, the super object will call **subclass overridden** method
- Runtime polymorphism cannot be achieved by data members.

  Since there is no overriding of data members.

```
class Animal{
void eat(){
System.out.println("animal is eating...");}
}
class Dog extends Animal{
void eat(){
System.out.println("dog is eating...");}
}
class BabyDog extends Dog{
public static void main(String args[]){
Animal a=new BabyDog();
a.eat();
}}
```

Variables

- 3 main types of variables
- Local variable- declared inside the method
- Instance variable- declared inside the class
- Static variable – its for the class not for the object.

```
class A{
    int data=50; //instance variable
    static int m=100; //static variable
    void method(){
    int n=90;//local variable
   }
}//end of class
```

Rules

Local variables

- Local variables resides inside the stack
- There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.
- Access modifiers cannot be used for local variables (like public, private)

Instance variables

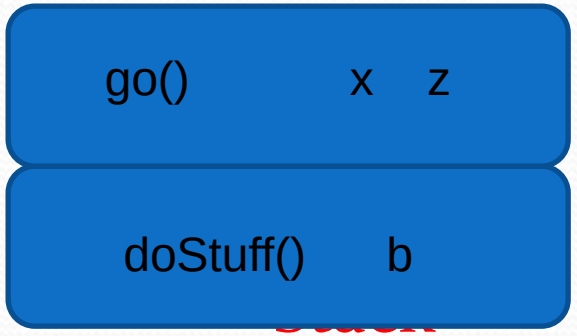
- Instance variables have default values. For numbers the default value is 0, for Booleans it is false and for object references it is null.
- Values can be assigned during the declaration or within the constructor.
- Access modifiers can be given for instance variables

Class(static) variables

- Class variables also known as static variables are declared with the *static* keyword
- Static variables are created when the program starts and destroyed when the program stops.
- Static variables can be accessed by calling with the class name

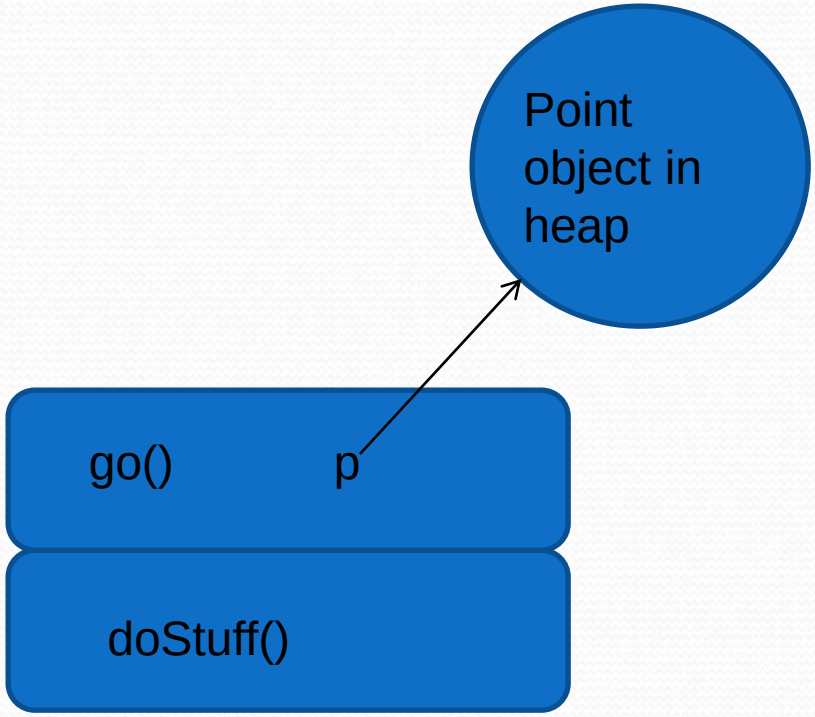
Stack scenario

```
public void doStuff()
 {
  boolean b = true;
  go(4);
 }
public void go (int x)
{
  int z = x + 24;
}
```



Local variables that are objects

```
public void doStuff()
 {
go(4);
 }
public void go (int x)
{
     Point p= new Point();
   //p is local variable since it is
   // inside the method so it will be inside the Stack
}
```



Life and scope

- A local variable is alive as long as its Stack frame is on the Stack. In other words, until the method completes.
- A local variable is in scope only within the method in which the variable was declared.

  When its own method calls another, the variable is alive, but not in scope until its method resumes.
- You can use a variable only when it is in scope
- Rule is same for reference variable also
- We can use the reference variable only if it is in scope.

Parameter passing in java

- In java parameters can be passed as a value or as a reference.
- Everything in Java is passed by value. Objects, however, are never passed *at all*.
- The values of variables are always primitives or references, never objects
- The value of the actual parameter is copied into the formal parameter when the procedure is invoked.
- Any modification of the formal parameter affects only the formal parameter and not the actual parameter.

Pass by value

```
Public class Animal{
public boolean sleep(int time)
{
   time=time+60;
   boolean f=true;
}
  return f;
}
```

```
Public static void main(String[] args)
{
Animal a= new Animal();
int t=50;
Boolean v=a.sleep(t);
}
```

Passing references- Different Scenarios

```
public class Duck {

private String name;
private int size;

//setter methods
}
```

```
public class Fish {

int size;
String name;

}
```

1st case

```
void eat(Fish f)
{
f=null;
}
```

2nd case

```
void eat(Fish f)
{
f= new Fish();
f.name="whityfish";
f.size=5;
}
```

3rd case

```
void eat(Fish f)
{
f.name="blackyfish";
f.size=15;
}
```

Static keyword

- static keyword allows to call without the help of an object
- A static variable is one that's associated with a class, not objects of that class.
- behavior not dependent on an instance variable, so no instance/object is required. Just the class.
- We may apply static keyword with variables, methods, blocks.
- Static things are loaded when the class is loaded.

Static variables

- The static variable can be used to refer the common property of all objects
- The static variable gets memory only once in class area at the time of class loading.

  so it is memory efficient
- instance variables: 1 per **instance**
- static variables: 1 per **class**

Static methods

- A static method belongs to the class rather than object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- static method can access static data member and can change the value of it.
- Ex: Math.min(1,2); (cannot instantiate the math class because constructor is private in math class)
- Calendar.Year- static field in Calendar class

```
class Calculate{
  static int cube(int x){
  return x*x*x;
  }

  public static void main(String args[]){
  int result=Calculate.cube(5);
  System.out.println(result);
  }
}
```

Restrictions

- The static method can not use non static data member or call non-static method **directly**.
- Static variables in a class are initialized before any *object of that* class can be created.
- Static Block is used to initialize the static data member.
- It is executed before main method at the time of classloading.

Final Keywords

- Final variable: Cannot change the value of the variable.
- Final methods: cannot be overridden
- Final class : cannot extend
- Final method can be inherited but cannot override it

Super keyword

- super is used to refer immediate parent class instance variable.
- super() is used to invoke immediate parent class constructor.
- super is used to invoke immediate parent class method.

Instance initializer

Instance Initializer block is used to initialize the instance data member. It run each time when object of the class is created.

Rules:

- The instance initializer block is created when instance of the class is created.
- The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).
- The instance initializer block comes in the order in which they appear.

this

- this. always refers to the currently executing object.
- this keyword can be used to invoke current class method
- We can use' this () ' with constructors 
- this should not be referenced in a static method because static methods are associated with the class

```java
public class ThisExamples {

    int i=10;

    public void method1()
    {
        System.out.println("var value is "+this.i);
    }
```

```java
public class ThisExamples {

    int i=10;

    public void method1()
    {
        System.out.println("var value is "+this.i);
        this.method2();
    }
    public void method2()
    {
        System.out.println("inside method2");
    }
```

```java
public Car getCar()
{
 return this;
}
```

this() with constructors

Back to constructors

Overloading of constructors

- Just like method overloading, constructors also can be overloaded.
- Same constructor declared with different parameters in the same class is known as constructor overloading.
- Compiler differentiates which constructor is to be called depending upon the number of parameters and their sequence of data types.
- Why to overload constructors?
  To create objects in different ways.
- Does constructor return any value?

Example

Rules of using this()

- If included, this() statement must be the first one in the constructor.
- With the above rule, there cannot be two this() statements in the same constructor (because both cannot be the first).
- this() must be used with constructors only, that too to call the same class constructor (but not super class constructor).

Superclass constructor

- Compiler provides a call to super() if we don't provide.
- By default , no –arg constructor is called by the compiler even if it has overloaded constructors.
- If the superclass constructor has arguments , we can pass it in a subclass constructor like,

```
public class Cat extends Animal{
  public Cat(int size){
    super(size); }
}
```

Invoking super class constructor

```
 public class Cat extends Animal
{
 int size;
  public Cat(int s)
{
 super();
 size=s;
}
}
```

Object lifecycle

- Objects life depends on the life of references referring to it.
- Depends whether reference variable is local or instance variable.
- A local variable lives only within the method that declared the variable
- An instance variable lives as long as the object does.

Garbage collection

- An object becomes eligible for GC when its last live reference disappears.
- The reference goes out of scope, permanently
- The reference is assigned to another object
- The reference is explicitly set to null
- When an object is no longer referred to by any variable, JVM makes it to be garbage collected.

Access specifiers

- private

  Only this class can use this constructor.

- protected

  Subclasses of this class and classes in the same package can use this constructor.

- public

  Any class can use this constructor.

- no specifier

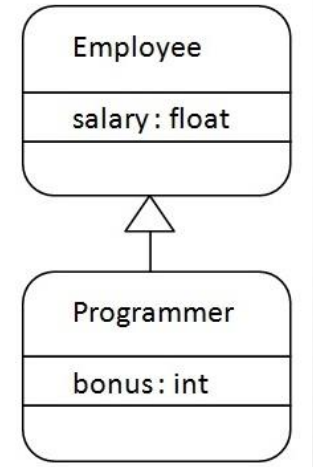
  Gives package access. Only classes within the same package and this class can use this constructor.

Naming conventions

class name	should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc.
interface name	should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc.
method name	should start with lowercase letter and be a verb e.g. actionPerformed(), main(), print(), println() etc.
variable name	should start with lowercase letter e.g. firstName, orderNumber etc.
package name	should be in lowercase letter e.g. java, lang, sql, util etc.
constants name	should be in uppercase letter. e.g. RED, YELLOW, MAX_PRIORITY etc.

JAVA PROGRAMMING

INHERITANCE



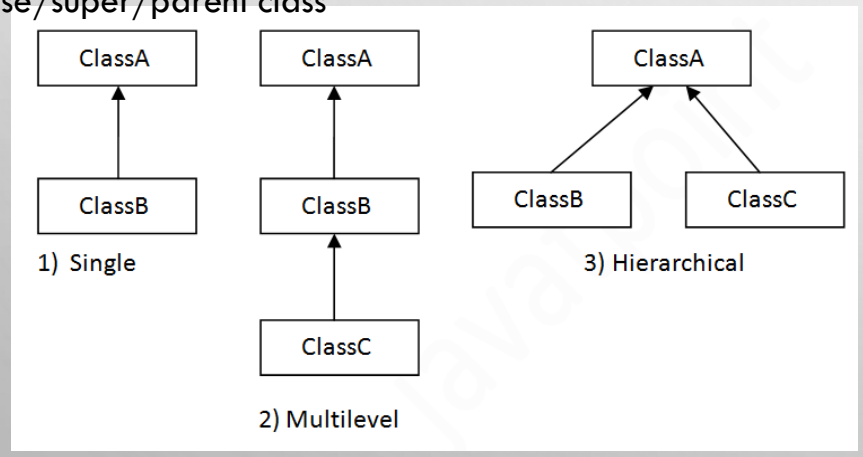
- **Inheritance** is a mechanism in which one object acquires all the properties and behaviours of parent object. Type of relationship is "is-a". Java uses "super" keyword.
- Advantages:
  - For Method Overriding (So Runtime Polymorphism).
  - For Code Reusability.
- **Syntax:** **class** Subclass-name **extends** Superclass-name

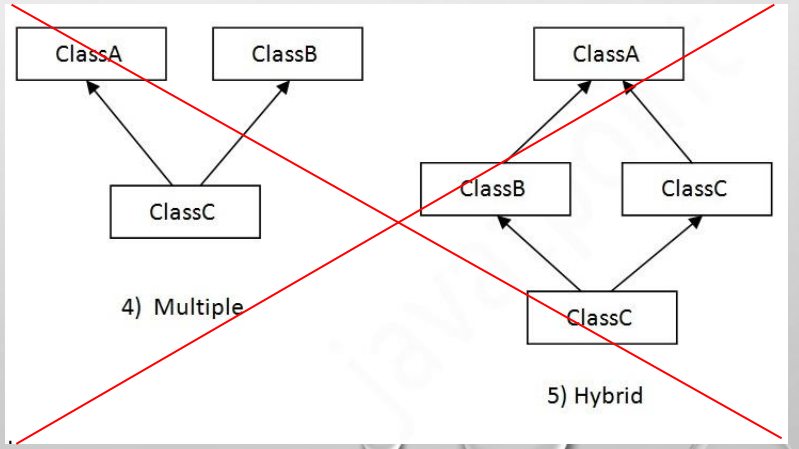
```
{
    //methods and fields
}
```

```
public class Animal{ }
public class Mammal extends Animal{ }
public class Reptile extends Animal{ }
public class Dog extends Mammal{ }
```

Base/super/parent class

Derived /sub /child class





POLYMORPHISM

Polymorphism in java is a concept by which we can perform a ***single action by different ways*.** Polymorphism is derived from 2 greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

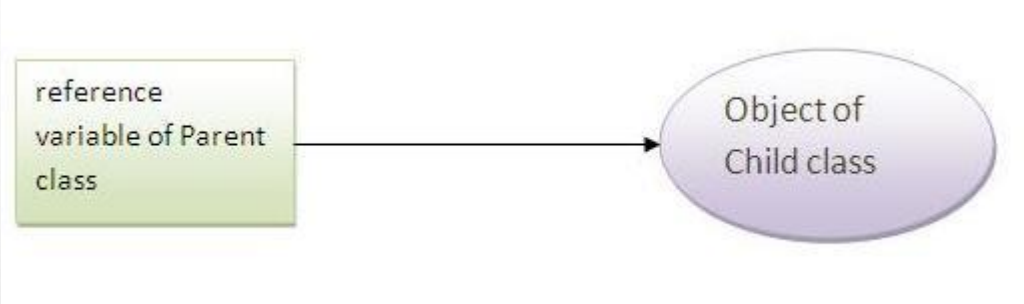
An overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Method is overridden not the data members, so **runtime polymorphism can't be achieved by data members.**

POLYMORPHISM

Upcasting

When reference variable of Parent class refers to the object of Child class, it is known as upcasting.



For example:

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

A a=new B();        //upcasting
```

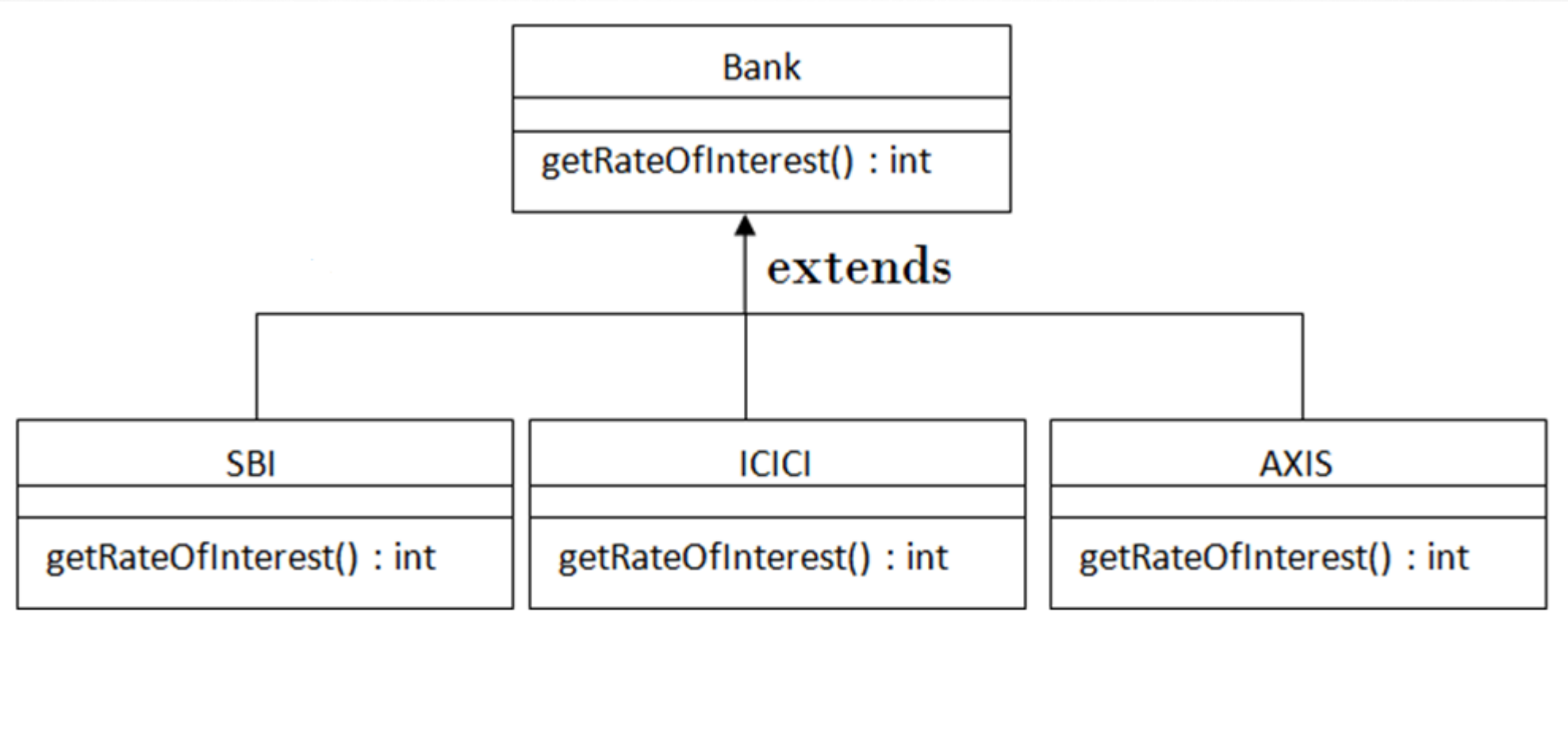
METHOD OVERRIDING

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding**.
- Advantages:
  - provide specific implementation of a method that is already provided by its super class.
  - for Runtime Polymorphism
- **Rules for Method Overriding**
  - method must have same name as in the parent class
  - method must have same parameter as in the parent class.
  - must be IS-A relationship (inheritance).

Method Overloading	Method Overriding
1) Method overloading is used to increase the readability of the program.	Method overriding is used to provide the specific implementation of the method that is already provided by its super class.
2) method overloading is performed within a class.	Method overriding occurs in two classes that use inheritance.
3) In case of method overloading parameter must be different.	In case of method overriding parameter must be same.

POLYMORPHISM

For Example:



POLYMORPHISM

Try output:

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

class Dog extends Animal{
void eat(){System.out.println("dog is eating...");}
}

class BabyDog1 extends Dog{
public static void main(String args[]){
Animal a=new BabyDog1();
a.eat();
}}
```

STATIC BINDING AND DYNAMIC BINDING

- Connecting a method call to the method body is known as binding.
- There are two types of binding
  1. static binding (also known as early binding).
  2. dynamic binding (also known as late binding).

Static binding:

- When type of the **object is determined at compiled time(by the compiler)**, it is known as static binding.
- If there is any **private, final or static method in a class**, there is static binding.

```
class Dog{
 private void eat(){System.out.println("dog is eating...");}

 public static void main(String args[]){
  Dog d1=new Dog();
  d1.eat();
 }
}
```

STATIC BINDING AND DYNAMIC BINDING

Dynamic binding

When type of the **object is determined at run-time**, it is known as dynamic binding.

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

class Dog extends Animal{
 void eat(){System.out.println("dog is eating...");}

 public static void main(String args[]){
  Animal a=new Dog();
  a.eat();
 }
}
```

ABSTRACT CLASS

A class that is declared with abstract keyword, is known as abstract class in java. It can have **abstract and non-abstract methods** (method with body).

An abstract class is one **that cannot be instantiated.** All other functionality of the class still exists, and its fields, methods, and constructors are all accessed in the same manner. You just cannot create an instance of the abstract class.

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

There are two ways to achieve abstraction in java

- Abstract class (0 to 100%)
- Interface (100%)

Syntax to declare the abstract class : **abstract class <class_name>{}**

Syntax to declare the abstract method: **abstract return_type <method_name>();//no braces{}**

ABSTRACT CLASS

Rule:

- If there is any abstract method in a class, that class must be abstract.
- An abstract class can have data member, abstract method, method body, constructor and even main() method.
- If you are extending any abstact class that have abstract method, you must either provide the implementation of the method or make this class abstract.

ABSTRACT CLASS

```
abstract class Bike{
   Bike(){System.out.println("bike is created");}
   abstract void run();
   void changeGear(){System.out.println("gear changed");}
}

class Honda extends Bike{
void run(){System.out.println("running safely..");}
}
class TestAbstraction2{
public static void main(String args[]){
 Bike obj = new Honda();
 obj.run();
 obj.changeGear();
}
}
```

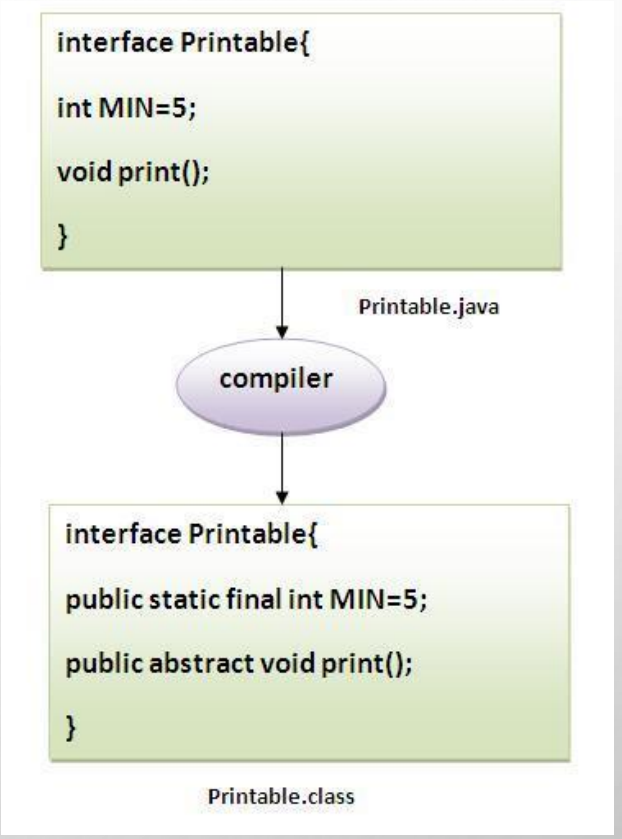
INTERFACE

1. An **interface** is a blueprint of a class. It has static constants and abstract methods.
2. The interface is **a mechanism to achieve fully abstraction** in java. There can be only abstract methods in the interface. It is used to achieve fully abstraction and multiple inheritance in Java.
3. Interface also **represents IS-A relationship**.

There are mainly three reasons to use interface. They are given below.

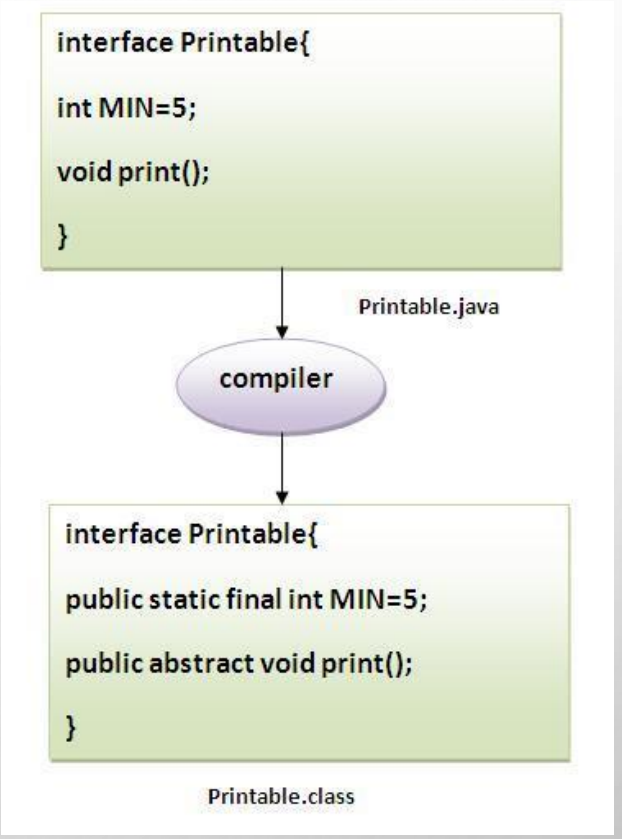
- It is used to achieve fully abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

Rule: The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.



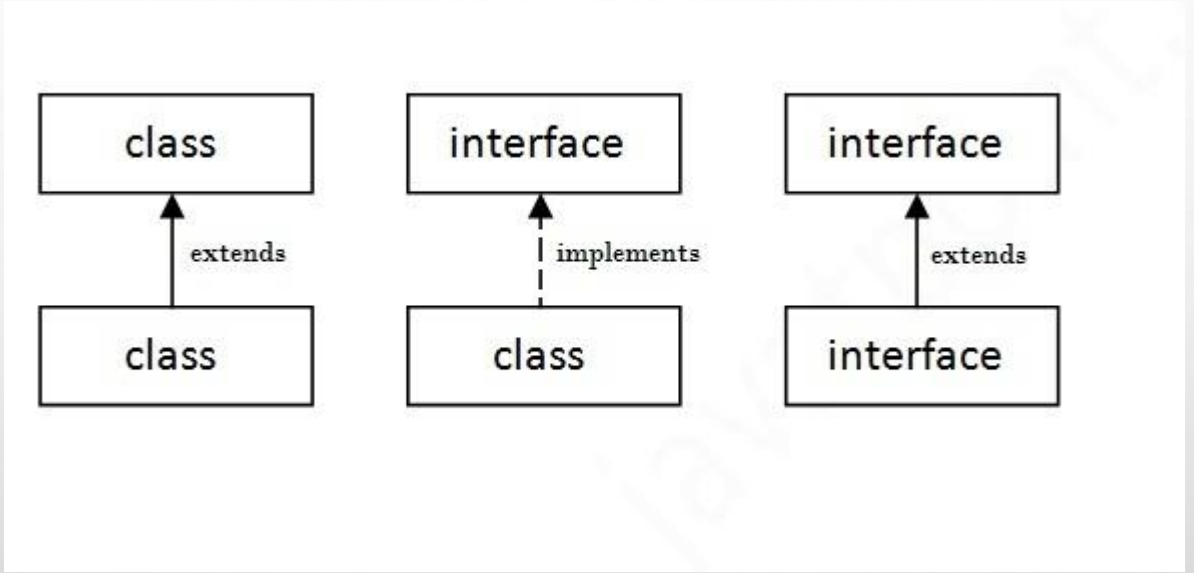
INTERFACE

```
interface printable{
void print();
}

class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
 }
}
```



RELATIONSHIP BETWEEN CLASSES AND INTERFACES

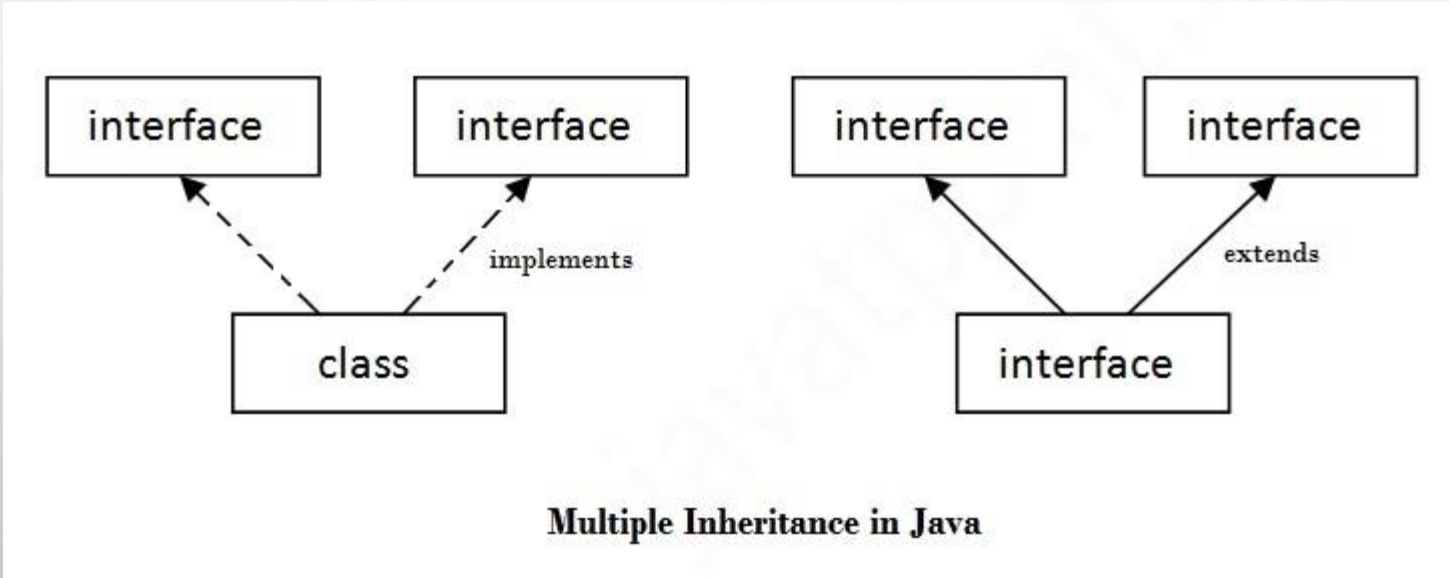
A class extends another class, an interface extends another interface but a **class implements an interface.**



A class implements interface but One interface extends another interface .

MULTIPLE INHERITANCE IN JAVA BY INTERFACE

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.



Multiple Inheritance in Java

INTERFACE

An interface that have no member is known as marker or tagged interface. For example: Serializable, Cloneable, Remote etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

```
//How Serializable interface is written?
public interface Serializable{}
```

An interface can have another interface i.e. known as nested interface

```
interface printable{
      void print();
      interface MessagePrintable{   void msg();
      }
}
```

Differences between Abstract and Interfaces

Abstract class	**Interface**
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods.
2) Abstract class **doesn't support multiple inheritance.**	Interface **supports multiple inheritance.**
3) Abstract class **can have final, non-final, static and non-static variables.**	Interface has **only static and final variables.**
4) Abstract class **can have static methods, main method and constructor.**	Interface **can't have static methods, main method or constructor.**
5) Abstract class **can provide the implementation of interface.**	Interface **can't provide the implementation of abstract class.**
The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

7) **Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

INTERFACE

How to compile the Package (if not using IDE)

javac -d directory javafilename

Ex: javac -d . Simple.java

Note:

How to run the Package (if not using IDE)

To Compile: javac -d . Simple.java
To Run: java mypack.Simple

How to access package from another package

There are three ways to access the package from outside the package.

- import package.*;
- import package.classname;
- fully qualified name. Ex: package.A obj = new package.A();

PACKAGES

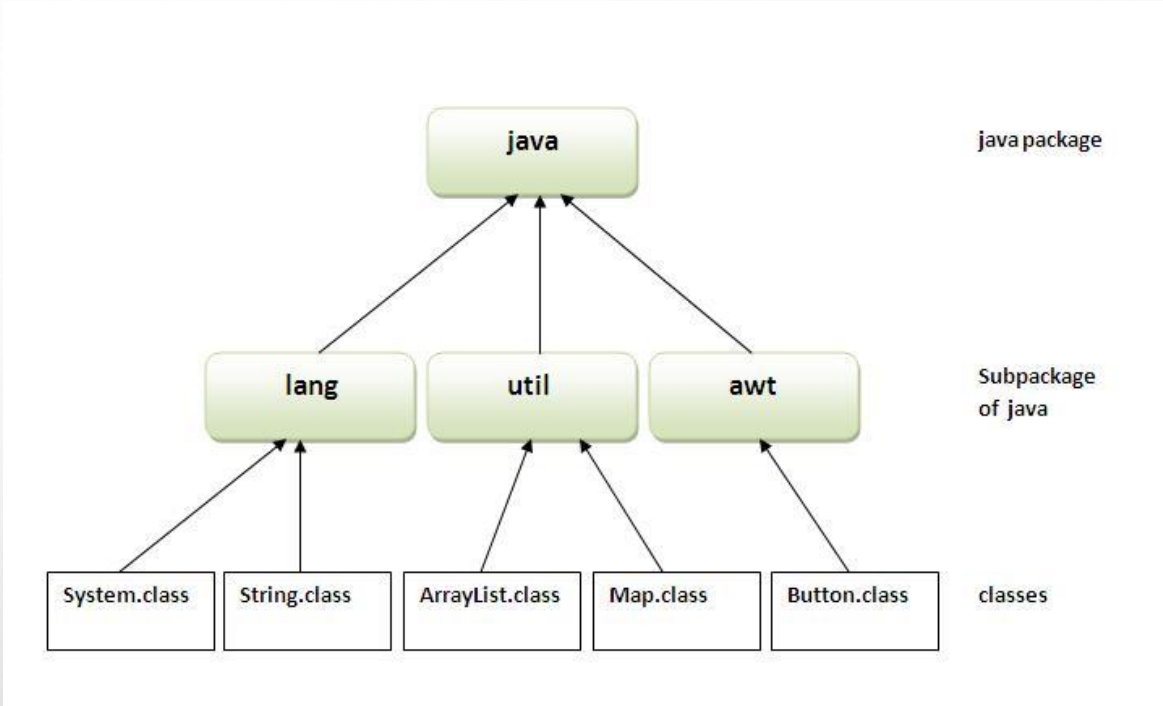
A package is a namespace for organizing classes and interfaces in a logical manner.
A **package** is a group of similar types of classes, interfaces and sub-packages.
Advantage : Placing your code into packages makes large software projects easier to manage.

Package can be categorized in two forms

- built-in package
- user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

The **package keyword** is used to create a package.



SUBPACKAGE

Package inside the package is called the **subpackage.**

If you import a package, subpackages will not be imported. you need to import the subpackage as well.

The standard of defining package is **domain.company.package e.g. com.javatpoint.bean or org.sssit.dao.**

There can be only one public class in a java source file and it must be saved by the public class name.

to put two public classes in a package code as below:

```
//save as A.java
package javatpoint;
public class A{}

//save as B.java
package javatpoint;
public class B{}
```

WRAPPER CLASSES

- Arise the need to consider variables of primitive data type as reference types.
- Wrapper classes are used to wrap the data in a new object which contains the value of that variable.
  - Ex: Integer intObject = new Integer (34); // boxing
    Integer intObject = new Integer ("34");
    int x = intObject.intValue(); // unboxing
- **Retrieving the value wrapped by a wrapper class object**
  the other seven wrapper classes also have methods *Value(), where star refers to the corresponding data type
- **Auto boxing and auto unboxing**
  - Auto boxing refers to an implicit call to the constructor
  - auto unboxing refers to an implicit call to the *value() method
- **Conversion between data types**
  - the format parse*() where * refers to any of the primitive data types except char
  - to convert any of the primitive data type value to a String, we use the valueOf() methods of the String class

Data Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

AUTO BOXING AND UNBOXING

The automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing and opposite operation is known as unboxing.

Advantage:

No need of conversion between primitives and Wrappers manually so less coding is required.

```
class BoxingExample1{
  public static void main(String args[]){
    int a=50;
        Integer a2=new Integer(a);//Boxing

        Integer a3=5;//Boxing

        System.out.println(a2+" "+a3);
 }
}
```

```
class UnboxingExample1{
  public static void main(String args[]){
    Integer i=new Integer(50);
        int a=i;

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

Autoboxing and unboxing with comparison operators

```
class UnboxingExample2{
  public static void main(String args[]){
    Integer i=new Integer(50);

        if(i<100){          //unboxing internally
        System.out.println(i);
        }
 }
}
```

Autoboxing and unboxing with method overloading

There are some rules for method overloading with boxing:

Widening beats boxing

Widening beats varargs

Boxing beats varargs

1. Widening beats boxing

```
class Boxing1{
  static void m(int i){System.out.println("int");}
  static void m(Integer i){System.out.println("Integer");}

  public static void main(String args[]){
   short s=30;
   m(s);
 }
}
```

Autoboxing and unboxing with method overloading

2. Widening beats varargs

```
class Boxing2{
  static void m(int i, int i2){System.out.println("int int");}
  static void m(Integer... i){System.out.println("Integer...");}

  public static void main(String args[]){
   short s1=30,s2=40;
   m(s1,s2);
 }
}
```

3. boxing beats varargs

```
class Boxing3{
  static void m(Integer i){System.out.println("Integer");}
  static void m(Integer... i){System.out.println("Integer...");}

  public static void main(String args[]){
   int a=30;
   m(a);
 }
}
```

Autoboxing and unboxing with method overloading

```
class Boxing4{
  static void m(Long l){System.out.println("Long");}

  public static void main(String args[]){
   int a=30;
   m(a);
 }
}
```

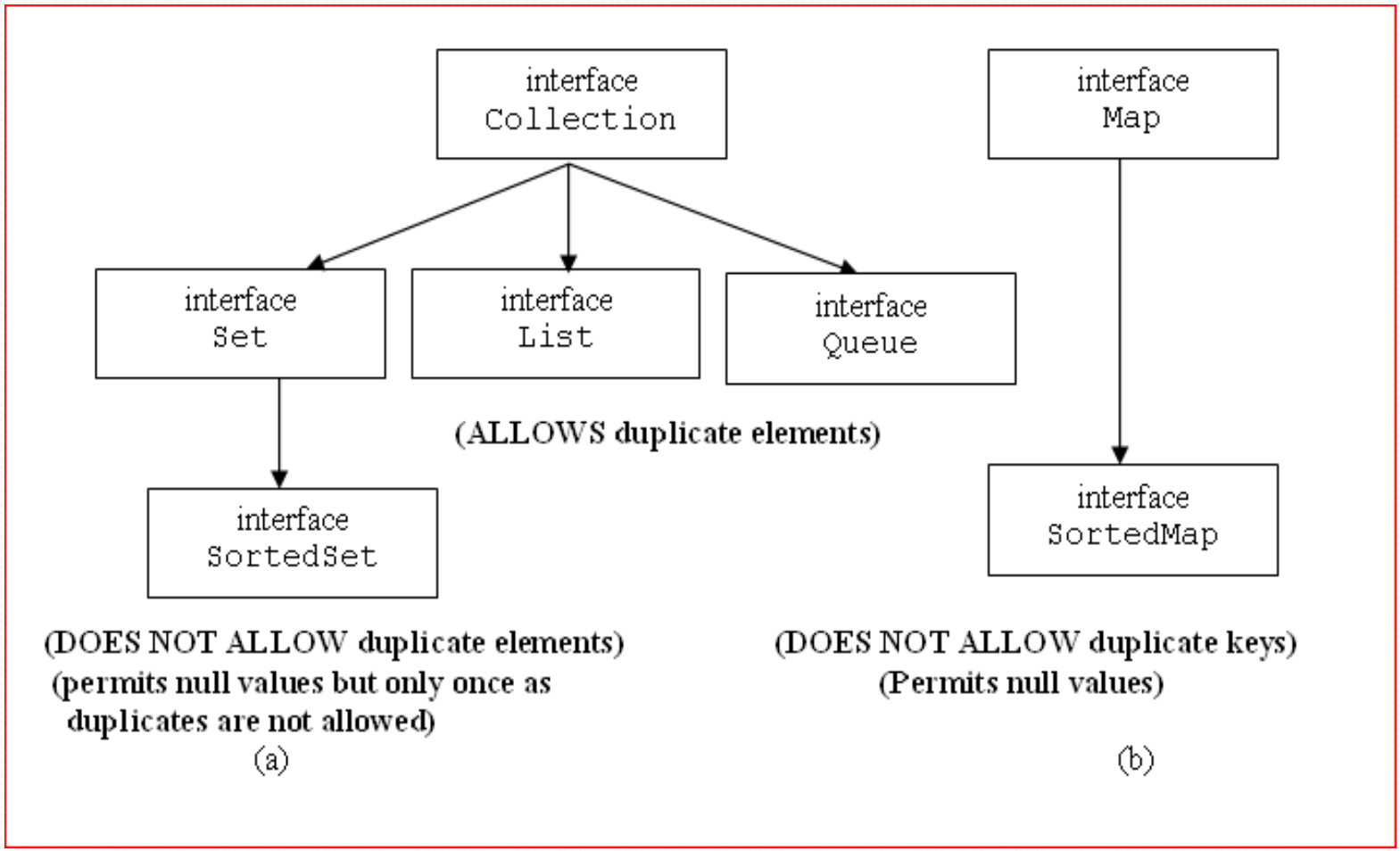
Output : compilation error

Collection Framework

Java Collection Framework

- A collection is a container object that represents a group of objects, often referred to as elements.
- In Java, a data structure is a class , grouping multiple elements as a single entity
- Contrary to arrays, a data structure grows dynamically when more elements are placed
- Operations like searching, sorting, insertion, deletion can be performed by java collection FW easily.
- The Java Collection Framework supports collections, named sets, lists and maps.

- java.util package contains all the classes and interfaces for collection framework



Interface	Description	Derived classes
Collection	It is a root interface . It comes with method s that define basic operations	Majority of the classes
List	Subclasses allow duplicate elements	LinkedList, ArrayList, Vector
Set	Subclasses does not allow duplicate elements	HashSet, LinkedHashSet
SortedSet	It prints the elements in **ascending** order implicitly. Being sub interface of Set, it allows only unique elements	TreeSet
Map	Allows the subclasses to store key/value pairs. Does not allow duplicate keys.	HashMap, LinkedHashMap, Hashtable
SortedMap	Prints elements in ascending order of keys. Being a sub interface of Map, it will not allow duplicate keys	TreeMap

Methods of Collection Interface

```
public interface Collection<E> extends Iterable {

  // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    boolean add(E o);
    boolean remove(Object o);
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection c);
    boolean addAll(Collection c);
    boolean removeAll(Collection c);
    boolean retainAll(Collection c);
    void clear();
    boolean equals(Object o);
    int hashCode();

    // Array operations
    Object[] toArray();
}
```

Methods	Description
public boolean add(Object o)	is used to insert an element in this collection. Returns true if it is successful
public boolean addAll(collection c)	is used to insert the specified collection elements in the invoking collection.
public boolean remove(Object o)	is used to delete an element from this collection. Returns true on successful deletion
public boolean removeAll(Collection c)	is used to delete all the elements of specified collection c , from the invoking collection.
public boolean retainAll(Collection c)	is used to delete all the elements of invoking collection except the specified collection c.
public boolean isEmpty()	checks if collection is empty.

Methods	Description
public int size()	return the total number of elements in the collection.
public void clear()	removes the total no of element from the collection.
public boolean contains(object o)	Determines if the o is present in collection or not
public boolean containsAll(Collection c)	is used to search the specified collection in this collection.
public Iterator iterator()	returns an iterator, to iterate elements of collection
public Object[] toArray()	converts collection into array.
public boolean equals(Object element)	matches two collection.
public int hashCode()	returns the hashcode number for collection.

Traversing collections

- **2 ways to traverse:**
- Enhanced for loop
- Iterators

- **enhanced for loop**

  for(object o:collection)
  SOP(o);

- Ex:

  Collection <String> c= **new ArrayList <String> ();**
  for(String s : c)
  System.*out.println(" "+s);*

Iterator

- An Iterator is an interface that enables us to traverse through a collection.

- We can use iterator for a collection by calling its `iterator()` method.

Iterator it= listName.Iterator();

Ex: Iterator it= c.iterator();

```
while(it.hasNext())
    System.out.println(it.next());
```

- The following is the `Iterator` interface.

```
public interface Iterator<E> {
 boolean hasNext();
 E next();
 void remove(); //rarely used
}
```

Iterator methods

- hasNext()→ returns true if the iteration has more elements
- next()→ returns next element.
- remove() → it is called only once per call to next() method.

  else it throws exception

List iterator

- used to traverse the list in backward and forward direction.
- It is available for the classes which implements List interface
- Methods used are:

```
public interface ListIterator<E> extends Iterator<E> {
 public boolean hasNext();
 public Object next();
 public boolean hasPrevious();
 public Object previous();
 public int nextIndex();
 public int previousIndex();
 public Void remove();
 public void add(Object o);
```

Collection classes

public interface **List** extends Collection

- List — an ordered collection (sometimes called a *sequence*).
- Like a dynamic array
- Lists can contain duplicate elements.
- In the list each element is inserted and can access elements by their integer index (position).
- Random access because it works at index basis.
- Maintains insertion order.

public interface **List** extends Collection

```
public interface List extends Collection {

    // Positional access
    Object get(int index);
    O set(int index,  object o);
    boolean add(object o);
    void add(int index, object o);
    Object remove(int index);

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator listIterator();
    ListIterator listIterator(int index);

    // Range-view
    List subList(int from, int to);
}
```

ArrayList → implements List

- The elements added to list are stored internally as an array.
- It implements all list operations and it also permits duplicate elements, includes null.
- It has 3 constructors :

ArrayList() This constructor is used to create an empty list with an initial capacity sufficient to hold default number of elements.
ArrayList(Collection c) This constructor is used to create a list containing the elements of the specified collection c.

LinkedList → implements List

- It is node- based list
- It also allows duplicate and null elements
- It has 2 constructors

LinkedList() This constructor is used to create an empty list
LinkedList (**Collection c**) This constructor is used to create a list containing the elements of the specified collection c.

LinkedList defines its own specific methods : few of them are

void addFirst(Object O)	Adds the element obj at the beginning
void addLast(Object O)	Adds the element obj at the last
Object peek()	Returns the first element but does not delete it
Object poll()	It works just like peek(), but deletes also
boolean offer(Object O)	Adds the element obj at the last in the list
Object getFirst()	Returns the first element in the list.
Object getLast()	Returns the last element in the list.
Object removeFirst()	Returns and deletes the first element.
Object removeLast()	Returns and deletes the last element.

- Ordered Lists

  - **List can be iterated using:**
  - Iterator interface
  - For-each loop(enhanced for loop)
  - Index based

ArrayList [AL] v/s LinkedList [LL]

- Search: AL searching is pretty faster than the LL

  since AL will do index based searching like arrays where as LL has to traverse through all the nodes

- Insertion :LL insertion is faster than AL

  since in AL inserting at any point needs to rearrange all the other elements , to fill the space where as LL has 2 pointers, with neighborhood elements so during insertion, only 2 pointers have to be adjusted

- Deletion : LL deletion is faster than AL
- Memory overhead: Al has only data but LL has data and pointers in it.

Collections class

- Collections is a class defined in java.lang package.
- Collections class contains many **static** methods with which data structures manipulation becomes easier –
- all the static methods are put together , which defines different algorithms of DS
- Class definition is :

  public class Collections extends Object

Methods	Desc
static void sort(List list1):	Sorts the list list1 into ascending order
static int binarySearch(List list1, Object obj1):	Searches the obj1 in the list list1. Returns the index number of the element obj1
static void reverse(List list1):	Existing order of the elements in the list list1 are reversed.
static void shuffle(List list1)	Shuffles the existing elelments of list1
static void swap(List list1, int index1, int index2)	In list1 , elements at the index, index1 and index2 are swapped
static void fill(List list1, Object obj1)	Replaces all the elements of list1 with obj1.
static void copy(List destination1, List source1)	Copies all the elements of list source1 to destination1
static Object min(Collection col1)	Returns minimum value in the collection
static Object max(Collection col1)	Returns maximum value in the collection

static void rotate(List list1, int dist1)	Rotates the elements in the list1 by specified distance dist1
static boolean replaceAll(List list1, Object oldObj, Object newObj)	Replaces the oldObj with the newObj in the list1
static int indexOfSubList(List source, List target)	Checks for the existence of the few elements in another list. Returns the index number of the first occurrence of the target list in the source list.
static int frequency(Collection col1, Object obj1)	Checks how many times obj1 exists in col1
static boolean disjoint(Collection col1, Collection col2)	Return true if collection doesnt have common elements

public interface **Set** extends Collection

- Set — a collection that cannot contain duplicate elements.
- It inherits the same methods from the collection interface but adds stronger contract on hashcode() and equals () method
- It allows **null** elements. But it permits only one null element as set does not allow duplicates

public interface **Set** extends Collection

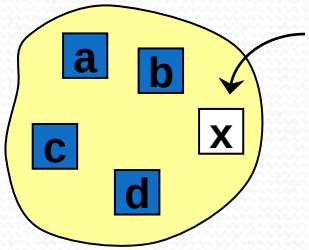
```
public interface Set<E> extends Collection<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E object);          //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optional
    boolean removeAll(Collection<?> c);        //optional
    boolean retainAll(Collection<?> c);        //optional
    void clear();                              //optional

    // Array Operations
    Object[] toArray();

}
```

Nothing added to Collection interface – except no duplicates allowed

Set - Example

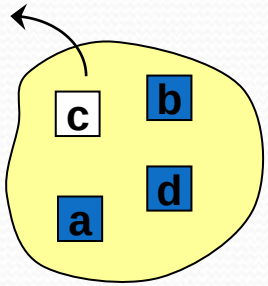
add(x)
→*true*

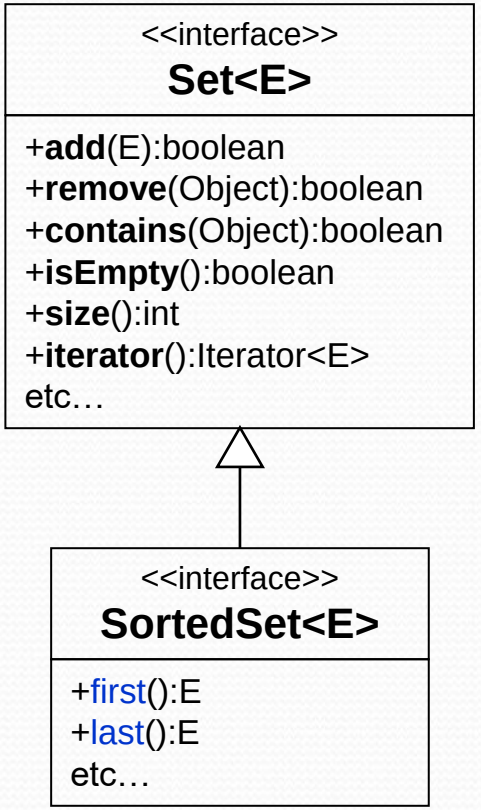
add(b)
→*false*



remove(c)
→*true*

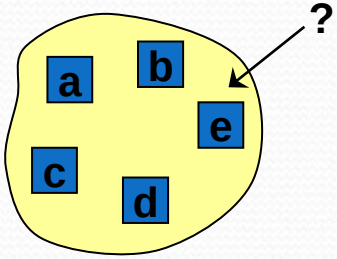
remove(x)
→*false*





contains(e)
→true

contains(x)
→*false*



isEmpty()
→*false*

size()
→*5*

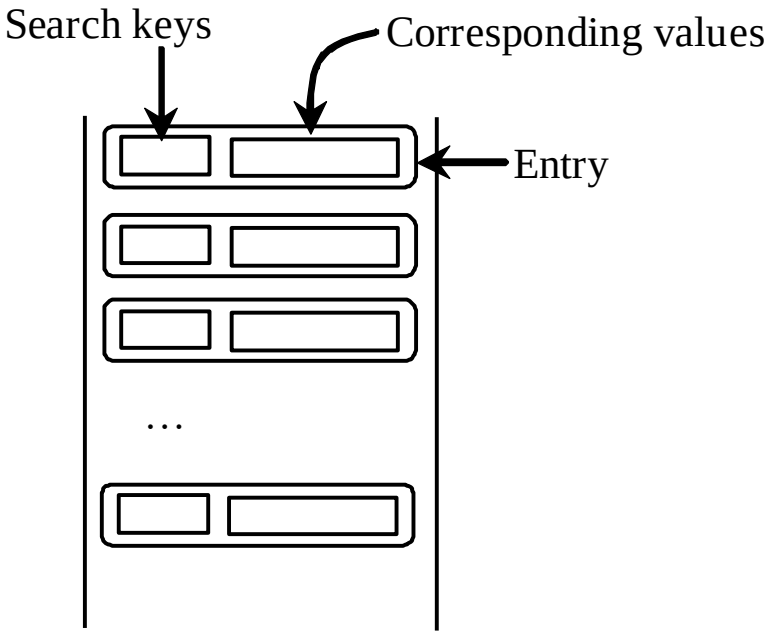
Unordered Sets

A **set** is an unordered collection of unique elements.

- **HashSet**
  - Uses hash tables to store the elements.
  - Permits a null element.
  - No guarantee of insertion order
- **TreeSet**
- Elements are kept in ascending order
- Uses a binary tree to store the elements.
- **LinkedHashSet**
- Implements doubly linked list
- Can retrieve elements based on insertion-order

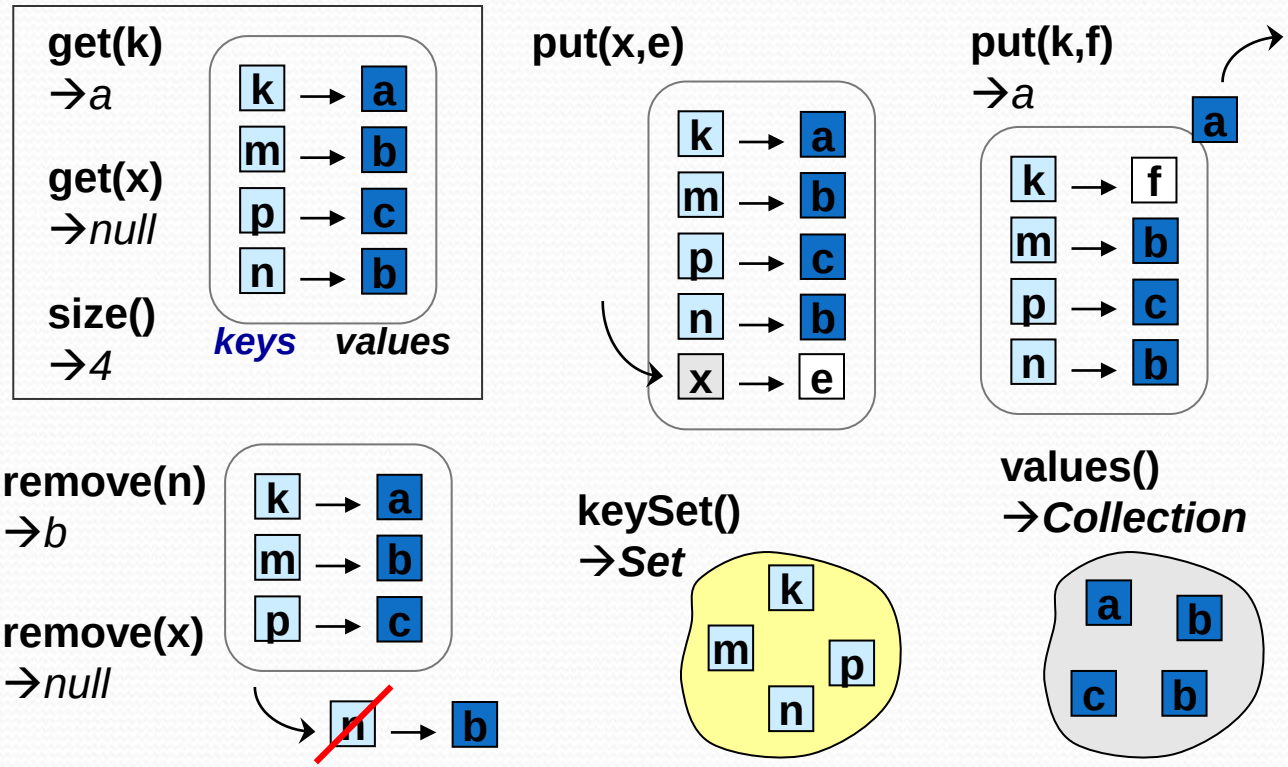
The Map Interface

- The Map interface maps keys to the values(elements).
- Map cannot have duplicate keys.
- The keys are like indexes.
- In List, the indexes are integer. In Map, the keys can be any objects.



Methods	Description
Object put(Object key1, Object value1)	Add the key/value pair
boolean containsKey(Object key1)	check the availability of a specific key, **key1**, in the map
boolean containsValue(Object value1)	Used to check the availability of a specific value, **value1**, in the map
Object get(Object key1)	Returns the value associated with the key, **key1**.
Object remove(Objevt O)	Deletes the key key1 from the map.
void putAll(Map m1):	Used to place the key/value pairs of one map into another
Set keySet()	Returns the entire keys of the map as a Set object.
Collection values()	Returns a collection object that contains the entire values of the map
boolean equals(Object obj)	Used to check whether two map objects contain the same key/value pairs
int size():	Returns the number of key/value pairs stored.
Boolean isEmpty()	Returns true if the map does not have any elements
void clear()	Deletes all the elements of the map

Map<K,V>


get(k)
→a
get(x)
→null
size()
→4
k → a
m → b
p → c
n → b
keys
values
put(x,e)
k → a
m → b
p → c
n → b
x → e
put(k,f)
→a
a
k → f
m → b
p → c
n → b
remove(n)
→b
remove(x)
→null
k → a
m → b
p → c
n → b
keySet()
→Set
k
m
p
n
values()
→Collection
a
b
c
b

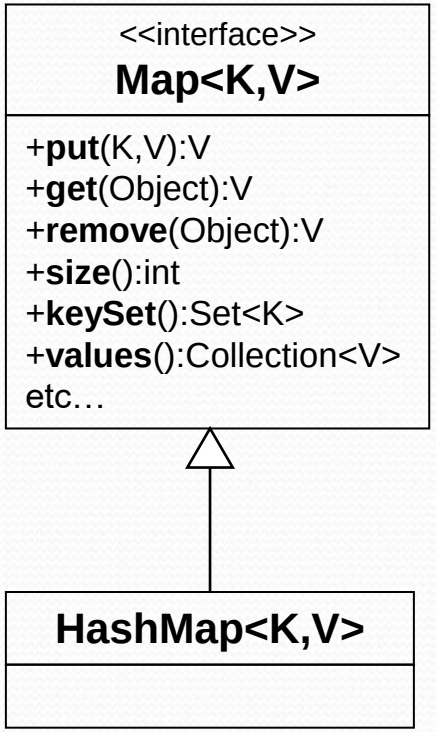
Hash map

- A HashMap contains values based on the key. It implements the Map interface.
- It contains only unique elements.(keys)
- It may have one null key and multiple null values.
- It maintains no order.

- Syntax:
-
```
Map<String,Integer> newMap=
                    new HashMap<String,Integer>();
```

HashMap<K,V>

```
Map<String, Integer> directory
                    = new HashMap<String, Integer>();

directory.put("Mum", 9998888);
directory.put("Dad", 9998888);
directory.put("Bob", 12345678);
directory.put("Edward", 5553535);
directory.put("Bob", 1000000);

System.out.println(directory.size());

for (String key : directory.keySet()) {
  System.out.print(key+"'s number: ");
  System.out.println(directory.get(key));
}

System.out.println(directory.values());
```



Linkedhashmap

- LinkedHashMap was introduced in JDK 1.4. It extends HashMap with a linked list implementation that supports an ordering of the entries in the map
- The entries in a LinkedHashMap can be retrieved in the order in which they were inserted into the map

- Syntax:

```
Map<String,Integer> newMap=
    new LinkedHashMap<String,Integer>();
```

TreeMap<K,V>

- A TreeMap provides an efficient means of storing key/value pairs in **sorted** order, and allows rapid retrieval.

```
Map<String, Integer> directory
                    = new TreeMap<String, Integer>();
directory.put("Mum", 9998888);
directory.put("Dad", 9998888);
directory.put("Bob", 12345678);
directory.put("Edward", 5553535);
directory.put("Bob", 1000000);

System.out.println(directory.size());

for (String key : directory.keySet()) {
  System.out.print(key+"'s #: ");
  System.out.println(directory.get(key));
}

System.out.println(directory.values());
```

4

Loop output?

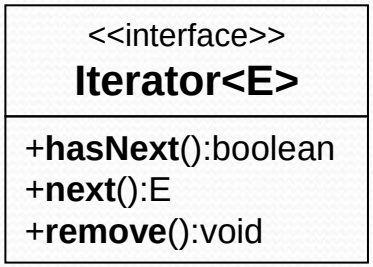
```
Bob's #: 1000000
Dad's #: 9998888
Edward's #: 5553535
Mum's #: 9998888
```

Using of Map.Entry

```
Set sset =m.entrySet();

for(Object i:sset)
    System.out.println("set "+(Map.Entry<Integer, String>)i);

Iterator it= sset.iterator();
while(it.hasNext())
{
    System.out.println("  "+it.next());
    Map.Entry ment=(Map.Entry)it.next();
    System.out.println(ment.getKey()+ "   " +ment.getValue());
}
```

Using an **Iterator**

```
Iterator<Item> it = items.iterator();
  while(it.hasNext()) {
    Item item = it.next();
    System.out.println(item.getTitle());
  }
```



Design notes:

- Above method takes in an object whose class implements Collection
  - List, ArrayList, LinkedList, Set, HashSet, TreeSet, Queue, MyOwnCollection, etc
- No need to know the exact implementation of Iterator we are getting.

Two ways to iterate over collection

```
for (Item item : items) {
  System.out.println(item);
}
```

~

~

```
Iterator<Item> it = items.iterator();
while(it.hasNext()) {
  Item item = it.next();
  System.out.println(item);
}
```

Exceptions in Java

Exception

- An *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions
- Indication of problem during execution of program
- Exception can occur for many different reasons, like:
  - A user has entered invalid data.
  - A file that needs to be opened cannot be found.
  - A network connection has been lost in the middle of communications
  - JVM has run out of memory.

Exception handling

Exception Handling is a mechanism to handle errors.

A java exception is an **Object** that describes an exception that occurred in the program.

These are automatically created when an unexpected situation arises.

Exceptions are always some subclass of **java.lang.Exception**

When an exceptional event occurs in Java, an exception is said to be **"thrown.“**

The code that's responsible for doing something about the exception is called an "exception handler," and it "catches" the thrown exception.

Exception handling works by transferring the execution of a program to an appropriate exception handler when an exception occurs.

Advantages of Exception handling

- normal flow of the application is maintained
- code to handle any particular exception that may occur in the governed region needs to be written only once.
- Separating error handling code from the regular code
- Grouping and differentiating the error types.

Categories of exceptions

- **2 types:**
- **Checked** Exceptions: this is an exception which can be predicted by the programmer.
- A checked exception must be handled.
- the compiler checks to make sure that they're handled or declared.
- Ex: if a file is to be opened, but the file cannot be found, an exception occurs.

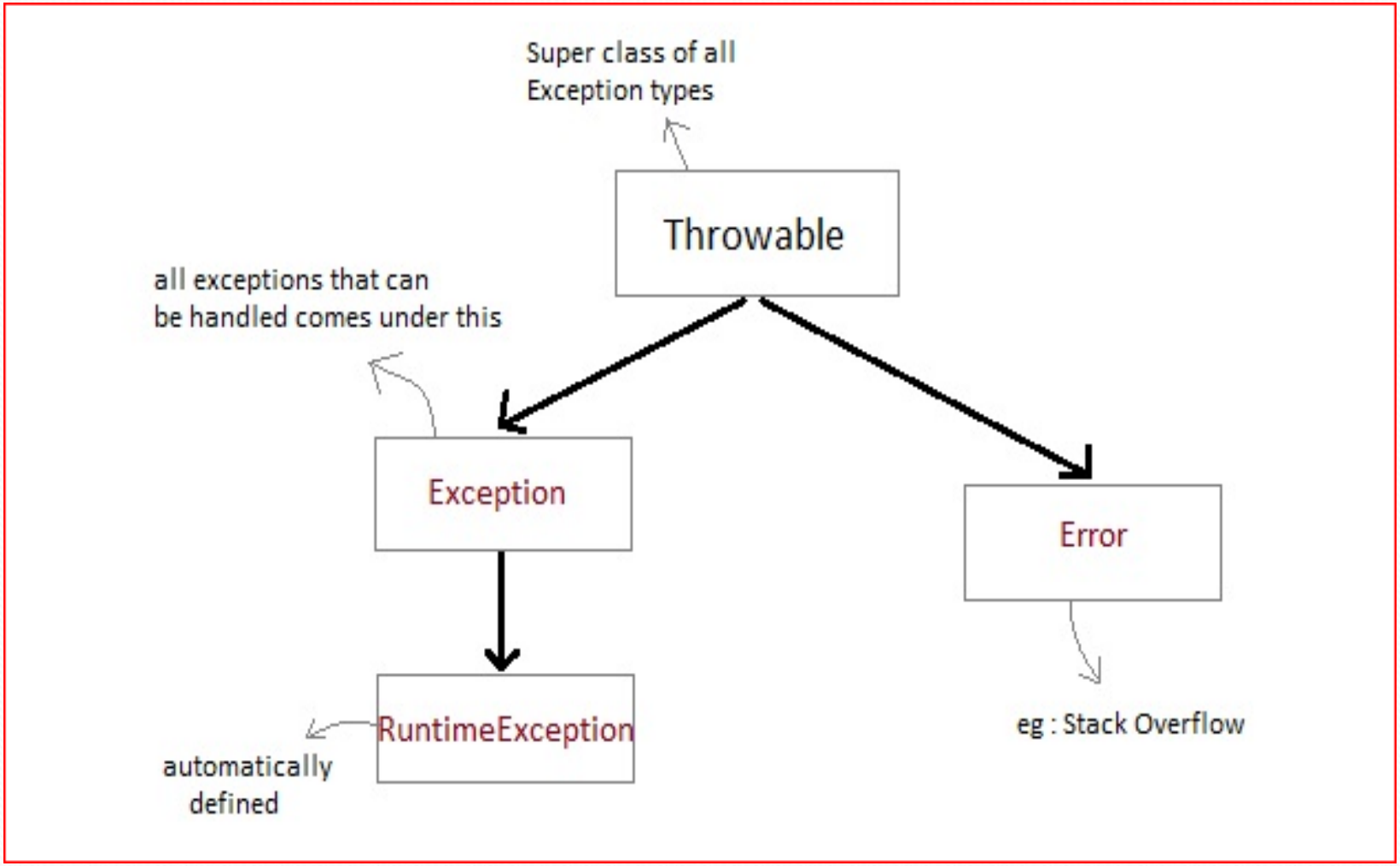
  These exceptions cannot be ignored at the time of compilation.

Categories

- **Unchecked** exception
- These are ignored at compile time.(avoided by the programmer)
- even if they are not handled or checked by the programmer, the program simply compiles.
- Like: ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.
- Compiler gives guarantee of syntax but not output or execution

- **Errors**:
- It is irrecoverable .
- These are the problems that arise beyond the control of the user or the programmer
- Like OutOfMemoryError, due to some HW problems

Exception Hierarchy:



- There are two ways to get information about an exception:
  1) The first is from the **type of the exception itself**.
  2) The next is the information that you can get from the **exception object**.
- If you call an exception object's printStackTrace() method, a stack trace from where the exception occurred will be printed.

Keywords used in Java Exception handling

- try
- catch
- finally
- throw
- throws

- Programmer can use the combination of these in exception handling.
- Exception can be handled in 2 ways :
- Either with **try – catch** block or **throws** exception to calling method

Try Catch block

- **risky code** which might throw an exception is enclosed in try block
- The statements of **try** block may or may not raise the exception.
- If the try throws the exception, the **catch** block is executed.
- The catch block includes an exception handler and some statements used to inform the user about the possible problem

try{

…

}**catch**(Exception_class_Name reference){}

Multiple catch blocks

- One **catch** block can handle only one exception; it is the best practice.
- There may be an occasion where the **try** block may have multiple problems, which can be handled by multiple catch blocks

```
try {
        //risky code here
}
catch(exception1 e)
{
}
catch(exception2 e)
{
}
```

```java
public static void main(String[] args) {
    try
    {
        String s=null;
        System.out.println("len of string is "+s.length());

        int a[]={0,2,3,4};
        System.out.println("array value is "+a[5]);


    }
    catch (NullPointerException e) {
        System.out.println("might be string is null");
    }
    catch(ArrayIndexOutOfBoundsException e)
    {
        System.out.println("please see the ArrayRange");
    }

    System.out.println("rest of the code");

}
```

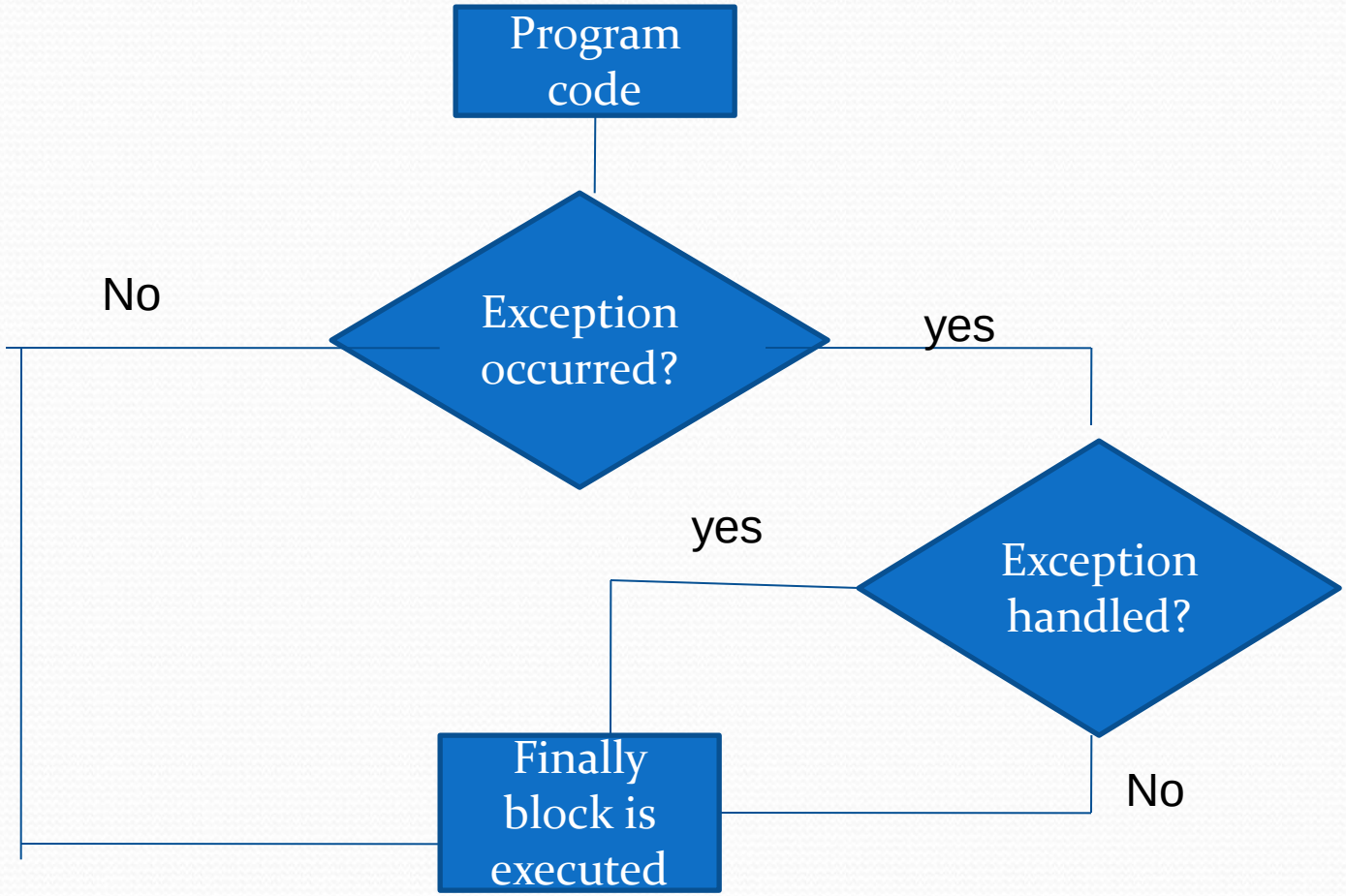
- In multiple catch blocks, each catch block is checked, one after another, for a suitable handler.
- At a time only one Exception is occured and at a time only one catch block is executed.
- When one catch block handles the exception, the next catch blocks are not executed. Control shifts directly after the last catch to execute the remaining part of the program.
- All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception

This is how try{ }catch( ){ } block looks like:

```
1. try {
2. // This is the first line of the risky code session
3. // that is governed by the try keyword.
4. // Put code here that might cause some kind of exception.
5. // We may have many code lines here or just one.
6. }
7. catch(MyFirstException) {
8. // Put code here that handles this exception.
9. // This is the next line of the exception handler.
10. // This is the last line of the exception handler.
11. }
12. catch(MySecondException) {
13. // Put code here that handles this exception
14. }
15.
16. // Some other unguarded (normal, non-risky) code begins here
```

Finally

- The finally block is a block that is always executed.(guaranteed code)
- finally block can be used to put "cleanup" code such as closing a file,closing connection etc.



Important points

- For each try block there can be zero or more catch blocks, but only one finally block.
- there can be try-finally without catch block
- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses when ever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.
- It is legal to omit either the catch clause or the finally clause, but not both.
- if there is a return statement in the try block, the finally block executes before the return executes!

Overall

```
try {
// do stuff
} catch (SomeException ex) {
// do exception handling
} finally {
// clean up
}
```

Question

```
try {
// do stuff
}
// need a catch or finally here
System.out.println("out of try block");

try {
// do stuff
}
System.out.println("out of try block");
catch(Exception ex) {}
```

Is this a legal try block??

Nested try

- Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error.
- We can use nested try there.

```
try
{
      statement 1;
      statement 2;
      try
      {
            statement 1;
            statement 2;
      }
      catch(Exception e)
      {
      }
}
catch(Exception e)
{
}
}
```

Examples

- **NumberFormatException**:

Wrong formatting of any value, may occur this exception .

Like:

String s="ten";

int i=Integer.parseInt(s);

ClassNotFoundException:

load a class manually with forName() method of class Class.

Like: Class.*forName("com.dac.exceptions.CustomExc");*

IndexOutOfBoundsException :

Inserting any value in the wrong index will cause this exception

It comes with 2 classes:

ArrayIndexOutOfBoundsException

StringIndexOutOfBoundsException

Examples

ArithmeticException: Thrown when an exceptional arithmetic condition has occurred

Ex: int a=50/0;//ArithmeticException ,Divide by 0

NullPointerException: Thrown when an application attempts to use null in a case where an object is required.

ClassCastException:

When JVM is unable to cast two objects which is done by the programmer

Ex: Object i=2;

System.*out.println((String)i);*

FileNotFoundException:

the file you would like to open may not exist all

NoSuchMethod:

Try to execute a class, without main()

NoSuchElementException: trying to retrieve element, which doesn't exist

IllegalStateException: trying to remove() without saying next()

ConcurrentModificationException : adding the element, while iteration is going on (fail-fast)

Examples of handling Exceptions

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

    int a[]={1,2,3,4};
    System.out.println("array value is "+a[5]);



}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at com.cdac.dac1.ExceptionEx.main(ExceptionEx.java:8)
```

```
public static void main(String[] args) {
    try
    {
    int a[]={1,2,3,4};
    System.out.println("array value is "+a[5]);
    }
    catch(ArrayIndexOutOfBoundsException e)
    {
        System.out.println("please see the ArrayRange");
    }
```

Throws keyword

- Another way of handling exception
- If a method does not handle a checked exception, the method must declare it using the **throws** keyword.
- The throws keyword appears at the end of a method's signature.
- Syntax:

```
void method_name() throws exception_class_name{
 …
}
```

```
public void methodToThrow() throws FileNotFoundException
{
    FileInputStream i1= new FileInputStream("D:\\DAC Material\\a.txt");

}

public static void main(String[] args) {

    try{
    new ThrowsExc().methodToThrow();
    System.out.println("just after the exc occured");
    }
    catch(IOException e1)
    {
        System.out.println("file nt ter");
    }
    catch(Exception e)
    {
        System.out.println("some error in input");
    }
}
```

Usage of throws

- One is claiming the exception where the designer warns the programmer, the possible problems, that may arise when using the method.
- The second one is as an alternative to try-catch; but not robust way.
- Ex:

public FileInputStream(String filename) ***throws*** *FileNotFoundException*

throw

- The throw keyword is used to explicitly throw an exception.
- The throw keyword is mainly used to throw custom exception.
- With throw keyword the exception object is thrown to the system.
- **throw** is used by the programmer to throw the exception to the system.
- System simply takes the exception object, displays the message and terminates the program.

throw

```
public class ThrowExc {

    public void methodToThrow()
    {
        throw new ArithmeticException("some error is thrown");
    }

    public static void main(String[] args) {

        new ThrowExc().methodToThrow();

    }

}
```

Custom exception or user defined exception

- Creating our own exception – for better messages
- We have to represent new exception by defining new java class which extends **Exception** class.
- May need to have a constructors to show the error messages
- Usage :

create the object of your user defined exception class and process it.

```
 class MyException extends Exception
{
  MyException(String msg)
  {
  super(msg); //calling Exception constructor
  }
}
```

Usage:

```
throw new MyException("Invalid User");
```

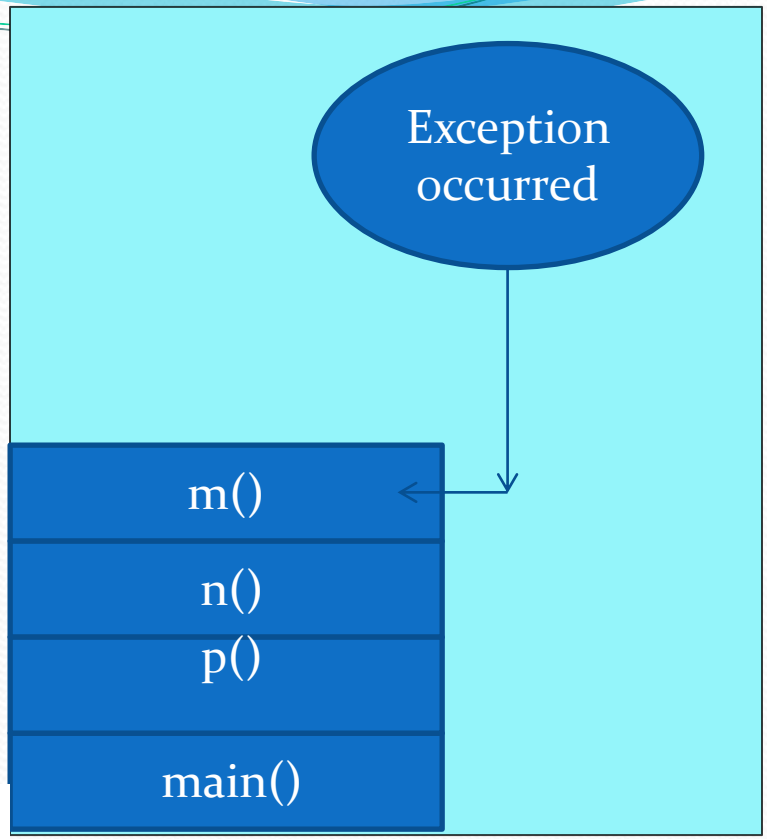
Exception propogation

- An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method, If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

- We can throw the exception out of main() as well. This results in the Java Virtual Machine (JVM) halting, and the stack trace will be printed to the output.

- So, at some point of time one has to handle the exception by putting the try and catch block around the lines of code)

```java
public class SimpleExpTesting {
    void m()
    {
        int i=50/0;
    }

    void n()
    {
        m();
    }

    void p()
    {
        try{
        n();
        }
        catch(Exception e)
        {
            System.out.println("its handled in method p()");
        }
    }

}
```




```java
    public static void main(String[] args) {
        SimpleExpTesting s= new SimpleExpTesting();
        s.p();

    }
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at com.cdac.dac.SimpleExpTesting.m(SimpleExpTesting.java:6)
        at com.cdac.dac.SimpleExpTesting.n(SimpleExpTesting.java:11)
        at com.cdac.dac.SimpleExpTesting.p(SimpleExpTesting.java:16)
        at com.cdac.dac.ExcChecking.main(ExcChecking.java:7)
```

Which exceptions to catch first

- the most specific exceptions must always be placed above those for more general exceptions.
- By specifying an superclass exception (general exception) most valuable information about the exception is discarded.

```
try {
// do risky IO things
} catch (FileNotFoundException ex) {
// handle just FileNotFoundException
}catch (IOException e) {
// handle general IOExceptions
}
```

Methods available

- **getMessage**() is a method of Throwable class
- The getMessage() method prints only the message part of the output printed by object
- Method signature is:

  public java.lang.String getMessage();

```
catch(Exception e)
{
System.out.println(e.getMessage());
}
```

Methods available

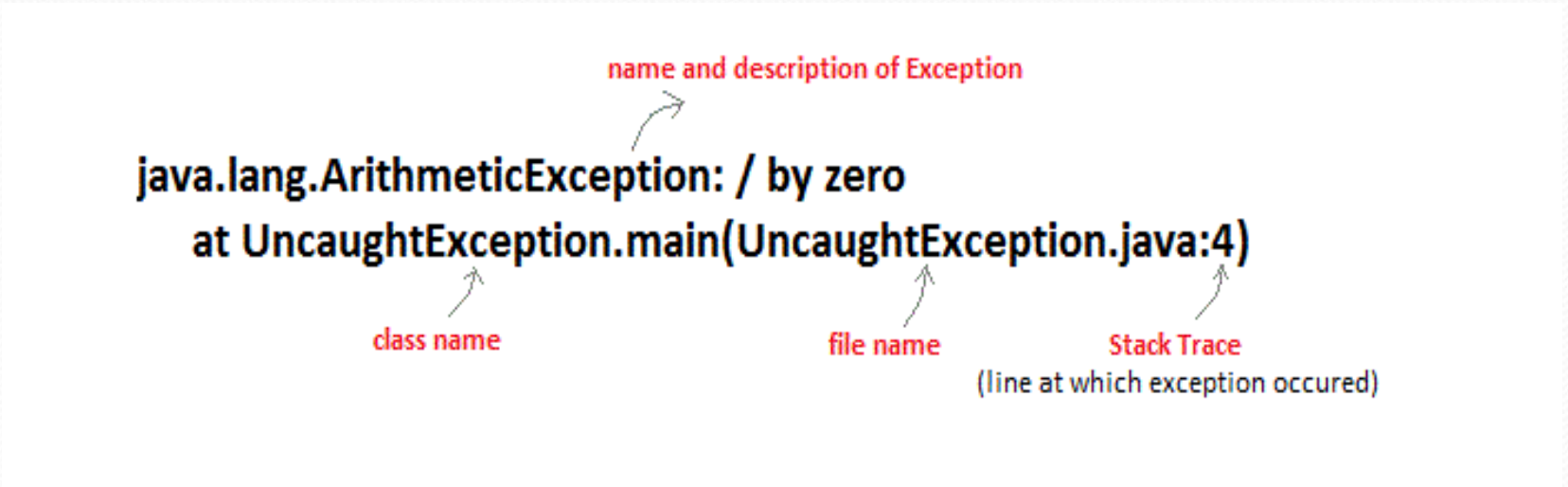
printStackTrace():

This method prints the same message of e object and also the line number where the exception occurred.

Method signature :

```
public void printStackTrace();
try {
//some code goes here
} catch (Exception ex) {
ex.printStackTrace();
}  //getting the info of the exception
```

Ex :


name and description of Exception
java.lang.ArithmeticException: / by zero
at UncaughtException.main(UncaughtException.java:4)
class name
file name
Stack Trace
(line at which exception occured)

Method overriding (some points)

- If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.
- If the superclass method declares an exception, subclass overridden method can declare

1. same exception
2. subclass exception
3. no exception

RumtimeExceptions:

- Some classes extending RuntimeException class are:

1) **ArithmeticException:** Thrown when an exceptional arithmetic condition has occurred
2) **ClassCastException:** Thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance.
3) **IllegalArgumentException:** Thrown to indicate that a method has been passed an illegal or inappropriate argument.
4) **IndexOutOfBoundsException:** Thrown to indicate that an index of some sort (such as to an array, to a string, or to a vector) is out of range.
5) **NullPointerException:** Thrown when an application attempts to use null in a case where an object is required.

ArithmeticException

- Thrown when an exceptional arithmetic condition has occurred. For example, an integer "divide by zero" throws an instance of this class.
- java.lang.ArithmeticException
- public class **ArithmeticException** extends RuntimeException

```
static void doMoreStuff()
{
System.out.println("In doMoreStuff()");
int x= 5/0;
}
```

ClassCastException

Thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance. For example, the following code generates a ClassCastException:

```
Object x = new Integer(0);
System.out.println((String)x);
```

This stacktrace will be printed:

java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String

NullPointerException:

Thrown when an application attempts to use null in a case where an object is required. These include:

- Calling the instance method of a null object.
- Accessing or modifying the field of a null object.
- Taking the length of null as if it were an array.
- Accessing or modifying the slots of null as if it were an array.
- Throwing null as if it were a Throwable value.

```
public class Example {
 public static void main(String[] args) {
 Object obj = null;
  obj.hashCode();
}
}
```

Some errors

OutOfMemoryError:

Thrown when the Java Virtual Machine cannot allocate an object because it is out of memory, and no more memory could be made available by the garbage collector.

VirtualMachineError:

Thrown to indicate that the Java Virtual Machine is broken or has run out of resources necessary for it to continue operating.

IOError:

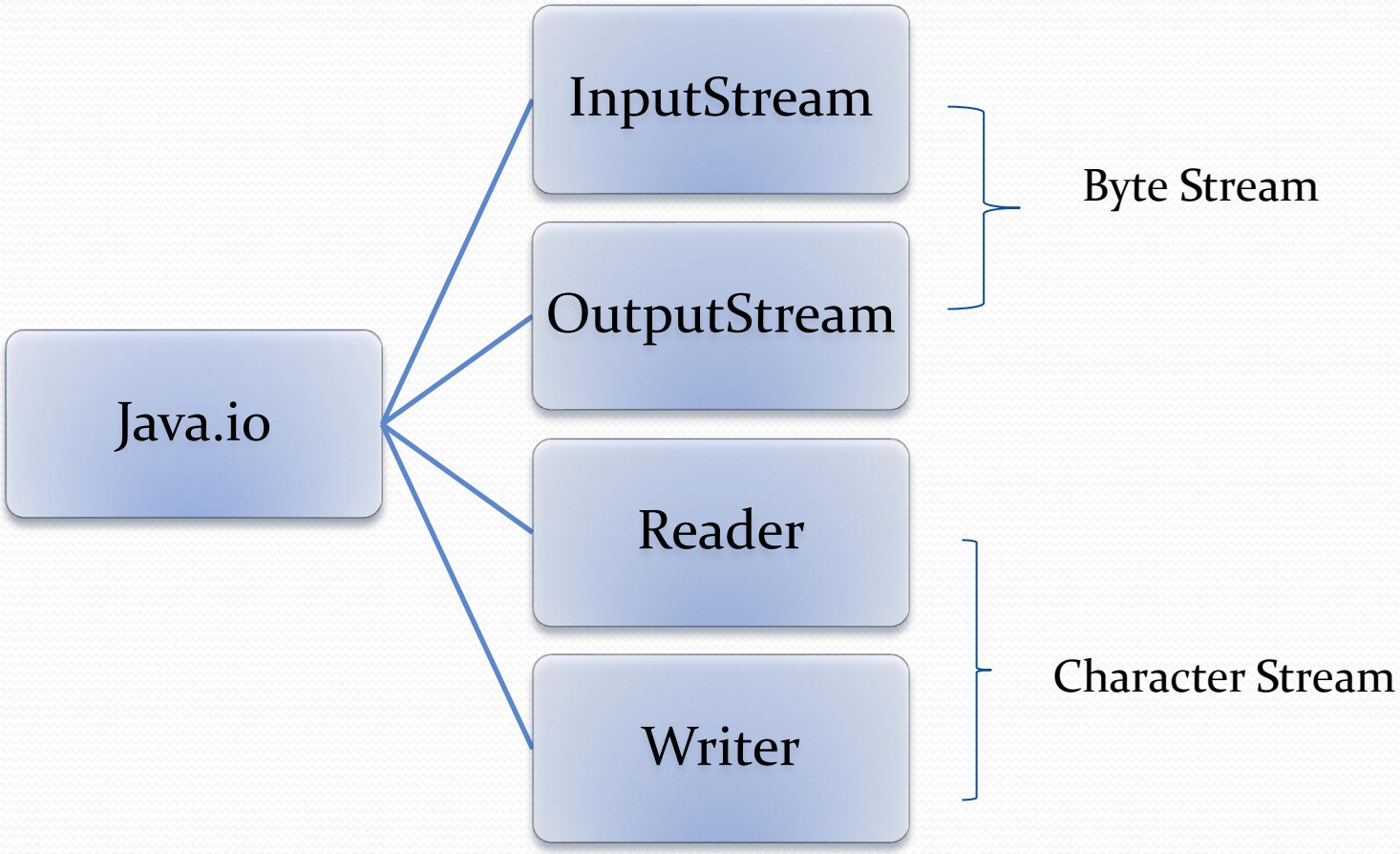
Thrown when a serious I/O error has occurred.

Java IO

Java IO Stream

- An *I/O* represents an input source or an output destination.
- Java performs IO through streams
- Stream means continuous flow of data
- The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.
  here source and destination can be file, array or any peripheral device or socket
- Java encapsulates stream under java.io package
- There are 2 types: byte Stream and Character Stream

Java IO summary



Types of Streams in java

Byte Streams

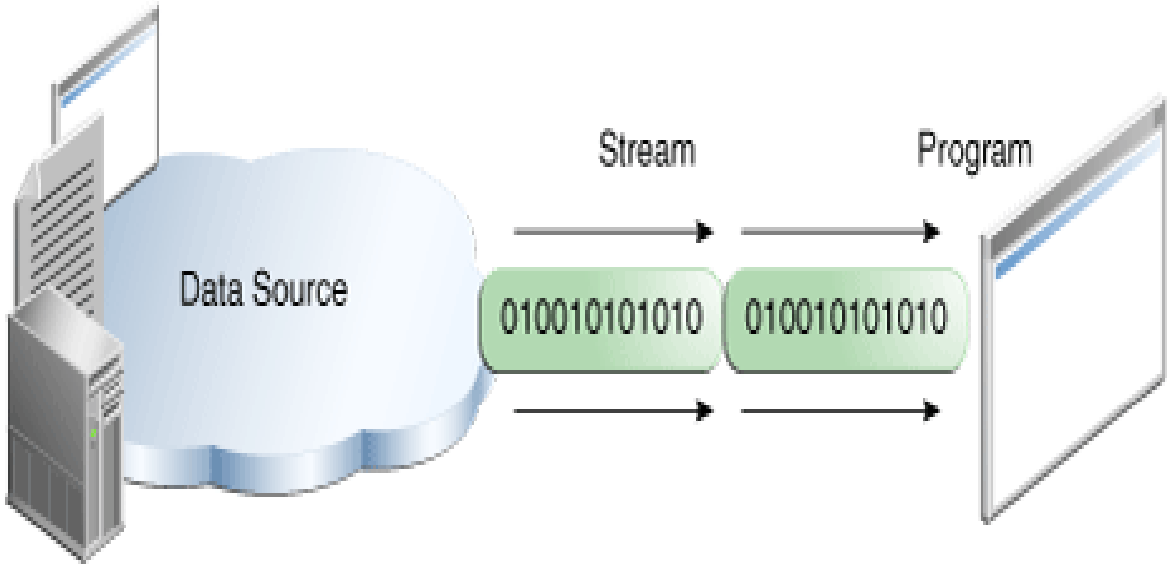
- Byte streams used to perform input/output with the 8 bit data
- Byte streams can access the files containing ASCII characters (only english)

Predefined Streams

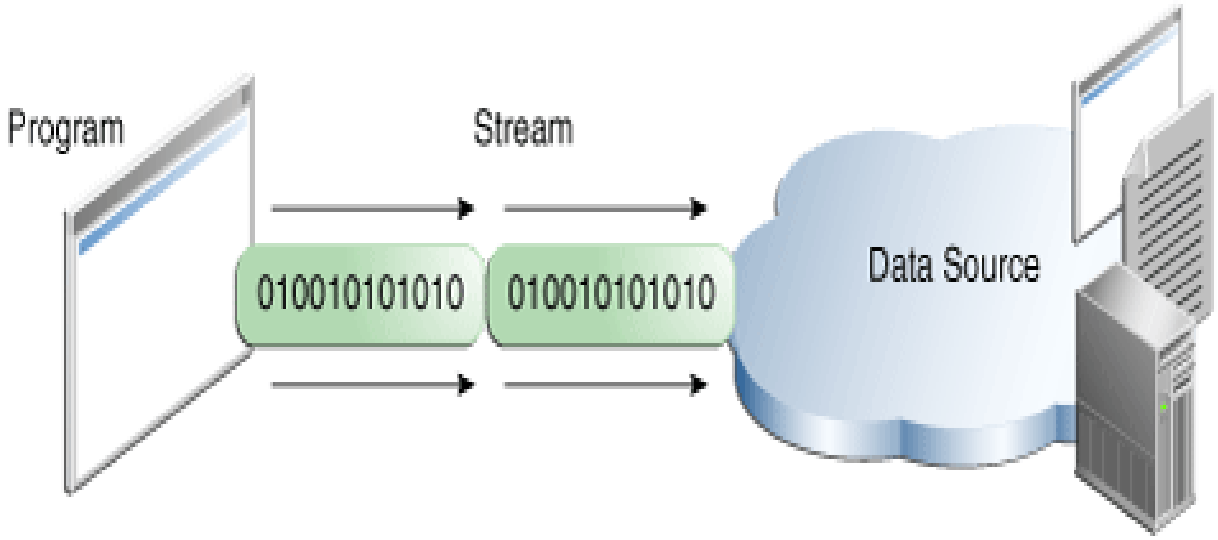
- System.in - Standard Input Stream (InputStream)
- System.out - Standard Output Stream (PrintStream)
- System.err - Standard Error Stream (PrintStream)

Character Streams

- Character streams used to perform input/output of Character data with the Unicode support.
- It supports with unicode characters (other languages)



Reading from the sources



Writing into the sources

How to do I/O

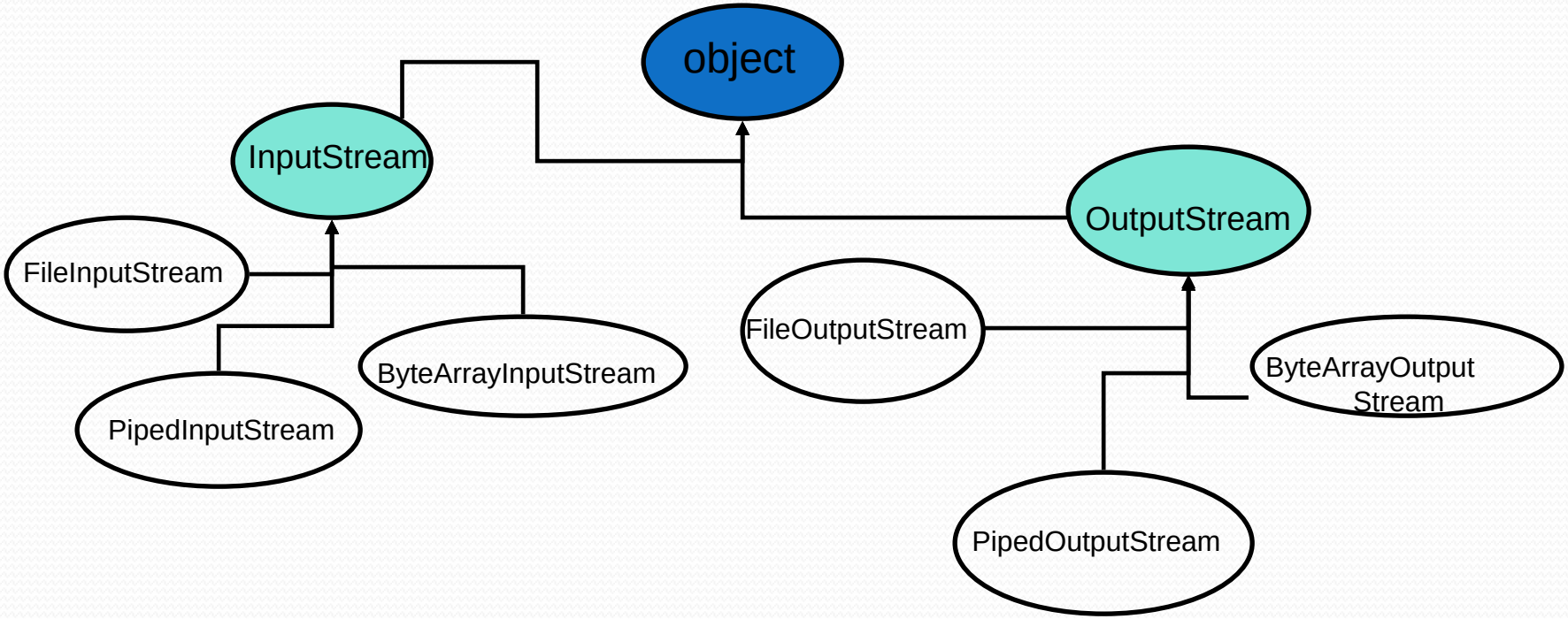
```
import java.io.*;
```

- *Open* the stream
- *Use* the stream (read, write, or both)
- *Close* the stream

Byte Streams

Byte Streams

- `InputStream` **and** `OutputStream`
  - Abstract classes
  - The parent class of all byte streams
  - Defines the methods common to all byte streams



Byte stream classes

Stream class	Description
BufferedInputStream	Used for Buffered Input Stream.
BufferedOutputStream	Used for Buffered Output Stream.
DataInputStream	Contains method for reading java standard datatype
DataOutputStream	An output stream that contain method for writing java standard data type
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that write to a file.
InputStream	Abstract class that describe stream input.
OutputStream	Abstract class that describe stream output.
PrintStream	Output Stream that contain `print()` and `println()` method

Stream class	Description
BufferedReader	Handles buffered input stream.
BufferedWriter	Handles buffered output stream.
FileReader	Input stream that reads from file.
FileWriter	Output stream that writes to file.
InputStreamReader	Input stream that translate byte to character
OutputStreamReader	Output stream that translate character to byte.
PrintWriter	Output Stream that contain `print()` and `println()` method.
Reader	Abstract class that define character stream input
Writer	Abstract class that define character stream output

File copying operations in JAVA

- Open the source file in read mode (say, using FileInputStream constructor)
- Open the destination file in write mode (say, using FileOutputStream constructor)
- Read data from the source file (say, using read() method)
- Write data to the destination file (say, using write(int) method)
- When the job is over, close the streams (say, using close() method

File Input and output stream

- FileInputStream and FileOutputStream classes are used to read and write data in file.
- they are used for file handling in java.

- Syntax :

```
InputStream f = new FileInputStream("C:/java1.txt");
OutputStream f = new FileOutputStream("C:/java1.txt")
```

```
public static void main(String[] args) throws IOException {
    InputStream i= new FileInputStream("E:\\dacJava\\java\\a.txt");
    OutputStream o= new FileOutputStream("E:\\dacJava\\java\\b.txt");

    int b=0;
    while((b=i.read()) != -1){
        o.write((byte)b);
        System.out.println("each one "+b);
        System.out.println("each one "+(char)b);
    }
```

- important methods used are:

read() : reads bytes of data
write() : writes byte of data
close() : closes the file handlers

public int read() throws IOException

read() method reads a byte, converts into its ASCII integer value and returns

if EOF reaches while reading, this method returns -1. In the while loop, every byte, the read() method reads is checked whether EOF is reached

public void write(int) throws IOException

This method takes the ASCII integer value returned by the read() method, converts into the original character and writes into the destination file

File operations

- As a good programming practice, after the usage of file streams, the file handles should be closed. By doing so, certain operations are done implicitly by the JVM, given below.
- Data in the buffers, if exist, is cleared.
- File format is checked (like .txt, .doc and .gif etc).
- When the format is okay, the OS adds EOF marker.
- When the EOF marker is added, the file definitely opens in the correct format, the way it is saved, even after 10 years.

FileWriter and FileReader

- FileWriter class is used to write character-oriented data to the file.
- FileReader class is used to read data from the file.

```
FileWriter fw=new FileWriter("E:\java\abc.txt");
fw.write(“good mng everyone"); //writing string
```

FiletrInput and FilterOutput Stream

- These are the high level streams which adds extra functionality to the existing streams
- DataInputStream and DataOutputStream: they can read or write integers, doubles and lines at a time, instead of byte by byte
- The LineNumberInputStream: adds line numbers that do not exist in the source.
- BufferedInput and BufferedOutput: using buffering to enhance the performance

Buffered input and output stream

- Buffered Stream used as an internal buffer.
- It adds more **efficiency** than to write data directly into a stream. So, it makes the performance fast.
- It gives system defined buffer for the data transfer
- textual information can be written into BufferedOutputStream object which is connected to the FileOutputStream object.

- FileOutputStream fout=new FileOutputStream("f1.txt");
- BufferedOutputStream bout=new BufferedOutputStream(fout);

Buffered reader and writer

- Reads / Writes text using a character-input/output stream, buffering characters so as to provide for the efficient reading/writing of characters, arrays, and lines.

```
FileReader f= new FileReader("D:\\DAC Material\\Aug 2014 batch\\a.txt");
BufferedReader bf= new BufferedReader(f);
System.out.println("each one aftre FR "+bf.readLine());
```

Sequence input stream

- It is used to read data from multiple streams.
- **SequenceInputStream(InputStream s1, InputStream s2):**
- **It** creates a new input stream by reading the data of two input stream in order, first s1 and then s2.

```
//sequence input
FileInputStream fin1=new FileInputStream("D:\\DAC Material\\Aug 2014 batch\\a.txt");
FileInputStream fin2=new FileInputStream("D:\\DAC Material\\Aug 2014 batch\\b.txt");

   SequenceInputStream sis=new SequenceInputStream(fin1,fin2);
   int i1;
   while((i1=sis.read())!=-1)
   {
    System.out.println((char)i1);
   }
```

Data from keyboard

- There are many ways to read data from the keyboard. For example:

- InputStreamReader
- Scanner
- DataInputStream

Input Stream Reader

- InputStreamReader class can be used to read data from keyboard. It performs two tasks:
- connects to input stream of keyboard
- converts the byte-oriented stream into character-oriented stream
- It is neither an input **stream** nor a **reader.**

- We can connect the **BufferedReader** stream with the InputStreamReader stream for reading the line by line data from the keyboard.
- BufferedReader class can be used to read data line by line by **readLine**() method.

```java
public class KBReadingEx {


    public static void main(String[] args) throws IOException {

        InputStreamReader is= new InputStreamReader(System.in);
        BufferedReader b= new BufferedReader(is);
        System.out.println("eneter the letter/letters");
        char s=(char) b.read();     //1 character
        System.out.println("read one is "+s);

        InputStreamReader is1= new InputStreamReader(System.in);
        BufferedReader b1= new BufferedReader(is1);
        System.out.println("eneter the letter/letters");
        String st= b1.readLine();  //String
        System.out.println("read one is "+st);
    }

}
```

Data input stream

- It is Also used to read data from keyboard
- Reads a line and returns string

```
DataInputStream d= new DataInputStream(System.in);
String sd=d.readLine();
System.out.println("line is "+sd);
```

Scanner class

- The Scanner class breaks the input into tokens using a delimiter which is whitespace by default.
- **Methods**:
- public String next(): it returns the next token from the scanner.
- public String nextLine(): it moves the scanner position to the next line and returns the value as a string.
- public byte nextByte(): it scans the next token as a byte.
- public short nextShort(): it scans the next token as a short value.
- public int nextInt(): it scans the next token as an int value.
- public long nextLong(): it scans the next token as a long value.
- public float nextFloat(): it scans the next token as a float value.
- public double nextDouble(): it scans the next token as a double value.

Example

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

    //scanner
    Scanner s= new Scanner(System.in);

    System.out.println("enter the string");
    String st=s.next();
    System.out.println("string "+st);

    System.out.println("enter the integer");
    int i=s.nextInt();
    System.out.println("string "+i);

    Scanner s1= new Scanner(System.in);
    System.out.println("enter the String again");
    String str=s1.nextLine();
    System.out.println("string "+str);
```

Print stream and

- **PrintStream:**
- The PrintStream class provides methods to write data to another stream.
- the methods of PrintStream class do not throw IOException.

PrintWriter

- The PrintWriter class enables you to write formatted data to an underlying Writer. For instance, writing int, long and other primitive data formatted as text, rather than as their byte values

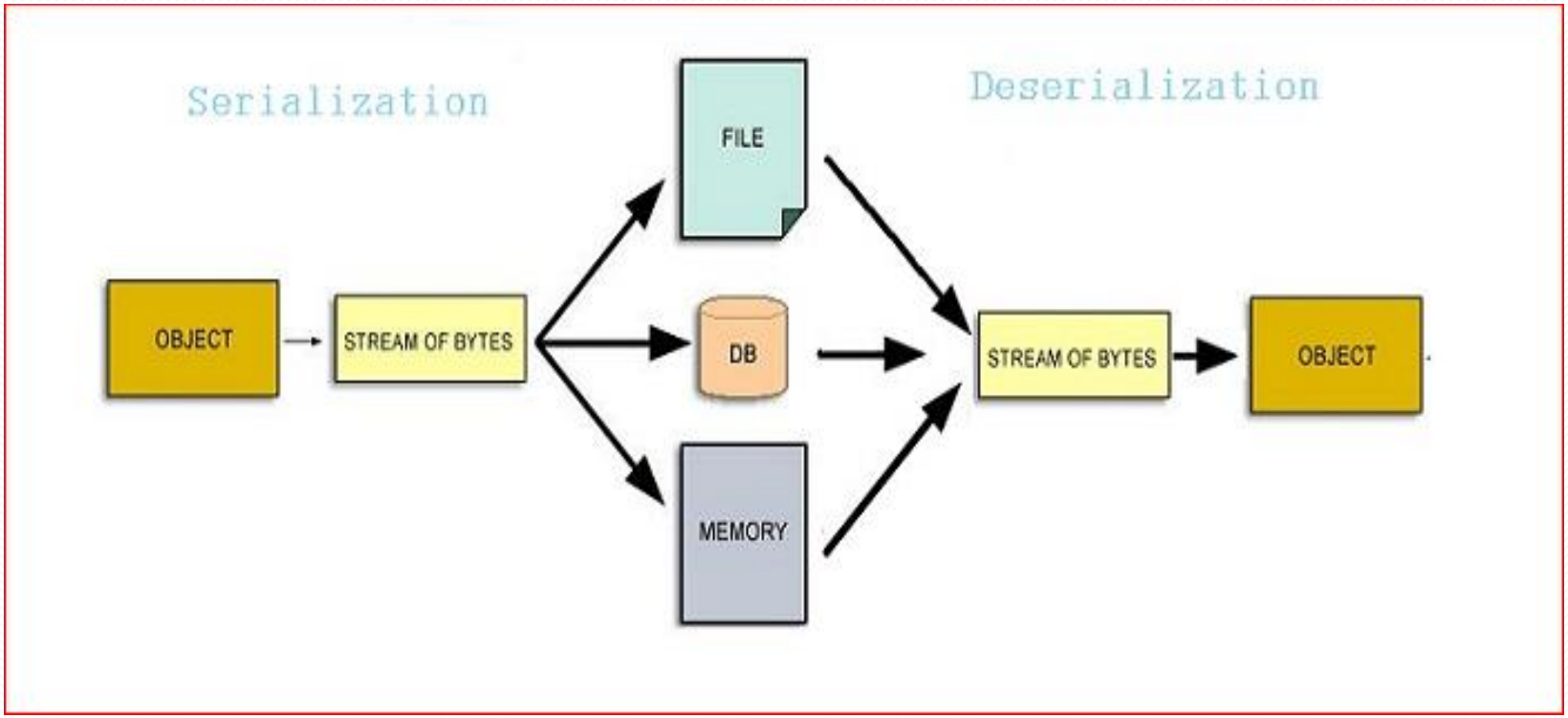
Examples

```
public static void main(String[] args) throws FileNotFoundException {
    //writing into file
    PrintStream p= new PrintStream("D:\\DAC Material\\Aug 2014 batch\\a.txt");
    p.println("hello");

    //writing into console
    PrintStream p1= new PrintStream(System.out);
    p1.println("hi");

    System.out.print("hello with static class - System");

}
```

Serialization

- Serialization is a mechanism of writing the state of an object into a byte stream.
- Serialization is the process converting an object into a special stream of bytes so that they can be written to a file or send across distributed network.
- Serialization gives guarantee that when the object is retrieved, the object maintains the same properties
- The reverse operation of the serialization is called deserialization.

- It is mainly used to travel object's state on the network.
- It is mainly used in Hibernate, JPA, EJB etc.


Serialization
Deserialization
FILE
OBJECT
STREAM OF BYTES
DB
STREAM OF BYTES
OBJECT
MEMORY

API's helping serialization

- All classes to be persisted should implement the marker interface provided by the java.io.Serializable

- An **ObjectOutputStream** is API used to write primitive data types and Java objects to an OutputStream.
- Only objects that support the java.io.Serializable interface can be written to streams.

- An **ObjectInputStream** deserializes objects and primitive data written using an ObjectOutputStream.

Object output stream

- Commonly used **Constructors**:

1) public ObjectOutputStream(OutputStream out) throws IOException {
}

- creates an ObjectOutputStream that writes to the specified OutputStream.

Commonly used **Methods**:

public final void writeObject(Object obj) throws IOException {
}

-write the specified object to the ObjectOutputStream.

public void flush() throws IOException {
}

-flushes the current output stream.

Object input stream

- Commonly used Constructors:

public ObjectInputStream(InputStream in) throws IOException {
}

-creates an ObjectInputStream that reads from the specified InputStream.

- Commonly used Methods:

public final Object readObject() throws IOException, ClassNotFoundException{
}

-reads an object from the input stream.

- public class Student implements Serializable
- When the Student class implements Serializble interface, automatically all the Student objects created are serialized
- If a class implements serializable then all its subclasses will also be serializable.
- If there is any static data member in a class, it will not be serialized because static is related to class not to instance.

Example

```
Animal a= new Animal(1,1.2f,"dog");
FileOutputStream fout= new FileOutputStream("D:\\sharadhi\\abc.txt");
ObjectOutputStream o= new ObjectOutputStream(fout);

o.writeObject(a);
o.flush();
fout.close();
//deserialization

FileInputStream fis= new FileInputStream("D:\\sharadhi\\abc.txt");
ObjectInputStream oo= new ObjectInputStream(fis);

Animal a1=(Animal)oo.readObject();
System.out.println(a1.name+a1.teeth);
fis.close();
```

Say no to Serialization

- **transient** keyword:
- While serializing the object, if we don't want certain member of object to be serialized, then we should use transient keyword with the data member.

```
public class Student implements Serializable{
  String name;
 transient int age;//Now it will not be serialized
 public Student(int id, String name,int age) {
  this.id = id;
  this.name = name;
  this.age=age;
 }
```

Some methods

- Skip() method – places the pointer at the specified byte in file
- available()- gives the file size

```
FileInputStream i= new FileInputStream("C:\\abc.txt");
System.out.println(i.available()); //say 3
i.skip(2);
System.out.println(i.available()); //1
```

RandomAccessFile

- It's a subclass of Object, placed in java.io
- It does job of both reading and writing
- It can read the file both sequentially and randomly
- Methods of DataInput and DataOutput are avialable with this.
- 2 important modes supported are: 'r' and 'w'
- There is a file pointer which helps in moving anywhere in the array.

```
RandomAccessFile raf= new RandomAccessFile("C:\\abc.txt", "rw");
for(int i=0;i<10;i++)
{
    raf.writeInt(i);
}
raf.seek(0); //point to first
for(int i=0;i<10;i++)
{
    System.out.println(raf.readInt());
}

System.out.println("pointer "+raf.getFilePointer());

raf.seek(5*4); //20th
raf.writeInt(333);//changing the 6th value
System.out.println("pointer "+raf.getFilePointer());
raf.seek(0);
for(int i=0;i<10;i++)
{
    System.out.println(raf.readInt());
}
```

File

- one more non stream class- File (not used for file operations)
- The API says that the class File is "An abstract representation of file and directory pathnames.“
- The File class isn't used to actually read or write data; it's used to work at a higher level, making new empty files, searching for files, deleting files, making directories

Methods

- String getName() – name of the file or directory
- String getAbsolutePath()- absolute path
- boolean canRead()
- boolean canWrite()
- boolean isFile() , isDirectory()
- long lastModified()- time that file is last modified
- Length() – returns file size, in bytes
- boolean delete()- deletes the file
- boolean mkdir()
- Boolean renameTo(filenameh)- renames the file