ACCESS CONTROLS

By
M. BABY ANUSHA,
ASST.PROF IN CSE DEPT.,
RGUKT,NUZVID

INTRODUCTION

- Access control is a mechanism, an attribute of encapsulation which restricts the access of certain members of a class to specific parts of a program. Access to members of a class can be controlled using the *access modifiers.*
- There are four access modifiers in Java. They are:
  - public
  - protected
  - default
  - private

CONTINUED…

- If the member (variable or method) is not marked as either *public* or *protected* or *private*, the access modifier for that member will be *default*. We can apply access modifiers to classes also.
- Among the four access modifiers, *private* is the most restrictive access modifier and *public* is the least restrictive access modifier.
- Syntax for declaring a access modifier is shown below:

access-modifier data-type variable-name;

CONTINUED…

- Example for declaring a *private* integer variable is shown below:

  private int side;

- In a similar way we can apply access modifiers to methods or classes although *private* classes are less common.

CONTINUED...

- Accessibility restrictions of the four access modifiers is as shown below:

Access Modifiers In Java

Access Modifier	Within the Class	Other Classes [Within the Package]	In Subclasses [Within the package and other packages]	Any Class [In Other Packages]
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	Same Package – Y Other Packages - N	N
private	Y	N	N	N

Y – Accessible
N – Not Accessible

WHAT IS THE USE OF ACCESS MODIFIERS?

- As I had said above, access modifiers are used to restrict the access of members of a class, in particular data members (fields).
- Let me explain this through Java code. Let's consider an employee class as shown below:

```
class Employee
{
int  empid;
String  empname;
float  salary;
//Methods which operate on above data
   members
}
```

CONTINUED…

- By looking at the above code we can say that the access modifier for all the three data members is *default*.
- As members with *default* (also known as **Package Private** or **no modifier**) access modifier are accessible throughout the package (in other classes), a programmer might, by mistake, try to make an employee's salary negative as shown below:

```
Employee e1 = new Employee();
e1.salary = -1000.00;
```

CONTINUED…

- Although above code is syntactically correct, it is logically incorrect.
- To prevent such things to happen, in general, all the data members are declared *private* and are accessible only through *public* methods.
- So, we can modify our *Employee* class as shown below:

SAMPLE PROGRAM :

```
Class Employee
{
    private int empid;
    private String emp;
    private float salary;
    public void setSalary(float sal)
    {
        if(sal < 0)
        {
            System.out.println("Salary cannot be negative");
        }
        else
        {
            salary = sal;
        }
      }
//Other methods
}
```

CONTINUED…

- By looking at the above code, we can say that one can access the *salary* field only through *setSalary()* method.
- Now, we can set the salary of an employee as shown below:

```
Employee e1 = new Employee();
e1.setSalary(-1000.00);  //Gives error as salary cannot be
   negative
e1.setSalary(2000.00);  //salary of e1 will be assigned
   2000.00ary of              an employee as shown below:
```

DEFAULT ACCESS MODIFIER

- When we do not mention any access modifier, it is called default access modifier.
- The scope of this modifier is limited to the package only. This means that if we have a class with the default access modifier in a package, only those classes that are in this package can access this class.
- No other class outside this package can access this class. Similarly, if we have a default method or data member in a class, it would not be visible in the class of another package.
- Lets see an example to understand this:

DEFAULT ACCESS MODIFIER EXAMPLE IN JAVA

- To understand this example, you must have the knowledge of packages in java.
- In this example we have two classes, Test class is trying to access the default method of Addition class, since class Test belongs to a different package, this program would throw compilation error, because the scope of default modifier is limited to the same package in which it is declared.

DEFAULT ACCESS MODIFIER EXAMPLE IN JAVA

Addition.java

```
package abcpackage;
 public class Addition
{
/* Since we didn't mention any access modifier here,
   it would be considered as default. */
   int addTwoNumbers(int a, int b)
   {
      return a+b;
   }
}
```

DEFAULT ACCESS MODIFIER EXAMPLE IN JAVA

```
package xyzpackage;

 /* We are importing the abcpackage but still we will get
    error because the  class we are trying to use has
    default access  modifier. */

import abcpackage.*;
 public class Test
{
    public static void main(String args[])
    {
    Addition obj = new Addition(); /* It will throw error
    because we are trying to access the default method in
    another package */
   obj.addTwoNumbers(10, 21);
    }
}
```

DEFAULT ACCESS MODIFIER EXAMPLE IN JAVA

Output:

```
Exception in thread "main" java.lang.Error:
Unresolved compilation problem: The method
addTwoNumbers(int, int) from the type
Addition is not visible at xyzpackage.Test.
main(Test.java:12)
```

PRIVATE ACCESS MODIFIER

- The scope of private modifier is limited to the class only.
- Private Data members and methods are only accessible within the class
- Class and Interface cannot be declared as private
- If a class has private constructor then you cannot create the object of that class from outside of the class.
- Let’s see an example to understand this:

PRIVATE ACCESS MODIFIER EXAMPLE IN JAVA

- This example throws compilation error because we are trying to access the private data member and method of class ABC in the class Example.
- The private data member and method are only accessible within the class.

PRIVATE ACCESS MODIFIER EXAMPLE IN JAVA

```
class ABC
{
   private double num = 100;
   private int square(int a)
   {
      return a*a;
    }
}
 public class Example{
   public static void main(String args[])
   {
   ABC obj = new ABC();
   System.out.println(obj.num); System.out.println(obj.
  square(10));
}}
```

Output:

Compile - time error

PROTECTED ACCESS MODIFIER

- Protected data member and method are only accessible by the classes of the same package and the subclasses present in any package.
- You can also say that the protected access modifier is similar to default access modifier with one exception that it has visibility in sub classes.
- Classes cannot be declared protected. This access modifier is generally used in a parent child relationship.

PROTECTED ACCESS MODIFIER EXAMPLE IN JAVA

- In this example the class Test which is present in another package is able to call the addTwoNumbers() method, which is declared protected.
- This is because the Test class extends class Addition and the protected modifier allows the access of protected members in subclasses (in any packages).

PROTECTED ACCESS MODIFIER EXAMPLE IN JAVA

Addition.java

```
package abcpackage;

public class Addition
 {
   protected int addTwoNumbers(int a, int b)
   {
      return a+b;
   }
}
```

PROTECTED ACCESS MODIFIER EXAMPLE IN JAVA

Test.java

```
package xyzpackage;
 import abcpackage.*;
class Test extends Addition
{
   public static void main(String args[])
   {
   Test obj = new Test(); System.out.println(obj.
   addTwoNumbers(11,22)); }
}
```

Output:

33

PUBLIC ACCESS MODIFIER

- The members, methods and classes that are declared public can be accessed from anywhere. This modifier doesn't put any restriction on the access.

public access modifier example in java

- Lets take the same example that we have seen above but this time the method addTwoNumbers() has public modifier and class Test is able to access this method without even extending the Addition class.
- This is because public modifier has visibility everywhere.

PUBLIC ACCESS MODIFIER EXAMPLE IN JAVA

Addition.java

```
package abcpackage;
 public class Addition
 {
   public int addTwoNumbers(int a, int b)
   {
      return a+b;
   }
 }
```

PUBLIC ACCESS MODIFIER EXAMPLE IN JAVA

Test.java

```
package xyzpackage;
 import abcpackage.*;
class Test
{
 public static void main(String args[])
{
 Addition obj = new Addition(); System.out.
   println(obj.addTwoNumbers(100,1));
}
}
```

Output:

101



shutterstock.com · 567687052

Object Oriented Programming in JAVA

prepared by

Dr.G.Shobana

Course Objectives

On completion of this course we will be able to:

- ✔ Identify the importance of Java .
- ✔ Identify the additional features of Java compared to C++ .
- ✔ Identify the difference between Compiler and Interpreter .
- ✔ Identify the difference between applet and application .
- ✔ Apply Object Oriented Principles of Encapsulations , Data abstraction, Inheritance, Polymorphism.
- ✔ Program using java API (Application Programming Interface).
- ✔ Program using Exception Handling, Files and Threads .
- ✔ Program Using applets and swings .

Why Java is Important?

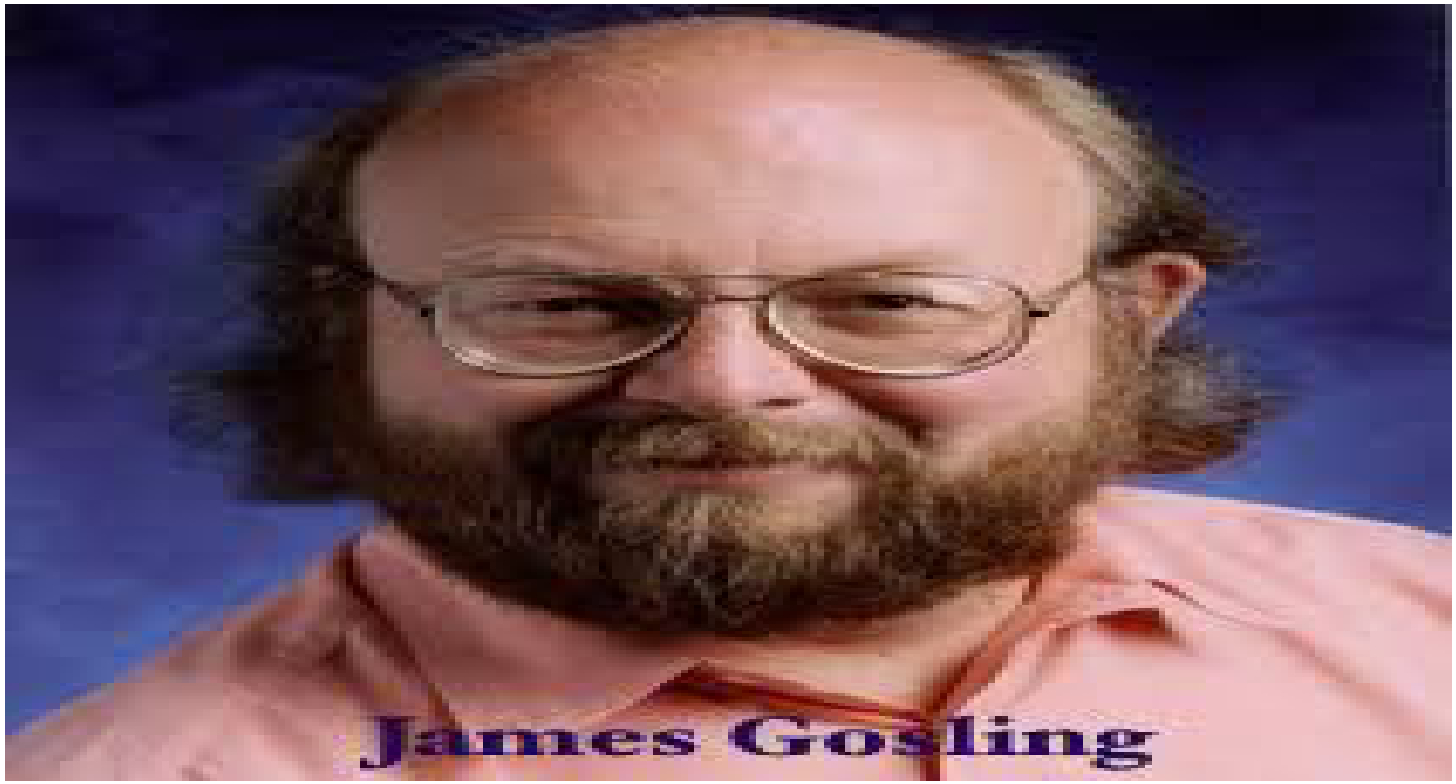
Two reasons :

- Trouble with C/C++ language is that they are not portable and are not platform independent languages.
- Emergence of World Wide Web, which demanded portable programs

Portability and security necessitated the invention of Java

History of Java

Java was developed by James Gosling, who is known as the father of Java, in 1995. James Gosling and his team members started the project in the early '90s. Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc.



There are given significant points that describe the history of Java.

James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991.

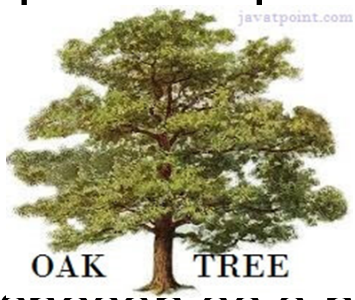
1)The small team of sun engineers called Green Team.

2) Initially designed for small, embedded systems in electronic appliances like set-top boxes.

3) Firstly, it was called "Greentalk" by James Gosling, and the file extension was .gt.

4) After that, it was called Oak and was developed as a part of the Green project.

Why Java named "Oak"?



5) Why Oak? Oak is a symbol of strength and chosen as a national tree of many countries like the U.S.A., France, Germany, Romania, etc.

6) In 1995, Oak was renamed as "Java" because it was already a trademark by Oak Technologies.

Why Java Programming named "Java"?

7) Why had they chosen java name for Java language? The team gathered to choose a new name. The suggested words were " dynamic", "revolutionary", "Silk", "jolt", "DNA", etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell and fun to say.

According to James Gosling, "Java was one of the top choices along with Silk". Since Java was so unique, most of the team members preferred Java than other names.

8) Java is an island of Indonesia where the first coffee was produced (called java coffee). It is a kind of espresso bean. Java name was chosen by James Gosling while having coffee near his office.

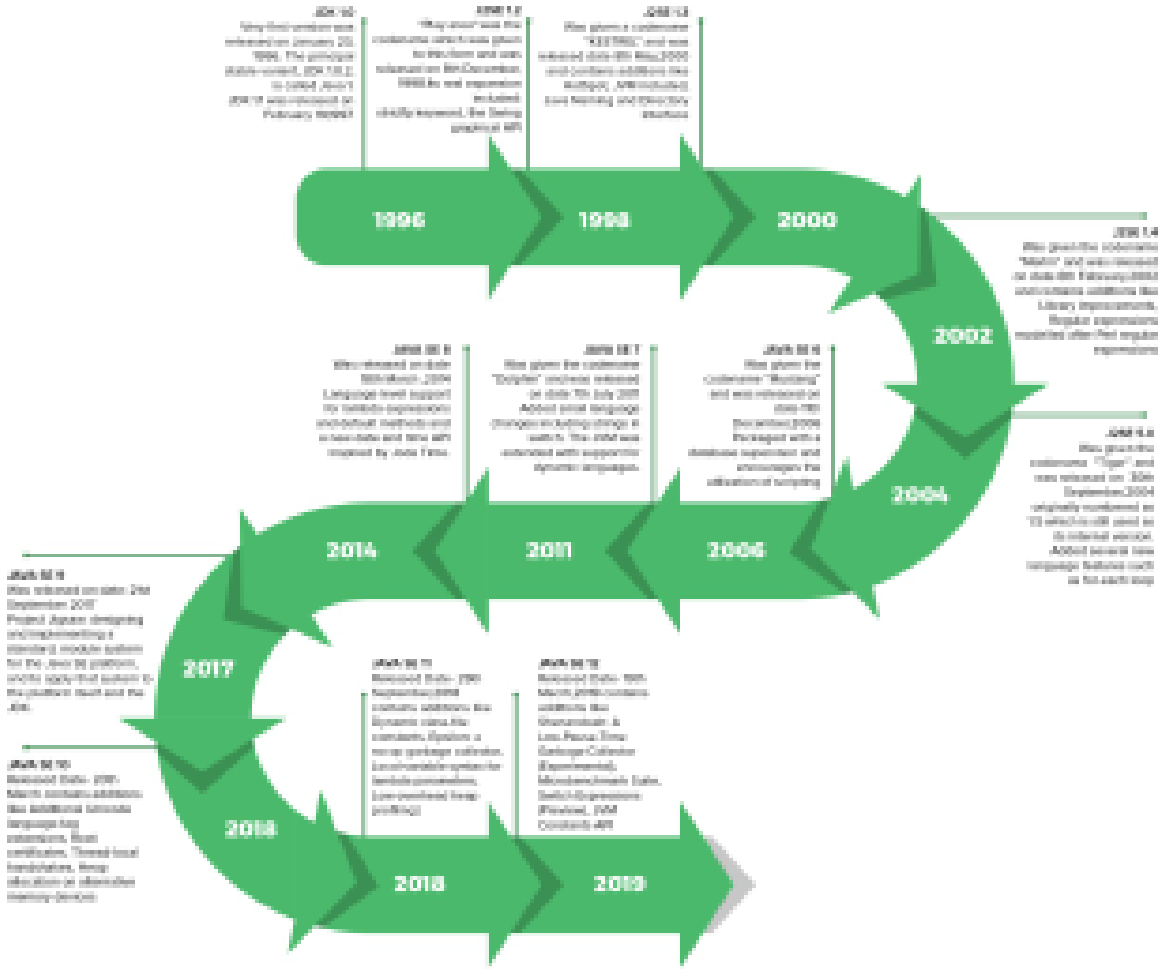
9) Notice that Java is just a name, not an acronym.

10) Initially developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.

11) In 1995, Time magazine called Java one of the Ten Best Products of 1995.

12) JDK 1.0 released in(January 23, 1996). After the first release

History



Java Editions

- **J2SE(Java 2 Standard Edition)** - to develop client side standalone applications or applets.
- **J2ME(Java 2 Micro Edition)** - to develop applications for mobile devices such as cellphones.
- **J2EE(Java 2 Enterprise Edition)** - to develop server-side applications such as Java servlets and Java Server Pages.

What is Java?

- A general-purpose object-oriented language.
- Write Once Run Anywhere (WORA).
- Designed for easy Web/Internet applications.
- Widespread acceptance.

Differences

	C++	java	c
1.Program paradigm	oops	purely oops	procedural
2.Origin	c	c++	Assembly langugae
3.Devoloper	Brian Stroustrup (1979)	James Gosling(1991)	Dennis Ritchie(1972)
4.Translator --	Compiler	Compiler& Interpreter	compiler
5.Platform Independent	dependent	JVM- independent	dependent
6.Code &Execution	direct .exe	.class	direct .exe
7.Aprroach	Bottom-up	bottom-up	top-dwn
8.Preprocessor Directives	#tag	import #tag	
9.Keyword	63	50	32
10.Datatype	sruct and union	no	struct and union
11.Inheritance	yes	yes	no
12.Overloading	yes	yes	no
13.Pointers	yes	no	yes
14.Memory allocation	new,delete	garbage collector	malloc,realloc,free
15.Exception Handling	yes	yes	no
16.Templates	yes	no	no
17.Destructor	yes	no	no
18.Multi-threaded	no	yes	no
19.DB connectivity	no	yes	no
20.Storage classes	yes	no	yes

How is Java different from C…

C Language:

Major difference is that C is a structure oriented language and Java is an object oriented language and has mechanism to define classes and objects.

- ✔ Java does not support an explicit pointer type
- ✔ Java does not have preprocessor, so we cant use #define, #include and #ifdef statements.
- ✔ Java does not include structures, unions and enum data types.
- ✔ Java does not include keywords like goto, sizeof and typedef.
- ✔ Java adds labeled break and continue statements.
- ✔ Java adds many features required for object oriented programming.

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

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.
- break and continue statements have been enhanced in java to accept labels as targets.
- The use of unicode characters ensures portability.

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.
- Java has replaced the destructor function with a finalize() function.
- C++ supports exception handling that is similar to java's.

Characteristics of Java

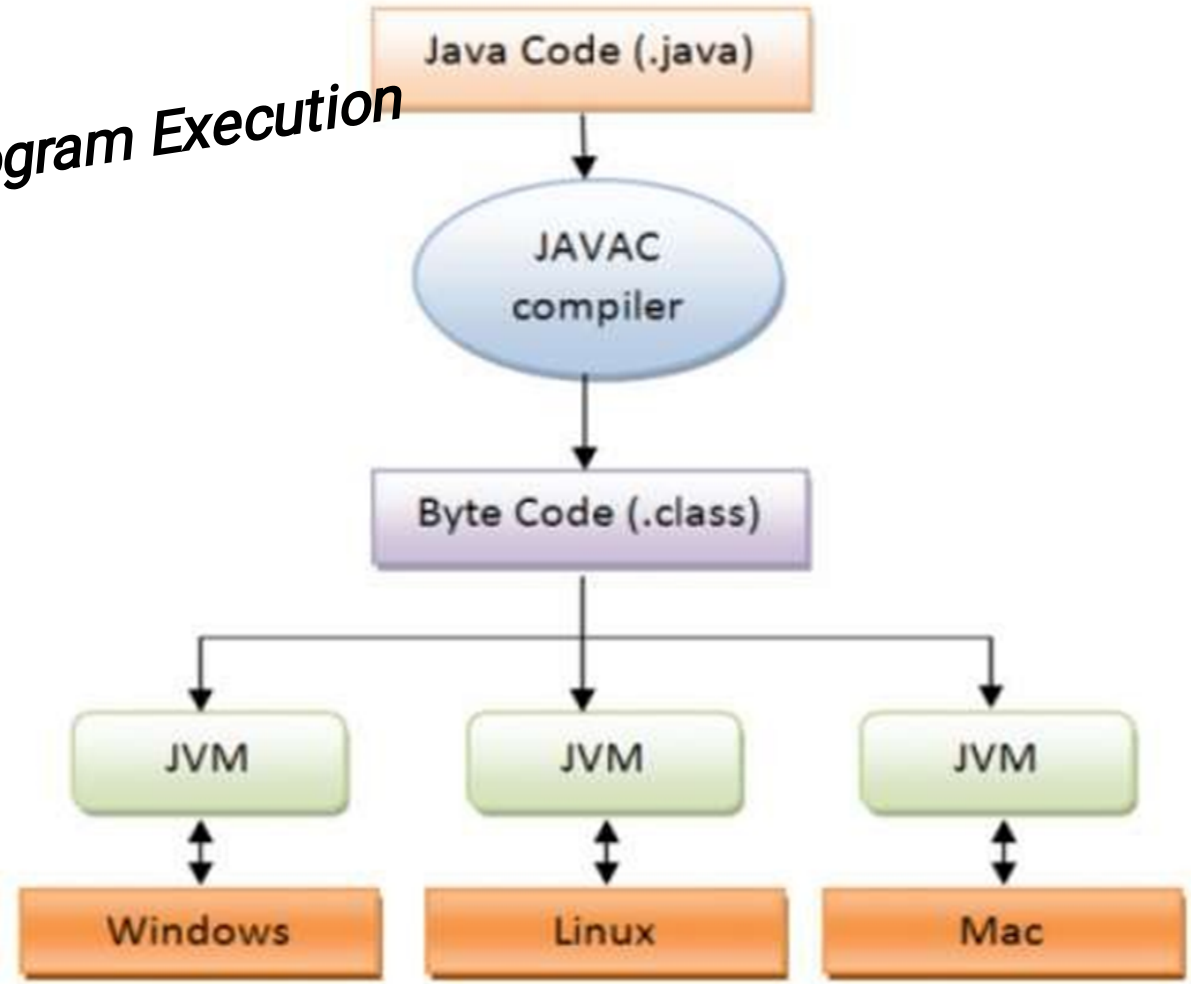
- ✔ Java is simple
- ✔ Java is object-oriented
- ✔ Java is distributed
- ✔ Java is interpreted
- ✔ Java is robust
- ✔ Java is architecture-neutral
- ✔ Java is portable
- ✔ Java's performance
- ✔ Java is multithreaded
- ✔ Java is dynamic
- ✔ Java is secure

Java Environment

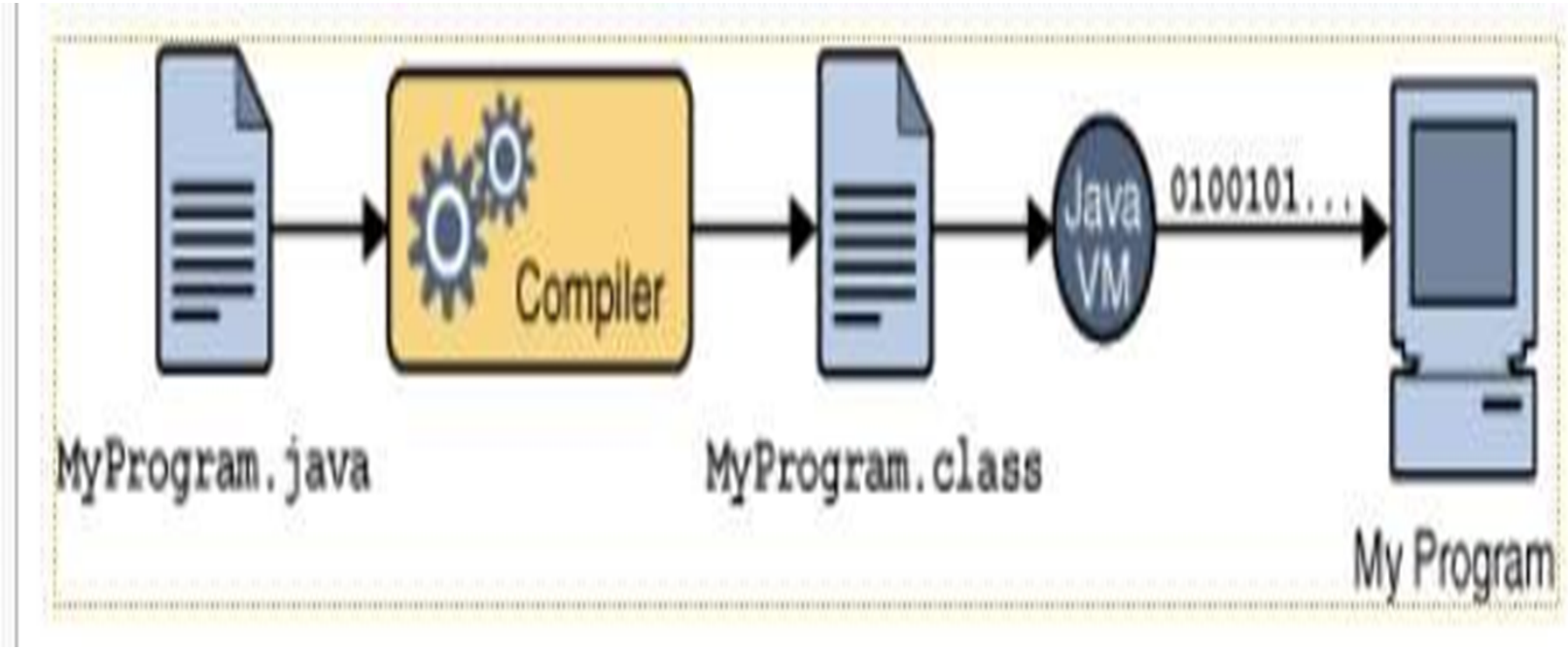
- Java includes many development tools, classes and methods
- Development tools are part of Java Development Kit (JDK) and The classes and methods are part of Java Standard Library (JSL),also known as Application Programming Interface (API).
- JDK constitutes of tools like java compiler, java interpreter and many.
- API includes hundreds of classes and methods grouped into several packages according to their functionality.

JAVA is Architectural Neutral

JAVA program Execution



WORA(Write Once Run Anywhere)



Introduction

Object oriented programming is the principle of design and development of programs using modular approach.

- Object oriented programming approach provides advantages in creation and development of software for real life application.
- The basic element of object oriented programming is the data.
- The programs are built by combining data and functions that operate on the data.
- Some of the OOP's languages are C++, Java, C #, Smalltalk, Perl, and Python.

Procedural programming

- The procedural programming focuses on **processing** of instructions in order to perform a desired computation.
- The top-down concepts to decompose main
- functions into lower level components for modular coding purpose.
- Therefore it emphasizes more on doing things like algorithms.
- This technique is used in a conventional programming language such as C and Pascal.

Object oriented programming

- Object oriented programming (OOP) is a concept that combines both the data and the functions that operate on that data into a single unit called the object.

  object=data+functions

- An object is a collection of set of data known as member data and the functions that operate on these data known as member function.

- OOP follows bottom-up design technique.

- Class is the major concept that plays important role in this approach. Class is a template that

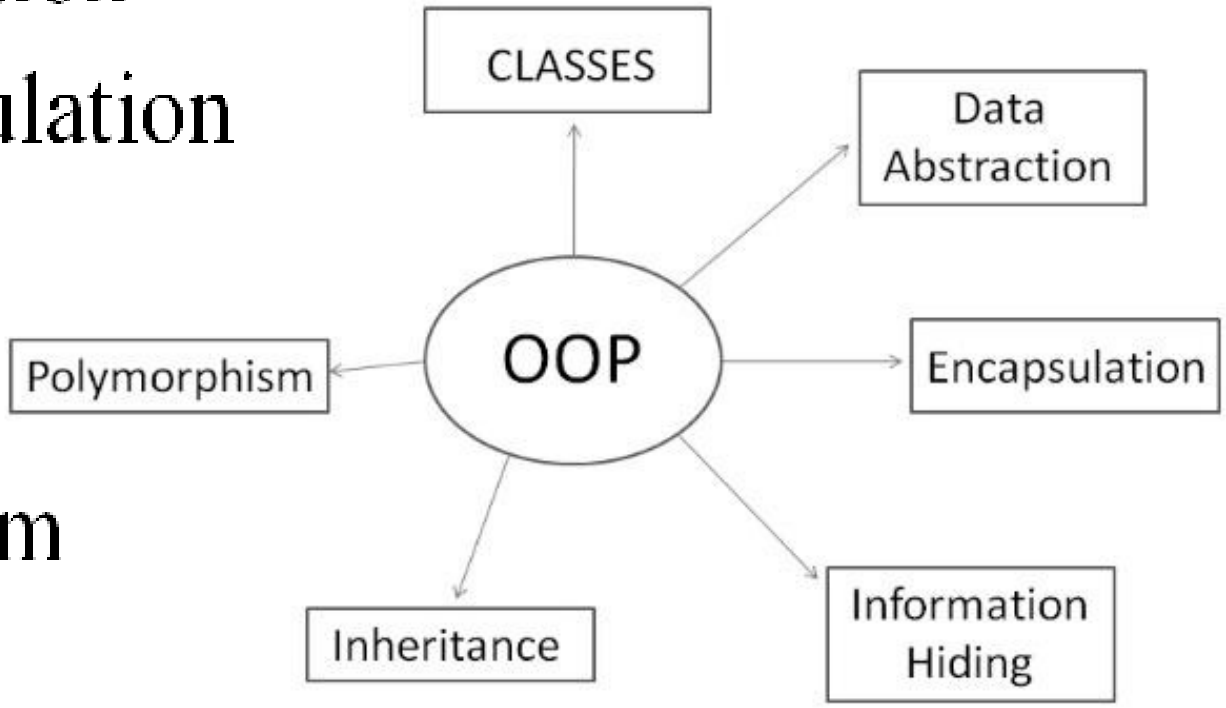
Differences

Procedural Programming	Object Oriented Programming
Large programs are divided into smaller programs known as functions	Programs are divided into objects
Data is not hidden and can be accessed by external functions	Data is hidden and cannot be accessed by external functions
Follow top down approach in the program design	Follows bottom-up approach in the program design
Data may communicate with each other through functions	Objects may communicate with each other through functions.
Emphasize is on procedure rather than data	Emphasize is on data rather than procedure

Basic Concepts of OOP's

The following are the major characteristics of OOP's:

- Objects
- Class
- Data abstraction
- Data encapsulation
- Inheritance
- Overloading
- Polymorphism
- Dynamic Binding
- Message Passing

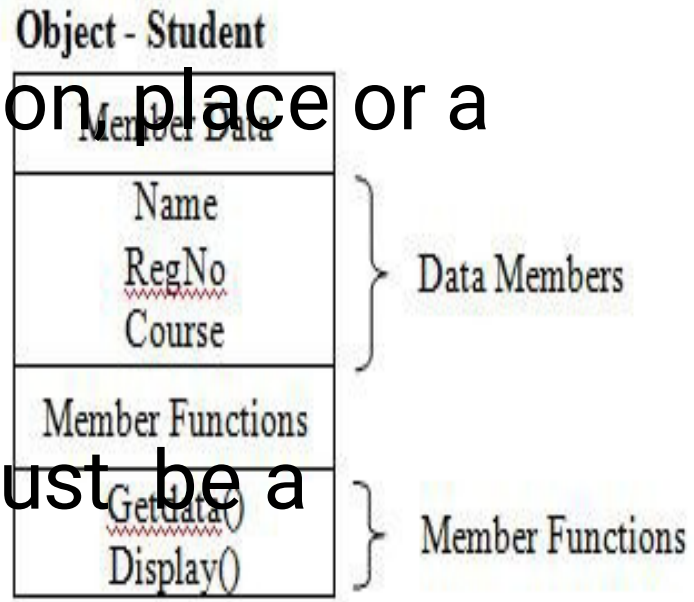


Objects

- Objects are basic building blocks for designing programs.

Object is a instance of a class

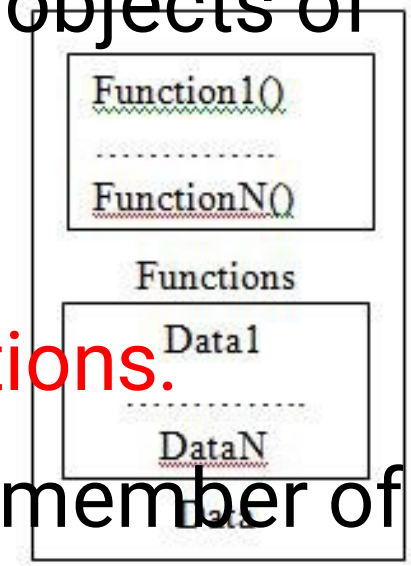
- ***An object is a collection of data members and associated member functions.***
- An object may represent a person, place or a table of data.
- Each object is identified by a unique name. Each object must be a member of a particular class.
- Example: Apple, orange, mango are the



Classes

- The objects can be made user defined data types with the help of a class.
- ***A class is a collection of objects that have identical properties, common behavior and shared relationship.***
- Once class is defined, any number of objects of that class is created.
- Classes are user defined data types.

  A class can hold both data and functions.
- For example: Planets, sun, moon are member of class solar system.



Class:

If u create then u can create any number of object

ex Student s1=new Student();

Student s2=new Student();

.

.

Object:

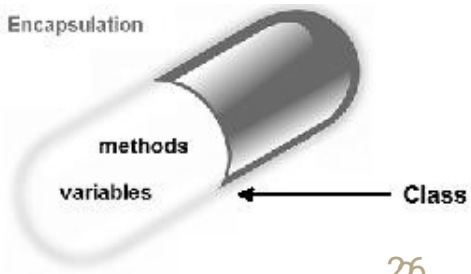
Object holds a memory,used to access all the data from the class

Object uniquely identified→

It will accessed only when class(data+functions)

DataAbstraction, Encapsulation & Data Hiding

- **Data Abstraction**
  - Data Abstraction refers to the process of representing essential features without including background details or explanations.
- **Data Encapsulation**
  - The wrapping of data and functions into a single unit (class) is called data encapsulation.
  - Data encapsulation enables data hiding and information hiding.
- **Data Hiding**
  - Data hiding is a method used in object oriented programming to hide information within computer code.



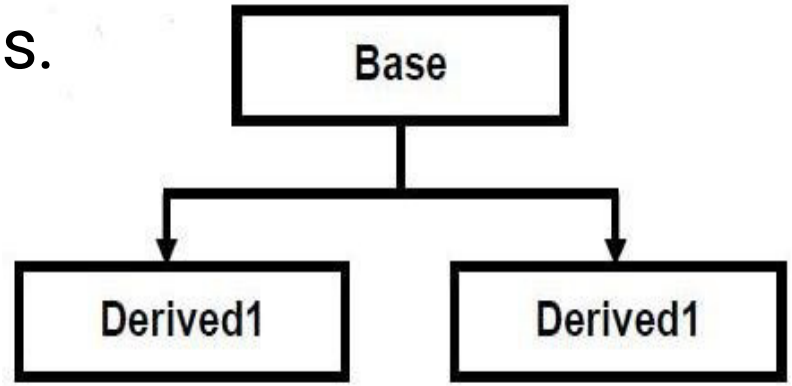
Inheritance

Inheritance is the process by which one object can acquire and use the properties of another object.

The existing class is known as ***base class or super*** class.

The new class is known as ***derived class or sub class.***

The derived class shares some of the properties of the base class. Therefore a code from a base class can be reused by a derived class.

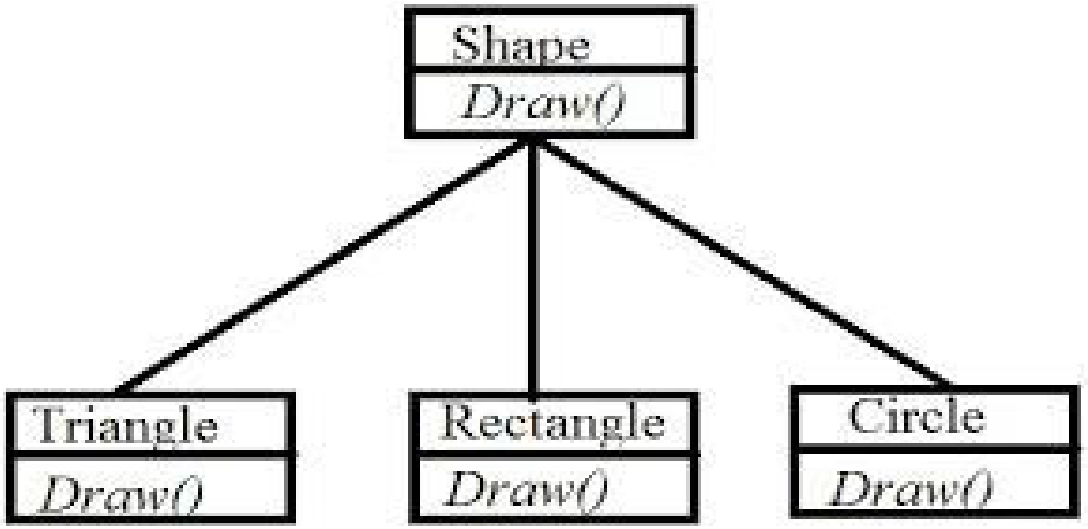


Overloading

- ***Overloading allows objects to have different meaning depending upon context.***
- There are two types of overloading viz.
  - Function Overloading
- When an existing operator operates on new data type is called ***operator overloading.***
- ***Function overloading*** *means two or more function have same name, but differ in the number of* arguments or data type of arguments.

Polymorphism

- ***The ability of an operator and function to take***
- ***multiple forms is known as Polymorphism.***
- The different types of polymorphism are operator
- overloa



Dynamic binding & Message Passing

Dynamic binding:

- Binding is the process of connecting one program to another.
- Dynamic binding is the process of linking the procedure call to a specific sequence of code or function at run time or during the execution of the program.

Message Passing:

- In OOP's, processing is done by sending message to objects.
- A message for an object is request for execution of procedure.
- Message passing involves specifying the name of the object, the name of the function (message) and the

Advantage of OOP's

- The programs are modularized based on the principles of **classes** and **objects.**
- Linking code & object allows related objects to share common code. This reduces **code duplication** and **code reusability**.
- Creation and implementation of OOP code is easy and reduces software development time.
- The concept of data abstraction separates object specification and object implementation.
- **Data encapsulated** along with functions. Therefore external non- member function cannot access or modify data, thus proving data security.
- Easier to develop complex software, because complexity can be minimized through inheritance.
- OOP can communicate through message passing which interface description with outside system, e.g.

Disadvantageof OOP's

- Larger program size: OOP's typically involves more lines of code than procedural programs.
- Slower Programs: OOP's typically slower than procedure based programs, as they typically require more instructions to be executed.
- Not suitable for all types of programs.
- To convert a real world problem into an object oriented model is difficult.
- OOP's software development, debugging and testing tools are not standardized.
- Polymorphism and dynamic binding also requires processing time, due to overload of function calls during run time.

Application of OOP's

- Computer graphics applications.
- CAD/CAM software
- Object-oriented database.
- User-Interface design such as windows
- Real-time systems.
- Simulation and Modeling
- Artificial intelligence and expert systems.

Thank you


CLASSES
Data Abstraction
OOP
Polymorphism
Encapsulation
All the best
Inheritance
Information Hiding

Java Programming

Shobana.G
Assistant Professor,
Rajiv Gandhi University of Knowledge& Technology

Objectives

- To create classes, objects and make use of arrays and strings.
- They will also learn the concepts of inheritance and garbage collection.

Class

- A class is a user-defined data type with the template that serves to define its properties.
- A class defines new data type. Once defined, this new data type can be used to create objects of that type.
- Class is a template for an object and object is the instance of the class.

Class definition

```
class class-name
{
        data-type instance-variable1;  data-type instance-
        variable2;
        - - - - -;
        data-type instance-variableN;
        data-type method-name1(parameter-list)
        {
                //body of the method1
        }
                data-type       method-name2(parameter-list)
        {
                //body of the method2
        }
                - - - - -;
                data-type method-nameN(parameter-list)
        {
                //body of the methodN
        }
}
```

Adding variables to class

```
class Country
{
        long population;  float
        currencyRate;  int
        noOfStates;
}
```

Creating objects

Creating the objects of the class is also called as instantiating an object.

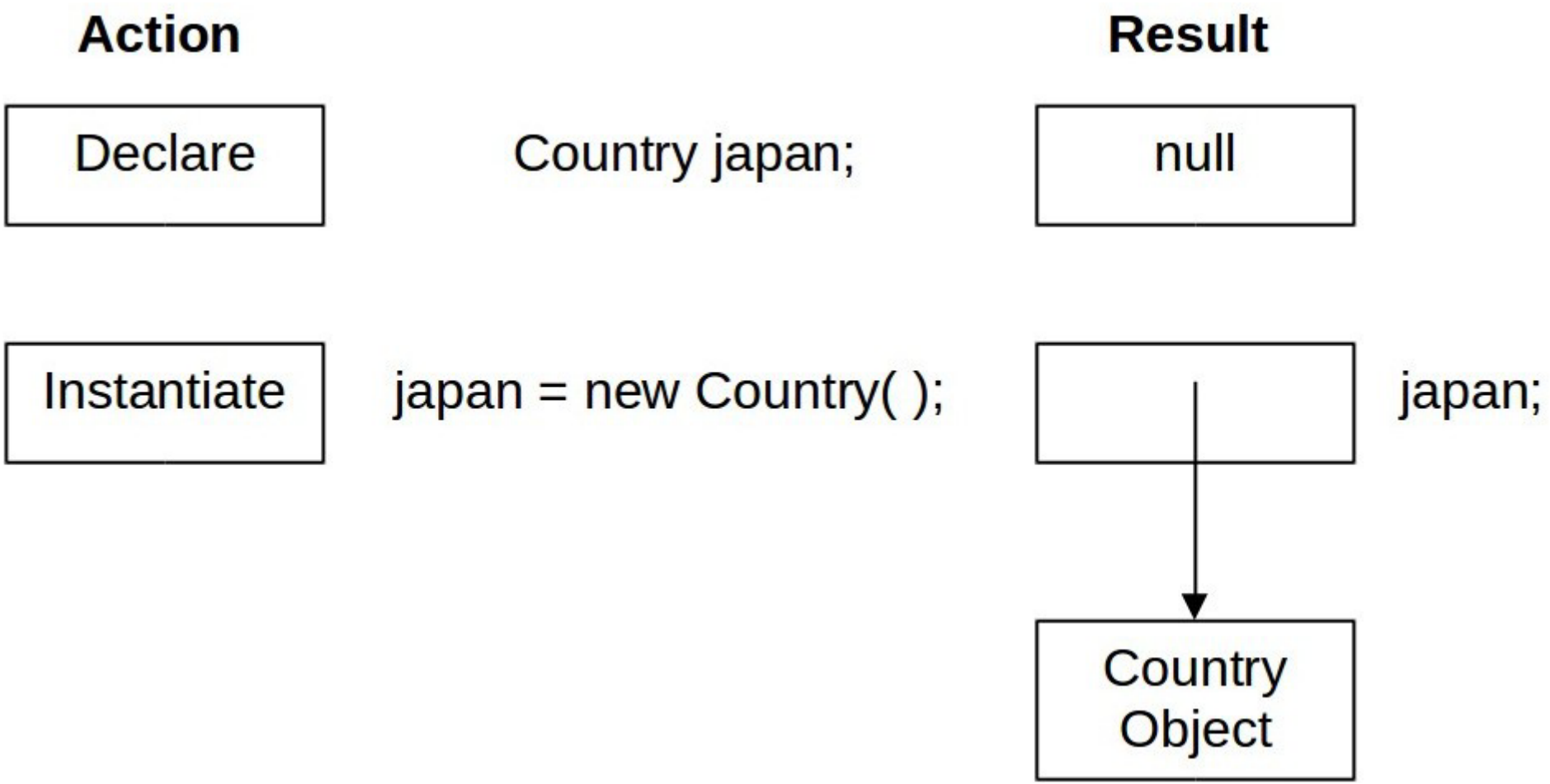
Example:

Country japan; // declaration

japan = new Country(); // instantiation or

Country japan = new Country();

Declaring and creating object



Accessing members

Member access follows the following syntax:

objectname.variablename

Example:

japan.population = bhutan. 58612453;

currancyRate = 0.236;

japan.noOfSates = 5;

Adding methods

The general form of the declaration of the method is,

```
data-type method-name (parameter-list)
{
    Body of the method;
}
```

The method declaration contains four different parts:

- Name of the method (method-name)
- Data type of the values returned by method (data-type)
- The list of parameters (parameter-list)
- Body of the method

What is a method?

```
float area(int radius)
{
    final       float pi = 3.14f;  val;
    float
    val = pi * (float) (radius * radius);  return(val);
}
```

Constructor

- A constructor initializes an object immediately upon creation.
- It has the same name as the class in which it resides and it is syntactically similar to a method.
- Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes.
- Constructors look a little strange because they have no return type, not even void.

Example:

```
class Country
{
        long population;  int noOfStates;  float currancyRate;
        Country(long x, int y)   {  population = x;  noOfStates = y;
        }
}
void display()
{
        System.out.println("Population:"+population);  System.out.println("No
        of states:"+noOfStates);
}
}
```

Garbage Collection

- Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects.
- An in use object, or a referenced object, means that some part of your program still maintains a pointer to that object.
- An unused object, or unreferenced object, is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed.

Finalize method

- finalize method in java is a special method much like main method in java. finalize() is called before Garbage collector reclaim the Object, its last chance for any object to perform cleanup activity i.e. releasing any system resources held, closing connection if open etc.

Example:

```
protected void finalize( ) throws Throwable {  try {
                System.out.println("Finalize of Sub Class");
                //release resources, perform cleanup ;
      }  catch(Throwable t){  throw t;
      }  finally   {
          System.out.println("Calling finalize of Super Class");
          super.finalize();
      }
  }
```

Type of constructors

- Default constructor
- Parameterized constructor

Example (Constructor Overloading):

```
class Box
{
        int height;
        int depth;  int
        length;
        Box()
        {
            height = depth          = length = 10;
        }
        Box(int x,int y)
        {
            height = x;  depth = y;
        }

}
```

The 'this' keyword

- Many times it is necessary to refer to its own object in a method or a constructor.
- To allow this Java defines the ‘this’ keyword. If we are familiar with C++, it is not hard to learn and understand the concept of ‘this’ keyword.
- The ‘this’ is used inside the method or constructor to refer its own object.
- That is, ‘this’ is always a reference to the object of the current class’ type.

Example:

```
class Box
{
        int height;  int depth;  int length;
        Box(int height, int depth,
        {
            this.height = height;  this.depth =
            depth;  this.length = length;
                                                    int length)
        }
}
```

Method overloading

- In Java it is possible to define two or more methods within the same class that are having the same name, but their parameter declarations are different.
- In the case, the methods are said to be overloaded, and the process is referred to as method overloading.
- Method overloading is one of the ways of Java implementation of polymorphism.

Example:

```
int add(int          x, int y) //version1
{
        cal = x + y;
        return(cal);
}
int add(int          z) //version2
{
      cal = z + 10;
      return(cal);
}
float add(float x, float                    y) //version3
{
      val    = x + y;
      return(val);
}
```

Static members

- Methods and variables defined inside the class are called as instance methods and instance variables. That is, a separate copy of them is created upon creation of each new object.
- But in some cases it is necessary to define a member that is common to all the objects and accessed without using a particular object.
- That is, the member belongs to the class as a whole rather than the objects created from the class. Such members can be created by preceding them with the keyword static.

Example:

```
static int cal;  static float min = 1;

static void display(int x)
```

Static members

- When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.
- We can declare both methods and variables to be static.
- The most common example of a static member is main( ) . main( ) is declared as static because it must be called before any objects exist.
- Instance variables declared as static are, essentially, global variables.

Restrictions to static

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

Argument passing to methods

- Call by value
- Call by reference

Call by value

```
class Test
{
        void meth(int           i, int j)
        {
                i++;  j++;


        }
}


         ob.meth(a, b);
```

Call by reference

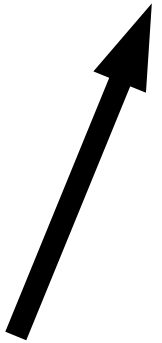
```
class Test
{
        int a, b;

        void meth(Test o)
        {
            a.  ++;
            b.  ++;
        }
}

Ob1.meth(obj2);
```

Command line arguments

public static void main(String args[ ])



Command line arguments

Command line and variable length of arguments

javac prog.java

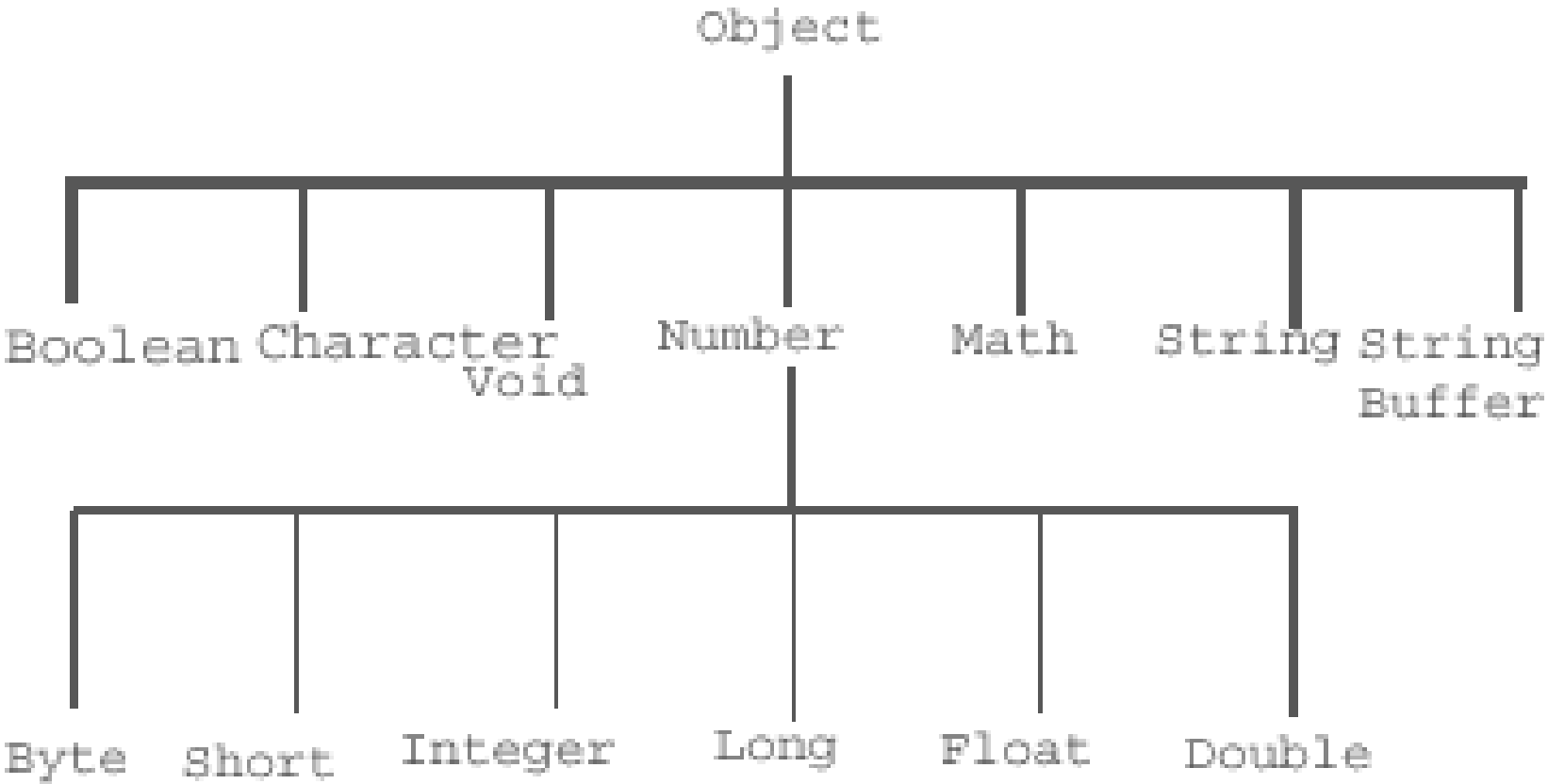
java prog Hello my name is Tushar

args[0] = Hello
args[1] = my
args[2] = name
args[3] = is
args[4] = Tushar

The Object class

- There is one special class, ‘Object’, defined by Java.
- All other classes are subclasses of ‘Object’. That is, ‘Object’ is a super class of all other classes.
- This means that a reference variable of type ‘Object’ can refer to an object of any other class.
- Also, since arrays are implemented as classes, a variable of type ‘Object’ can also refer to any array.

The object class



Class visibility controls

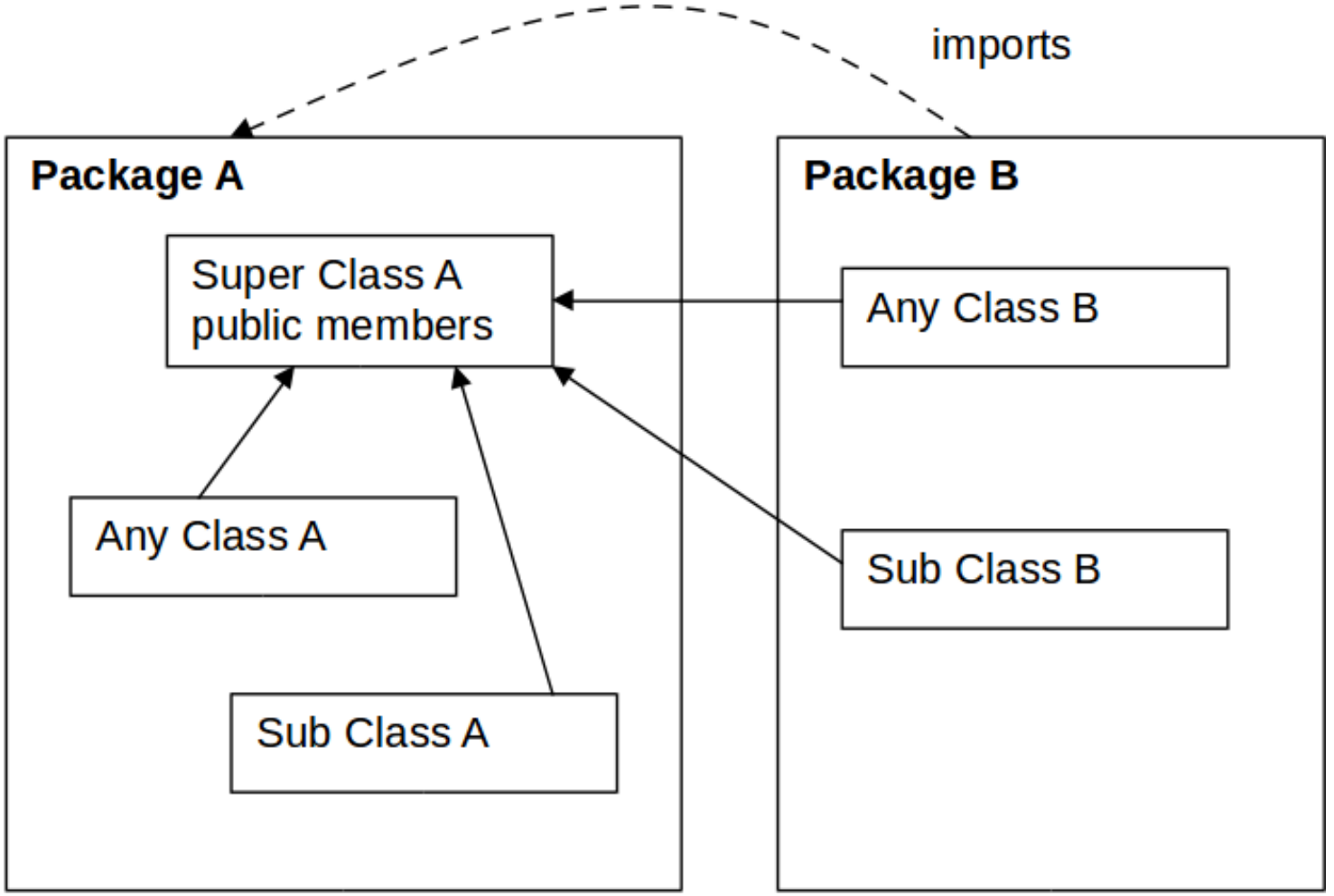
- Public
- Private
- Protecte
- d Default

Public access

- If we declare any variable and method as ‘public’, it will be accessible to all the entire class and visible to all the classes outside the class. It can be accessed from outside package also. For example,

```
public int val;
public void display();
```

Public access



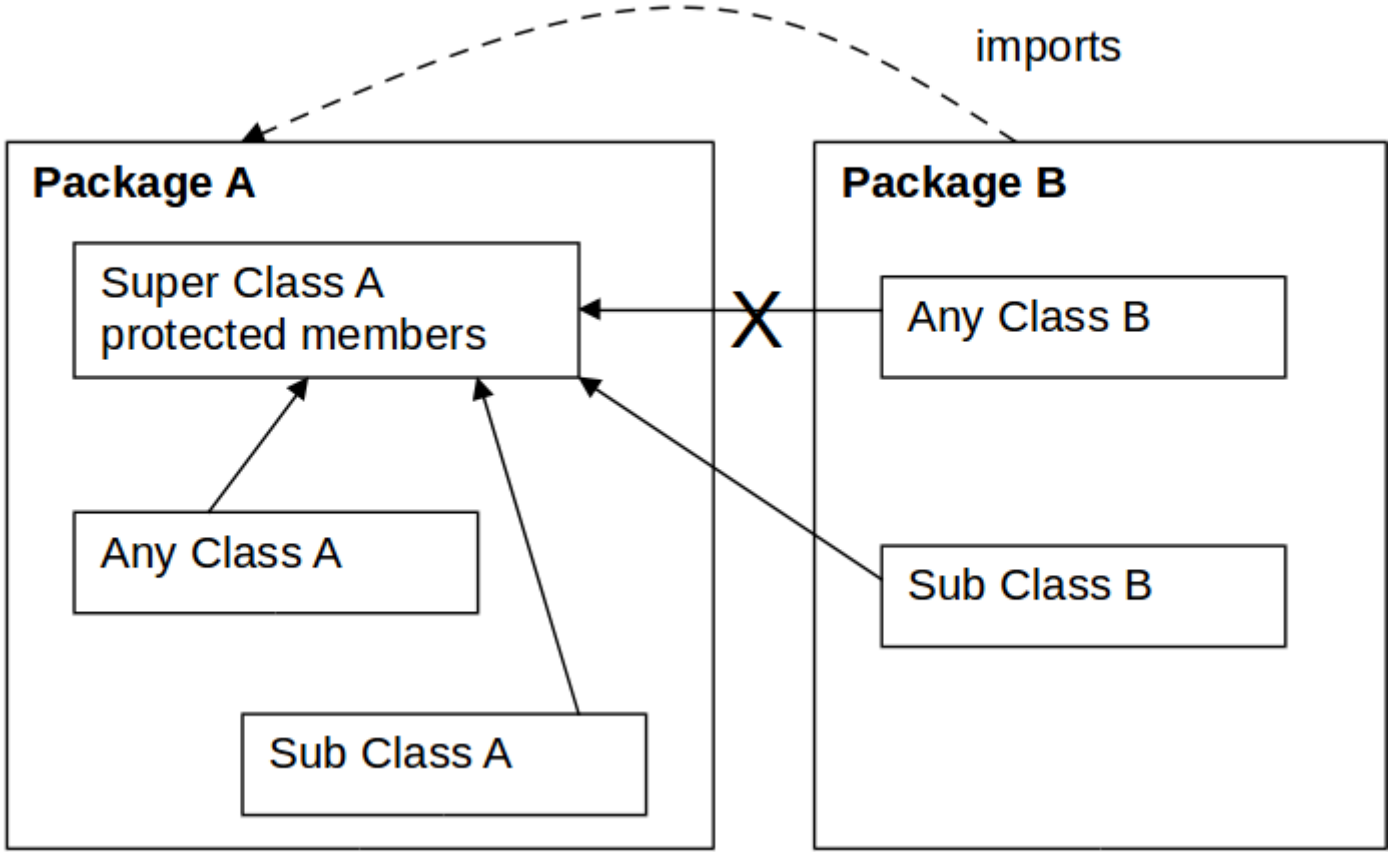
Protected access

- A protected member is accessible in all classes in the package containing its class and by and by all subclasses of its class in any package where this class is visible.
- In other words, non-subclasses in other packages cannot access protected members from other packages. Means, this visibility control is strongly related to inheritance.

  For example,

```
protected int val;
protected void display();
```

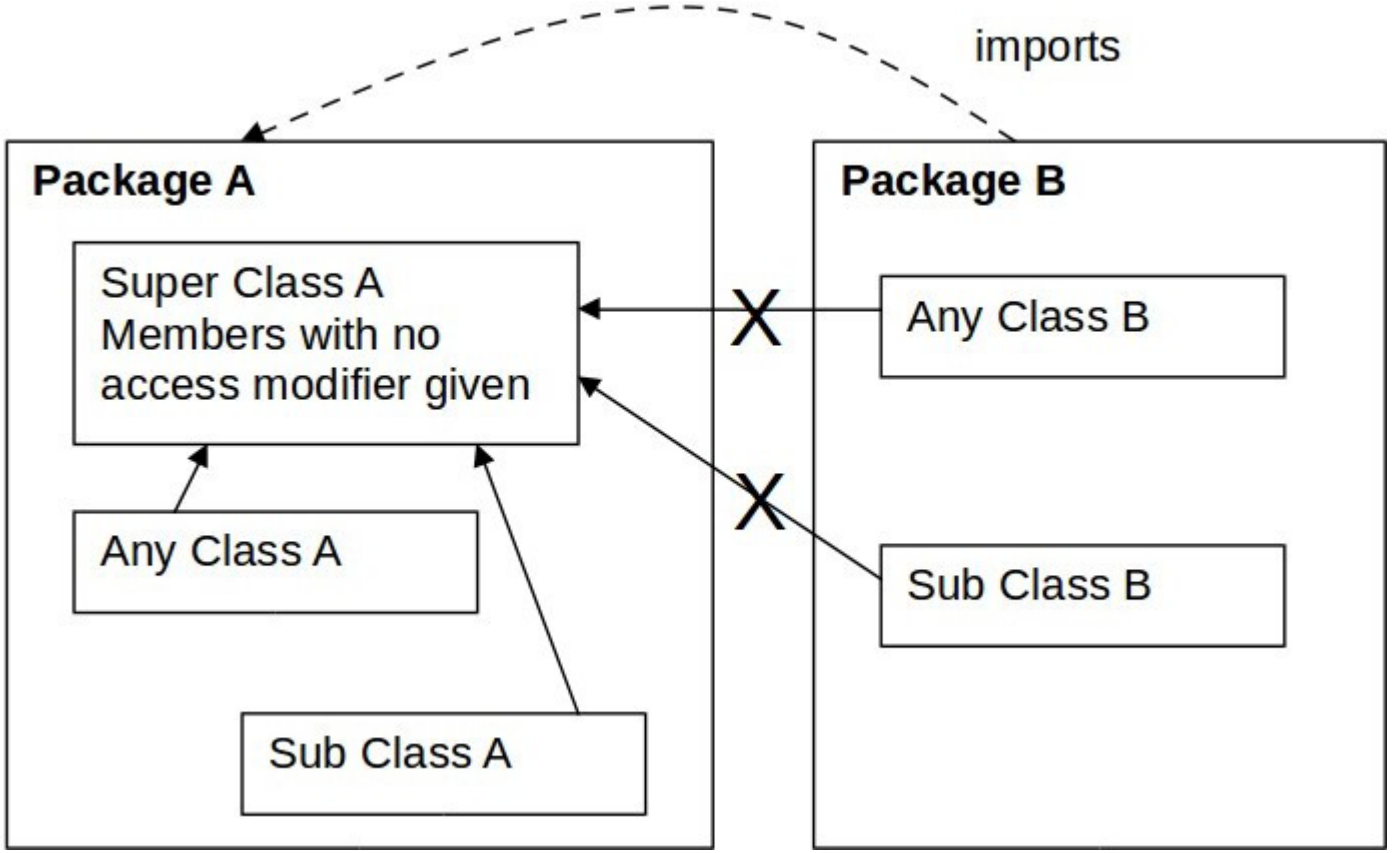
Protected access



Default access/package access/friendly access

- The default access can also be called as package access or friendly access.
- When no member accessibility modifier is specified, the member is only accessible inside its own package only.
- Even if its class is visible in another package, the member is not accessible there.
- Default member accessibility is more restrictive than the protected member accessibility.

Default access

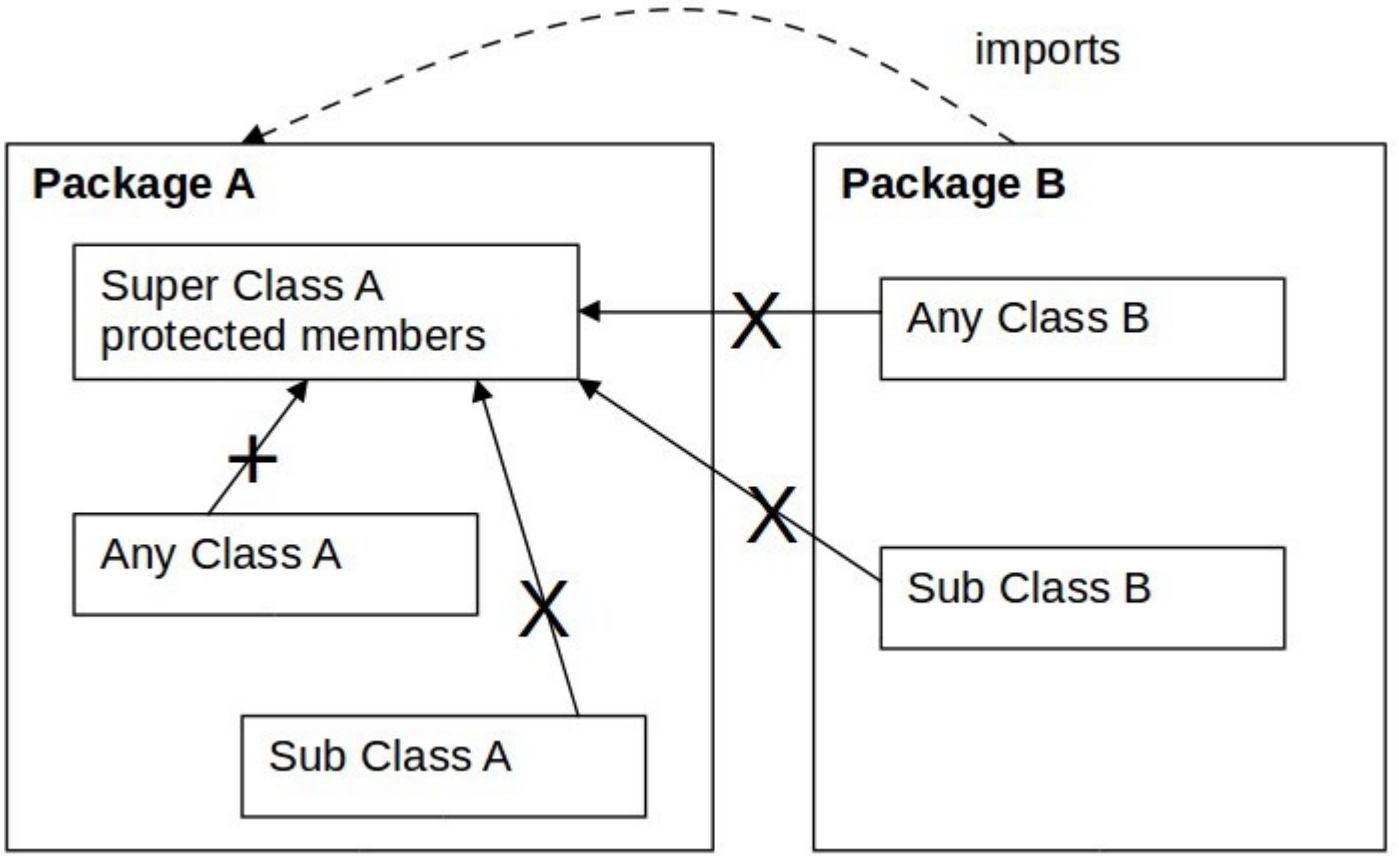


Private access

- This is most restrictive than all other visibility controls.
- Private members are not accessible from any other class.
- They can not be accessed even from any subclass or any class from the same package. In order to declare a member as private, we just need to write ‘private’ in front of the member as,

```
private int val;
private void display();
```

Private access



Arrays

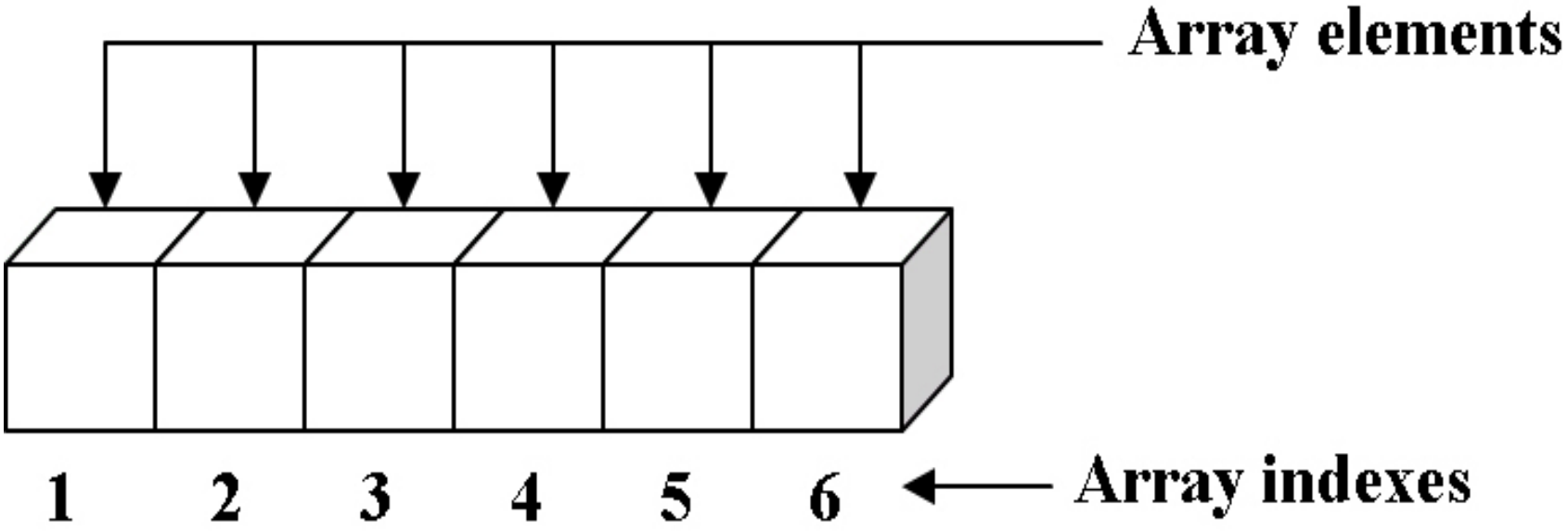
- An array is contiguous or related data items that share the common name.
- It offers the convenient means of grouping information. This means that all elements in the array have the same data type.
- A position in the array is indicated by a non-negative integer value called as index.
- An element at the given position is accessed by using this index.
- The size of array is fixed and can not increase to accommodate more elements.

Types of array

- One dimensional array
- Multidimensional

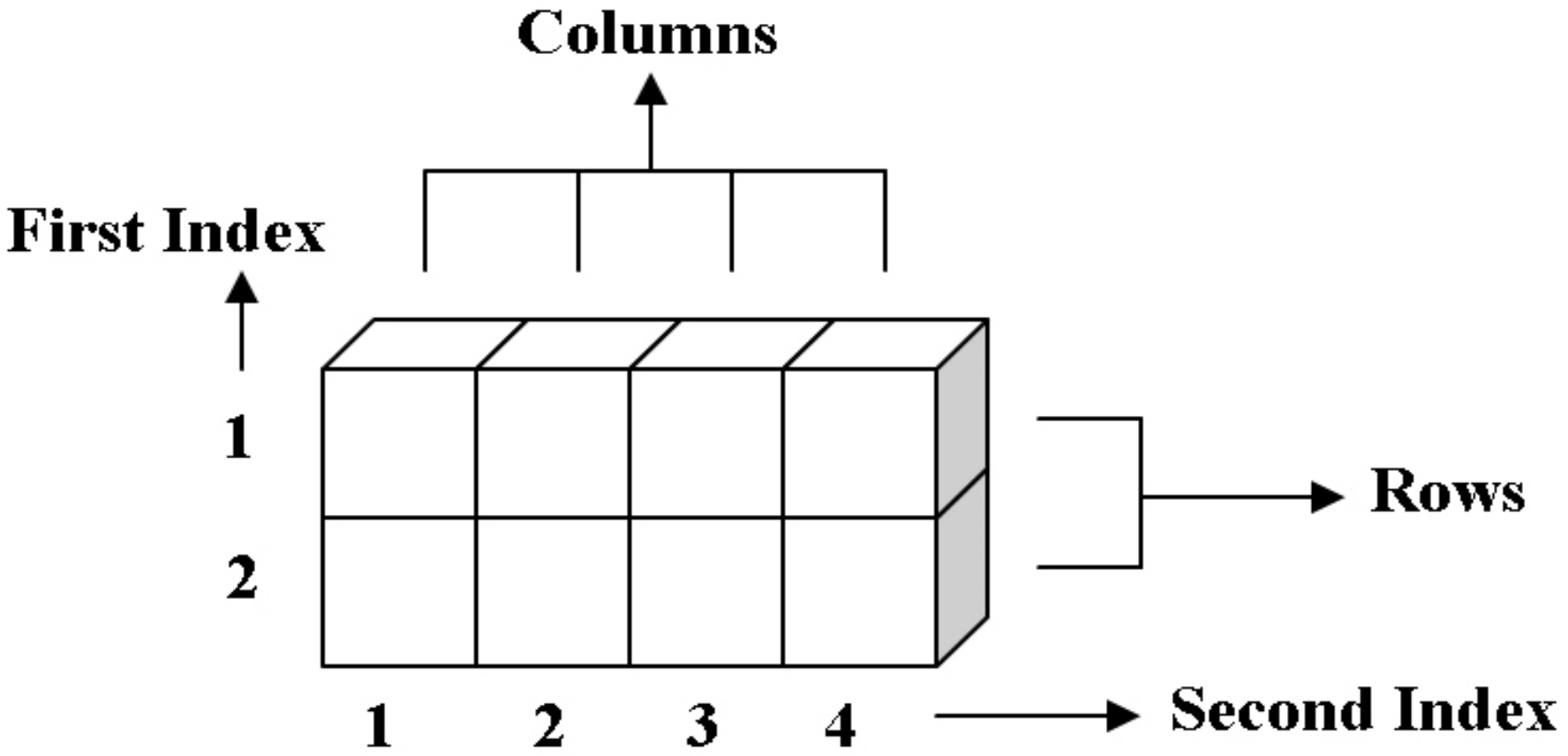
  array

1-D array



One-dimensional array with six elements

2-D array



Two-dimensional array with eight elements

Array declaration

- One dimensional array is essentially, a list of same-typed variables. A list of items can be given one variable name using only one subscript. The general form of creating one dimensional array is,

datatype variablename[ ] = new datatype[size];

Array declaration

```
int val[5] = new int[5];

int arr[] = {8, 6, 2, 4, 9, 3, 1};
```

The strings

- In Java, a string is a sequence of characters. But, unlike many other languages that implement strings as character arrays, Java implements strings as objects of type String.
- Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient.
- For example, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string.
- Also, String objects can be constructed a number of ways, making it easy to obtain a string when needed.

Creating strings

- The String class has several constructors defined. To create an empty string the default constructor is used. For example:

  String s = new String();

- It will create the instance of the string with no characters in it. If we want to initialize the string with values we can use following constructor.

  String(char chars[])

- For example:

  char chars[] = {'h', 'e', 'l', 'l', 'o'}; String s

  = new String(chars);

String length

String s = “Pramod”; int len =

s.length();

String concat

```
String      s =  “First”;
String       t =  “Second”;
s.concat(t);
```

Character at

String s = “Indian”; char ch = s.

charAt(2);

//ch will be ‘d’ after this

Equality

```
String a = “Hello”;  String b = “HELLO”;

if(a.equals(b))   //false  System.out.
    println(“Equals in case”);
if(a.equalsIgnoreCase(b))//true
    System.out.println(“Equals in characters”);
```

String Buffer

- String represents fixed-length, immutable character sequences.
- In contrast, StringBuffer represents growable and writeable character sequences.
- It is possible to insert the characters anywhere inside the StringBuffer.
- StringBuffer will automatically grow to make room for such additions and often has more characters pre-allocated than are actually needed, to allow room for growth.

Creating StringBuffer

StringBuffer class defines three different constructors:

StringBuffer() **StringBuffer(int size)** **StringBuffer(String str)**

How to use StringBuffer?

```
StringBuffer sb = new StringBuffer("Oriya");


int len =          sb.length());  be 5
//len will


int cap =          sb.capacity());  be 21
//cap will
```

Vecto
r

- The package java.util contains a library of Java’s utility classes.
- One of them is Vector class. It implements a dynamic array which can hold the array of any type and any number.
- These objects do not have to be homogenous. Capacity of the Vector can be increased automatically.

Creation of Vector

Vector class defines three different constructors:

Vector()

Vector(int size)

Vector(int size, int incr)

Operations of Vector

void addElement(Object element) int capacity()

Object elementAt(int index) int size()

boolean removeElement(Object element)

Example:

```
Vector v = new Vector(3, 2);  System.out.println("
Initial size: " + v.size( ));  System.out.println("Initial
capacity: " +
                                                    v.capacity( ));
v.addElement(new Integer(23));  v.addElement(new
Float(9.6f));
```

Wrapper classes

- The primitive data types of Java such as int, char, float are not the part of any object hierarchy.
- They are always passed by value to the method not by reference.
- Object representations of primitive data types are called as wrapper classes.
- In essence, these classes encapsulate or wrap the primitive data types within the class. All the wrapper classes are defined in package java.lang.

Wrapper classes

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

Example:

```
Character ch = new Character('X');  System.out.print("Char
value: ");  System.out.println(ch.charValue());

if(Character.isDigit('0'))  System.out.println("0 is digit");

if(Character.isLowerCase('R'))  System.out.println("R is
    lower case");
```

Enumerated data type

- Enum is type like class and interface and can be used to define a set of Enum constants.
- Enum constants are implicitly static and final and you can not change there value once created.
- Enum in Java provides type-safety and can be used inside switch statment like int variables.

Example:

```
public enum Color
{
    WHITE, BLACK, RED, YELLOW, BLUE;
}
```

Inheritance

- Reusability is one of the most useful advantages of the Object Oriented Programming.
- Reusability means using something again and again without creating a new one.
- Java programming language supports this concept. The reusability can be obtained by creating new classes and reusing the properties of the existing one.
- The mechanism of deriving a new class from existing one is called as inheritance.
- The old class or existing class is known as super class and new class is known as subclass.
- Thus, a subclass is specialized version of super class. We can call super class as base class or parent class and subclass as derived class or child class.

Types of inheritance

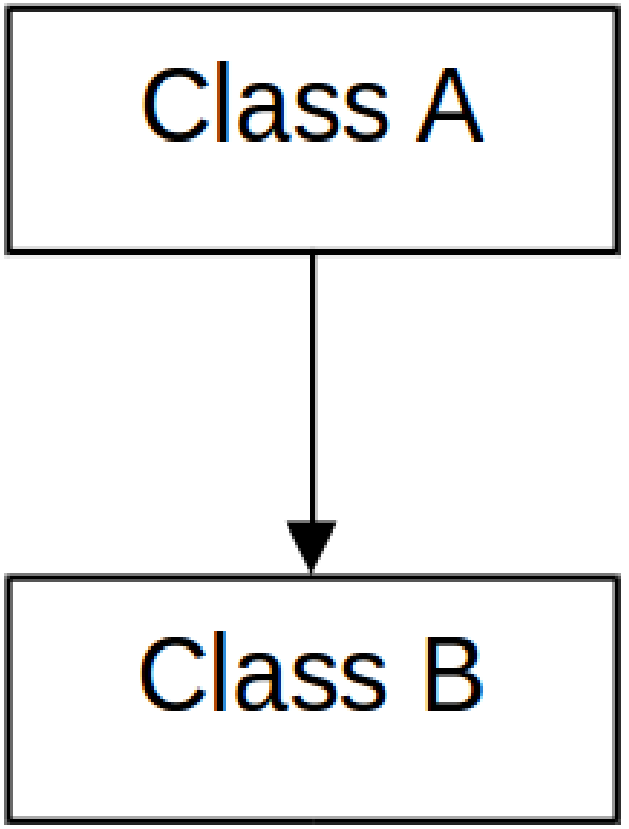
- Single Inheritance (only one super and subclass)
- Multiple Inheritance (several super classes for a subclass)
- HierarchicalInheritance (one super class with many subclasses)
- Multilevel Inheritance (subclass issuper class of another class)
- Hybrid Inheritance (combination of above three types)

Creating inheritance

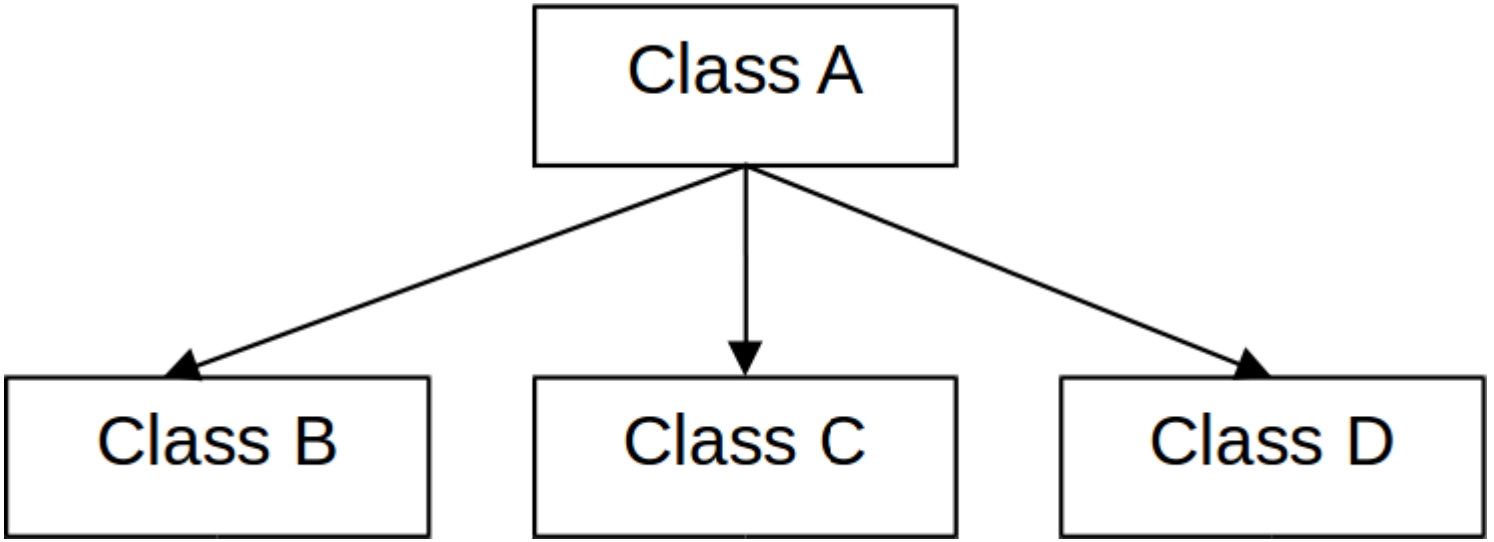
The class can derived from another class by following the syntax:

```
class subclassname extends
superclassname
{
        //Body of the class;
}
```

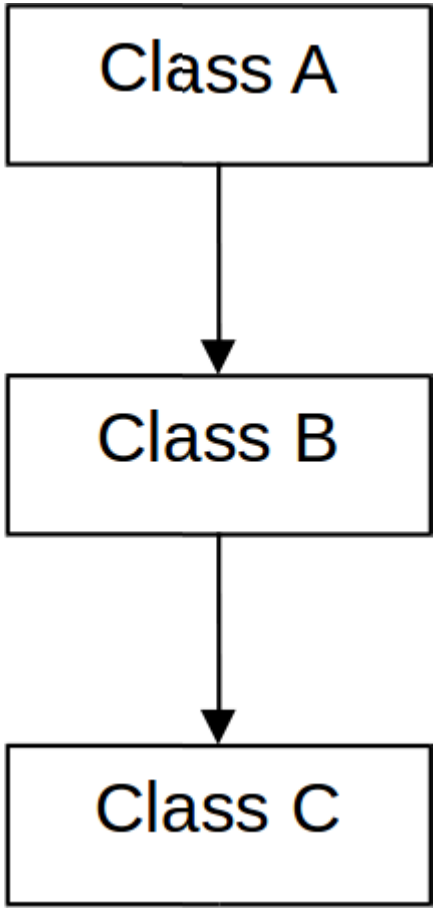
Single inheritance



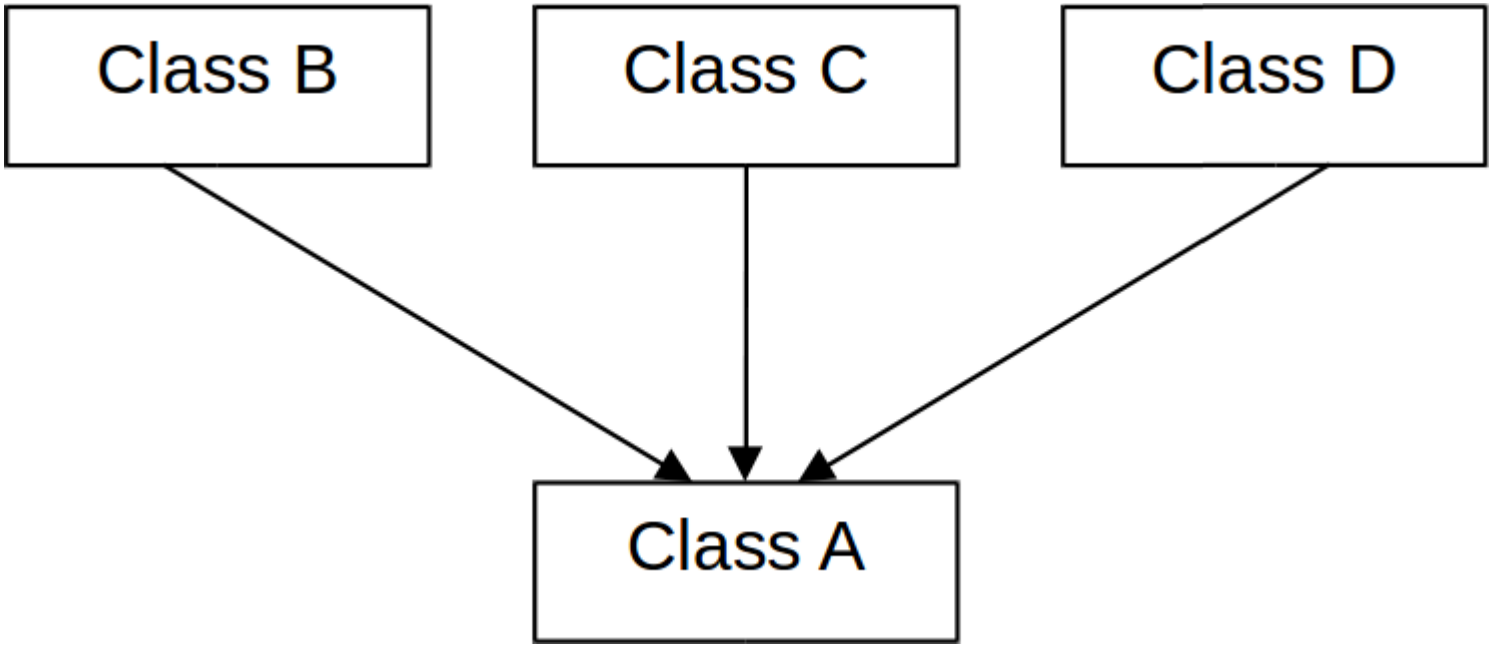
Hierarchical inheritance



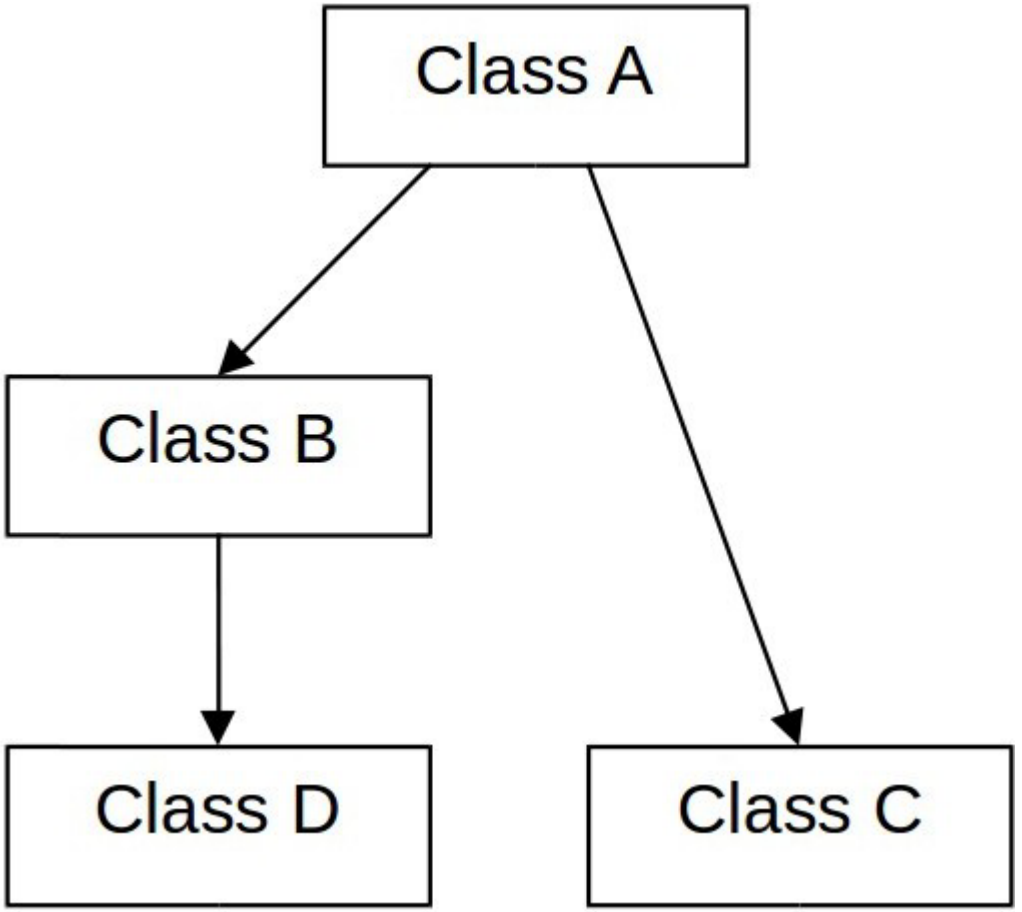
Multilevel inheritance



Multipleinheritance



Hybrid inheritance



The 'super' keyword

- The keyword ‘super’ has specially been added for inheritance.
- Whenever the subclass needs to refer to its immediate super class, ‘super’ keyword is used. There are two different applications of the ‘super’. That are,
  - Accessing members from super class that has been hidden by members of the subclass
  - Calling super class constructor

Usesof 'super'

- **Accessing members from super class**
- In order to access the members from super class following form of the ‘super’ is used.

super.variablename;

- **Calling super class constructor**
- A subclass can call the constructor defined by super class by using the following form of the ‘super’ keyword.

super(argumentlist);

- The argument list specifies any arguments needed by constructor in the super class. Remember ‘super ( )’ must always be the first statement executed inside a subclass’ constructor.

Method overriding

- In a class hierarchy, when a method in a subclass has the same name, same arguments and same return type as a method in its super class, then the method in the subclass is said to override the method in the super class.
- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass.
- The version of the method defined by the super class will be hidden.

Example:

```
class Maths
    {
            int var1, var2, var3;  Maths(int x,
            int y)
            {
                    var1 = x;  var2 = y;
            }


            void calculate()                    //statement1
            {
                    var3 = var1 +               var2;
                    System.out.println("Addition : "+var3);
            }
    }
    class Arithmetic extends Maths
    {
            Arithmetic(int x,int y)
            {
                  super(x,y);
            }

            void calculate()                    //statement2
            {
                    var3 = var1 -               var2;
                    System.out.println("Subtraction : "+var3);
            }
    }
```

Dynamic method dispatch

- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- Dynamic method dispatch is important because this is the only way in which Java implements run-time polymorphism.
- If a super class contains a method that is overridden by a subclass, then when different types of objects are referred to through super class reference variable, different versions of the method are executed.

  This concept is C++ counterpart of Virtual function in Java.
-

Example:

```
class Principal  {
        void message(){  System.out.println("This
            is                                          Principal");
        }
    }
    class Professor extends Principal {
        void message(){  System.out.println("This
            is                                          Professor");
        }
    }
    class Lecturer extends Professor  {
        void message(){  System.out.println("This
            is                                          Lecturer");
        }
    }
```

Example continued...

```
class    Dynamic   {
         public static void main(String args[])  Principal x = new            {
              Principal();  Professor y = new Professor();  Lecturer
              z = new Lecturer();  Principal ref; //reference
              variable
                                                                  of super class

              ref = x;   //statement1  ref.
              message();

              ref = y;   //statement2  ref.
              message();

              ref = z;   //statement3  ref.
              message();
         }
   }
```

Abstract classes and methods

- Sometimes we will want to create a super class that only defines a generalized form that will be shared by all of its subclasses by leaving it to each subclass to fill in the details.
- Such a class determines the nature of the methods that the subclasses must implement. This can be done by creating a class as an abstract.

Abstrac
t

- We can indicate that a method must always be redefined in a subclass by making the overriding compulsory.
- This can be achieved by preceding the keyword ‘abstract’ in the method definition with following form:

abstract datatype methodname(parameters);

Abstrac
t

```
abstract class White
{
        ……
        abstract void paint();
        ……
        ……
}
```

- We can not create the objects of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator.

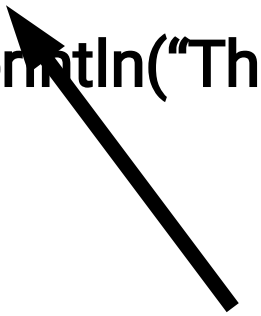
Application of 'final' in inheritance

- Prevent the method
- overriding Prevent the inheritance

Prevent method overriding

```
class First {
    final void display() {  System.out.println("This is first class");
    }

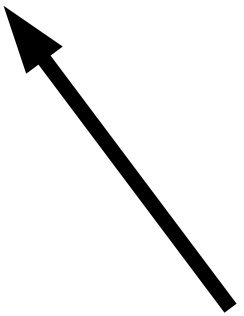

}
class Second extends  void     First {
    display() {
      System.out.println("This is second class");
    }
}
```



error

Prevent inheritance

```
final class First
{
        // ….
}
class Second        extends      First
{
        // ….
}
```



error

References



222. Core Java Programming—A Practical Approach *Tushar B. Kute*

Contents

1. Introduction to Java; **2.** Classes, Objects and Methods; **3.** Arrays, Strings and Vectors; **4.** Packages and Interfaces; **5.** Exceptions and Multithreading; **6.** Applets and Graphics Programming; **7.** Streams of File I/O; Appendix; Index.

ISBN: 978-93-83828-29-6 **EDITION:** First, 2015 **SIZE:** $7\frac{1''}{4} \times 9\frac{1''}{2}$

Thank
you



This presentation is created using LibreOffice Impress 4.2.6.3 and available freely under GNU GPL

Command-line Arguments in Java

By
M. BABY ANUSHA,
ASST.PROF IN CSE DEPT.,
RGUKT,NUZVID

Objectives

At the end of the lesson, the student should be able to:

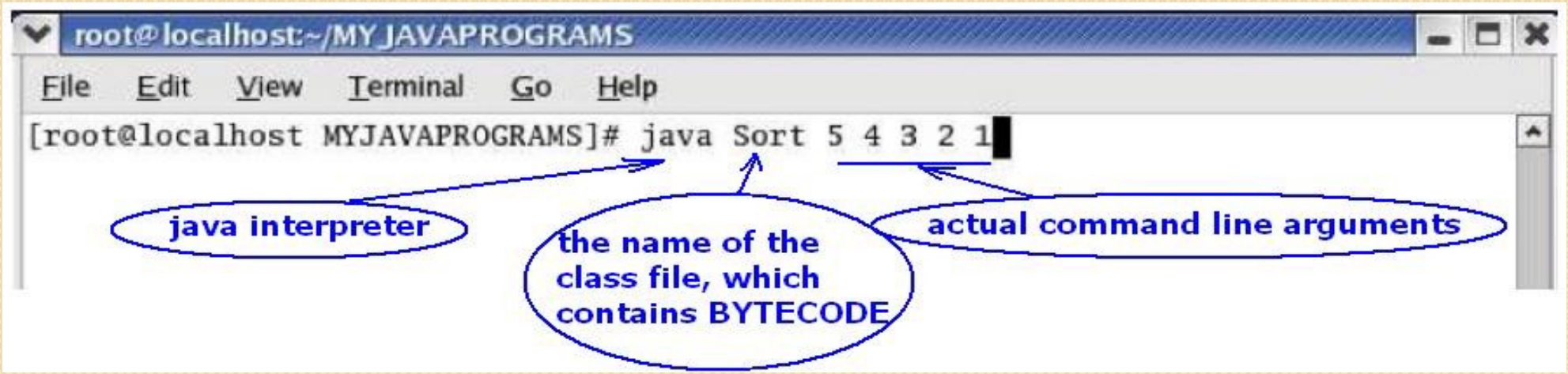
- Know and explain what a command-line argument is
- Get input from the user using command-line arguments
- How to pass command-line arguments in Command prompt

Command-line Arguments

- Command line arguments means a way to pass parameters to the main function while executing the program.
- A Java application can accept any number of arguments from the command-line.
- Command-line arguments allow the user to affect the operation of an application.
- The user enters command-line arguments when invoking the application and specifies them after the name of the class to run.

Command-line Arguments

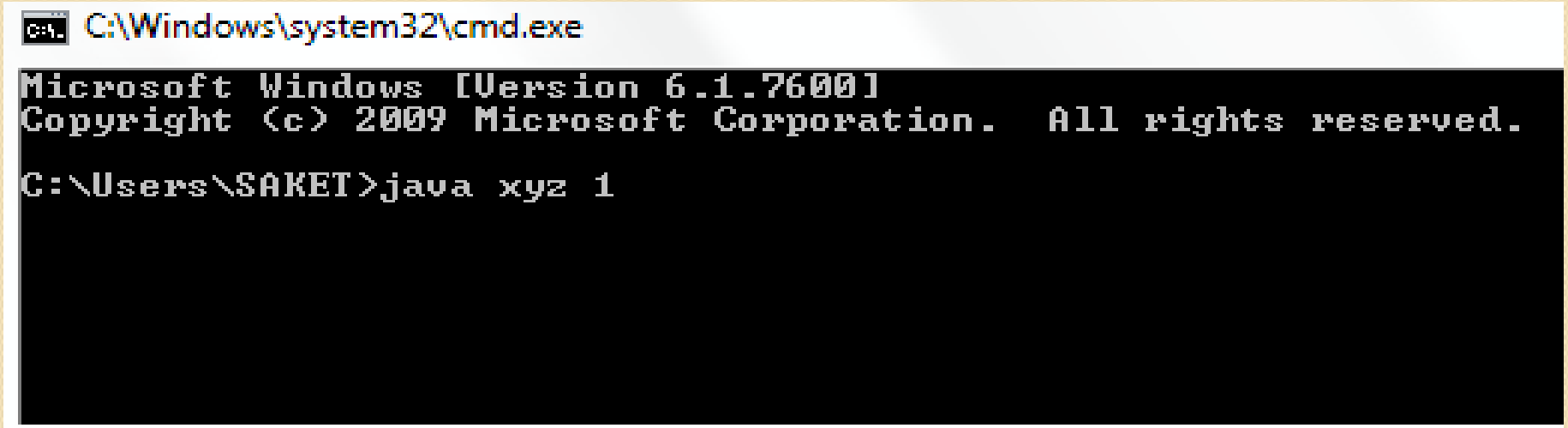
- For example, suppose you have a Java application, called Sort, that sorts five numbers, you run it like



- **Note: The arguments are separated by spaces.**

Command line arguments in DOS

- The following screen image shows **Command line arguments in DOS:**



The user enters command-line arguments when invoking the application and specifies them after the name of the class to be run.

How Java Application Receive Command-line Arguments

- In Java, when you invoke an application, the runtime system passes the command-line arguments to the application's main method via an array of Strings.

  public static void main(String[] args)

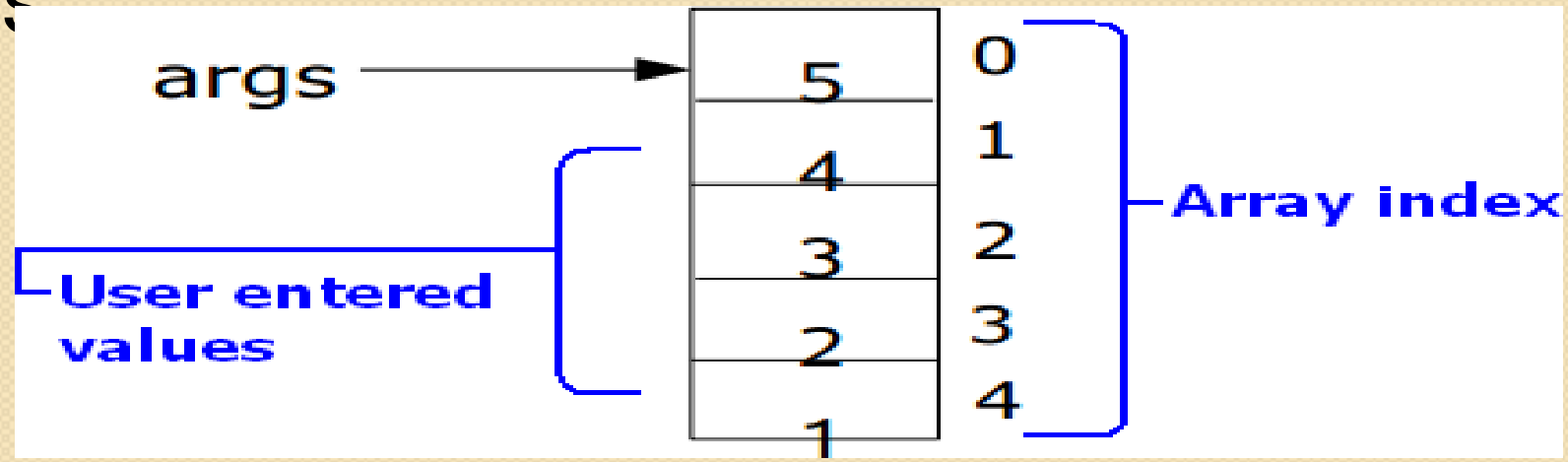
- Each String in the array contains one of the command-line arguments.

args[ ] array

- Given the previous example where we run:

java Sort 54321

- the arguments are stored in the args array of the main method declaration as



Example :

To print the array of arguments, we write:

```
public class CommandLineSample
{

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

Conversion of Command-line Arguments

- If your program needs to support a numeric command-line argument, it must convert a String argument that represents a number, such as "34", to a number.
- Here's a code snippet that converts a command- line argument to an integer,

```
int firstArg = 0;
        if (args.length > 0){
        firstArg = Integer.parseInt(args[0]);
}
```

Command-line Arguments: Coding Guidelines

- Before using command-line arguments, always check the number of arguments before accessing the array elements so that there will be no exception generated.
- For example, if your program needs the user to input 5 arguments,

```
if( args.length!= 5 ){
    System.out.println("Invalid number of arguments");
    System.out.println("Please enter 5 arguments");
}
else{
    //some statements here
}
```

Example :

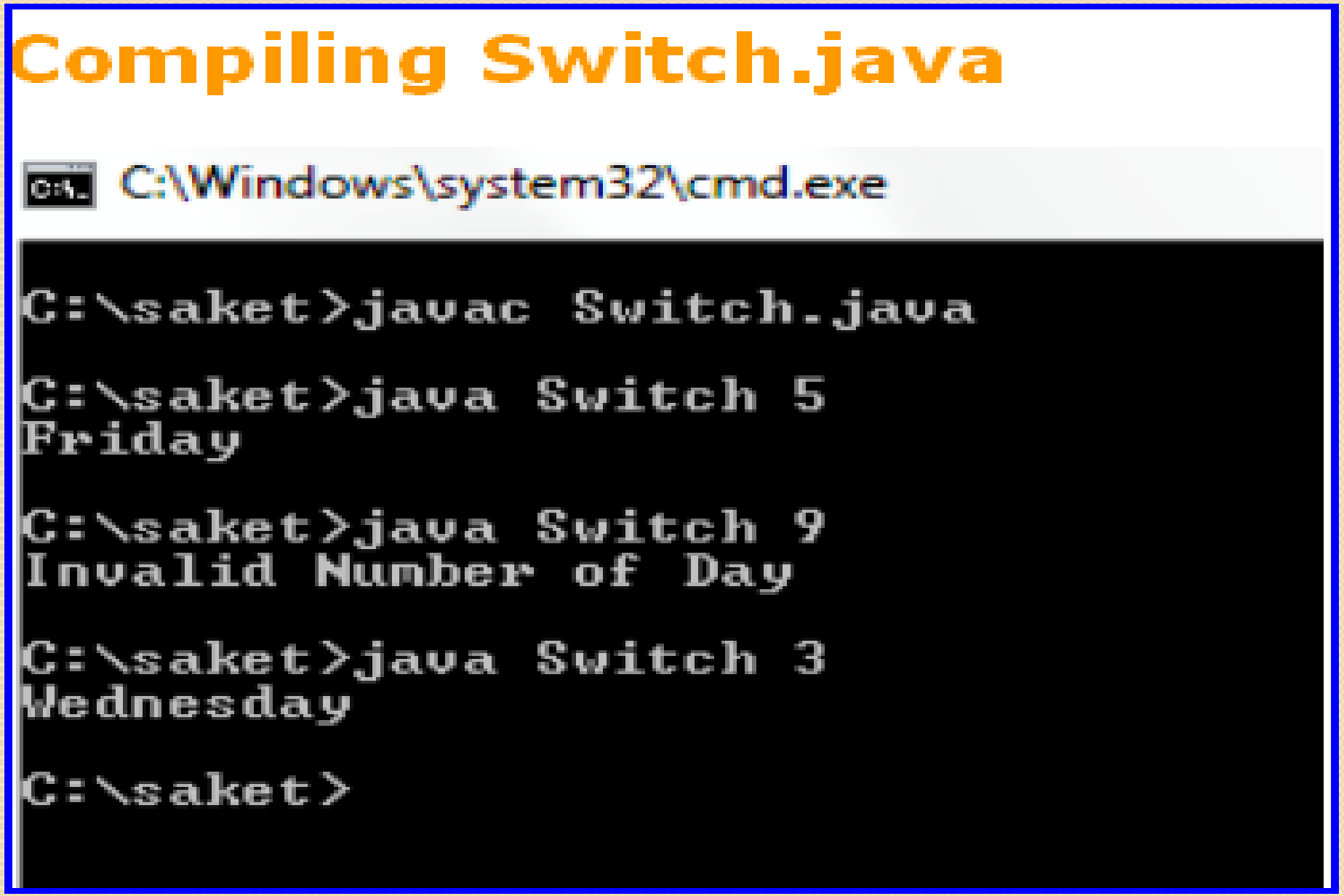
- The code snippet shows the switch method working through command line

Switch.java

```
/**
 * Switch.java
 * Integer passing in command-line arguments.
 */
public class Switch
{
public static void main(String key[])
     {

     int x=Integer.parseInt(key[0]);
     switch(x){
     case 1: System.out.println("Monday");
     break;
     case 2: System.out.println("Tuesday");
     break;
     case 3: System.out.println("Wednesday");
     break;
     case 4: System.out.println("Thursday");
     break;
     case 5: System.out.println("Friday");
     break;
     case 6: System.out.println("Saturday");
     break;
     case 7: System.out.println("Sunday");
     break;
     default :System.out.println("Invalid Number of Day");
          }
     }
}
```

Compiling and executing Switch.java



Another Example…

StringCLA.java

StringCLA.java displays each of its command line arguments on a line by itself.

```
/**
 * StringCLA.java
 * String passing in command-line arguments.
 */
class StringCLA {

    public static void main(String[] str) {
        int length = str.length;
        if (length <= 0) {
            System.out.println("Enter Some String");
        }
        for (int i = 0; i < length; i++) {
            System.out.println(str[i]);
        }
    }
}
```

Summary :

- Command-line arguments
  - How to access the arguments
  - How to convert String arguments to integer using Integer.parseInt method
  - How to pass command-line arguments in Command prompt



shutterstock.com · 567687052

Garbage Collection in Java



BY
M. BABY ANUSHA,
ASST.PROF IN CSE DEPT.,
RGUKT,NUZVID

Introduction :



- In C/C++, programmer is responsible for both creation and destruction of objects. Usually programmer neglects destruction of useless objects.
- Due to this negligence, at certain point, for creation of new objects, sufficient memory may not be available and entire program will terminate abnormally causing **OutOfMemoryErrors**.
- But in Java, the programmer need not to care for all those objects which are no longer in use. Garbage collector destroys these objects.

Garbage Collection



- Java Garbage collector tracks the live object and objects which are no more need are marked for garbage collection. It relieves developers to think of memory allocation/deallocation issues.
- JVM uses the heap, for dynamic allocation. In most of the cases, the operating systems allocate the heap in advance which is then to be managed by the JVM while the program is running.

Garbage Collection



It helps in following ways −

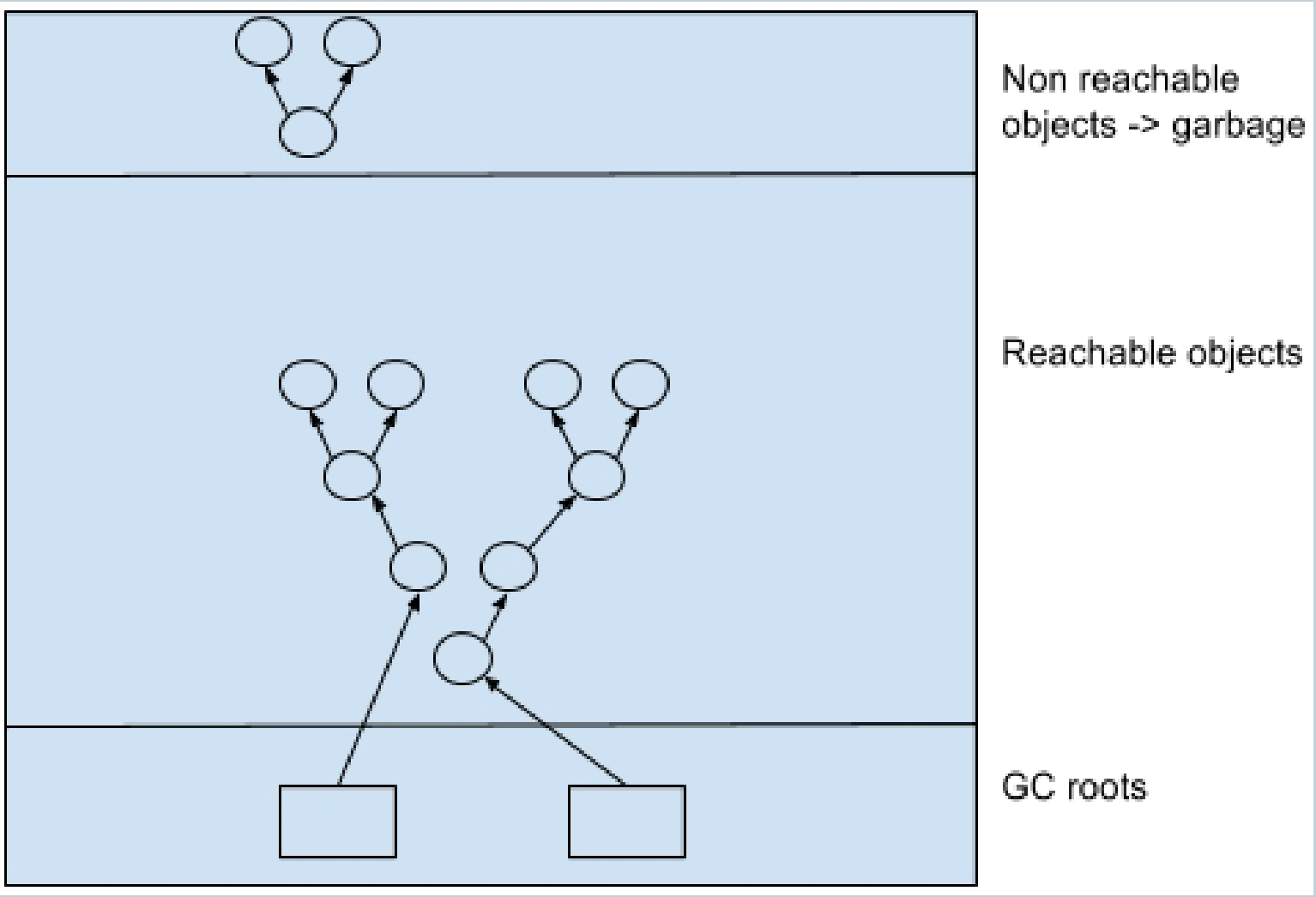
- Faster object creation is possible when heap memory is used.. Object Allocation takes some memory .
- When an object is not required, garbage collector reuses the object's memory for further allocation.

Garbage Collection



- As objects forms tree, they have one or more root objects. If root objects are reachable, the whole tree is reachable.
- There are some special objects as well which are garbage collection roots (GC roots) and are always reachable.

Garbage Collection



Garbage Collection



- Main objective of Garbage Collector is to free heap memory by destroying **unreachable objects**.

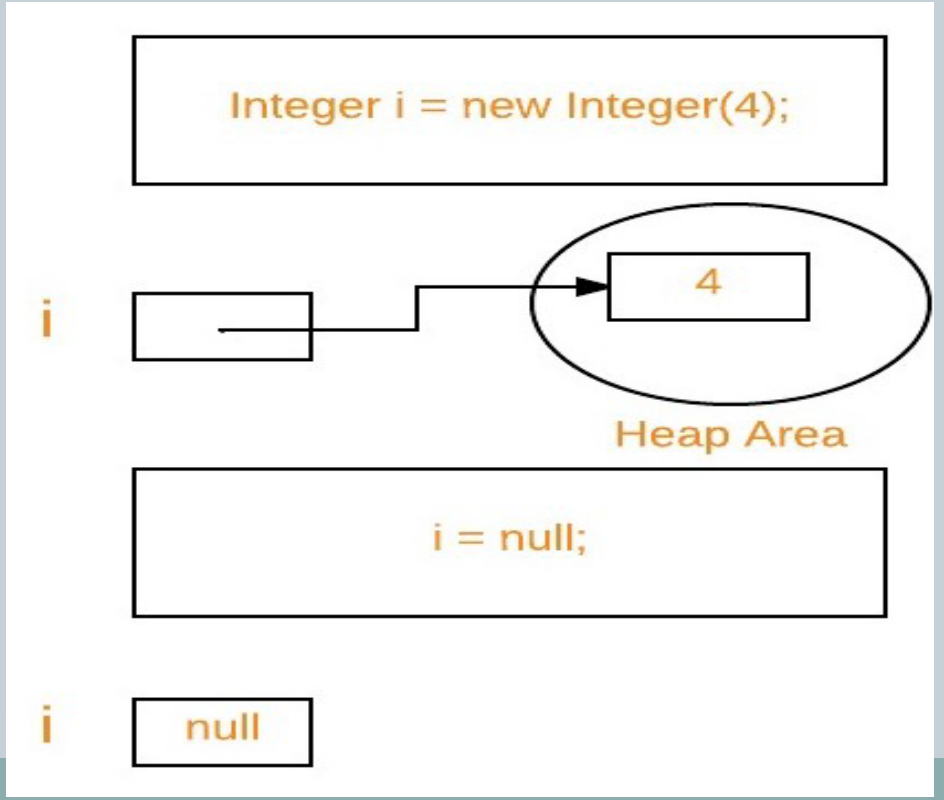
Important terms :

- **Unreachable objects :** An object is said to be unreachable iff it doesn't contain any reference to it.
- **Eligibility for garbage collection :** An object is said to be eligible for GC(garbage collection) iff it is unreachable.

Garbage Collection

Integer i = new Integer(4); // the new Integer object is reachable via the reference in ‘i’

i = null; // the Integer object is no longer reachable.



Ways to make an object eligible for GC



- Even though the programmer is not responsible to destroy useless objects but it is highly recommended to make an object unreachable(thus eligible for GC) if it is no longer required.
- There are generally different ways to make an object eligible for garbage collection.
  - Nullifying the reference variable
  - Re-assigning the reference variable
  - Object created inside method

Continued…



- Once we made object eligible for garbage collection, it may not destroy immediately by the garbage collector.
- Whenever JVM runs the Garbage Collector program, then only the object will be destroyed. But when JVM runs Garbage Collector, we can not expect.

Request JVM to run Garbage Collector



- We can also request JVM to run Garbage Collector. There are two ways to do it :
  - **Using System.gc() method** : System class contain static method gc() for requesting JVM to run Garbage Collector.
  - **Using Runtime.getRuntime().gc() method** :

  Runtime class allows the application to interface with the JVM in which the application is running. Hence by using its gc() method, we can request JVM to run Garbage Collector

```
// Java program to demonstrate requesting
// JVM to run Garbage Collector
public class Test
{
    public static void main(String[] args) throws InterruptedException
    {
        Test t1 = new Test();
        Test t2 = new Test();

        // Nullifying the reference variable
        t1 = null;

        // requesting JVM for running Garbage Collector
        System.gc();

        // Nullifying the reference variable
        t2 = null;

        // requesting JVM for running Garbage Collector
        Runtime.getRuntime().gc();

    }
```

```
// finalize method is called on object once
    // before garbage collecting it
    protected void finalize() throws Throwable
    {
        ; System.out.println("Garbage collector called");
        System.out.println("Object garbage collected : " + this)
    }
}
```

Output:

Garbage collector called Object garbage collectedTest@46d08f12
Garbage collector called Object garbage collected : Test@481779b8

Note :

- There is no guarantee that any one of above two methods will definitely run Garbage Collector.
- The call System.gc() is effectively equivalent to the call : Runtime.getRuntime().gc()

Finalization

- Just before destroying an object, Garbage Collector calls *finalize()* method on the object to perform cleanup activities. Once *finalize()* method completes, Garbage Collector destroys that object.

- *finalize()* method is present in Object class with following prototype.

  protected void finalize() throws Throwable

- Based on our requirement, we can override *finalize()* method for perform our cleanup activities like closing connection from database.

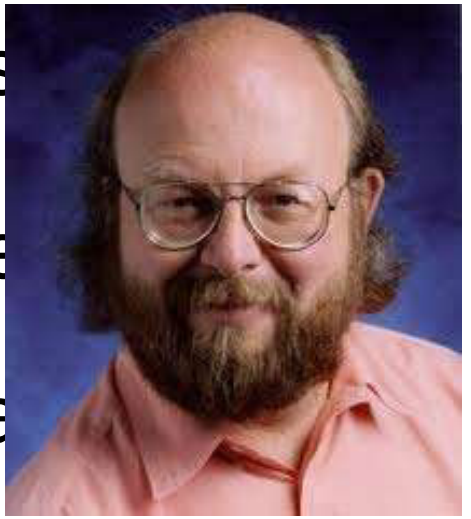


shutterstock.com · 567687052

Introduction to Java

Introduction to Java

- Java is a purely an object – oriented language.
- In 1991, James Gosling and Patrick Naughton developed named “OAK” at Sun Micros



- Java is simple and platform-independe
- In 1995, this language was renamed a
- **Java history** is interesting to know. The history of java starts from Green

Introduction to Java

- Java team members (also known as **Green Team**), initiated a revolutionary task to develop a language for digital devices such as set-top boxes, televisions etc.
- For the green team members, it was an advance concept at that time. But, it was suited for internet programming. Later, Java technology as incorporated by Netscape.
- Currently, Java is used in internet programming,

Introduction to Java - History

- There are given the major points that describes the history of java.
- **James Gosling**, **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.
- Originally designed for small, embedded systems in electronic appliances like set-top boxes.
- Firstly, it was called **"Greentalk"** by James Gosling

Introduction to Java – History – Why Oak name?

- After that, it was called **Oak** and was developed as a part of the Green project.
- **Why Oak?** Oak is a symbol of strength and choosen as a national tree of many countries like U. S.A.,France, Germany, Roman
- In 1995, Oak was renamed as **"Java"** because it was already Technologies.



Introduction to Java – History – Why Java name?

- **Why they choosed java name for java language?**

  The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", " Silk", "jolt", "DNA" etc.

- They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell and fun to say.

- According to James Gosling "Java was one of the top choices along with **Silk**". Since java was so

Introduction to Java – History

- Java is an island of Indonesia where first coffee was produced called java coffee.
- Notice that Java is just a name not an acronym.
- Originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.
- In 1995, Time magazine called **Java one of the Ten Best Products of 1995**. JDK 1.0 released in(January 23, 1996).

Introduction to Java – Versions

- Sun Microsystems has been releasing new versions of JDK. The originally version of Java released in 1995, was called JDK1.0 version.
- The versions are JDK1.1, JDK1.2, JDK1.3, JDK1.4 and JDK1.5, JDK1.6, JDK1.7 and JDK1.8.
- **Homework** - History of different versions and its features

Introduction to Java - JVM

- CPUs are capable of executing only the machine instructions.
- Different types of CPUs have different sets of machine instructions.
- To enable Java programs to be executed on multiple CPUs without modifications, java programs are compiled into machine instructions that can be understood and executed by an idealized CPU. Such a CPU is called **Java Virtual Machine (JVM)**.

Introduction to Java – Bytecode, JIT

- A Java program is first compiled into the machine code that is understood by JVM. Such a code is called **Byte Code**.
- Although the details of the JVM will differ from platform to platform, they all interpret the **Java Byte Code** into executable native machine code.
- The **Just In Time (JIT)** compiler compiles the byte code into executable machine code in real time, on a piece-by-piece demand basis.

Introduction to Java – JDK

- The JIT cannot compile the entire byte code of a Java program into executable machine code all at once, because Java Runtime Environment carries out various run time checks that can be performed only at run time.
- The collection of the three entities name the ***Java language***, the ***Java language packages*** and the set of ***Java development tools*** is called the **Java Development Kit (JDK)**.

Introduction to Java - JDK Tools

- The important development tools included in JDK are Java compiler, Java interpreter, Java disassembler, Java debugger, tool for C header files, tool for creating HTML documents and tool for viewing Java applets.
- The purpose of each one of these development tools is briefly described in the following table.

Introduction to Java – JDK Tools

Tool in JDK	Purpose
appletviewer	This tool helps us to execute a special type of Java program known as applets.
java	This tool is the Java ***interpreter***. It interprets the bytecode into machine code and then executes it
javac	This tool is Java ***compiler***. It compiles the Java source code into the bytecode, which can be understood by JVM.
javadoc	This tool is used to create documentation in HTML.
javah	This tool produces C header files for use with a special type of Java methods, known as native methods.

Introduction to Java – Types of applications

- Three types of programs, can be developed using Java, Namely *applets, servlets* and *stand-alone* applications.
- *Applet* is a Java program that is developed exclusively for the Internet.
- Applets are embedded in HTML documents that take care of the design of web pages.
- *Servlets* are used to create dynamic web content.
- *Stand-alone* applications can be executed without the internet connectivity. These applications are executed

Features of Java

Features of Java

- There is given many features of java. They are also known as java buzzwords. The Java Features given below are simple and easy to understand.

1. Simple
2. Object-Oriented
3. Platform independent
4. Secured
5. Robust
6. Architecture neutral
7. Portable
8. Dynamic
9. Interpreted
10. High Performance
11. Multithreaded
12. Distributed

Features of Java - Simple

- According to Sun, Java language is simple because:
  - syntax is based on C++ (so easier for programmers to learn it after C++).
  - removed many confusing and/or rarely-used features e.g., explicit pointers, operator overloading etc.
  - No need to remove unreferenced objects because there is Automatic Garbage

Features of Java - Object-oriented

- Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior.
- Object-oriented programming(OOPs) is a methodology that simplify software development and maintenance by providing some rules.
- Basic concepts of OOPs are:
  - Object
  - Class
  - Inheritance
  - Polymorphism

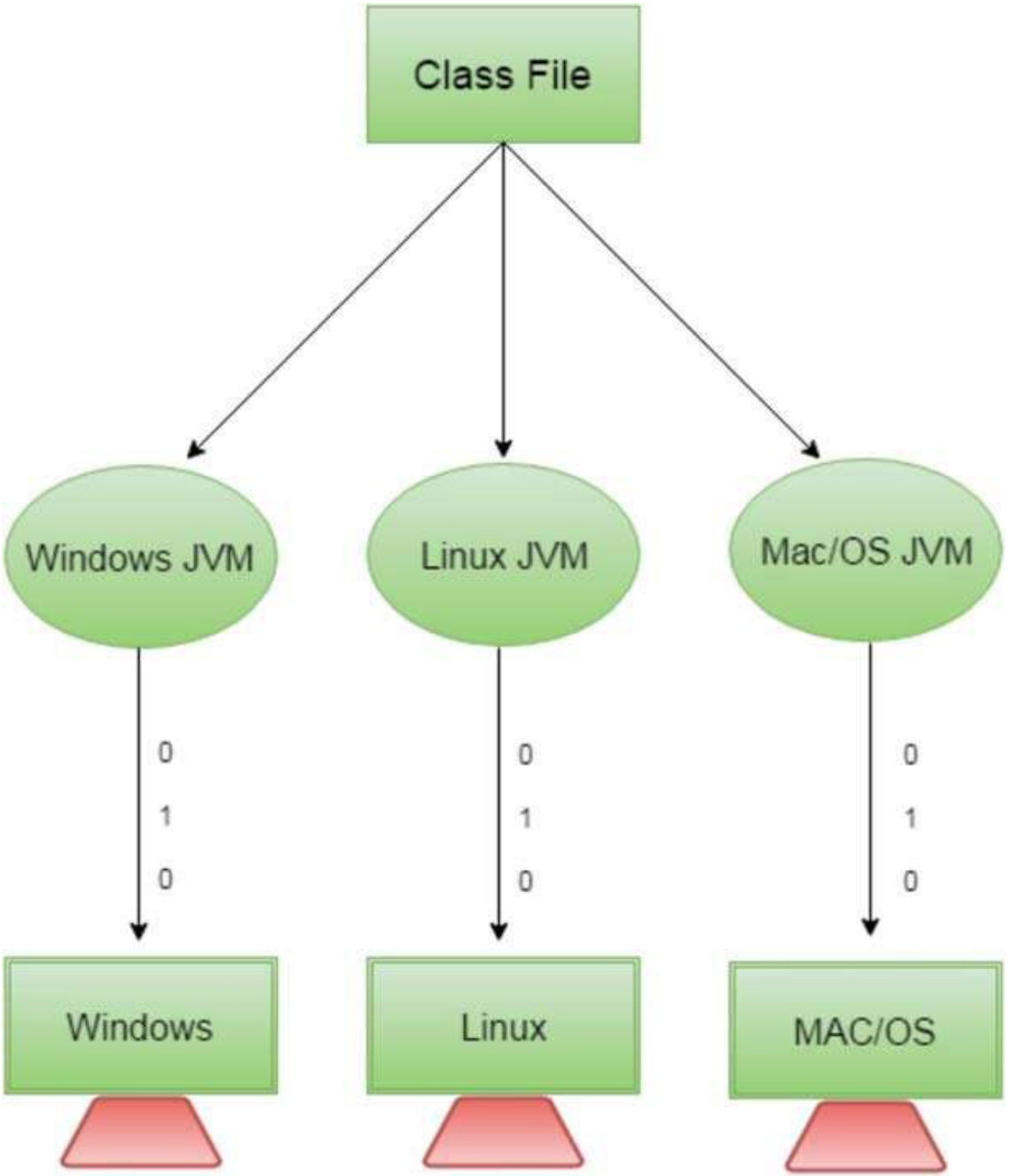
Features of Java - Platform Independent

- A platform is the hardware or software environment in which a program runs.
- There aretwo types of platforms, software-based and hardware-based.
- Java provides software-based platform.
- The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms.
- It has two components:
  1. Runtime Environment
  2. API(Application Programming Interface)

Features of Java - Platform Independent

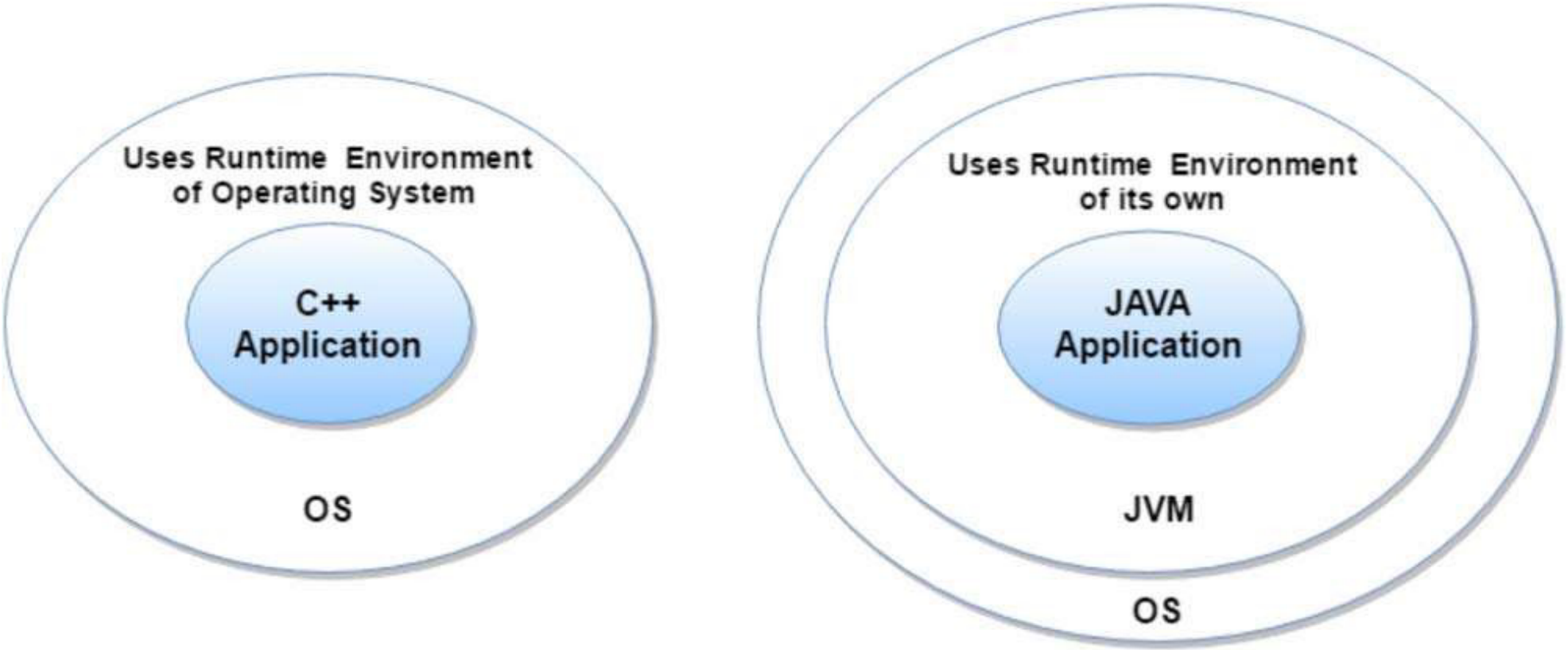
- Javacode can be runon multiple platforms e.g. Windows, Linux, Sun Solaris, Mac/OS etc.
- Java code is compiled by the compiler and converted into bytecode.
- This bytecode is a platform-independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere(WORA).

Features of Java - Platform Independent



Features of Java - Secured

- Java is secured because:
  - No explicit pointer
  - Java Programs run inside virtual machine sandbox



Features of Java - Secured

- **Classloader:** adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier:** checks the code fragments for illegal code that can violate access right to objects.
- **Security Manager:** determines what resources a class can access such as reading and writing to the local disk.
- These security are provided by java language.

Features of Java - Robust

- Robust simply means strong.
- Java uses strong memory management.
- There are lack of pointers that avoids security problem.
- There is automatic garbage collection in java.
- There is exception handling and type checking mechanism in java.
- All these points makes java robust.

Features of Java - Architecture-neutral

- There is no implementation dependent features

  e.g. size of primitive types is fixed.

- In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture.

- But in java, it occupies 4 bytes of memory for both 32 and 64 bit architectures.

Features of Java - Portable

- We may carry the java bytecode to any platform.

Features of Java – High-performance

- Java is faster than traditional interpretation since byte code is "close" to native code still somewhat slower than a compiled language (e.g. , C++)

Features of Java – Distributed

- We can create distributed applications in java.
- RMI and EJB areused for creating distributed applications.
- We may access files by callingthe methods from any machine on the internet.

Features of Java – Multi-threaded

- A thread is like a separate program, executing concurrently.
- We can write Java programs that deal with many tasks at once by defining multiple threads.
- The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area.
- Threads are important for multimedia, Web applications etc.

Object Oriented Concepts

OOPS and Java

- Java is a purely an object – oriented language.
- The central idea behind object-oriented programming is to divide a program into isolated parts called Objects.
- Each object contain two parts.
  - Data
  - Functions
- The **functions** in an object operate on the **data** involved in that object.

OOPS and Java – Objects and Classes

- Data and routines (functions) required to process it are combined into a single entity called an Object.
- A collection of similar objectsisknown as a class. (Blueprint of Object).
- In order to clearly understand the idea of objects, lets us now consider an example. Suppose we are planning a birthday party and we wish to play some songs available in our compact disk. there is an index card file, in which each card contains a song's name, its author and its playing time in the following format.
  - Song
  - name

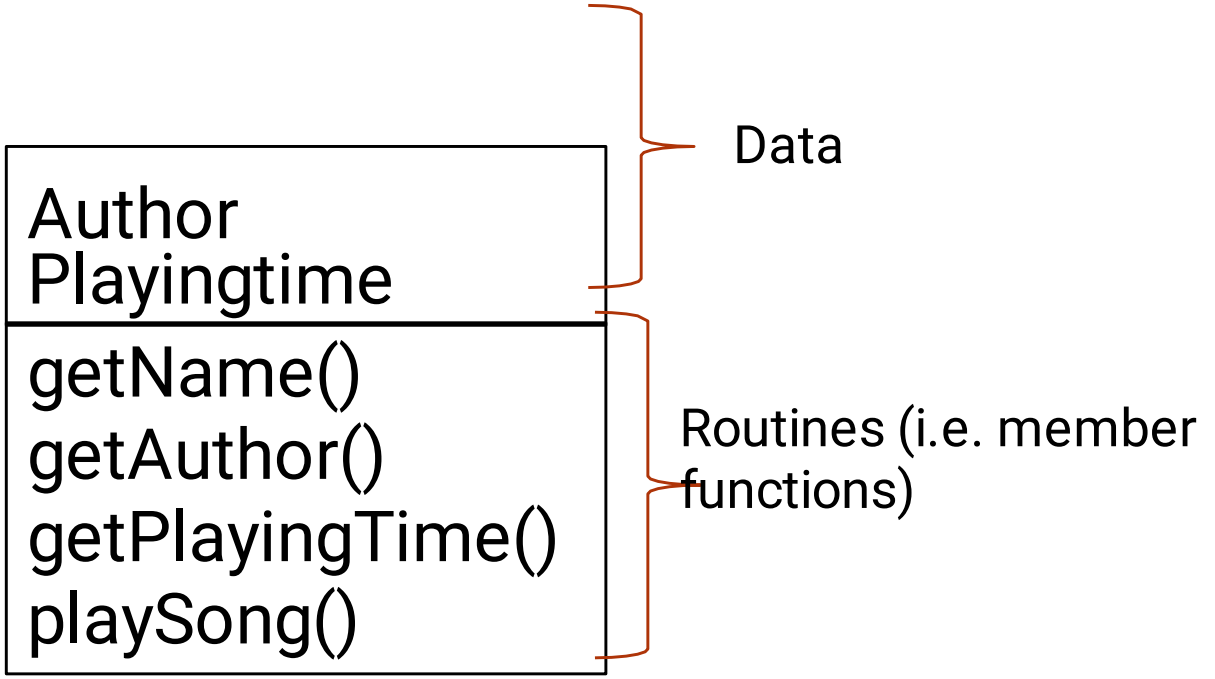
OOPS and Java - Objects and Classes

- The above mentioned **data** in the index card file can be used to get the following types of information about a particular song of interest.
  - Name of the song
  - Name of the Author,
  - Playing time of the song.
- Name the routines (funtions) to get the above mentioned three types of information from the index card file as **getName()**, **getAuthor()**, and **getPlayingTime()** and the routine for instructing the computer to play a song as **playSong()**.

OOPS and Java - Objects and Classes

- It should be carefully note here that all the above mentioned four routines are concerned with the three data items.

- The data and the related routines are combined to together to form



OOPS and Java - Objects and Classes

- Once we define a class, we can declare a group of objects for that class.
- Each object in such a group contains the data and member functions that have been defined in that class.
- In other words, each object in such a group of objects imitates the exact characteristics of the class, which serves as a plan or template.

OOPS and Java - Encapsulation

- One of the most powerful and important points of OOP is that access to a class is strictly regulated.
- If we wish to modify or access the data in an object, we have to issue a command to that object, which will use its member function to retrieve the requested data and communicate them.
- The above mentioned mode of retrieval means that the data are wantonly hidden, so that they are not directly accessible to the user. Here, the data and its related

OOPS and Java - Inheritance

- Inheritance is one the most powerful capabilities of object-oriented programming.
- Using this concept, a new class of objects can be derived from an old one. This process is called **inheritance** because the new class inherits the characteristics of the original class.
- The new class is called a **derived class** of the original class and the original class is called the **base class** of the new class.

OOPS and Java - Inheritance

- The derived class can called as sub class or child class.
- The base class can called as super class or parent class.
- There are five types of Inheritance
  - Singe Inheritance – One base class, one derived class.
  - Multilevel Inheritance – The derived class canact as base class.
  - Multiple Inheritance – More than one base class, one

OOPS and Java – Polymorphism

- We can build up subclasses from a base class by using the concept of inheritance.
- We can specify a unique behavior to each subclass by using the concept of polymorphism.
- Thus, we can design the derived classes in such a way that they respond to a message that a base class responds to, but at the same time they can perform different functions by making use of concepts of polymorphism.

OOPS and Java – Polymorphism

- If we redefine the functionality of a base class member function in a subclass, we say that we override its base class functionality.
- Overloading an important feature of OOP is that process of attaching different functionalities to a function or a operator depending on the different contexts.
- Two types.
  - Runtime Polymorphism – Virtual Functions

A Simple Java Program
(Creating and Executing)

A Simple Java Program (Creating & Executing)

- File name is ***Hello.java***

```
class Hello
{
        public static void main(String[] args)
        {
                System.out.println("Hello,  Welcome to
Java Application");
        }
}
```

- Type the above program in notepad editor.
- Save it as ***Hello.java*** in bindirectory of java software.
- Compile theprogram using command ***javac Hello.java***

A Simple Java Program (Creating & Executing)

- In java, a source program is first compiled and interpreted. After the compilation is over, the compiled version of the source file is stored in special format, namely the bytecode format.
- The source file is stored under the name *Hello.java*
  - The compiled code is stored under the name *Hello.class* During the interpretation stage, *Hello.class* file is converted into the machine code

A Simple Java Program (Creating & Executing)

- Hello.class file contains the byte code corresponding to the code that we have written in Hello.java.
- The Java interpreter, name java, loads the bytecode in Hello.class file, starts the execution of our program, and loads the necessary library files.

Syntax: *java class name* Example: *java Hello*

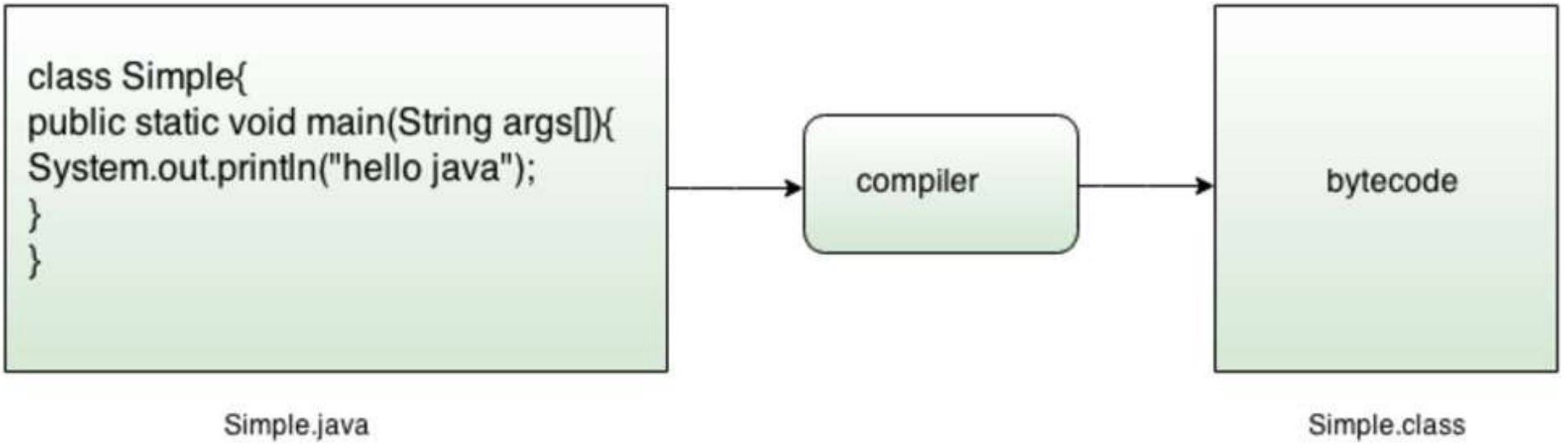
- ***Ouput will be***

Hello, Welcome to Java Applicaiton

What happens at Compile Time?

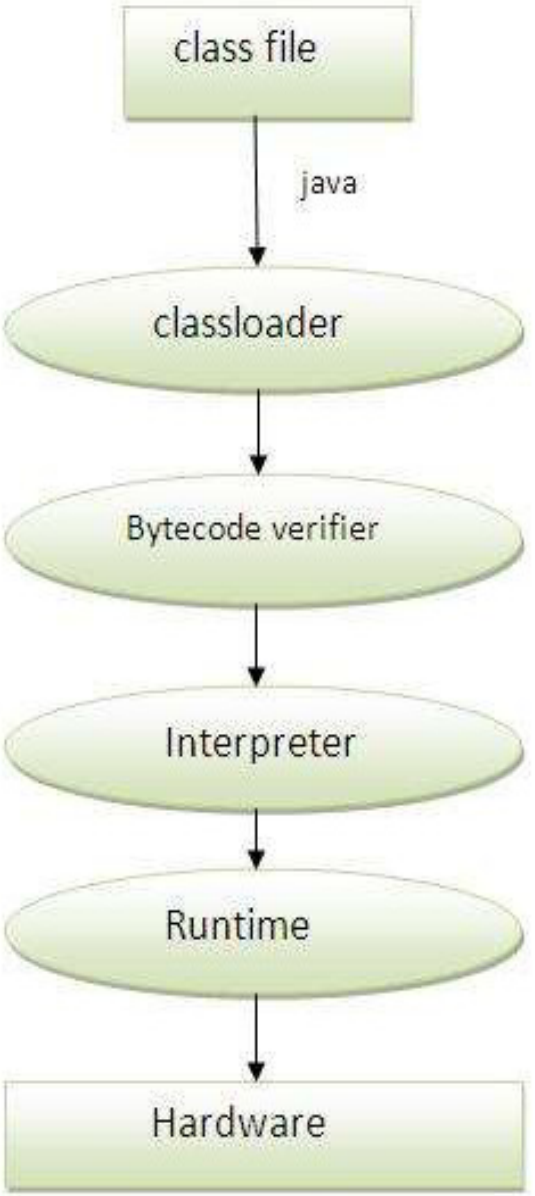
time?

- At compile time, java file is compiled by Java Compiler (It does not interact with OS) and converts the java code into bytecode.



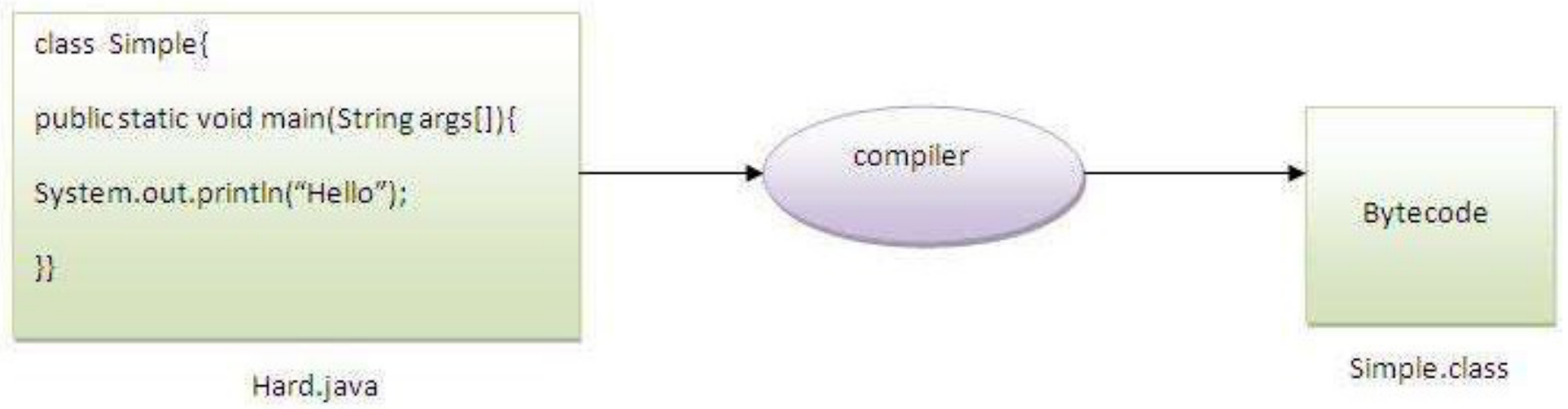
time?

- **Classloader:** is the subsystem of JVM that is used to load class files.
- **Bytecode Verifier:** checks the code fragments for illegal code that can violate access right to objects.
- **Interpreter:** read bytecode stream then



by other name than the class name?

- Yes, if the class is not public. It is explained in the figure given below:
- **To compile:** javac Hard.java
- **To execute:** java Simple



A Simple Java Program
(Understanding)

A Simple Java Program (Understanding)

- File name is *Hello.java*

```
class Hello
{
        public static void main(String[] args)
        {
                System.out.println("Hello,   Welcome to
Java Application");
        }
}
```

- As Java is a pure OOP-based language, each simple Java program is written in the form of a class.
- In java, a class consists of certain **data members** and the related **methods.**

A Simple Java Program (Understanding)

- Since the Hello.java program is a very simple one involving any input or output variables, it doesn't contain any data members.
- It contains only one method, namely the main(), which must be present in every Java stand-alone application. Since this method is not expected to return any value, the return type of this method is indicated in the above program as void.
- In java, we usually create an object of a class and

A Simple Java Program (Understanding)

- However, in certain cases, we prefer to access the methods in a class without creating an object for that class. Such methods are known as static methods.
- It shall be noted here that main() is a static method.
- As is apparent from the above program, the elements of an array of String type, called args, constitute the parameters of the main() method.

A Simple Java Program (Understanding)

- But it is not necessary to always provide the values for these parameters.
- The first periodin ***System.out.println*** means "locate the ***out*** object in the System class".
- The Second periodmeans "apply the***println()*** method to that object".
- The System class, the ***out*** object and the ***println()*** method have all been predefined by the Java developers.

Character Set and Tokens

Character Set

- The set of characters allowed in Java constitutes its character set. All the constituents of a java program, such as the constants, variables, expressions and statements are written only by using characters in the following character set:

  Numerals : 0-9

  Alphapets : a-z (Lower Case), A-Z (Upper Case)

  Special characters: + - * / % = < > ( ) . \ ; , ‘ ’ : ? # $! ^

  & ~ [ ] { }

- Java uses the uniform 16-bit coding scheme called Unicode to represent characters internally. The Unicode can

Tokens - Keywords

- The smallest individual entities in a Java program are known as tokens.
- Five important types of tokens are listed below:
  - Reserved keywords, identifiers, literals, operators, separators.
- Around 60 words have been reserved as the keywords in Java. They have predefined specific meanings. They should be written in lower-case letters. Here are some valid keywords.
  - break, continue, switch, private, while

Tokens – Rules for defining identifiers

- The names that the programmer assign for the classes, methods, variables, objects, packages and interfaces in a java program are known as **identifiers**. They should follow the following rules.
  - They shall contain digits, alphabets, underscore and dollar sign characters.
  - No white spaces are allowed.
  - A keywords should not be used as a identifier.
  - The first character should not be a digit.
  - They shall have any number of characters.

Tokens – Valid & Invalid Identifiers

- Some valid identifiers are furnished below
  - ***rectangle, perimeter$, count, r2, shape2d, tv_Channel***
- Some invalid identifiers are furnished below
  - ***rect angle, 2day, tv&Channel, good morning, wel@com***

Tokens – Literals

- A sequence of valid characters that represents a constant value is called a literal. Literals are also know as constants.
- Five major types of literals available in Java are as follows
  - Integer
  - Floating point
  - Character
  - String

Tokens – Operator and Separator

- A symbol that represents an operation, such as multiplication or division, is called an **operator**.
- Some of the well-known operators are available in Java are as follows.
  - + - * / %< > && || ! ++ -- += -= *= /=
- A symbol that is used to separate one group of code from another group is called a **separator**.

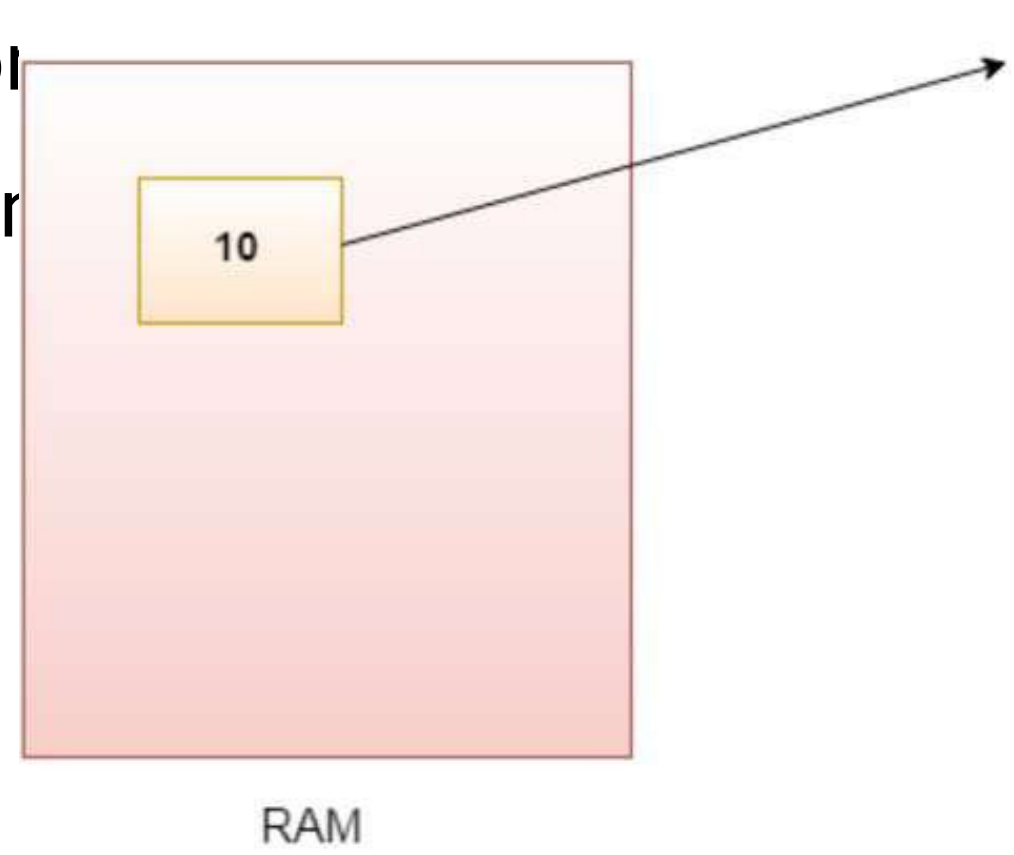
Some of the well-known separators are listed below

( ) [] ; , ‘ ’ “ ”

Variables and Constants

Variables

- **Variable** is name of *reserved area allocated in memory*. In other words, it is a *name of memory location*.
- It is a co s value ca



Variables - Rules

- The first character in a variable name should not be digit.
- A variable name may consist of alphabets, digits, the underscore character and the dollar character.
- A keyword should not be use as a variable name.
- White spaces are not allowed within a variable name.
- A variable name can be of any length

Variables

- Every variable has a **data type**.
- Every variable has a **value**.
- Every variable is the name of a **storage location.**
- Every variable has its **size**.

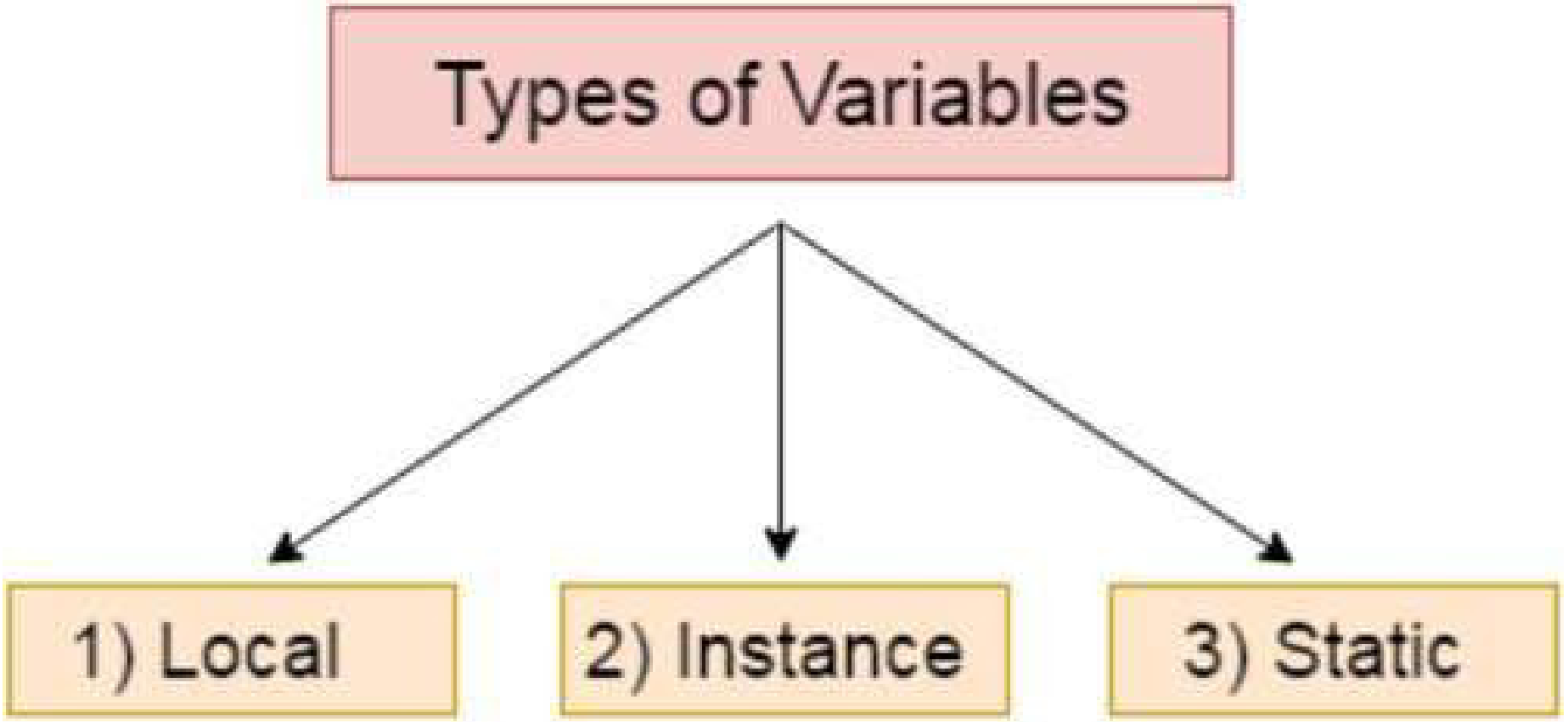
Variables - Declaration

- Variable can declared as follows

  access-specifier data-type variable-name

- Here,
  - ***access-specifier*** specifies which methods can access the variable. Most variables are declared as private.
  - ***data-type*** refers to the type of the variable. Here is a valid type declaration statement.

Variables – Types of Variables



Variables – Types of Variables

1) **Local Variable**

A variable which is declared inside the method is called local variable.

2) **Instance Variable**

A variable which is declared inside the class but outside the method, is called instance variable . It is not declared as static.

3) **Static variable**

A variable that is declared as static is called static variable. It cannot be local.

Variables – Examples to understand types of Variables

```
class A
{
        int data=50;//instance variable
        static int m=100;//static variable

        void method()
        {
                int n=90;//local variable
        } //end of method.
}//end of class
```

Constants

- ***Constants*** means it can not be changed, that means fixed value. Declaration as follows:

  const const-name=value;

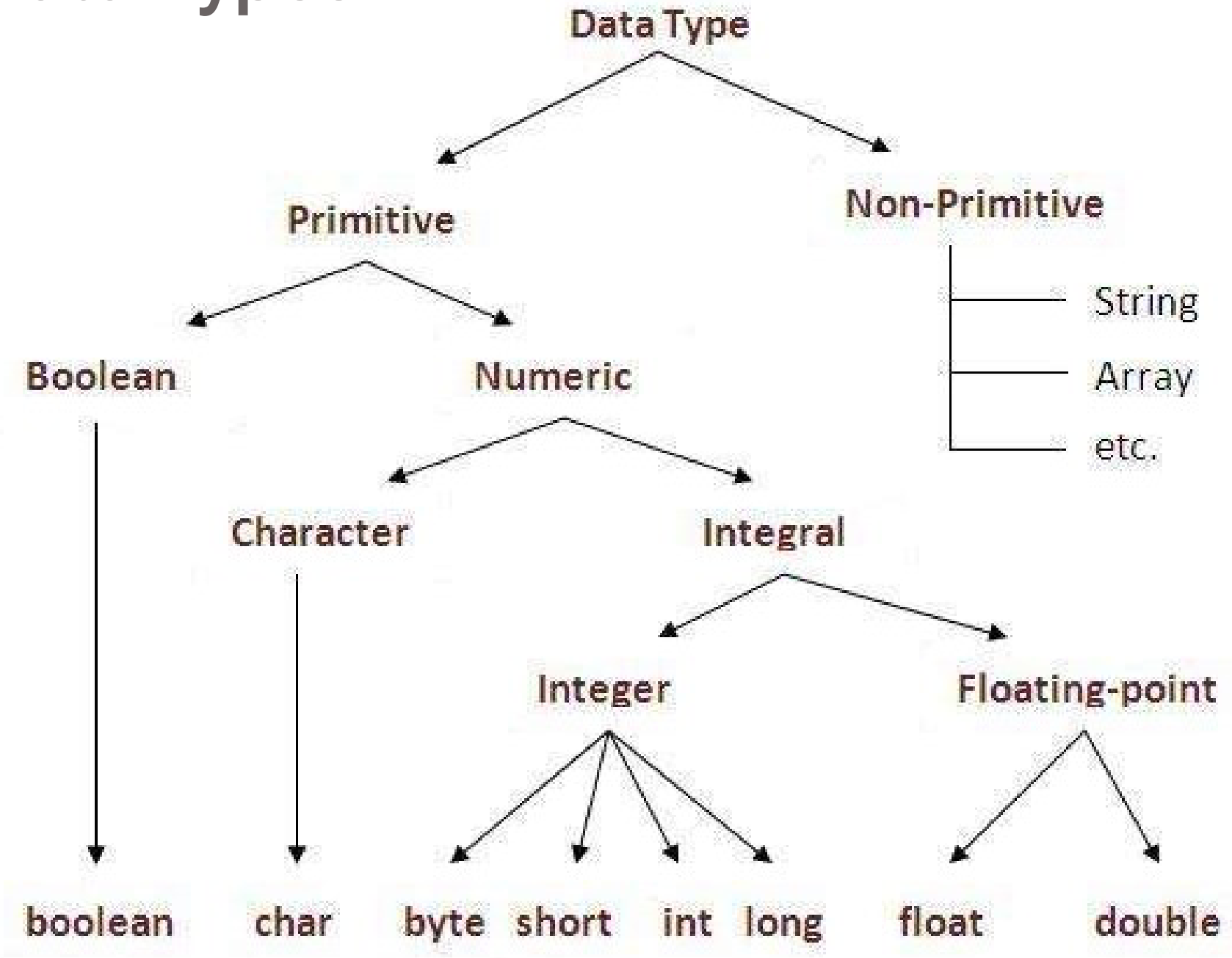
- ***Types are***
  - Integer constant
  - Floating point constant
  - Character constant
  - Boolean constant
  - Backslash character constant or Escape Sequence Character

Constants – Backslash Character Constants

Backslash character constant	Meaning
\b	Backspace
\n	New line
\t	Tab
\r	Carriage return
\f	Form feed
\\	Backslash
\’	Single quote
\”	Double quote

Data Types

Data Types


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

Data Types – Default Values

Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

HomeWork: Listthe data types with range

Operators and Expressions

Operators and Expressions

- **Operator** in java is a symbol that is used to perform operations.
- There are many types of operators in java such as unary operator, arithmetic operator, relational operator, shift operator, bitwise operator, ternary operator and assignment operator.
- These operators are classified into three
  - Unary Operator – works with one operand
  - Binary Operator – works with two operands.

Operators and Expressions

- Combination of operators and operands are called expressions.
- When we assign an expression to a variable, the types of the variable and the expression should be compatible.

double sum = 10.1; int total = sum;

We must convert the floating point expression to integer by using a cast, as shown below

Library Methods

Library Methods

- Library methods are normally used to write the expressions in a compact manner. For example, the expression

$$x = \frac{\sqrt{b \pm b^2 - 4ac}}{2a}$$

can be compactly represented as shown below, by using the sqrt() library methods:

```
( -b + Math.sqrt(b*b-4*a*c))/(2*a)
```

Library Methods

Library Method	Purpose
Math.sqrt(x)	To find square root of x (>=0)
Math.pow(x,y)	To find X^y (x >0 and x =0 and y>0 or x<0 and y is an integer)
Math.sin(x)	To find Sine of x (x in radians)
Math.cos(x)	To find Cosine of x
Math.tan(x)	To find Tangent of x
Math.asin(x)	To find Arc sine of x
Math.acos(x)	To find Arc cosine of x
Math.atan(x)	To find Arc tangent of x
Math.toRadians(x)	To convert x radians to degrees

Library Methods

Library Method	Purpose
Math.toDegrees (x)	To convert x degrees to radians
Math.exp(x)	To find e^x
Math.log(x)	To find natural log
Math.round(x)	To find closest integer to x
Math.ceil(x)	To find smallest integer >=x
Math.floor(x)	To find largest integer <=x
Math.abs(x)	To find the absolute value |x|

Strings

Strings

- ***string*** is the most important data type used in programs.
- A ***string*** is a sequence of characters. In, Java a string is enclosed in quotation marks, as in the case of the example “Don Bosco College”.
- The ***String*** keyword, which represents the ***string*** data type, starts with an uppercase letter. This is because of the fact that the keyword ***String*** is the name of a predefined class.

Strings – *length()* Method

- The number of characters in a string is provided by the length() method of the String class, as illustrated below

```
String collegeName = “Don Bosco College”;  int wordLength =
collegeName.length();
```

- Since the string “Don Bosco College” consists of 17 characters, the *length()* method will return the value 17.
- A string contains no is called an

Strings – Concatenation Operator (+)

- Two strings shall be concatenated by using the + operator. This operator is a powerful operator. If one of the two operands, either to the left or the right of the + operator is a string, then the other operand is automatically converted into a string and both strings are concatenated. Here is an example.

```
String s = “Price:”;  int price = 83;
```

Strings

- Here, the expression s + price has strings operand and an integer operand. They will be concatenated into the single string “Price : 83” and assigned to the variable result.

Input and Output Statements (Scanner and System Class)

Input and Output Statements

- Every program needs some data as input. Similarly, every program provides some processed data as output.
- While the input operations enable us to provide input data to a program, the output operations help us to get the output displayed in a desired format.
- Java provides a rich set of I/O functions for performing a variety of operations.

I/O statement – Console Output

- The *out* object in the *System* class represents the standard output stream.
- The *out* object has two methods for sending the output to the Console screen. They are *print()* and the *println()* methods.
- While the *print()* method displays the output without a newline character, the *println()* method displays the output with a newline character.
- Since both the *print()* and the *println()* methods are heavily overloaded methods, they can output many different types of data, such as ***int, double, char,***

I/O statement – Console Output

System.out.print("Welcome to Java."); System.out.println("Have a nice time.");

- Here, the *System.out.print()* method displays the message Welcome to Java on the screen and retains the cursor on the same line.
- The *System.out.println()* method displays the string Have a nice time and moves the cursor to the beginning of the next line. Here is a sample output:

Welcome to Java.Have a nice time.

I/O statement – Console Output

- Sometimes, we may wish to have a blank line between two sets of output values. We shall print a blank line by using *println()* method, as shown below:

```
System.out.println("Value of A = 1000");  System.out.println();
System.out.println("Value of B = 2000");
```

- The above snippet will render the output in the following form:

```
Value of A = 1000
```

I/O statement – Console Input

- In java, any input is read in as a string.

```
System.out.print("Enter Your Name :");  InputStreamReader reader =
new
                                        InputStreamReader(System.in);
BufferedReader in = new BufferedReader( reader);
String name = in.readLine();
```

- In the above program, the name that is keyed in by the user is received by the system and stored in the buffer System.in terms of bytes.

I/O statement – Console Input

- When the second statement in the above snippet is executed, the bytes in the buffer *System.in* are converted into characters and stored in the object *reader*.
- An object *reader* has *InputStreamReader*, can read characters, but it cannot read a while string at time.
- Inorder to overcome this limitation, we turn the input stream reader into a *BufferedReader* object,

I/O statement – Console Input

- When the fourth statement in the following snippet is executed, the *readLine()* method reads in a string (i.e. a name) and stores it in the *String* type variable name.
- Suppose that we provide 1000 as the input to the following snippet.
- This value will be initially read in as the string “1000” and stored as the value of a string type variable text.
- The *parseInt()* method, which is available in the

I/O statement – Console Input

```
System.out.print("Enter the amount in dollars:");
InputStreamReader reader = new
                                InputStreamReader( System.in );
BufferedReader in = new BufferedReader( reader );  String text = in.
readLine();
int dollar = Integer.parseInt( text );
```

- Here, *parseInt()* method is used to convert string into int.

I/O - Console Input using Scanner Class

- There are various ways to read input from the keyboard, the java.util.Scanner class is one of them.
- The **Java Scanner** class breaks the input into tokens using a delimiter that is whitespace by default.
- It provides many methods to read and parse various primitive values.
- Java Scanner class is widely used to parse text for

Methods

Method	Description
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.

I/O - Scanner Class Example

```
import java.util.Scanner;

class ScannerTest

{

public static void main(String args[])
{

        Scanner sc=new Scanner(System.in);  System.out.

        println("Enter your rollno");  int rollno=sc.nextInt();

        System.out.println("Enter your name");  String

        name=sc.next();
```

I/O - Scanner Class Example

```
        System.out.println("Enter your fee");

        double fee=sc.nextDouble();

        System.out.println("Rollno:"+rollno+"name:  "+name+"
fee:"+fee);

        sc.close();

}

}
```

Output:

```
Enter your rollno  111
Enter your name  Vijay
Enter  450000
Rollno:111 name:Vijay fee:450000
```

Control Statements (Conditional, Looping, Jumping)

Control Statements

- The control statements are used to control over the **order of execution of the statements.**
- The two major types of control statements available in Java are the **decision-making statements** and the **repetitive statements**.
  - **Decision-making Statements** – if, if-else, switch
  - **Repetitive statements** – for, while, do-

Control Statements (Conditional or Decision-making)

Control Statements

- The Java *if statement* is used to test the condition.

  It checks boolean condition: *true* or *false*.

- There are various types of if statement in java.
  - if statement
  - if-else statement
  - nested if statement
  - if-else-if ladder
- Switch statement

Control Statements – if statement

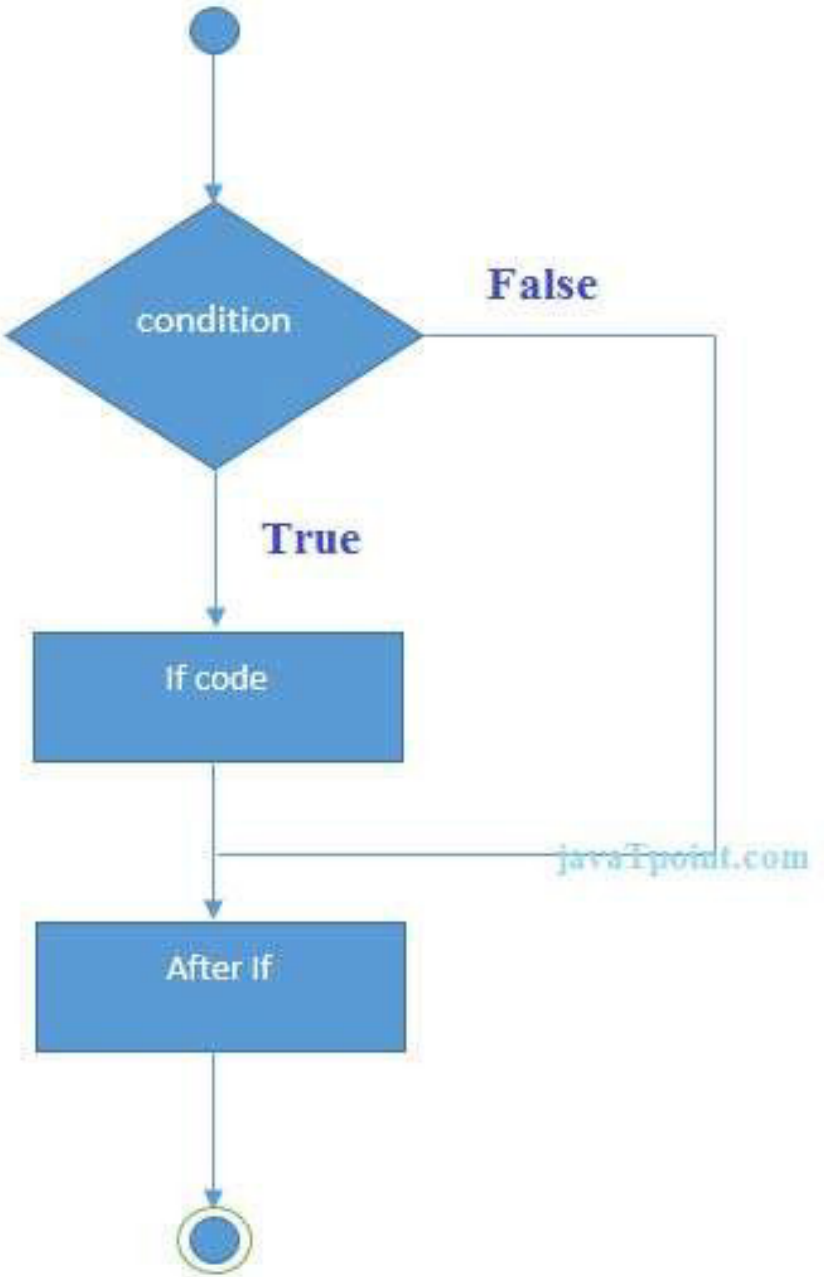
- The Java if statement tests the condition.

  It executes the *if block* if condition is true.

- The syntax as follows

```
if(condition)
{
        //code to be executed
}
```

Control Statements – if statement – flow chart



Control Statements – if statement - example

```
public class IfExample
{
        public static void main(String[] args)
        {
                int age=20;
                if(age>18)
                {
                        System.out.print("Age is  greater than 18");
                }
        }
}
```

Output:

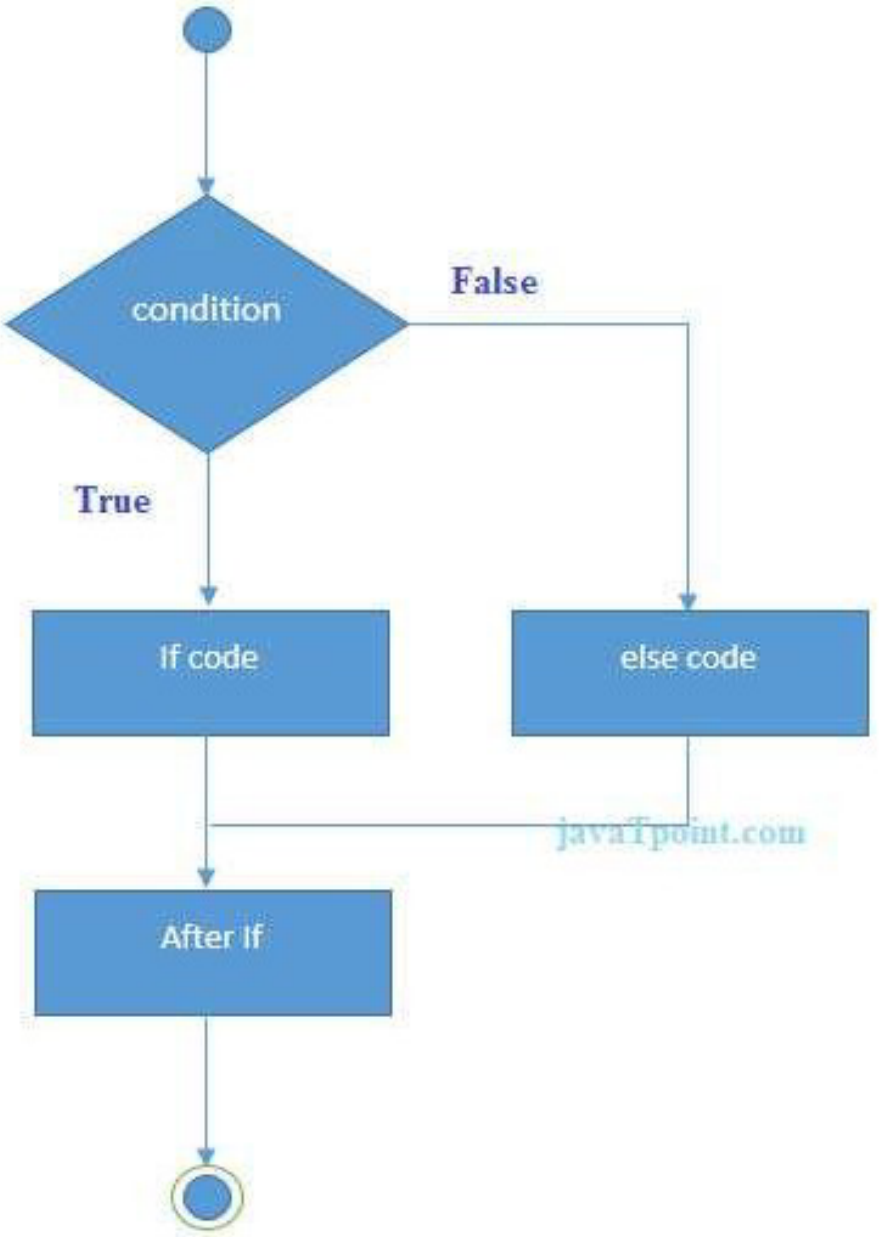
Age is greater than 18

Control Statements – if-else statement

- The Java if-else statement also tests the condition.
- It executes the if block if condition is true otherwise else block is executed.
- Syntax as follow:

```
if(condition)
{
        //code if condition is true
}
else
{
        //code if condition is false
}
```

Control Statements – if-else – flow chart



Control Statements – if-else - Example

```
public class IfElseExample
{
        public static void main(String[] args)
        {
                int number=13;
                if(number%2==0)
                {
                        System.out.println("even
number");
                }
                else
                {
                        System.out.println("odd number");
                }

        }
}
```

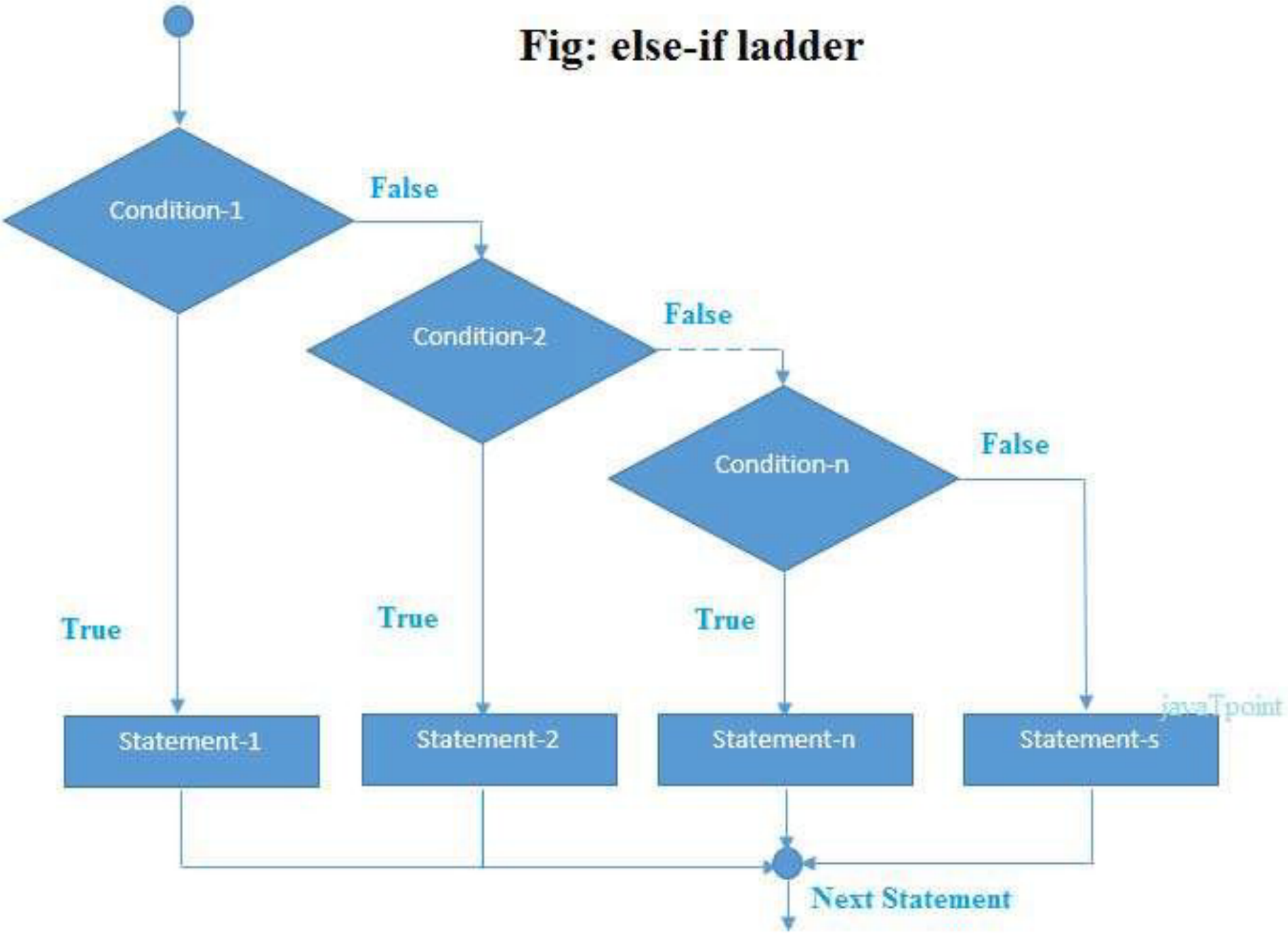
Output:
odd number

Control Statements – if-else-if ladder

- The if-else-if ladder statement executes one condition from multiple statements. The syntax as follows:

```
if(condition1){
        //code to be executed if condition1 is true
}else if(condition2){
        //code to be executed if condition2 is true
}
else if(condition3){
        //code to be executed if condition3 is true
}
...
else{
        //code to be executed if all the conditions are  false
}
```

Control Statements – if-else-if – flow chart



Control Statements – if-else-if Example

```
public class IfElseIfExample
{
        public static void main(String[] args)
        {
                int marks=65;
                if(marks<50)
                {
                        System.out.println("fail");
                }
                else if(marks>=50 && marks<60)
                {
                        System.out.println("D grade");
                }
```

Control Statements – if-else-if Example

```
else if(marks>=60 && marks<70)
{
        System.out.println("C grade");
}
else if(marks>=70 && marks<80)
{
        System.out.println("B grade");
}
else if(marks>=80 && marks<90)
{
        System.out.println("A grade");
}
```

Control Statements – if-else-if Example

```
            else if(marks>=90 && marks<100)
            {
                    System.out.println("A+ grade");
            }
            else
            {
                    System.out.println("Invalid!");
            }
        }
}
```

Output:

C grade

Control Statements: Nested –if:

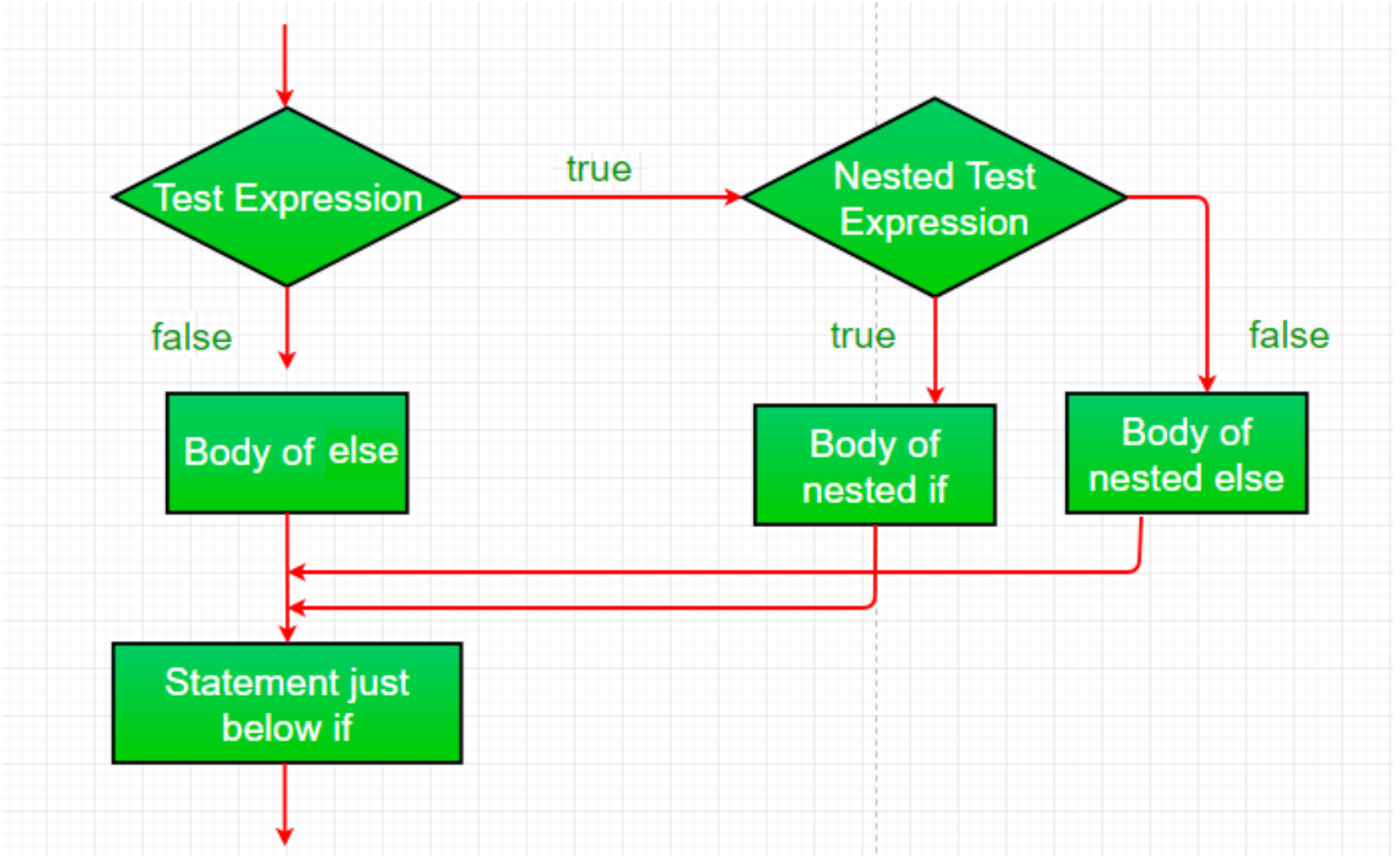
A nested if is an if statement that is the target of another if or else. Nested if statements means an if statement inside an if statement. Yes, java allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.

Control Statements: Nested –if :

Syntax:

```
if (condition1)
{
      // Executes when condition1 is true
   if (condition2)
   { // Executes when condition2 is true
   }
         ……
 }
```

Control Statements: Nested –if- Flow chart



Control Statements: Nested –if- Example

```
class NestedIfDemo
{
    public static void main(String args[])
    {
        int  i = 10;

        if (i == 10)
        {
            // First if statement
            if (i < 15)
                System.out.println("i is smaller than 15");

            // Nested - if statement
            // Will only be executed if statement above
            // it is true
            if (i < 12)
                System.out.println("i is smaller than 12 too");
            else
                System.out.println("i is greater than 15");
        }
    }
}
```

Output:

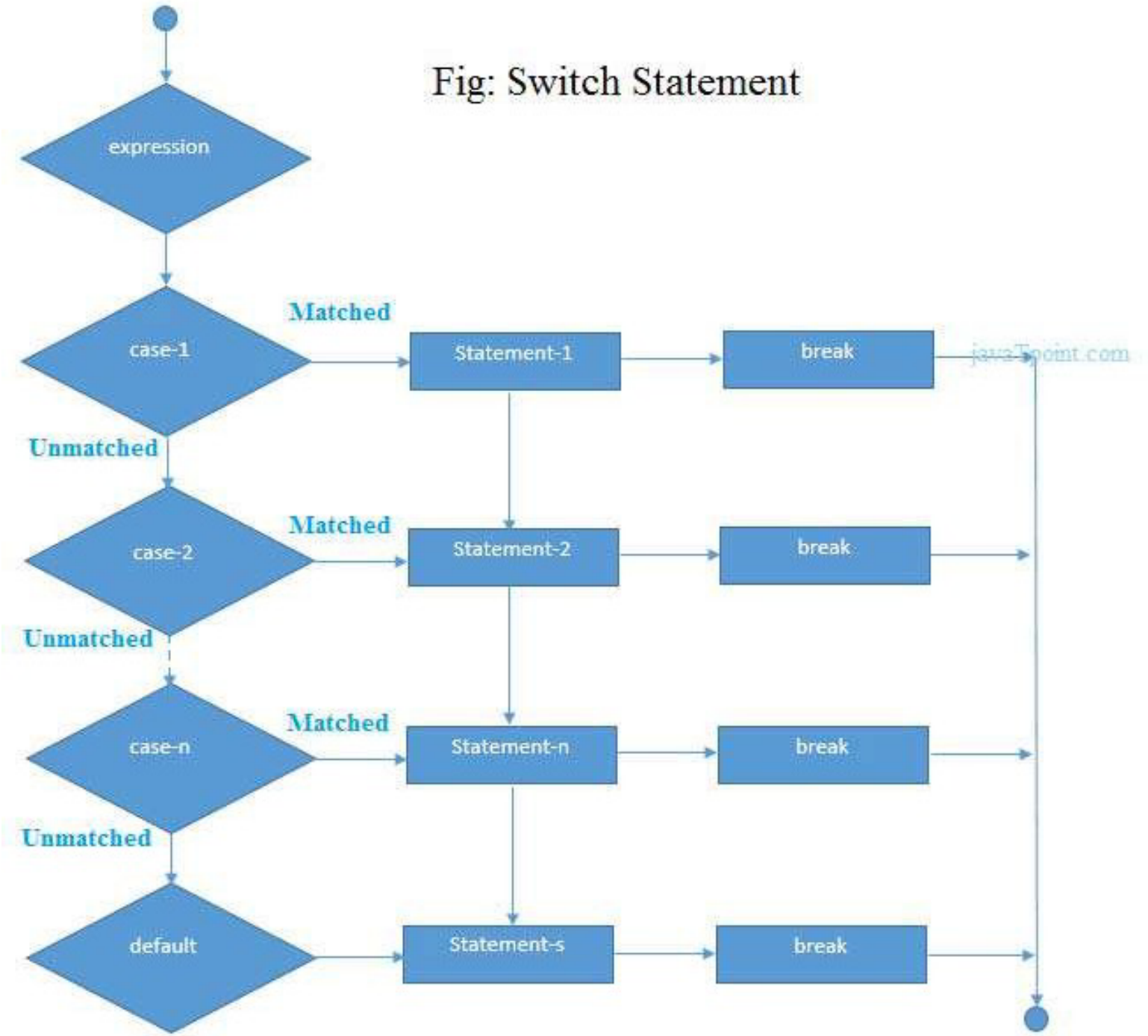
i is smaller than 15
i is smaller than 12 too

Control Statements – switch statement

- The Java *switch statement* executes one statement from multiple conditions. It is like if-else- if ladder statement. The syntax as follows:

```
switch(expression)
{
    case value1:
                    //code to be executed;
                    break; //optional
    case value2:
                    //code to be executed;
                    break; //optional
                    ......
    default:
                    code to be executed if all  cases are not
matched;
}
```

chart


Fig: Switch Statement
expression
case-1
Matched
Statement-1
break
Unmatched
case-2
Matched
Statement-2
break
Unmatched
case-n
Matched
Statement-n
break
Unmatched
default
Statement-s
break
javaTpoint.com

Control Statements – switch – Example

```
public class SwitchExample
{
        public static void main(String[] args)
        {
                int number=20;
                switch(number)
                {
case 10: System.out.println(“Ten"); break;  case 20: System.
out.println(“Twenty"); break;  case 30: System.out.
println(“Thirty"); break;  default: System.out.println("Not in 10,
20 or
30");
                }
        }
}
```

Output

Twenty

Control Statements (Looping or repetitive)

Looping Statements

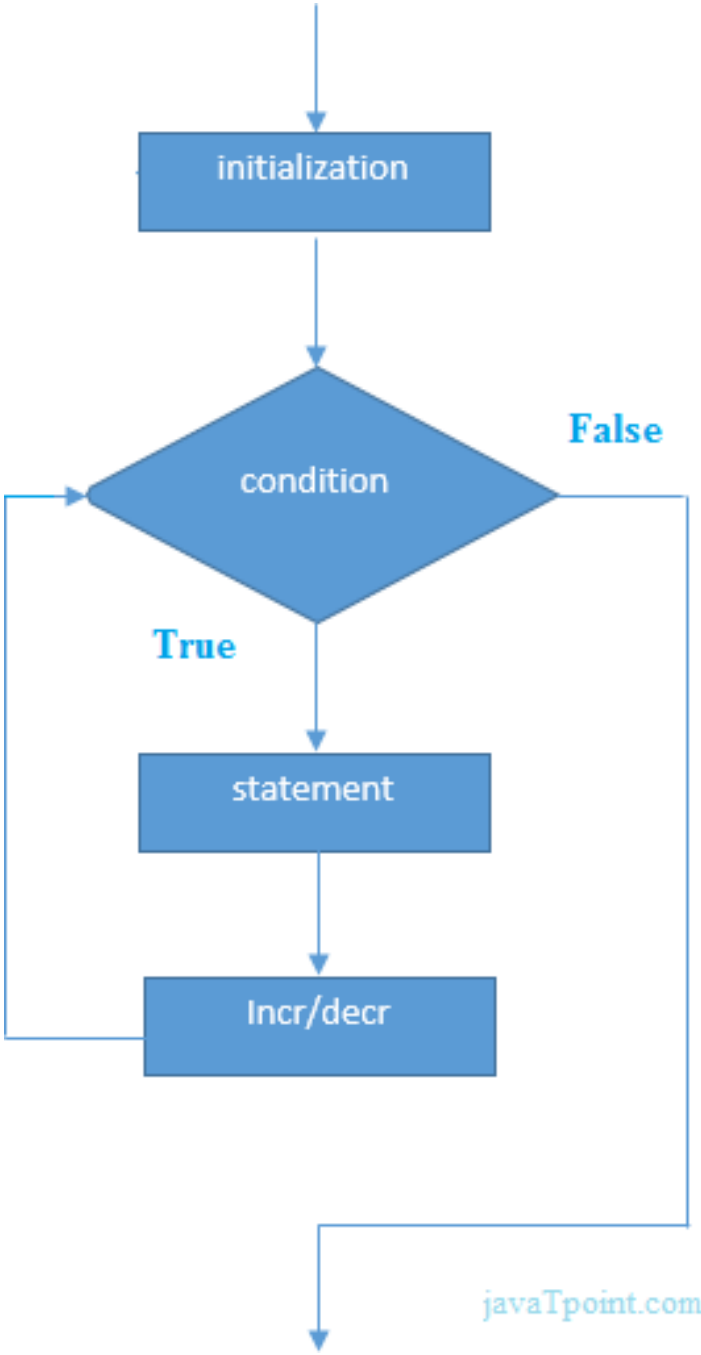
- The Java *for loop* is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop.
- There are three types of for loop in java.
  - Simple For Loop
  - For-each or Enhanced For Loop
  - Labeled For Loop

Looping Statements – for loop

- The simple for loop is same as C/C++. We can initialize variable, check condition and increment/ decrement value. The syntax as follows:

```
for(initialization; condition; incr/decr)
{
        //code to be executed
}
```

Looping Statements – for loop – flow chart



Looping Statements – for loop - Example

```
public class ForExample
{
        public static void main(String[] args)
        {
                for(int i=1;i<=10;i++)
                {
                        System.out.println(i);
                }
        }
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

Looping Statements – for each

- The for-each loop is used to traverse array or collection in java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation.
- It works on elements basis not index. It returns element one by one in the defined variable. The syntax as follows

```
for(Type var:array)
{
        //code to be executed
```

Looping Statements – for each - Example

```
public class ForEachExample
{
        public static void main(String[] args)
        {
                int arr[]={12,23,44,56,78};
                for(int i:arr)
                {
                        System.out.println(i);
                }
        }
}
```

Output:

```
12
23
44
56
78
```

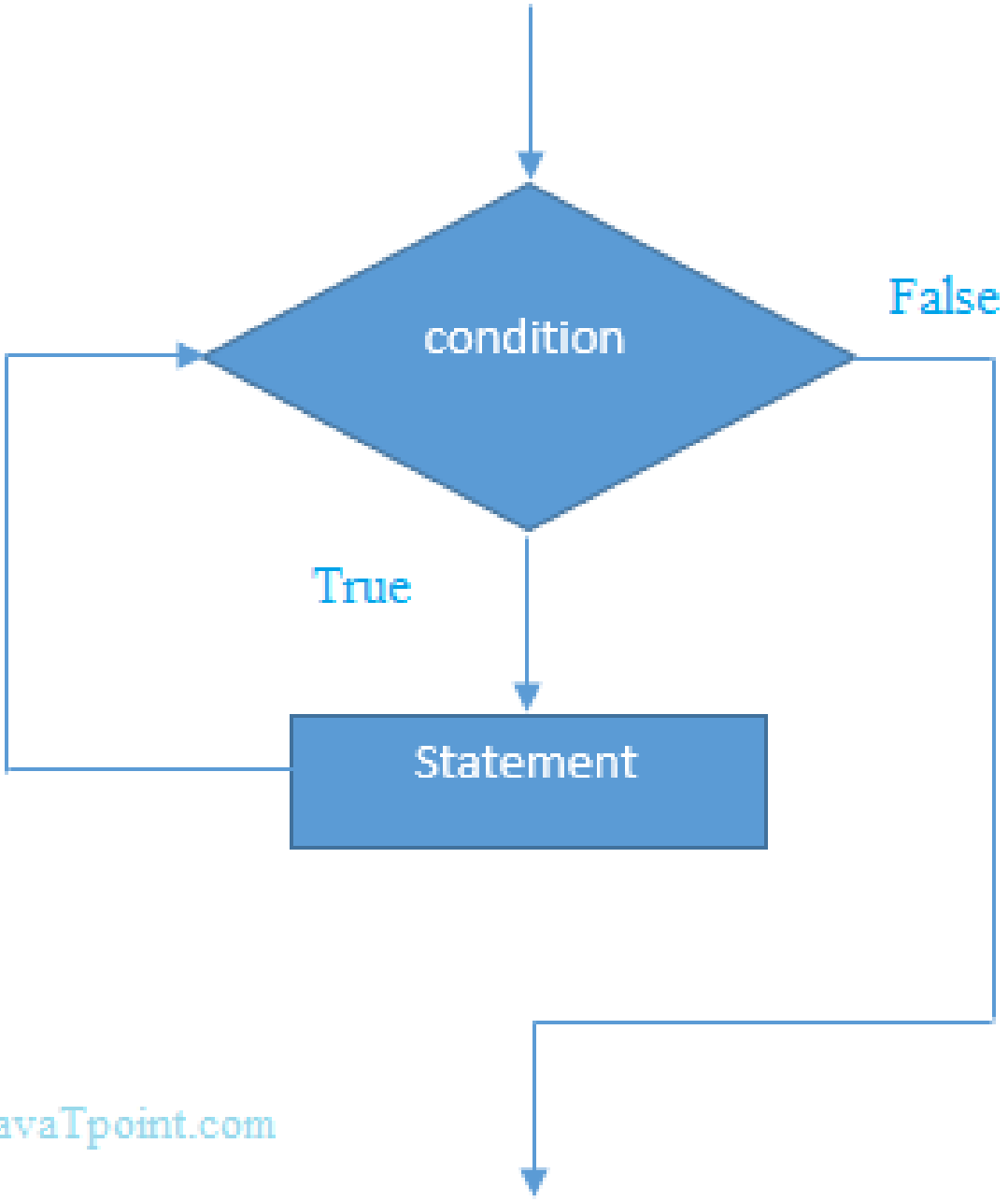
Looping Statements – While loop–Entry controlled

- The Java *while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.

Syntax:

```
while(condition)
{
        //code to be executed
}
```

Looping Statements – While loop–Entry controlled



Example

```
public class WhileExample
{
        public static void main(String[] args)
        {
                int i=1;
                while(i<=10)
                {
                        System.out.println(i);  i++;
                }

        }
}
```

Output:

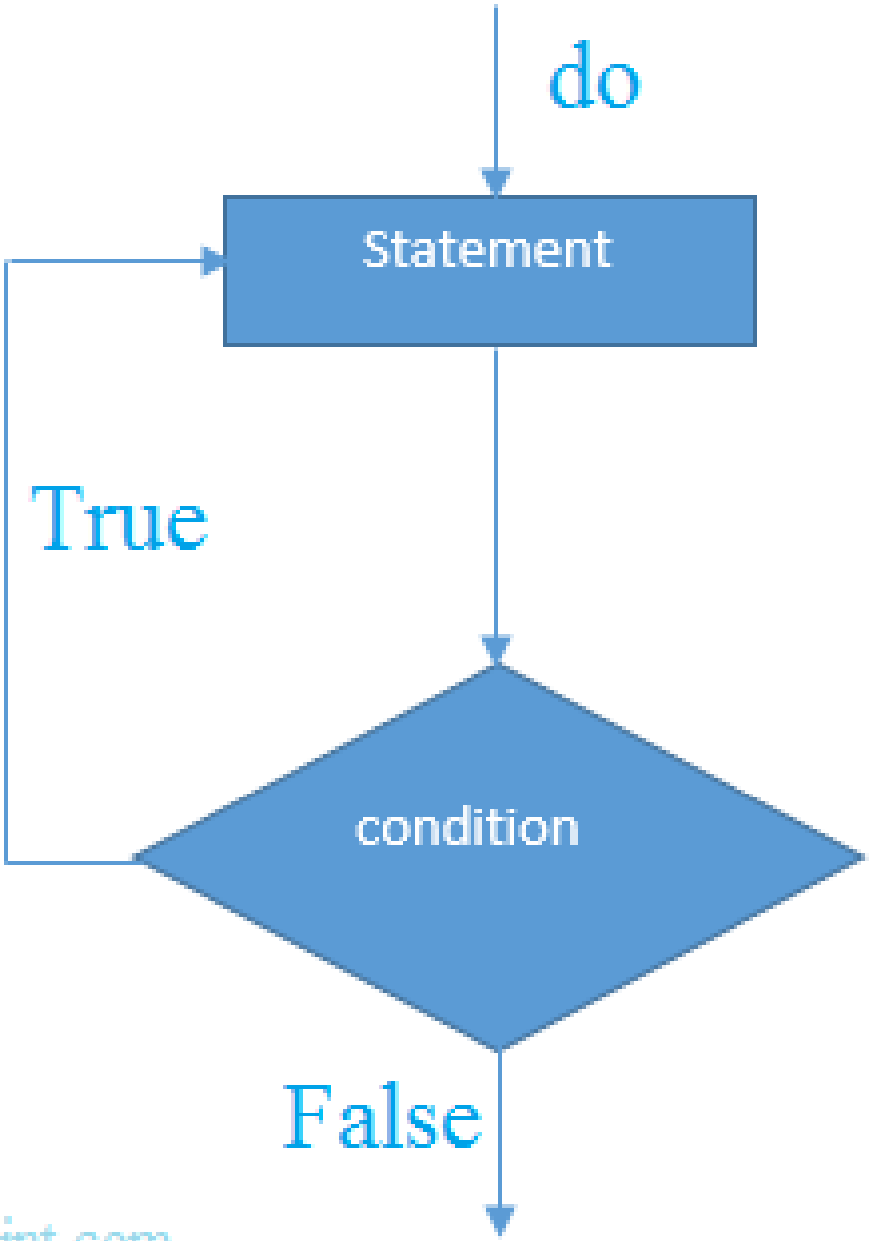
```
1
2
3
4
5
6
7
8
9
10
```

Looping Statements–do-while loop–Exit controlled

- The Java *do-while loop* is used to iterate a part of the program several times.
- If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.
- The Java *do-while loop* is executed at least once because condition is checked after loop body. The syntax as follows:

```
do
{
```

Looping Statements–do-while loop–Exit controlled



Example

```
public class DoWhileExample
{
        public static void main(String[] args)
        {
                int i=1;
                do
                {
                        System.out.println(i);  i++;
                }while(i<=10);

        }
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

Control Statements (Jumping)

Jumping Statements – break statement

- The Java *break* is used to break loop or switch statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop.

Syntax:

```
jump-statement;
break;
```

Jumping Statements – break statement

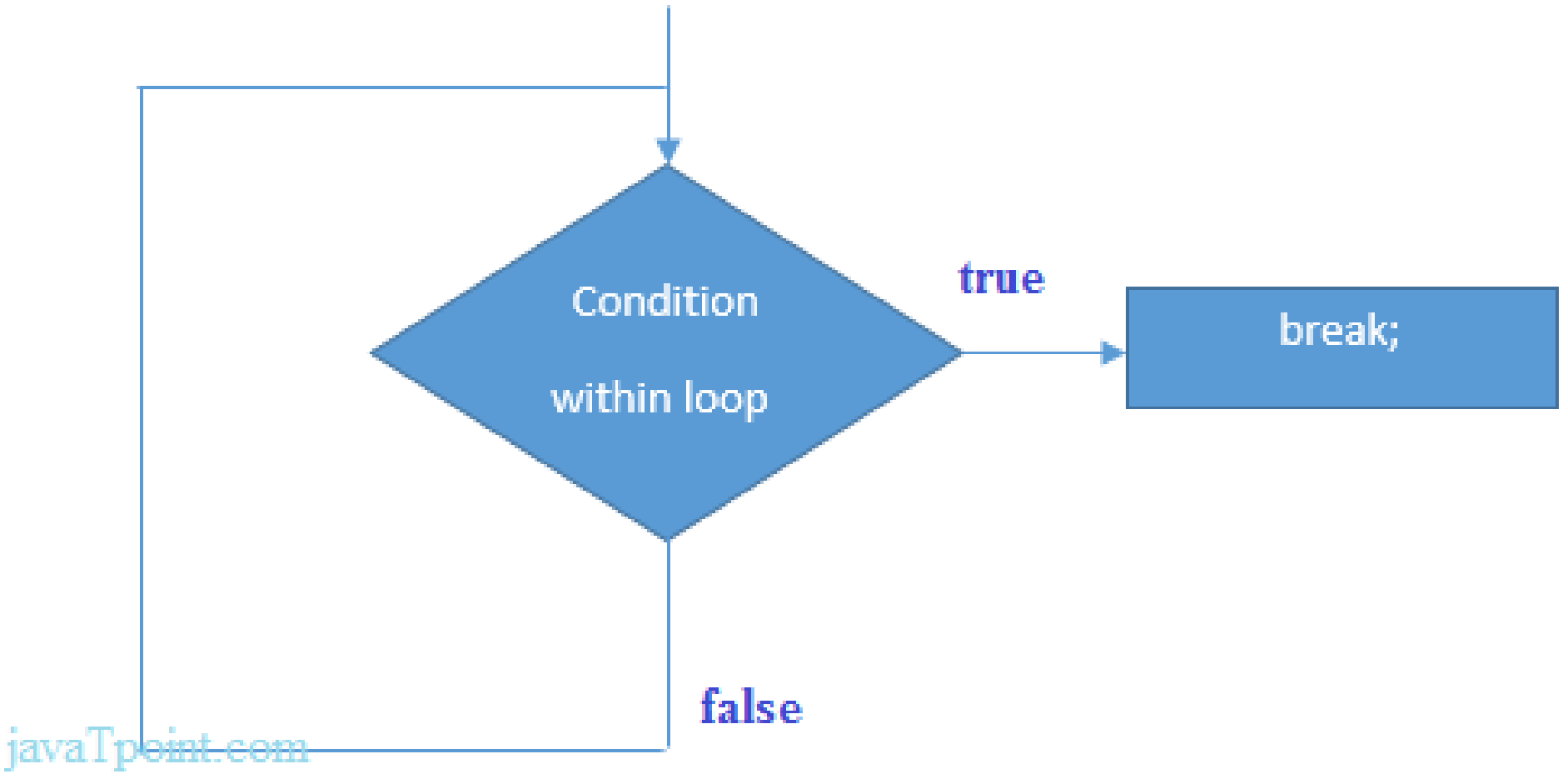


Figure: Flowchart of break statement

Jumping Statements – break statement-example

```
public class BreakExample
{
        public static void main(String[] args)
        {
                for(int i=1;i<=10;i++)
                {
                        if(i==5)
                        {
                                break;
                        }
                System.out.println(i);
                }
        }
}
```

Output:

1
2
3
4

Jumping Statements – continue statement

- The Java *continue statement* is used to continue loop. It continues the current flow of the program and skips the remaining code at specified condition. In case of inner loop, it continues only inner loop.

Syntax:

```
jump-statement;
continue;
```

Jumping Statements – continue Statement

```
public class ContinueExample
{
        public static void main(String[] args)
        {
                for(int i=1;i<=10;i++)
                {
                        if(i==5)
                        {
                                continue;
                        }
                        System.out.println(i);
                }
        }
}
```

Output:

1
2
3
4
6
7
8
9
10

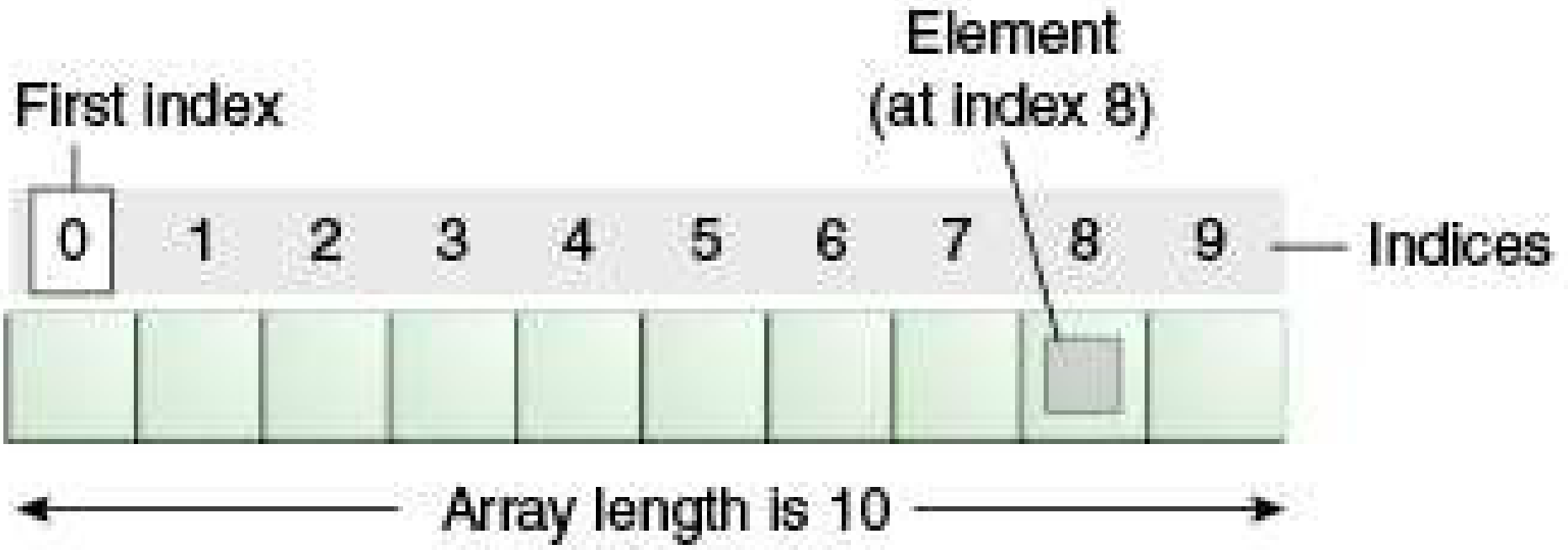
Arrays

Arrays

- Normally, Array is a collection of similar type of elements that have contiguous memory location.
- **Java array** is an object the contains elements of similar data type.
- It is a data structure where we store similar elements.
- We can store only fixed set of elements in a java array.

Arrays

- Array in java is index based, first element of the array is stored at 0 index.



Arrays – Advantage & Disadvantages

- Advantages:
  - **Code Optimization:** It makes the code optimized, we can retrieve or sort the data easily.
  - **Random access:** We can get any data located at any index position.
- Disadvantages:
  - **Size Limit:** We can store only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection

Arrays

One Dimensional Array

One dimensional arrays –

Syntax to declare

dataType[] arr; (or) dataType []arr; (or) dataType arr[];

- Instantiation of an Array in java

arrayRefVar=**new** datatype[size];

Ex. int marks=new int[5];

One dimensional arrays – Example

```
class Testarray {

public static void main(String args[]){

int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization  a[1]=20;

a[2]=70;

a[3]=40;

a[4]=50;

//printing array
for(int i=0;i<a.length;i++)//length is the property of array  System.out.println(a[i]);

}}
```

Output:

```
10
20
70
40
50
```

Declaration, Instantiation, and Initialization

- We can declare, instantiate and initialize the java array together by:

```
int a[]={33,3,4,5};//declaration, instantiation and initialization
```

Passing array to method in Java

- We can pass the java array to method so that we can reuse the same logic on any array.

```
class Testarray2{
static void min(int arr[]){
        int min=arr[0];
        for(int i=1;i<arr.length;i++)
                if(min>arr[i])
                                min=arr[i];
                        System.out.println(min);
}
public static void main(String  args[]){
        int a[]={33,3,4,5};
        min(a);//passing array to method    } }
```

Output:3

Arrays

Two Dimensional Array

Two Dimensional Arrays

- In such case, data is stored in row and column based index (also known as matrix form).
- **Syntax to Declare Multidimensional Array in java**

```
dataType[][] arrayRefVar; (or)
dataType [][]arrayRefVar; (or)
dataType arrayRefVar[][]; (or)
dataType []arrayRefVar[];
```

- Example to instantiate Multidimensional Array in java

Example to initialize two dimensional Array in java

```
arr[0][0]=1;

arr[0][1]=2;

arr[0][2]=3;

arr[1][0]=4;

arr[1][1]=5;

arr[1][2]=6;

arr[2][0]=7;

arr[2][1]=8;

arr[2][2]=9;
```

Two Dimensional Array - Example

```
class Testarray3{
public static void main(String args[]){
//declaring and initializing 2D array
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
//printing 2D array  for(int i=0;i<3;i++){
for(int j=0;j<3;j++){
System.out.print(arr[i][j]+" ");
}
System.out.println();
}
}}
```

```
Output:1  2 3
       2  4 5
       4  4 5
```

Methods

General Form of a Method

Methods

- A method is a self-contained entity that carries out a well-defined task of some kind.
- For example, a method named ***rectangleArea()*** may be defined so as to compute the area of rectangle. The methods in class are its basic operation entities.
- Methods are also called functions, member functions (in class), routine or procedures.
- In java we are calling Methods.

Methods – General Form

- Each method carried out well-defined task.
- Each method has a **name** and a list of **parameters**.
- Any valid identifier can be assigned as the name of a method. However, this names of predefined methods and keywords are not to be used as the names of user-defined methods.
- The syntax of a method is as follows

```
access-specifier return-type method-name (parameter  list)
{
                    statement 1;
                    …
                    …
                    statement n;
```

Methods – General Form

- Here,

1. ***access-specifier*** specifies which other methods can call this method. Most methods shall be declared as public.
2. ***Return-type*** is the type of the value that is returned by the method to its caller. If no value returned by a method, then the keyword ***void*** should be specified in the place of ***type***.

Methods – General Form

- Here,

3. ***Method-name*** is the name assigned to the method by the user. It shall be any valid identifier.
4. ***Parameter-list*** is a list of parameters. This list, always enclosed in parentheses contains parameter names and their types. The values that we want to pass on to the method as input

Methods – General Form

- The parameters in the parameter list should be separated by commas.
- In no input data are to be provided to a method, then the method name should be followed by pair of ***empty*** parentheses.
- The ***body*** of the method contains a set of variables and a set of statements.
- The statements in a method describe the operations that are performed by the method.

Methods – General Form

- The variables that are declared in the method are know as ***local variables***.
- The local variables and the parameters associated with a method. They are removed from the memory, when the control comes out of the method.
- Local variables should be always initialized.
- A method may be instructed to return a value after its execution is over. A special statement namely the ***return*** statement of the following form shall be used for this purpose

return expression;

Methods – General Form - Example

- Here, ***data type*** of the value yielded by the ***expression*** should be same as the ***return-type*** of the method.

  Example,

```
int rectAngleArea()
{
        return length * width;
}
```

- Here the value yielded by the expression **length * width** should be *int*, as the ***return type*** of the method *rectangleArea()* is specified as int.
- Method name is rectAngleArea(), return type is int, no parameters list, inside body only one statement that is

Methods

Method of Invoking a Method

Methods – Invoking a Method

- Suppose we define a method named product() for finding the product of two numbers as shown below:

```
int product( int a, int b)
{
        return a * b;
}
```

- Once we define a method, such as the product() method, we shall invoke it any number of times.
- To use a method, we have to invoke it by specifying its name with the values for its

Methods – Invoking a Method

- For example, we shall invoke the *product()* method, as shown below

```
product(10,6);
```

- When the method *product()* is invoked as specified above, the numbers 10 and 6 will be passed to the method *product()*.
- As a result, the product **a * b** will be evaluated as **60** and this value will be returned by the *product()* method

Methods – Invoking a Method

- The value that is returned by a method shall very well be assigned to a variable and then involved in some computation. For example, consider the following **assignment statement.**

int p = product (10, 6);

- Here the value 60, which is returned by the product() method, will be stored as the value of the variable p. The value that is returned by a method shall also be directly displayed on the screen with the help of a statement of the following form:

Methods – Invoking a Method

- Here the value **60**, which is returned by the *product()* method will be displayed on the screen along with thestring *"Product of two numbers = "* as shown below:

Product of two numbers = 60

Methods

Method Overloading

Methods –Method Overloading

- When two methods in a class have the same name but different parameter lists, those two methods are said to be **overloaded**.

- **Advantage of method overloading?**

  Method overloading **increases the readability of the program**.

- There are two ways to overload the method in java, By changing
  1. Number of arguments
  2. By changing the data type

Methods –Method Overloading Example

```
class Calculation{

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

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

        obj=new Calculation();  obj.sum(10,10,10);



        obj.sum(20,20);

}
}
```

Output:30
40

Methods

Recursion

Methods – Recursion

- A method can invoke another method. It can also invoke itself. Such a method is known as a **recursive** method.
- The definition of a recursive method has the following two parts
  1. The **basis**, which defines the value of a recursive function for the first one or few values of the parameter(s).
  2. The **recurrence relationship**, which expresses

Methods – Recursion

- For example, let us consider the following recursive method *factorial()*.

```
static int factorial (int m)
{
        if( m==1 )
                return 1;
        return m *  factorial ( m – 1);
}
```

Methods – Recursion

- Here the factorial of 1 is 1. This is the **basis**.
- The factorial of m can be found from the expression ***m * factorial (m – 1).*** This is the **recurrence relationship.**
- Suppose we invoke the above recursive method *factorial()* with the argument 3, as shown below.

factorial(3);

- Now, the *factorial()* method will return the following expression:

*3 * factorial(2);*

Methods – Recursion

- So, the next call to the *factorial()* method will be as follows:

factorial(2);

- Now, the *factorial()* method will return the following expression

*2 * factorial(1);*

- So, the next call to the *factorial()* method will be as follows:

factorial(1);

Methods – Recursion

- Since the *factorial()* method is invoked with the argument 1at this stage, this method will return the value 1.
- At this stage, *factorial(2)* will be computed as **2**.
- Then the *factorial(3)* will be computed as **6**.

Random Class

java.util.Random Class

- The **java.util.Random** class instance is used to generate a stream of pseudorandom numbers.

  Following are the important points about Random:

  - The class uses a 48-bit seed, which is modified using a linear congruential formula.
  - The algorithms implemented by class Random use a protected utility method that on each invocation can supply up to 32 pseudorandomly generated bits.

java.util.Random Class Constructors

1. Random()

This creates anewrandom number generator.

2. Random(*long seed)*

This creates anewrandom numbe r generator using a single long seed.

java.util.Random Class Methods

1. **protected int next(int bits)**

This method generates the next pseudorandom number.

2. **boolean nextBoolean()**

This method returns the next pseudorandom, uniformly distributed boolean value from this random number generator's sequence.

3. **void nextBytes(byte[] bytes)**

This method generates random bytes and

java.util.Random Class Methods

4. **double nextDouble()**

This method returns the next pseudorandom, uniformly distributed double value between 0.0 and 1.0 from this random number generator's sequence.

5. **float nextFloat()**

This method returns the next pseudorandom, uniformly distributed float value between 0.0 and 1.0 from this random number generator's sequence.

6. **double nextGaussian()**

This method returns the next pseudorandom, Gaussian ("normally") distributed double value with mean 0.0 and standard deviation 1.0 from this random number generator's sequence

java.util.Random Class Methods

7. int nextInt()

This method returns thenext pseudorandom, uniformly distributed int value from this random number generator's sequence.

8. int nextInt(int n)

This method returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.

java.util.Random Class Methods

9. **long nextLong()**

This method returns the next pseudorandom, uniformly distributed long value from this random number generator's sequence.

10. **void setSeed(long seed)**

This method sets the seed of this random number generator using a single long seed.


Thank you!

METHOD OVERLOADING IN JAVA

By

M. BABY ANUSHA,

ASST.PROF IN CSE DEPT.,

RGUKT,NUZVID

Introduction :

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
- If we have to perform only one operation, having same name of the methods increases the readability of the program.

Continued…

- Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.
- So, we perform method overloading to figure out the program quickly.

Advantage of method overloading :

Method overloading *increases the readability of the program*. There are two ways to overload the method in java.

- By changing number of arguments.
- By changing the data type of arguments.

Method Overloading: changing no. of arguments :

- In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.
- In this example, we are creating static methods so that we don't need to create instance for calling methods.

Example Program :

```
class Adder
{
static int add(int a,int b)
{
   return a+b;
}
static int add(int a,int b,int c)
{
return a+b+c;}
}
```

Example Program :

```
class TestOverloading1
{
  public static void main(String[] args)
  {
    System.out.println(Adder.add(11,11));
    System.out.println(Adder.add(11,11,11));
  }
}
```

Output:

```
22
 33
```

Method Overloading: changing data type of arguments

- In this example, we have created two methods that differs in data type.
- The first add method receives two integer arguments and second add method receives two double arguments.

Example Program :

```
class Adder
{
   static int add(int a, int b)
   {
       return a+b;
    }
   static double add(double a, double b)
   {
      return a+b;
   }
}
```

Example Program :

```
class TestOverloading2
{
        public static void main(String[] args)
        {
            System.out.println(Adder.add(11,11));
            System.out.println(Adder.add(12.3,12.6));
        }
    }
```

Output:

22

24.9

Method Overloading is not possible by changing the return type of method only?

- In java, method overloading is not possible by changing the return type of the method only because of ambiguity.
- Let's see how ambiguity may occur:

Example Program :

```
class Adder
{
   static int add(int a,int b)
   {
      return a+b;
   }
   static double add(int a,int b)
   {
      return a+b;
   }
}
```

Example Program :

```
class TestOverloading3
{
   public static void main(String[] args)
   {
   System.out.println(Adder.add(11,11));//
    ambiguity
   }
}
```

Output:

Compile Time Error: method add(int,int) is already defined in class Adder

Can we overload java main() method?

- Yes, by method overloading. You can have any number of main methods in a class by method overloading.
- But JVM calls main() method which receives string array as arguments only. Let's see the simple example:

Example Program :

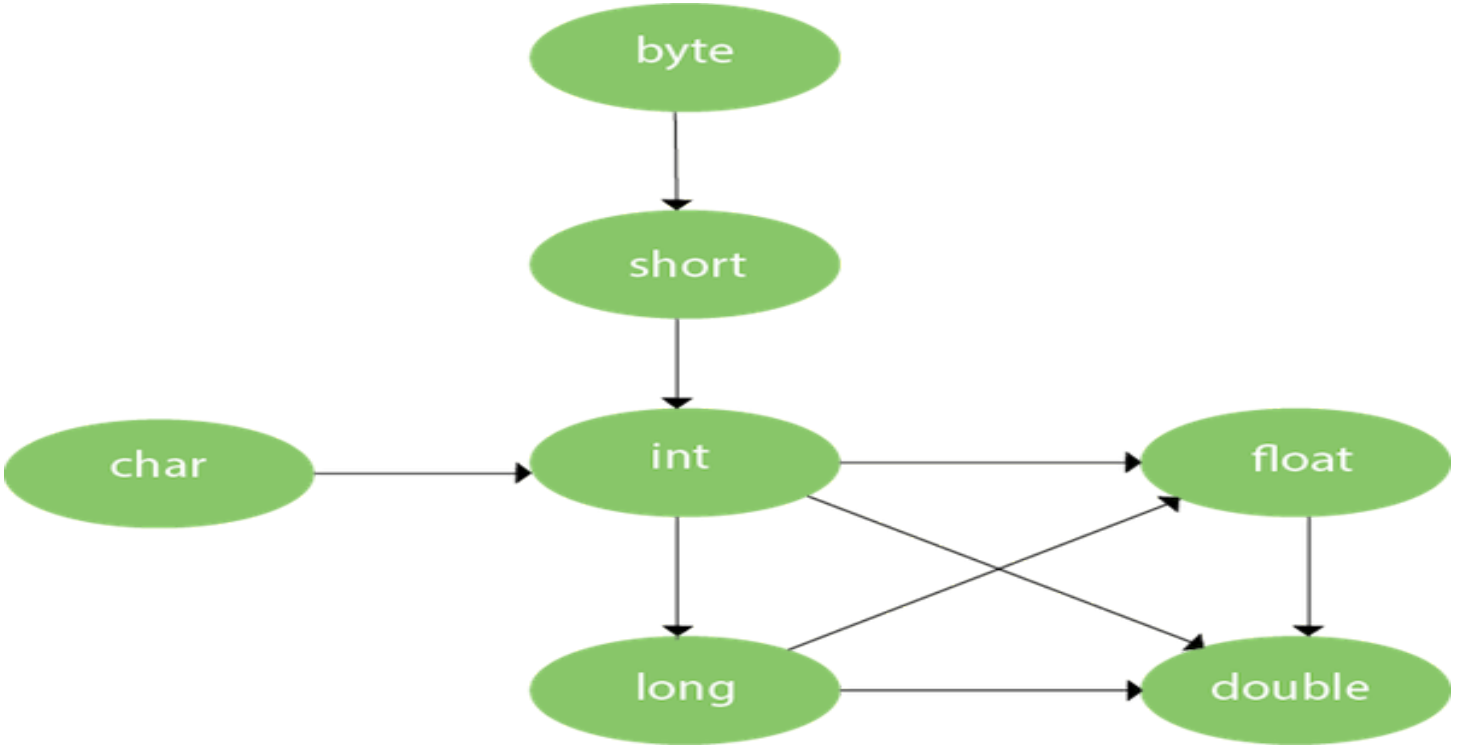
```
class TestOverloading4
{
public static void main(String[] args)
{System.out.println("main with String[]");}
public static void main(String args)
{System.out.println("main with String");}
public static void main()
{System.out.println("main without args");}
}
```

Output:

main with String[]

Method Overloading and Type Promotion :

- One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:



Method Overloading and Type Promotion :

- As displayed in the above diagram, byte can be promoted to short, int, long, float or double.
- The short datatype can be promoted to int, long, float or double.
- The char datatype can be promoted to int,long, float or double and so on.

Example of Method Overloading with TypePromotion :

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

Example of Method Overloading with TypePromotion :

```
 public static void main(String args[])
{
  OverloadingCalculation1 obj=new OverloadingCalculation1();

obj.sum(20,20);//
    now second int literal will be promoted to long

obj.sum(20,20,20);
    }
}
```

Output:

40

60



shutterstock.com · 567687052

Class and Objects in Java

By
M. BABY ANUSHA,
ASST.PROF IN CSE DEPT.,
RGUKT,NUZVID

OBJECTIVES

- Classes
- Objects
- Methods
- Constructors

CLASS IN JAVA

```
class
One
 {
   }
```

```
class
College
 {
    }
```

```
class
Book
{
  }
```

```
class AnyThing
{
      }
```

CLASS DEFINITION

A class contains a name, several variable declarations (instance variables) and several method declarations. All are called members of the class.

General form of a class:

```
class    classname   {
    type    instance-variable-1;
    …
    type    instance-variable-n;

    type    method-name-1(parameter-list)          {   … }
    type    method-name-2(parameter-list)          {   … }
    …
    type    method-name-m(parameter-list)          {   … }
}
```

EXAMPLE: CLASS

A class with three variable members:

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

A new Box object is created and a new value assigned to its width variable:

```
Box  myBox  =  new  Box();
myBox.width    = 100;
```

EXAMPLE: CLASS USAGE

```
class   BoxDemo{
    public  static  void main(String args[]) {
        Box  mybox  =  new Box();
        double vol;

        mybox.width   =  10;
        mybox.height  =  20;
        mybox.depth    = 15;

        vol   =  mybox.width   *  mybox.height  *  mybox.depth;
        System.out.println("
        Volume       is                          depth; "   +  vol);
    }
}
```

COMPILATION AND EXECUTION

Place the Box class definitions in file Box.java:

```
class   Box  {   … }
```

Place the BoxDemo class definitions in file BoxDemo.java:

```
class   BoxDemo{
    public  static  void main(…)  {   … }
}
```

Compilation and execution:

```
>  javac   BoxDemo.java
>  java  BoxDemo
```

VARIABLE INDEPENDENCE 1

Each object has its own copy of the instance variables: changing the variables of one object has no effect on the variables of another object.

Consider this example:

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

        mybox1.width     =  10;
        mybox1.           =  20;
        height           = 15;
        mybox1.depth
```

VARIABLE INDEPENDENCE 2

```
        mybox2.width =  3;
        mybox2.height=  6;
        mybox2.depth = 9;

        vol   =  mybox1.width *  mybox1.depth;
        System.out.println("mybox1.height is " + vol);
        Volume

        vol   =  mybox2.width *  mybox2.depth;
    }   System.out.println("mybox2.height is " + vol);
}       Volume
```

What are the printed volumes of both boxes?

OBJECTS

```
class Book
{

}


class JavaBook
{
  public ststic void main(String args[])
        {
          Book b=new Book();
        }
}
```

```
class College
      {

            }


class Kits
{
Public static void
   main(String args[])
{
    College c=new College();
}}
```

b and c are called as objects

DECLARING OBJECTS

Obtaining objects of a class is a two-stage process:

1) Declare a variable of the class type: Box myBox;

   The value of myBox is a reference to an object, if one exists, or null. At this moment, the value of myBox is null.

2) Acquire an actual, physical copy of an object and assign its address to the variable. How to do this?

OPERATOR NEW

Allocates memory for a Box object and returns its address:

```
Box myBox = new Box();
```

The address is then stored in the myBox reference variable.

Box() is a class constructor - a class may declare its own constructor or rely on the default constructor provided by the Java environment.

MEMORY ALLOCATION

Memory is allocated for objects dynamically.

This has both advantages and disadvantages:

1) as many objects are created as needed
2) allocation is uncertain – memory may be insufficient

Variables of simple types do not require new: `int n = 1;`

In the interest of efficiency, Java does not implement simple types as objects. Variables of simple types hold values, not references.

ASSIGNING REFERENCE VARIABLES

Assignment copies address, not the actual value:

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

Both variables point to the same object.

Variables are not in any way connected. After

```
b1  = null;
```

b2 still refers to the original object.

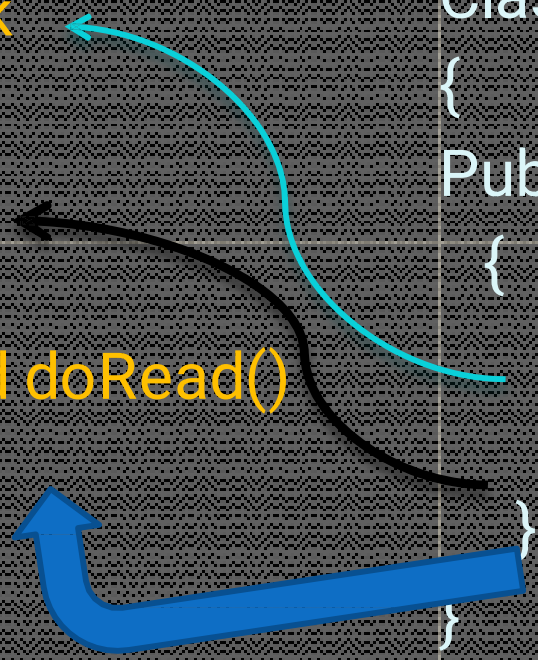
METHODS

```
class Book
{
Int pages;

Public void doRead()
{
}

  }
```

```
Class JavaBook
{
Public static void main(String args[])
  {
    Book b=new Book();  b.
    pages=200;  b.doRead();
  }
}
```



METHODS

General form of a method definition:

```
type name(parameter-list)  {
     … return   value;   …
}
```

Components:

1) type - type of values returned by the method. If a method does not return any value, its return type must be void.
2) name is the name of the method
3) parameter-list is a sequence of type-identifier lists separated by commas
4) return value indicates what value is returned by the method.

EXAMPLE: METHOD 1

Classes declare methods to hide their internal data structures, as well as for their own internal use:

Within a class, we can refer directly to its member variables:

```
class    Box  {
     double width,        height, depth;
     void volume()       {
          System.out.print("                    is ");
          Volume  System.out.                  *  height      *  depth);
     }     println(width
}
```

EXAMPLE: METHOD 2

When an instance variable is accessed by code that is not part of the class in which that variable is defined, access must be done through an object:

```
Class  BoxDemo3 {
public static  void main(String   args[]) {
            Box  mybox1       new Box();
            = Box  mybox2     new Box();
            = mybox1.         =  10;        mybox2.width =  3;
            width  mybox1.     =  20;       mybox2.              =  6;
            height  mybox1.  = 15;          height               = 9;
            depth                           mybox2.
            mybox1.volume();                depth
            mybox2.volume();
        }
    }
```

VALUE-RETURNING METHOD 1

The type of an expression returning value from a method must agree with the return type of this method:

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

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

VALUE-RETURNING METHOD 2

```
class   BoxDemo4  {
    public  static  void main(String args[]) {
        Box  mybox1       new Box();
        =                 new Box();
        Box mybox2
        =
        double vol; = mybox1.
        width   =  10;
        mybox2.width  =  3;
        mybox1.height=  20;
        mybox2.height=  6;
        mybox1.depth = 15;
        mybox2.depth = 9;
```

VALUE-RETURNING METHOD 3

The type of a variable assigned the value returned by a method must agree with the return type of this method:

```
        vol   =  mybox1.volume(); System.out.println("Volume  is "   +  vol);
        vol   = mybox2.volume(); System.out.println("Volume   is "   +  vol);
    }
}
```

PARAMETERIZED METHOD

Parameters increase generality and applicability of a method:

1) method without parameters

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

2) method with parameters

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

Parameter: a variable receiving value at the time the method is invoked.

Argument: a value passed to the method when it is invoked.

EXAMPLE: PARAMETERIZED METHOD 1

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

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

    void setDim(double w, double h, double d) {
        width = w; height = h; depth = d;
    }
}
```

EXAMPLE: PARAMETERIZED METHOD 2

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

        mybox1.setDim(10,      20,   15);
        mybox2.setDim(3,      6, 9);

        vol   =  mybox1.volume();
        System.out.println("                    is "   +  vol);
        Volume  vol  =  mybox2.
        volume();  System.out.                  is "   +  vol);
    }     println("Volume
}
```

CONSTRUCTOR

```
class One
  {
  One()
{
//Initialization
}
    }
```

```
class
College
    {

          }
```

```
class Book
{
  Book()
{
// Some Initialization
}
  }
```

```
class AnyThing
{

        }
```

CONSTRUCTOR

A constructor initializes the instance variables of an object.

It is called immediately after the object is created but before the new operator completes.

1) it is syntactically similar to a method:
2) it has the same name as the name of its class
3) it is written without return type; the default return type of a class constructor is the same class

When the class has no constructor, the default constructor automatically initializes all its instance variables with zero.

EXAMPLE: CONSTRUCTOR 1

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

    Box()  {
        System.out.println("                    Box");
        Constructing                    depth  =  10;
    }   width  = 10; height  =  10;

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

EXAMPLE: CONSTRUCTOR 2

```
class   BoxDemo6  {
    public  static  void main(String args[]) {
        Box  mybox1       new Box();
        =                 new Box();
        Box mybox2
        double vol;
        =

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

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

PARAMETERIZED CONSTRUCTOR 1

So far, all boxes have the same dimensions.

We need a constructor able to create boxes with different dimensions:

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

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

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

PARAMETERIZED CONSTRUCTOR 2

```
class   BoxDemo7  {
    public  static  void main(String              args[]) {
        Box  mybox1       new  Box(10,    20,     15);
        =                 new  Box(3,  6, 9);
        Box mybox2
        double vol;
        =

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

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

Thank you

History of Java:

Before starting to learn Java, let us plunge into its history and see how the language originated. In 1990, Sun Microsystems Inc. (US) was conceived a project to develop software for consumer electronic devices that could be controlled by a remote. This project was called *Stealth Project* but later its name was changed to *Green Project*.

In January of 1991, Bill Joy, **James Gosling**, Mike Sheradin, Patrick Naughton, and several others met in Aspen, Colorado to discuss this' project. Mike Sheradin was to focus on business development; Patrick Naughton was to begin work on the graphics system; and James Gosling was to identify the proper programming language for the project. Gosling thought C and C++ could be used to develop the project.

But the problem he faced with them is that they were system dependent languages and hence could not be used on various processors, which the electronic devices might use. So he started developing a new language, which was completely system independent. This language was initially called *Oak.* Since this name was registered by some other company, later it was changed to *Java.*

Why the name Java? James Gosling and his team members were consuming a lot of tea while developing this language. They felt that they were able to develop a better language because of the good quality tea they had consumed. So the tea also had its own role in developing this language and hence, they fixed the name for the language as *Java.* Thus, the symbol for *Java* is tea cup and saucer.



- *James Gosling was born on **May 19, 1955**.*
- *James Gosling received a Bachelor of Science from the **University of Calgary** and his M.A. and Ph.D. from **Carnegie Mellon University**. He built a multi-processor version of Unix for a 16-way computer system while at Carnegie Mellon University, before joining Sun Microsystems.*

By September of 1994, Naughton and Jonathan Payne started writing *WebRunner-a* Java-based Web browser, which was later renamed as *HotJava.* By October 1994, HotJava was stable and was demonstrated to Sun executives. HotJava was the fIrst browser, having the capabilities of executing *applets,* which are programs designed to run dynamically on Internet. This time, Java's potential in the context of the World Wide Web was recognized.

Sun formally announced Java and HotJava at SunWorld conference in 1995. Soon after, Netscape Inc. announced that it would incorporate Java support in its browser Netscape Navigator. Later, Microsoft also announced that they would support Java in their Internet Explorer Web browser, further solidifying Java's role in the World Wide Web. On **January 23rd 1996, JDK 1.0** version was released**.** Today more than 4 million developers use Java and more than 1.7 5 billion devices run Java. Thus, Java pervaded the world.

Features of Java:

- **Simple:** Java is a simple programming language. Rather than saying that this is the feature of Java, we can say that this is the design aim of Java. JavaSoft (the team who developed Java is called with this name) people maintained the same syntax of C and C++ in Java, so that a programmer who knows C or C++ will find Java already familiar.

- **Object-oriented:** Java is an object oriented programming language. This means Java programs use objects and classes. What is an object? An object is anything that really exists in the world and can be distinguished from others.

- **Distributed:** Information is distributed on various computers on a network. Using Java, we can write programs, which capture information and distribute it to the clients. This is possible because Java can handle the protocols like TCP/IP and UDP.
- **Robust:** Robust means *strong.* Java programs are strong and they don't crash easily like a C or C++ program. There are two reasons -for this. Firstly, Java has got excellent inbuilt exception handling features.
- **Secure:** Security problems like eavesdropping, tampering, impersonation, and virus threats can be eliminated or minimized by using Java on Internet.
- **System independence:** Java's byte code is not machine dependent. It can be run on any machine with any processor and any operating system.
- **Portability:** If a program yields the same result on every machine, then that program is called *portable.* Java programs are portable. This is the result of Java's *System independence* nature.
- **Interpreted:** Java programs are compiled to generate the byte code. This byte code can be downloaded and interpreted by the interpreter in JVM. If we take any other language, only an interpreter or a compiler is used to execute the programs. But in Java, we use .both compiler and interpreter for the execution.
- **High Performance:** The-problem with interpreter inside the JVM is that it is slow. Because of this, Java programs used to run slow. To overcome this problem, along with the interpreter, JavaSoft people have introduced JIT (Just In Time) compiler, which enhances the speed of execution. So now in JVM, both interpreter and JIT compiler work together to run the program.
- **Multithreaded:** A thread represents an individual process to execute a group of statements. JVM uses several threads to execute different blocks of code. Creating multiple threads is called 'multithreaded'.
- **Scalability:** Java platform can be implemented on a wide range of computers with varying levels of resources-from embedded devices to mainframe computers. This is possible because Java is compact and platform independent.
- **Dynamic:** Before the development of Java, only static text used to be displayed-in the browser. But when James Gosling demonstrated an animated atomic molecule where the rays are moving and stretching, the viewers are dumbstruck. This animation was done using an *applet* program, which are the dynamically interacting programs on Internet.

Java Virtual Machine:

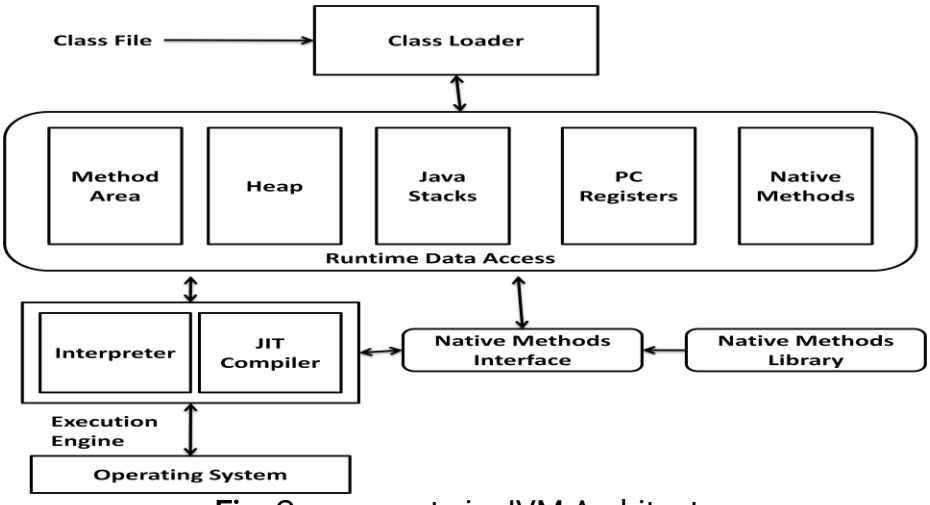


Fig: Components in JVM Architecture

First of all, the .java program is converted into a .class file consisting of byte code instructions by the java compiler. Remember, this java compiler is outside the JVM: Now this
.class file is given to the JVM. In JVM, there is a module (or program) called *class loader sub system,* which performs the following functions:

- First of all, it loads the .class file into memory.
- Then it verifies whether all byte code instructions are proper or not. If it finds any instruction suspicious, the execution is rejected immediately.
- If the byte instructions are proper, then it allocates necessary memory to execute the program.

This memory is divided into 5 parts, called *run time data areas,* which contain the data and results while running the program. These areas are as follows:

- **Method area:** Method area is the memory block, which stores the class code, code of the variables, and code of the methods in the Java program. (Method means functions written in a class)
- **Heap:** This is the area where objects are created. Whenever JVM loads a class, a method and a heap area are immediately created in it.
- **Java Stacks:** Method code is stored on Method area. But while running a method, it needs some more memory to store the data and results. This memory is allotted on Java stacks.
- **PC (Program Counter) registers:** These are the registers (memory areas), which contain memory address of the instructions of the methods. If there are 3 methods, 3 PC registers will be used to track the instructions of the methods.
- **Native method stacks:** Java methods are executed on Java stacks. Similarly, native methods (for example *C/C++* functions) are executed on Native method stacks. To execute the native methods, generally native method libraries (for example *C/C++* header files) are required. These header files are located and connected to JVM by a program, called *Native method interface.*

Execution engine contains interpreter and JIT (Just In Time) compiler, which are responsible for converting the byte code instructions into machine code so that the processor will execute them. Most of the JVM implementations use both the interpreter and JIT compiler simultaneously to convert the byte code. This technique is also called *adaptive optimizer.* Generally, any language (like C/C++, Fortran, COBOL, etc.) will use either an interpreter or a compiler to translate the source code into a machine code. But in JVM, we got interpreter and JIT compiler both working at the same time on byte code to translate it into machine code.

Java Program Structure:

- Documentation Section
- Package Statement
- Import Statements
- Interface Statement
- Class Definition
- Main Method Class
    - o Main Method Definition

Documentation Section	It comprises of a comment line which gives the names program, the programmer's name and some other brief details. In addition to the 2 other styles of comments i.e. /**/ and //, Java also provides another style of comment i.e. /**….*/
Package statement	The first statement allowed in Java file is the Package statement which is used to declare a package name and it informs the compiler that the classes defined within the program belong to this package. package package_name;
Import statements	The next is the number of import statements, which is equivalent to #include statement in C++. Example: import calc.add;
Interface statement	Interfaces are like class that includes a group of method declarations. This is an optional section and can be used only when programmers want to implement multiple inheritance(s) within a program.
Class Definition	A Java program may contain multiple class definitions. Classes are the main and important elements of any Java program. These classes are used to plot the objects of real world problem.
Main Method Class	Since every Java stand alone program requires a main method as the starting point of the program. This class is essentially a part of Java program. A simple Java program contains only this part of the program.

Sample Code of Java "Hello Java"
Program Example:

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

Program Output:

Hello Java

Comments:

- **Single line comments:** These comments are for marking a SL'1gle line as a comment. These comments start with double slash symbol / / and after this, whatever is written till the end of the line is taken as a comment. For example,

```
// This is single line comment
```

- **Multi-line comments:** These comments are used for representing several lines as comments. These comments start with / * and end with * /. In between / * and * /, whatever is written is treated as a comment. For example,

```
/* This is Multi line
comment, Line 2 is here
Line 3 is here */
```

Importing Classes:

In C/C++ compiler to include the header file <stdio.h>. What is a header file? A *header file* is a file, which contains the code of functions that will be needed in a program. In other words, to be able to use any function in a program, we must first include the header file containing that function's code in our program. For example, <stdio.h> is the header me that contains functions, like printf (), scanf (), puts (), gets (), etc. So if we want to use any of these functions, we should include this header file in our *C/C++* program.

A similar but a more efficient mechanism, available in case of Java, is that of importing classes. First, we should decide which classes are 'needed in our program. Generally, programmers are interested in two things:

- Using the classes by creating objects in them.
- Using the methods (functions) of the classes.

In Java, methods are available in classes or interfaces. What is an interface? An *interface* is similar to a class that contains some methods. Of course, there is a lot of difference between an interface and a class, which we will discuss later. The main point to be kept in mind, at this stage, is that a class or an interface contains methods. A group of classes and interfaces are contained in a package. A *package* is a kind of directory that contains a group of related Classes and interfaces and Java has several such packages in its library,

Java library
|
Packages
|
Classes and interfaces
|
Method
s

In our first Java program, we are using two classes namely, System and String. These classes belong to a package called java .lang (here lang represents language). So these two classes must be imported into our program, as shown below:

```
import java.lang.System;
import java.lang.String;
```

Whenever we want to import several classes of the same package, we need not write several import statements, as shown above; instead, we can write a single

```
import java.lang.*;
```

statement as:

Here, * means all the classes and interfaces of that package, i.e. java.lang, are imported made available) into our program. In import statement, the package name that we have written acts like a reference to the JVM to search for the classes there. JVM will not copy any code from the classes or packages. On the other hand, when a class name or a

method name is used, JVM goes to the Java library, executes the code there, comes back, and substitutes the result in that place of the program. Thus, the Java program size will not be increased.

After importing the classes into the program, the next step is to write a class, Since Java is purely an object-oriented programming language and we cannot write a Java program without having at least one class or object. So, it is mandatory that every Java program should have at least one class in it, How to write a class? We should use class keyword for this purpose and then write the class name.

```
class
  Hello{ sta
  tements;
```

A class code starts with a { and ends with a }. We know that a class or an object contains variables and methods (functions), 'So we can create any number of variables and methods inside the class. This is our first program, so we will create only one method, i.e. compulsory, main () method.

The main method is written as follows:

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

- Here **args[ ]** is the array name and it is of String type. This means that it can store a group of strings. Remember, this array can also store a group of numbers but in the form of strings only. The values passed to main( ) method are called *arguments.* These arguments are stored into args[ ] array, so the name args[ ] is generally used for it.
- If a method is not meant to return any value, then we should write *void* before that method's name. ***void*** means *no value.* main() method does not return any value, so void should be written before that method's name.
- ***static*** methods are the methods, which can be called and executed without creating the objects. Since we want to call main()method without using an object, we should declare main( ) method as static. Then, how is the main( ) method called and executed? The answer is by using the classname.methodname(). JVM calls main() method using its class name as Hello.main() at the time of running the program.
- JVM is a program written by *JavaSoft* people (Java development team) and main() is the method written by us. Since, main() method should be available to the JVM, it should be declared as ***public***. If we don't declare main() method as public, then. It doesn't make itself available to JVM and JVM cannot execute it:

So, the main()method should always be written as shown here:

public static void main(String args[ ])

If at all we want to make any changes, we can interchange public and static and write it as follows:

static public void main(String args[])

Or, we can use a different name for the string type array and write it as:

public static void main(String x[])

Q) When main() method is written without String args[args]:

public static void main ()

Ans: The code will compile but JVM cannot run the code because it cannot recognize the main () method as the method from where it should start execution of the Java program. Remember JVM always looks for

main () method with string type array as parameter.

print method in java:

In Java, print() method is used to display something on the monitor. So, we can write it as:

print("Welcome to java");

But this is not the correct way of calling a method in Java. A method should be called by using objectname .methodname( ). So, to call print() method, we should create an object to the class to which print() method belongs. print() method belongs to PrintStream class. So, we should call print() method by creating an object to PrintStream class as:

PrintStream obj.print("Welcome to java");

But as it is not possible to create the object to PrintStream class directly, an alternative is given to us, i.e. ***System.out***. Here, ***System*** is the class name and ***out*** is a static variable in *System* class. *Out* is called a field in *System* class. When we call this field, a PrintStream class object will be created internally. So, we can call the print() method as shown below:

System.out.print("Welcome to java");

System.out gives the PrintStream class object. This object, by default, represents the standard output device, i.e. the monitor. So, the string "Welcome to Java" will be sent to the monitor.

```
class Hello
{
  public static void main(String[] args)
  {
        System.out.print("Welcome to java");
  }
}
```

Save the above program in a text editor like *Notepad* with the name Hello.java.
Now, go to System prompt and compile it using *javac* compiler as:

```
javac Hello.java
```

The compiler generates a file called Hello.class that contains byte code instructions. This file is executed by calling the JVM as:

```
java Hello
```

Then, we can see the result.

Note: In fact, JVM is written in C language.

Output:

```
C:\> javac Hello.java
C:\> java Hello
Welcome to Java
```

Program: Write a program to find the sum of addition of two numbers.

```
class Add
{
        public static void main(String args[])
        {
                int x, y;
                x = 27;
                y = 35;
                int  z = x + y;
                System.out.print("The Sum is
                "+z);
        }
```

Variables:

A variable provides us with named storage that our programs can manipulate. Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

You must declare all variables before they can be used. Following is the basic form of a variable declaration –

```
DataType variable [ = value][, variable [ = value] ...] ;
```

Here *DataType* is one of Java's datatypes and *variable* is the name of the variable. To declare more than one variable of the specified type, you can use a comma-separated list.

Following are valid examples of variable declaration and initialization in Java – Example

```
int a, b,          // Declares three ints, a, b,
int a = 10, b = 10; // Example of initialization
byte B = 22;       // initializes a byte type variable B.
double pi = 3.14159; // declares and assigns a
value of PI.
```

There are three kinds of variables in Java –

a) Local variables
b) Instance variables
c) Class/Static variables

a) Local Variables

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor, or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor, or block.
- Local variables are implemented at stack level internally.
- 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.

Example 1

Here, *age* is a local variable. This is defined inside *pupAge()* method and its scope is limited to only this method.

```
class Test {
  void pupAge( )
    { int  age = 0;
    age = age +
    7;
    System.out.println("Puppy age is : " + age);
  }

  public static void main(String
    args[]) { Test test = new Test();
    test.pupAge();
  }
```

This will produce the following result –

```
Puppy age is: 7
```

b) Instance Variables

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
- Instance variables can be declared in class level before or after use.
- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However, visibility for subclasses can be given for these variables with the use of access modifiers.
- 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.
- Instance variables can be accessed directly by calling the variable name inside the class. However, within static methods (when instance variables are given accessibility), they should be called using the fully qualified name. *ObjectReference.VariableName*.

Example 1

```
import java.io.*;
public class Employee
  { public String name;
  public Employee (String empName)
    { name = empName;
  }
  public void printEmp()
    { System.out.println("name : " + name
     );
  }
  public static void main(String args[]) {
    Employee empOne = new
    Employee("Shobana"); empOne.printEmp();
  }
```

This will produce the following result – Output

```
name : Shobana
```

c) Class/Static Variables

- Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final, and static. Constant variables never change from their initial value.

- Static variables are stored in the static memory. It is rare to use static variables other than declared final and used as either public or private constants.
- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables. Static variables can be accessed by calling with the class name *ClassName.VariableName*.

Example 1

```
import java.io.*;
public class Employee
  { private static double
  salary;
  public static void main(String
    args[]) { salary = 1000;
    System.out.println("average salary:" + salary);
  }
```

This will produce the following result –

```
average salary:1000
```

Identifiers:

Identifiers are the names of variables, methods, classes, packages and interfaces. In the Hello program, *Hello, String, args, main* and *print* are identifiers. There are a few rules and conventions related to the naming of variables.

The rules are:

1. Java variable names are case sensitive. The variable name money is not the same as Money or MONEY.
2. Java variable names must start with a letter, or the $ or _ character.
3. After the first character in a Java variable name, the name can also contain numbers (in addition to letters, the $, and the _ character).
4. Variable names cannot be equal to reserved key words in Java. For instance, the words int or forare reserved words in Java. Therefore you cannot name your variables int or for.

Here are a few valid Java variable name examples:

myvar	myVar	MYVAR	_myVar
$myVar	myVar1	myVar_1	

Data types:

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

There are two data types available in Java –

a) Primitive Data Types
b) Reference/Object Data Types

a) Primitive Data Types

There are eight primitive datatypes supported by Java. Primitive datatypes are predefined by the language and named by a keyword. Let us now look into the eight primitive data types in detail.

Integer Data Types:

Data Type	Memory Size
byte	1 byte
short	2 bytes
int	4 bytes
long	8 bytes

Float Data Types:

Data Type	Memory Size
float	4 byte
double	8 bytes

Character Data Types:

The Character Data type is having 2 bytes of memory size.

String Data Types:

A String represents a group of characters, like New Delhi, AP123, etc. The simplest way to create a String is by storing a group of characters into a String type variable as:

String str="New Delhi";

Now, the String type variable str contains "New Delhi". Note that any string written directly in a program should be enclosed by using double quotes.

There is a class with the name String in Java, where several methods are provided to perform different operations on strings. Any string is considered as an object of String class.

b) Reference Datatypes

- Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy, etc.
- Class objects and various type of array variables come under reference datatype.
- Default value of any reference variable is null.
- A reference variable can be used to refer any object of the declared type or any compatible type.
- Example: Animal an = new Animal("giraffe");

Java Literals

A literal is a value that is stored into a variable directly in the program. There are 5 types of literals.

a) Integer Literals
b) Float Literals
c) Character Literals
d) String Literals
e) Boolean Literals

a) Integer Literals

```
class Test {
   public static void main(String[] args)
   {
      int dec = 101; // decimal-form
      literal int oct = 0100; // octal-form
      literal
      int hex = 0xFace; // Hexa-decimal form literal

      System.out.println(dec); //101
      System.out.println(oct); //64
      System.out.println(hex); //64206
   }
```

b) Float Literals

```
class Test {
   public static void main(String[] args)
   {
      double d1 = 123.4;
      double d2 = 1.234e2;   //Same value as d1, but in scientific
      notation float f1 = 123.4f;

      System.out.println(d1);
      //123.4
      System.out.println(d2);
      //123.4
      System.out.println(f1);
```

c) Character Literals

```
public class Test {
   public static void main(String[] args)
   {
                  char   ch1   =
                  'a';  char  ch2
                  =    '\'';    char
                  ch3 = '\n';
                  char ch4 = '\u0015';

                 System.out.println(ch1); //
                  a
                  System.out.println(ch2);
                  //
                  " System.out.println(ch3);
```

d) String Literals

String literals represent objects of String class. For example, Hello, Anil Kumar, AP1201, etc. will come under string literals, which can be directly stored into a String object.

e) Boolean Literals

Boolean literals represent only two values-true and false, It means we can store either true or false into a boolean type variable.

Operators:

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups −

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

The Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators − Assume integer variable A holds 10 and variable B holds 20, then −

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator.	A + B will give 30
- (Subtraction)	Subtracts right-hand operand from left-hand operand.	A - B will give -10
* (Multiplication)	Multiplies values on either side of the operator.	A * B will give 200
/ (Division)	Divides left-hand operand by right-hand operand.	B / A will give 2
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.	B % A will give 0
++ (Increment)	Increases the value of operand by 1.	B++ gives 21
-- (Decrement)	Decreases the value of operand by 1.	B-- gives 19

The Relational Operators

There are following relational operators supported by Java language.
Assume variable A holds 10 and variable B holds 20, then −

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

The Bitwise Operators

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60 and b = 13; now in binary format they will be as follows –

a = 0011 1100

b = 0000 1101

---

a&b = 0000 1100
a|b = 0011 1101
a^b = 0011 0001
~a = 1100 0011

Assume integer variable A holds 60 and variable B holds 13 then –

Operator	Description	Example
& (bitwise and)	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
| (bitwise or)	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^ (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~ (bitwise complimen t)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>> (zero fill right shift)	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

The Logical Operators

The following table lists the logical operators –

Assume Boolean variables A holds true and variable B holds false, then –

Operator	Description	Example
&& (logical and)	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false
|| (logical or)	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A || B) is true

! (logical not)	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.	!(A && B) is true

The Assignment Operators

Following are the assignment operators supported by Java language –

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C – A
*=	Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Miscellaneous Operators

There are few other operators supported by Java Language.

Conditional Operator (? :)

Conditional operator is also known as the **ternary operator**. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. The operator is written as –

```
variable x = (expression) ? value if true : value if false
```

Example

```
public class Test {

  public static void main(String
    args[]) { int a, b;
    a = 10;
    b = (a == 1) ? 20: 30;
    System.out.println( "Value of b is : " + b );

    b = (a == 10) ? 20: 30;
    System.out.println( "Value of b is : " + b );
  }
}
```

This will produce the following result −

Output

```
Value of b is :
30 Value of b is
```

instanceof Operator

This operator is used only for object reference variables. The operator checks whether the object is of a particular type (class type or interface type). instanceof operator is written as −

```
( Object reference variable ) instanceof
(class/interface type)
```

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is an example **Example**

```
public class Test {

  public static void main(String

    args[]) { String name = "James";

    // following will return true since name is type of
    String boolean result = name instanceof String;
    System.out.println( result );
  }
}
```

This will produce the following result −

Output

```
true
```

This operator will still return true, if the object being compared is the assignment compatible with the type on the right. Following is one more example −

Example

```
class Vehicle {}
public class Car extends Vehicle
  { public static void main(String
  args[]) {
    Vehicle a = new Car();
    boolean result = a instanceof
```

```
  }
}
```

This will produce the following result −

Output

```
true
```

Precedence of Java Operators

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator −

For example, x = 7 + 3 * 2; here x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3 * 2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

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

Primitive Type Conversion and casting:

Java supports two types of castings – primitive data type casting and reference type casting. Reference type casting is nothing but assigning one Java object to another object. It comes with very strict rules and is explained clearly in Object Casting. Now let us go for data type casting.

Java data type casting comes with 3 flavors.

a) Implicit casting
b) Explicit casting
c) Boolean casting.

a) Implicit casting (widening conversion)

A data type of lower size (occupying less memory) is assigned to a data type of higher size. This is done implicitly by the JVM. The lower size is widened to higher size. This is also named as automatic type conversion.

Examples:

```
int x = 10;                // occupies 4 bytes
double y = x;              // occupies 8
bytes System.out.println(y);            //
```

In the above code 4 bytes integer value is assigned to 8 bytes double value.

b) Explicit casting (narrowing conversion)

A data type of higher size (occupying more memory) cannot be assigned to a data type of lower size. This is not done implicitly by the JVM and requires explicit casting; a casting operation to be performed by the programmer. The higher size is narrowed to lower size.

```
double x = 10.5;          // 8 bytes
int y = x;                // 4 bytes ; raises compilation error
```

In the above code, 8 bytes double value is narrowed to 4 bytes int value. It raises error. Let us explicitly type cast it.

```
double x =
10.5; int y =
(int) x;
```

The double x is explicitly converted to int y. The thumb rule is, on both sides, the same data type should exist.

c) Boolean casting

A boolean value cannot be assigned to any other data type. Except boolean, all the remaining 7 data types can be assigned to one another either implicitly or explicitly; but boolean cannot. We say, boolean is incompatible for conversion. Maximum we can assign a boolean value to another boolean.

Following raises

```
error. boolean x
= true;
int y = x;                //
error boolean x = true;
int y = (int) x;          // error
```

byte –> short –> int –> long –> float –> double

In the above statement, left to right can be assigned implicitly and right to left requires explicit casting. That is, byte can be assigned to short implicitly but short to byte requires explicit casting.

Following char operations are possible

```
class Demo {
  public static void main(String args[])
    { char ch1 = 'A';
    double d1 = ch1;

    System.out.println(d              // prints
    System.out.println(ch1 * ch1);   // prints 4225 , 65 *

    double d2 = 66.0;
    char ch2 = (char)
    d2;                               // prints
  }
}
```

Procedure oriented approach:

The Languages like C, Fortran, Pascal, etc., are called procedure oriented programming languages since in these languages, a programmer uses procedures or functions to perform a task. When the programmer wants to write a program, he will first divide the task into several subtasks, each of which expressed as a function. So, C program generally contains several functions which are called and controlled from main() function. This approach is called *procedure oriented approach*.

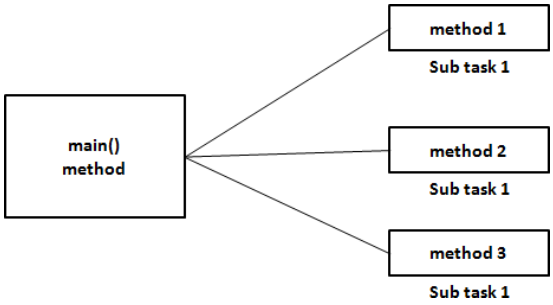


Fig: Procedure oriented approach

Object oriented approach:

On the other hand, languages like C++ and Java use classes and objects in their programs and are called Object Oriented Programming languages. A class is a module which itself contains data and methods (functions) to achieve the task. The main task is divided into several modules, and these are represented as, classes. Each class can perform some tasks for which several methods are written in a class. This approach is called *Object oriented approach.*

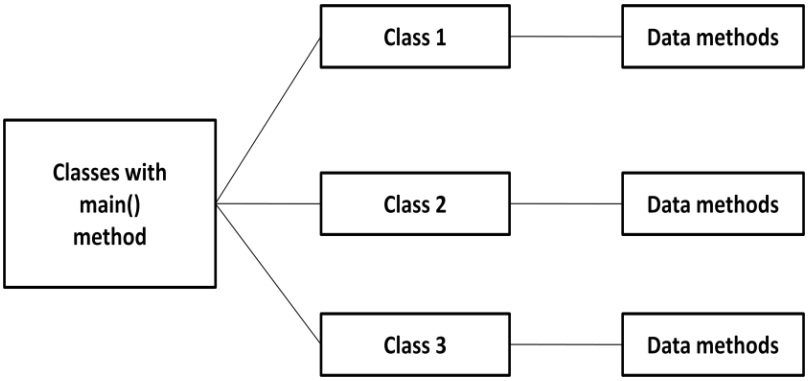


Fig: Object oriented approach

Problems in Procedural oriented approach:

- There is NO Reusability.
- The lines of code are very high.
- Security is very less.
- Don't have any access specifier.

Difference between POP and OOP:

Procedure Oriented Programming	Object Oriented Programming
In POP, program is divided into small parts called functions.	In OOP, program is divided into parts called objects.
In POP, Importance is not given to data but to functions as well as sequence of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a real world.
POP follows Top Down approach.	OOP follows Bottom Up approach.
POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
POP does not have any proper way for hiding data so it is less secure.	OOP provides Data Hiding so provides more security.
In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples of POP are: C, VB, FORTRAN, Pascal.	**Examples of OOP are:** C++, JAVA, VB.NET, C#.NET.

Principles of OOP:

The key ideas of the object oriented approach are:

- Class / Objects
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Class / Objects

Entire OOP methodology has been derived from a single root concept, called 'object'. An object is anything that really exists in the world and can be distinguished from others. This definition specifies that everything in this world is an object. For example, a table, a ball, a car, a dog, a person, etc., everything will come under

objects.

Every object has properties and can perform certain actions. For example, let us take a person whose name is 'Raju'. Raju is an object because he exists physically. He has properties

like name, age, gender, etc. These properties can be represented by variables in our programming.

```
String
name;    int
age;
char gender;
```

Similarly, Raju can perform some actions like talking, walking, eating and sleeping. We may not write code for such actions in programming. But, we can consider calculations and processing of data as actions. These actions are performed by methods (functions), So an object contains variables and methods.

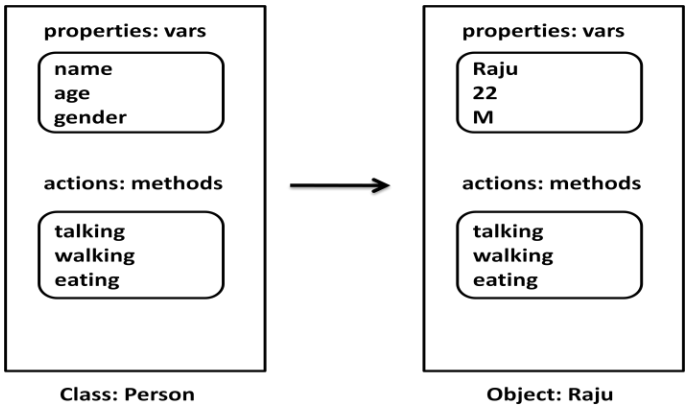


Fig: Person class and Raju Object

From the above diagram we can understand that Person is class and Raju is an object i.e., instance of class.

***Note**: A class is a model for creating objects and does not exist physically. An object is anything that exists physically. Both the class and objects contain variables and methods.*

Abstraction

There may be a lot of data, a class contains and the user does not need the entire data. The user requires only some part of the available data. In this case, we can hide the unnecessary data from the user and expose only that data that is of interest to the user. This is called abstraction.

A bank clerk should see the customer details like account number, name and balance amount in the account. He should not be entitled to see the sensitive data like the staff salaries, profit or loss of the bank amount paid by the bank and loans amount to be recovered etc,. So such data can abstract from the clerk's view. Whereas the bank manager is interested to know this data, it will be provided to the manager.

Encapsulation

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding**.

To achieve encapsulation in Java –

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.

Inheritance

It creates new classes from existing classes, so that the new classes will acquire all the features of the existing classes is called Inheritance. A good example for Inheritance in nature is parents producing the children and children inheriting the qualities of the parents.

There are three advantages inheritance. First, we can create more useful classes needed by the application (software). Next, the process of creating the new classes is very easy, since they are built upon already existing classes. The last, but very important advantage is managing the code becomes easy, since the programmer creates several classes in a hierarchical manner, and segregates the code into several modules.

Polymorphism

The word polymorphism came from two Greek words 'poly' meaning 'many' and 'morphos' meaning 'forms'. Thus, polymorphism represents the· ability to assume several different forms. In programming, we can use a single variable to refer to objects of different types and thus using that variable we can call the methods of the different objects. Thus method call can perform different tasks depending on the type of object.

Polymorphism provides flexibility in writing programs in such a way that the programmer uses same method call to perform different operations depending on the requirement.

Application of OOP:

The promising areas includes the followings,

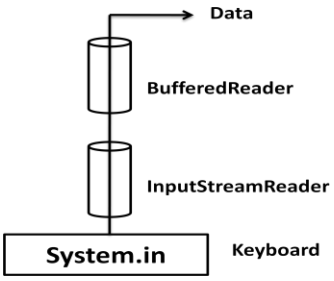
1. *Real Time Systems* Design
2. Simulation and Modeling System
3. Object Oriented Database
4. Object Oriented Distributed Database
5. Client-Server System
6. Hypertext, Hypermedia
7. Neural Networking and Parallel Programming
8. Decision Support and Office Automation Systems
9. CIM/CAD/CAM Systems
10. AI and Expert Systems

Accessing Input from the keyboard:

- Input represents data given to a program and output represents data displayed as a result of a program.
- A stream is required to accept input from the keyboard. A stream represents flow of data from one place to another place.
- It is like a water-pipe where water flows. Like a water-pipe carries water from one place to another, a stream carries data from one place to another place.
- A stream can carry data from keyboard to memory or from memory to printer or from memory to a file.
- A stream is always required if we want to *move* data from one place to another.
- Basically, there are two types of streams: input streams and output streams.

- Input streams are those streams which receive or read data coming from some other place.
- Output streams are those streams which send or write data to some other place.
- All streams are represented by classes in java.io (input and output) package.

- This package contains a lot of classes, all of which can be classified into two basic categories: input streams and output streams.
- Keyboard is represented by a field, called in System class. When we write System.in, we are representing a standard input device, i.e. keyboard, by default. System class is found in java.lang (language) package and has three fields' as shown below.
- All these fields represent some type of stream:
  - *System.in*: This represents Ii1putStream object, which by default represents standard input device, i.e., keyboard.
  - *System.out*: This represents PrintStream object, which by default represents standard output device, i.e., monitor.
  - *System.err*: This field also represents PrintStream object, which by default represents monitor.
- Note that both *System.out* and *System.err* can be used to represent the monitor and hence any of these two can be used to send data to the monitor.
- To accept data from the keyboard i.e., *System.in*. we need to connect it to an input stream. And input stream is needed to read data from the keyboard.



- Connect the keyboard to an input stream object. Here, we can use

  InputStreamReader that can read data from the keyboard.

  InputStreamReader isr = new InputStreamReader(System.in);

- Connect InputStreamReader to BufferedReader, which is another input type of stream.

  BufferedReader br = new BufferedReader(isr);

- Now we can read data coming from the keyboard using read( ) and readLine( ) methods available in BufferedReader class.

Ex: Write a JAVA Program to read Single character from the keyboard.

```
import java.io.*; import
java.lang.*; class Test {
        public static void main(String[] args)throws IOException
        {
                InputStreamReader isr = new InputStreamReader(System.in);
                BufferedReader br = new BufferedReader(isr); System.out.print("Enter a
                single Character: ");
                char ch = (char)br.read(); System.out.print("\n The
                Character is "+ch);
        }
}
```

Output:

```
javac Test.java
java Test
Enter a single Character:
m The Character is m
```

Ex: Write a JAVA Program to read String from the keyboard.

```
import java.io.*; import
java.lang.*; class Test {
        public static void main(String[] args)throws IOException
        {
                InputStreamReader isr = new InputStreamReader(System.in);
                BufferedReader br = new BufferedReader(isr); System.out.print("Enter the
                String: ");
                String str = br.readLine(); System.out.print("\n The
                String is "+str);
        }
}
```

Output:

```
javac Test.java
java Test
Enter the String:
mothi The String is
mothi
```

Ex: Write a JAVA Program to read Integer from the keyboard.

```
import java.io.*; import
java.lang.*; class Test {
        public static void main(String[] args)throws IOException
        {
                InputStreamReader isr = new InputStreamReader(System.in);
                BufferedReader br = new BufferedReader(isr); System.out.print("Enter the
                Number: ");
                int n = Integer.parseInt(br.readLine());
                System.out.print("\n The number is "+n);
        }
}
```

Output:

```
javac Test.java
java Test
Enter the Number:
56 The Number is 56
```

- To accept Float value we have to use
  float f = Float.parseFloat(br.readLine());
- To accept Double value we have to use
  double d = Double.parseDouble(br.readLine());

Reading Input with *java.util.Scanner* Class:

We can use Scanner class of java.util package to read input from the keyboard or a text file. When the Scanner class receives input, it breaks the input into several pieces, called. *tokens.* These tokens can be retrieved from the Scanner object using the following methods

- **next() - to read a string**
- **nextByte() - to read byte value**
- **nextInt() - to read an integer value**
- **nextFloat() - to read float value**

- nextLong()- to read long value.
- nextDouble () -to read double value

To read input from keyboard, we can use Scanner class as:

```
Scanner sc = new Scanner(System.in);
```

Ex: Write a JAVA Program to read different types of data separated by space, from the keyboard using the Scanner class.

```
import java.util.Scanner; import
java.lang.*; class Test {
        public static void main(String[] args)
        {
                Scanner sc = new Scanner(System.in);
                System.out.print("Enter id name sal: "); int id =
                sc.nextInt();
                String name = sc.next(); float sal =
                sc.nextFloat();
                System.out.println("Id is: "+id);
                System.out.println("Name is: "+name);
                System.out.println("Sal is: "+sal);

        }
}
```

Output:

```
javac Test.java
java Test
Enter id name sal: 09 sudheer
40000.00 Id is: 09
Name is:
sudheer Sal is:
40000.00
```

Strings:

In java, string is basically an object that represents sequence of char values. An array of characters works same as java string.

String is a class in java.lang package. But in java, all classes are also considered as data types. So, we can take string as a data type also. A class is also called as user-defined data type.

Creating Strings:

There are three ways to create strings in Java:

- We can create a string just by assigning a group of characters to a string type variable:

```
String s;
s = "Hello";
```

- Preceding two statements can be combined and written as:

```
String s = "Hello";
```

In this, case JVM creates an object and stores the string: "Hello" in that object. This object is referenced by the variable's'. Remember, creating object means allotting memory for storing data.

- We can create an object to String class by allocating memory using new operator. This is just like creating an object to any class, like given here:

```
String s = new String("Hello");
```

  Here, we are doing two things. First, we are creating object using new operator. Then, we are storing the string: "Hello" into the object.

- The third way of creating the strings is by converting the character arrays into strings. Let us take a character type array: arr[] With some characters, as:

```
char arr[]={'H','e','l','l','o'};
```

- Now create a string object, by pass.ing the array name to it, as:

```
String s = new String(arr);
```

String Methods:

No.	Method	Description
1	char **charAt**(int index)	returns char value for the particular index
2	**int length**()	returns string length
3	static String **format**(String format, Object... args)	returns formatted string
4	String **substring**(int beginIndex, int endIndex)	returns substring for given begin index and end index
5	boolean **contains**(CharSequence s)	returns true or false after matching the sequence of char value
6	static String **join**(CharSequence delimiter, CharSequence... elements)	returns a joined string
7	boolean **equals**(Object another)	checks the equality of string with object
8	boolean **isEmpty**()	checks if string is empty
9	String **concat**(String str)	concatenates specified string
10	String **replace**(char old, char new)	replaces all occurrences of specified char value

11	static String equalsIgnoreCase(String another)	Compares another string. It doesn't check case.

12	String[] split(String regex, int limit)	returns splitted string matching regex and limit
13	int indexOf(int ch)	returns specified char value index
14	String toLowerCase()	Returns string in lowercase.
15	String toUpperCase()	Returns string in uppercase.
16	String trim()	Removes beginning and ending spaces of this string.
17	static String valueOf(int value)	Converts given type into string. It is overloaded.

equals method in java:

Program:

```
class StringComapre
{
        public static void main(String args[])
        {
                String s1 = "Hello";
                String s2 = new String("Hello");
                System.out.println(s1==s2); //false
                System.out.println(s1.equals(s2));
                //true
        }
}
```

Immutability of String:

- In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.
- Once string object is created its data or state can't be changed but a new string object is created.

```
class Testimmutablestring{
 public static void main(String
  args[]){ String s="Sachin";
  s.concat(" Tendulkar");
  System.out.println(s);
 }
}
```

Output: Sachin

- But if we explicitely assign it to the reference variable, it will refer to "Sachin Tendulkar" object.For example:

```
class Testimmutablestring1{
 public static void main(String
  args[]){ String s="Sachin";
  s=s.concat("
  Tendulkar");
  System.out.println(s);
 }
}
```

Output: Sachin Tendulkar

StringBuffer Class:

- Java *StringBuffer* class is used to create mutable (modifiable) string. The *StringBuffer* class in java is same as String class except it is mutable i.e. it can be changed.
- *String* class objects are immutable and hence their contents cannot be modified. *StringBuffer* class objects are mutable; so they can be modified. Moreover the methods that directly manipulate data of the object are not available in String class. Such methods are available in *StringBuffer* class.

Creating StringBuffer objects:

We can create *StringBuffer* object by using new operator and pass the string to the object, as:

```
StringBuffer sb = new StringBuffer("Hello");
```

Methods in StringBuffer class:

S.No.	Method & Description
1	**StringBuffer append(String str)** This method appends the specified string to this character sequence.
2	**char charAt(int index)** This method returns the char value in this sequence at the specified index.
3	**StringBuffer delete(int start, int end)** This method removes the characters in a substring of this sequence.
4	**void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)** This method characters are copied from this sequence into the destination character array dst.
5	**int indexOf(String str, int fromIndex)** This method returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
6	**StringBuffer insert(int offset, String str)** This method inserts the string into this character sequence.
7	**int lastIndexOf(String str)** This method returns the index within this string of the rightmost occurrence of the specified substring.
8	**int length()** This method returns the length (character count).
9	**StringBuffer replace(int start, int end, String str)** This method replaces the characters in a substring of this sequence with characters in the specified String.
10	**StringBuffer reverse()** This method causes this character sequence to be replaced by the reverse of the sequence.
11	**void setCharAt(int index, char ch)** The character at the specified index is set to ch.
12	**String substring(int start, int end)** This method returns a new String that contains a subsequence of characters currently contained in this sequence.
13	**String toString()** This method returns a string representing the data in this sequence.
14	**void trimToSize()** This method attempts to reduce storage used for the character sequence.

Program: Write a JAVA Program to delete single character. class DeleteCharacter

```
{
        public static void main(String args[])
        {
                StringBuffer sb = new StringBuffer("Hello");
                sb.delete(1,2);
                System.out.println(sb); //Hllo
                sb.delete(3,4);
                System.out.println(sb); //Hll
        }
}
```

Classes and Objects:

We know a class is a model for creating objects. This means, the properties and actions of the objects are written in the class. Properties are represented by variables and actions of the objects are represented by methods. So, a class contains variables and methods. The same variables and methods are also available in the objects because they are created from the class. These variables are also called 'instance variables' because they are created inside the object (instance).

If we take Person class, we can write code in the class that specifies the properties and actions performed by any person. For example, a person has properties like name, age, etc. Similarly a person can perform actions like talking; walking, etc. So, the class Person contains these properties and actions as shown

```
class Person
{
        String name;
        int age; void
        talk()
        {
                System.out.println("Hello my name is "+name);
                System.out.println("and my age is "+age);
        }
}
```

here:

Object Creation:

We know that the class code along with method code is stored in 'method area' of the JVM. When an object is created, the memory is allocated on 'heap'. After creation of an object, JVM produces a unique reference number for the object from the memory address of the object. This reference number is also called hash code number.

To know the hashcode number (or reference) of an object, we can use hashCode() method object class, as shown here:

```
Person p = new Person();
System.out.println(p.hashCode());
```

The object reference, (hash code) internally represents heap memory where instance- variables are stored. There would be a pointer (memory address) from heap memory to a special structure located in method area. In method area, a table

is available which contains pointers to static variables and methods.

```
class Person
{
	String name;
	int age; void
	talk()
	{
		System.out.println("Hello my name is "+name);
		System.out.println("and my age is "+age);
	}
}
class Demo
{
	public static void main(String args[])
	{
		Person p = new Person();
		System.out.println(p.hashCode());	//
		705927765
	}
```

Initializing the instance variables:

It is the duty of the programmer to initialize the instance depending on his requirements. There are various ways to do this.

First way is to initialize the instance variables of Person class in other class, i.e., Demo class. For this purpose, the Demo class can be rewritten like this:

```
class Demo
{
	public static void main(String args[])
	{
		Person p = new Person();
		p.name="Raju";  p.age=22;
		System.out.println(p.hashCode());	// 705927765
	}
}
```

Second way is to initialize the instance variables of Person class in the Same class. For this purpose, the Person class can be rewritten like this:

```
class Person
{
	String name="Raju";
	int age=22;
	void talk()
	{
		System.out.println("My Name is "+name);
		System.out.println("My Age is "+age);
	}
}
```

```
class Demo
{
	public static void main(String args[])
	{
		Person p1 = new Person();
		System.out.println("p1 hashcode "+p1.hashCode());
		p1.talk();
		Person p2 = new Person();
		System.out.println("p2 hashcode "+p2.hashCode());
		p2.talk();
	}
}
```

Output:

```
p1 hashcode
705927765 My Name
is Raju
My Age is 22
p2 hashcode
705934457 My Name
is Raju
My Age is 22
```

But, the problem in this way of initialization is that all the objects are initializing with same data.

Constructors:

The third possibility of initialization is using constructors. A constructor is similar to a method is used to initialize the instance variables. The sole purpose of a constructor is to initialize the instance variables. A constructor has the following characteristics:

- The constructor's name and class name should be same. And the constructor's name should end with a pair of simple braces.

```
Person()
{
}
```

- A constructor may have or may not have parameters: Parameters are variables to receive data from outside into the constructor. If a constructor does not have any parameters, it is called 'Default constructor'. If a constructor has 1 or more parameters, it is called 'parameterized constructor'; For example, we can write a default constructor as:

```
Person()
{
}
```

And a Parameterized constructor with two parameters, as:

```
Person(String s, int i)
{
}
```

- A constructor does not return any value, not even 'void'. Recollect, if a method does not return any value, we write 'void' before the method name. That means, the method is returning ‘void’ which means 'nothing'. But in case of a constructor, we should not even write 'void' before the constructor.
- A constructor is automatically called and executed at the time of creating an object. While creating an object, if nothing is passed to, the object, the default constructor is called and executed. If-some values are passed to the object, then the parameterized constructor is called. For example, if we create the

```
Person p = new Person(); // Default Constructor
Person p = new Person(Raju, 22); // Parameterized Constructor
```

object as:

- A constructor is called and executed only once per object. This means .when we create an object, the constructor is called. When we create second object, again the constructor is called second time.

```
class Person
{
 String name;
 int age;
 Person()
 {
       name="Raju"; age=22;
 }
 void talk()
 {
       System.out.println("Hello my name is "+name);
       System.out.println("and my age is "+age);
 }
}
class Demo
{
 public static void main(String args[])
 {
       Person p = new Person(); // Default Constructor
       System.out.println("p  hashcode  "+p.hashCode());
       p.talk();
       Person p1 = new Person();
       System.out.println(p1.hashCode());
       p1.talk();
 }
}
```

Output:

```
p hashcode
705927765 My
Name is Raju
My Age is 22
p1 hashcode
705934457 My Name
```

is Raju
My Age is 22

From the output, we can understand that the same data "Raju" and 22 are store in both objects p1 and p2. p2 object should get p2's data, not p1's data. Isn't it? To mitigate the problem, let us try parameterized constructor, which accepts data from outside and initializes instance variables with that data.

```
class Person
{
	String name;
	int age;
	Person()
	{
		name="Raju";
		age=22;
	}
	Person(String s, int i)
	{
		name=s;
		age=i;
	}
	void talk()
	{
		System.out.println("My Name is "+name);
		System.out.println("My Age is "+age);
	}
}
class Demo
{
	public static void main(String args[])
	{
		Person p1 = new Person();
		System.out.println("p1 hashcode "+p1.hashCode()); p1.talk();
		Person p2 = new Person("Sita",23);
		System.out.println(p2.hashCode());
		p2.talk();
	}
}
```

Output:

```
p1 hashcode
705927765 My Name
is Raju
My Age is 22
p2 hashcode
705934457 My Name
is Sita
My Age is 23
```

Q: When is a constructor called, before or after creating the object?

Ans: A constructor is called concurrently when the object creation is going on. JVM first allocates memory for the object and then executes the constructor to initialize the instance variables. By the time, object creation is completed; the constructor execution is also completed.

Difference between Default Constructor and Parameterized Constructor:

Default Constructor	Parameterized Constructor
Default constructor is useful to initialize all the objects with same data.	Parameterized constructor is useful to initialize each object with different data.
Default constructor does not have any parameters.	Parameterized constructor will have 1 or more parameters.
When the data is not passed at the time of creating object, default constructor is called.	When the data is passed at the time of creating object, default constructor is called.

Difference between Constructor and Method:

Java Constructor	Java Method
Constructor is used to initialize the state of an object.	Method is used to expose behaviour of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name may or may not be same as class name.

Q: *What is constructor overloading?*

Ans: *Writing two or more constructors with the same name but with difference in the parameters is called constructor overloading. Such constructors are useful to perform different tasks.*

Write a Java Program for Constructor Overloading?

```
class Box
{
	double width, height, depth; Box()
	{
		width = height = depth = 0;
	}
	Box(double len)
	{
		width = height = depth = len;
	}
	Box(double w, double h, double d)
	{
		width = w;
		height = h;
		depth = d;
	}
```

```
        double volume()
        {
                return width * height * depth;
        }
}
public class Test
{
        public static void main(String args[])
        {
                Box mybox1 = new Box(10, 20, 15);
                Box mybox2 = new Box();
                Box mybox3 = new Box(7);
                double vol;
                vol = mybox1.volume();
                System.out.println(" Volume of mybox1 is " + vol); vol =
                mybox2.volume();
                System.out.println(" Volume of mybox2 is " + vol); vol =
                mybox3.volume();
                System.out.println(" Volume of mybox3 is " + vol);
        }
}
```

Output:

Volume of mybox1 is
3000.0 Volume of mybox2
is 0.0 Volume of mybox3 is
343.0

Java Garbage Collection

- In java, garbage means unreferenced objects. Garbage Collection is process of reclaiming the runtime unused memory automatically.
- In other words, it is a way to destroy the unused objects. To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

Advantage of Garbage Collection

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector (a part of JVM) so we don't need to make extra efforts.

How can an object be unreferenced?

- Even though programmer is not responsible to destroy useless objects but it is highly recommended to make an object unreachable (thus eligible for GC) if it is no longer required.
- There are many ways:
  - By nulling the reference
  - By assigning a reference to another

- By annonymous object etc.

1) By nulling a reference:

```
Employee e=new Employee();
e=null;
```

2) By assigning a reference to another:

```
Employee e1=new Employee();
Employee e2=new Employee();
e1=e2;//now the first object referred by e1 is available for garbage collec
tion
```

3) By annonymous object:

```
new Employee();
```

finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as: `protected void finalize(){}`

gc() method

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

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

Program:

```
public class TestGarbage1
{
   public void finalize()
   {
     System.out.println("object is garbage collected");
   }
    public static void main(String args[])
   {
      TestGarbage1 s1=new TestGarbage1();
      TestGarbage1 s2=new TestGarbage1();
      s1=null;
      s2=null;
      System.gc();
   }
}
```

Output:

```
object is garbage
collected object is
garbage collected
```

Methods in Java:

- A method represents a group of statements that performs a task. Here 'task' represents a calculation or processing of data or generating a report etc.
- A Method has two types:
  - Method Header
  - Method Body

- **Method Header:** It contains method name, method parameters and method return data type.

```
returntype methodName(parameters)
```

- **Method Body:** Method body consists of group of statements which contains logic to perform the task.

```
{
    Statements;
}
```

- If a method returns some value, then a return statement should be written within the body of method, as:

```
return somevalue;
```

- There are two types of methods in java:
  a) Instance methods
  b) Static methods

a) Instance Methods:

Instance methods are the methods which ***act*** on the instance variables of the class. To call the instance methods, we should use the form:

```
objectname. methodname ();
```

Example-1: Write a program to perform addition of two numbers using instance method without return statement.

```
class Addition
{
    void add(double a,double b)
    {
        double c=a+b;
        System.out.println("Addition is
        "+c);
    }
}
class MethodDemo
{
    public static void main(String args[])
    {
        Addition a1=new
        Addition(); a1.add(27,35);
    }
```

Output:

```
Addition is 62
```

Example-2: Write a program to perform addition of two numbers using instance method with return statement.

```
class Addition
{
    double add(double a,double b)
    {
        double c=a+b;
        return c;
    }
```

```
}
class MethodDemo
{
      public static void main(String args[])
      {
            Addition a1=new
            Addition(); double
            c=a1.add(27,35);
           System.out.println("Addition is "+c);

      }
```

Output:

```
Addition is 62
```

b) **Static Methods:**

Static methods are the methods which ***do not act*** on the instance variables of the class. static methods are declared as 'static'. To call the static methods, we need not create the object. we call a static method, as:

```
ClassName. methodname ();
```

Example-1: Write a program to perform addition of two numbers using static method without return statement.

```
class Addition
{
      static void add(double a,double b)
      {
            double c=a+b;
            System.out.println("Addition is
            "+c);
      }
}
class MethodDemo
{
      public static void main(String args[])
      {
            Addition.add(27,35);
      }
```

Output:

```
Addition is 62
```

Example-2: Write a program to perform addition of two numbers using static method with return statement.

```
class Addition
{
      static double add(double a,double b)
      {
            double c=a+b;
            return c;
      }
}
class MethodDemo
{
```

```
        public static void main(String args[])
        {
             double c= Addition.add(27,35);
            System.out.println("Addition is
            "+c);

        }
```

Output:

```
Addition is 62
```

Q: why instance variables are not available to static methods?
Ans: *After executing static methods, JVM creates the objects. So, the instance variables are not available to static methods.*

Example: Write a JAVA program to implement method overloading.

Static Block:

- A static block is a block of statements declared as 'static'.

```
static {
                statements;
}
```

- JVM executes a static block on highest priority basis. This means JVM first goes to static block even before it looks for the `main()` method in the program.

Example-1: Write a program to test which one is-executed first by JVM, the static block or the static method.

```
class Demo
{
   static
   {
      System.out.println("Static block");
   }
   public static void main(String args[])
   {
      System.out.println("Static Method");
   }
}
```

Output:

```
Static block
Static
```

Example-2: Write a java program without `main()` method.

```
class Demo
{
   static
   {
      System.out.println("Static block");
   }
}
```

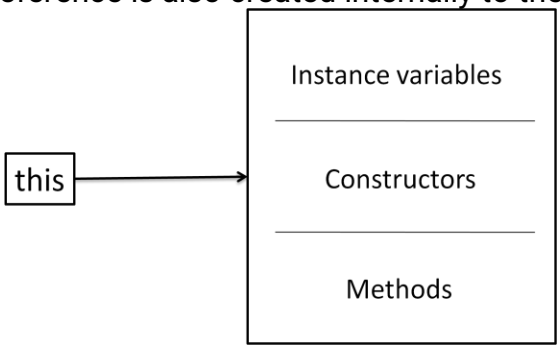
Output:

```
Static block
```

Q: Is it possible to compile and run a Java program without writing `main()` *method? Ans: Yes, it is possible by using a static block in theJava program.*

'this' keyword:

- 'this' is a keyword that refers to the current object of the class where it is used. In other words, 'this' refers the object of the present class.
- Generally, we write instance variables, constructors and methods in a class. All these members are referenced by 'this'. When an object is created to a class, a default reference is also created internally to the object.



Instance variables are not initialized then default value

int -0
float -0.0
Char- emptyspace
String-null
Boolean-false

Example: Write a program for 'this' keyword.

```
class Demo
{
      int x;
      Demo(int
      x)
      {
          this.x=x+3;
       System.out.println("x="+x);
      System.out.println("this.x="+this.x);
      }
      public static void main(String args[])
      {
            Demo d=new Demo(5);
      }
```

Output:

```
x=5
this.x=8
```

Recursion:

A method calling itself is known as recursive method', and that process is called 'recusrion'. It is possible to write recursive methods in Java. Let us, take an example to find the factorial value of the given number. Factorial value of number num is defined as: num * (num-1) * (num-2) * .....

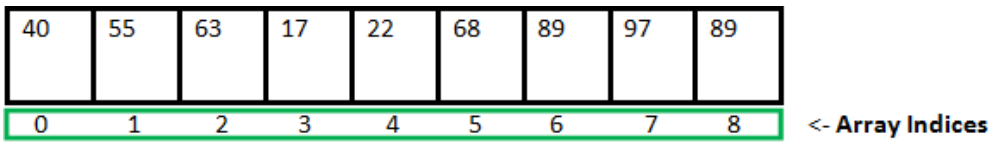
Example: Write a java Program to find factorial of a given number using recursion.

```
import java.util.Scanner;
class Factorial
{
     static long fact(long num)
     {
          if(num==1 || num==0)
               return 1;
          else
```

```
        {
            return num*fact(num-1);
        }
    }
    public static void main(String args[])
    {
        Scanner sc=new Scanner(System.in);
        System.out.print("Enter number: ");
        long num=sc.nextInt();
        long f=Factorial.fact(num);
        System.out.println("Factorial is
        "+f);
    }
```

Arrays:

Java provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.



Array Length = 9
First Index = 0
Last Index = 8

Advantage of Java Array

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data easily.
- **Random access:** We can get any data located at any index position.

Disadvantage of Java Array

- **Size Limit:** We can store only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in java.

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Declaring Single Dimensional Array:

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable –

Syntax:

```
dataType[] varName;
      or
dataType varName[];
```

Example:

```
int a[];        or        int[] a;
```

Instantiating an Array in Java

When an array us declared, only a reference of array is created. To actually create or give memory to array, you create an array like this: The general form of *new* as it applies to one-dimensional arrays appears as follows:

```
varName = new type [size];
```

Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *varName* is the name of array variable that is linked to the array.

That is, to use *new* to allocate an array, you must specify the type and number of elements to allocate.

Single -Dimensional Array:

The declaration and instantiating of an 1-D array is, as follows

Example:

```
int[] a=new int[5];
```

Example: Write a JAVA program to find sum of all numbers in an array.

```
import java.util.Scanner;
class SumArray
{
	public static void main(String[] args)
	{
		Scanner sc=new Scanner(System.in);
		System.out.print("Enter the no. of elements: ");
		int n=sc.nextInt();
		int[] a=new int[n];
		System.out.print("Enter the "+n+" elements ");
		for(int i=0;i<n;i++)
			a[i]=sc.nextInt();
		int sum=0;
		for(int i=0;i<n;i++)
		{
			sum=sum+a[i];
		}
		System.out.print("The Sum of elements is "+sum);
	}
}
```

Output:

```
Enter the no. of elements:
5 Enter the 5 elements 5 4
2 1 3 The Sum of elements
is 15
```

Two-Dimensional Array:

The declaration and instantiating of an 1-D array is, as follows

Example:

```
int[][] a=new int[2][3];
```

Example: Write a java Program to perform Matrix addition.

```
import java.util.Scanner;
class MatrixAddition
{
	public static void main(String[] args)
	{
		Scanner sc=new Scanner(System.in);
		int[][] a=new int[2][2];
		int[][] b=new int[2][2];
		int[][] c=new int[2][2];
		System.out.println("Enter the A Matrix elements ");
		for(int i=0;i<2;i++)
			for(int j=0;j<2;j++)
				a[i][j]=sc.nextInt();
		System.out.println("Enter the B Matrix elements ");
		for(int i=0;i<2;i++)
			for(int j=0;j<2;j++)
				b[i][j]=sc.nextInt();
		for(int i=0;i<2;i++)
			for(int j=0;j<2;j++)
				c[i][j]=a[i][j]+b[i][j];
		System.out.println("The Addition of Matrix is ");
		for(int i=0;i<2;i++)
		{
			for(int j=0;j<2;j++)
			{
				System.out.print(c[i][j]+" ");
			}
			System.out.print("\n");
		}
	}
}
```

Output:

```
Enter the A Matrix
elements 1 2
3 4
Enter the B Matrix
elements 5 6
7 8
The Addition of Matrix
is 6 8
10 12
```

Example: Write a java Program to perform Matrix multiplication.

```
import java.util.Scanner;
class
MatrixMultiplication
{
 public static void main(String[] args)
 {
      Scanner sc=new Scanner(System.in);
      int[][] a=new int[2][2];
      int[][] b=new int[2][2];
      int[][] c=new int[2][2];
      System.out.println("Enter the A Matrix elements ");
      for(int i=0;i<2;i++)
           for(int j=0;j<2;j++)
                a[i][j]=sc.nextInt();
      System.out.println("Enter the B Matrix elements ");
      for(int i=0;i<2;i++)
           for(int j=0;j<2;j++)
                b[i][j]=sc.nextInt();
      for(int i=0;i<2;i++)
           for(int j=0;j<2;j++)
                for(int k=0;k<2;k++)
                c[i][j]+=a[i][k]*b[k][j];
      System.out.println("The Multiplication of Matrix is ");
      for(int i=0;i<2;i++)
      {
           for(int j=0;j<2;j++)
           {
                System.out.print(c[i][j]+" ");
           }
           System.out.print("\n");
      }
 }
```

Output:

```
Enter the A Matrix
elements 1 2
3 4
Enter the B Matrix
elements 5 6
7 8
The Multiplication of Matrix
is 19 22
43 50
```

Command line arguments:

- The java command-line argument is an argument i.e. passed at the time of running the java program.
- The arguments passed from the console can be received in the java program and it can be used as an input.
- So, it provides a convenient way to check the behaviour of the program for the different values. You can pass **N** (1,2,3 and so on) numbers of arguments from the command prompt.

Example: Write a JAVA Program to read input from command line.

```
class CommandLine
{
  public static void main(String[] args)
  {
       System.out.println("Command line argument values are ");
       for(int i=0;i<args.length;i++)
       {
            System.out.println(args[i]);
       }
  }
}
```

Output:

```
Command line argument values
are 27
2.5
krishna
```

Nested Classes:

- Nested classes are the classes that contain other classes like inner classes. Inner class is basically a safety mechanism since it is hidden from other classes in its outer class.
- To make instance variables not available outside the class, we use 'private' access specifier before the variables. This is how we provide the security mechanism to variables. Similarly, in some cases we want to provide security for the entire class.
- In this case, can we use 'private' access specifier before the class? The problem is, if we use private access specifier before a class, the class is not available to the Java compiler or JVM. SO it is illegal to use 'private' before a class name in Java.
- But, private specifier is allowed before an inner class and thus it is useful to provide security for the entire inner class.
- An inner class is a class that is defined inside another class.
- Inner class is a safety mechanism.
- Inner class is hidden from other classes in its outer class.
- An object to inner class cannot be created in other classes.
- An object to inner class can be created only in its outer class.

- Inner class _can access the members of outer class directly.
- Inner class object and outer class objects are created in separate memory locations.

Example: Write a JAVA program to create the outer class BankAccount and the inner class Interest in it.

```
import java.util.Scanner;
class BankAccount
{
	double bal;
	BankAccount(double
	b)
	{
		bal=b;
	}
	void contact(double r)
	{
		Scanner sc=new Scanner(System.in);
		System.out.print("Enter Password: ");
		String password=sc.next();
		if(password.equals("cse556"))
		{
			Interest in=new
			Interest(r);
			in.calculateInterest();
		}
	}
	private class Interest
	{
		double rate;
		Interest(double
		r)
		{
			rate=r;
		}
		void calculateInterest()
		{
			double
			inter=bal*rate/100;
			bal=bal+inter;
			System.out.println("Updated Balance= "+bal);
		}
	}
	public static void main(String[] args)
	{
```

Output:

Enter Password: cse556
Updated Balance=
10950.0

JAVA.UTIL.RANDOM CLASS IN JAVA

By
M. BABY ANUSHA,
ASST.PROF IN CSE DEPT.,
RGUKT,NUZVID

JAVA.UTIL.RANDOM CLASS IN JAVA

- Random class is used to generate pseudo-random numbers in java.
- This class provides various method calls to generate different random data types such as float, double, int.
- The **Java Random class** is a part of the java. util package and contains inbuilt methods to generate random numbers.
- The following import statement must be included in your code when using this class.

```
import java.util.Random;
```

JAVA.UTIL.RANDOM CLASS IN JAVA

Constructor:

- **Random()**: Creates a new random number generator.

METHODS OF RANDOM CLASS:

- **java.util.Random.doubles():** Returns an effectively unlimited stream of pseudo random double values, each between zero (inclusive) and one (exclusive)
  Syntax:
  public DoubleStream doubles()
- **Returns:** a stream of pseudo random double values

JAVA.UTIL.RANDOM.INTS():

- **java.util.Random.ints():**

  Returns an effectively unlimited stream of pseudo random int values

  Syntax:

  public IntStream ints()

  Returns: a stream of pseudo random int values

JAVA.UTIL.RANDOM.LONGS():

- **java.util.Random.longs():**

  Returns an effectively unlimited stream of pseudo random long values

  Syntax:

- **public LongStream longs()**

  Returns: a stream of pseudorandom long values

JAVA.UTIL.RANDOM.NEXT(INT BITS)

- **next(int bits): java.util.Random.next(int bits)**

  Generates the next pseudo random number

  Syntax:
- **protected int next(int bits) Parameters:**
- bits - random bits
- **Returns:** the next pseudo random value from this random number generator's sequence

JAVA.UTIL.RANDOM.NEXTBOOLEAN():

- **java.util.Random.nextBoolean():**

  Returns the next pseudo random, uniformly distributed boolean value from this random number generator's sequence

  Syntax:

- **public boolean nextBoolean()**

  Returns: the next pseudorandom, uniformly distributed boolean value from this random number generator's sequence

JAVA.UTIL.RANDOM.NEXTBYTES(BYTE[] BYTES):

- **java.util.Random.nextBytes(byte[] bytes):**

  Generates random bytes and places them into a user-supplied byte array

  Syntax:

- **public void nextBytes(byte[] bytes)**

JAVA.UTIL.RANDOM.NEXTDOUBLE():

- **java.util.Random.nextDouble():**
- Returns the next pseudo random, uniformly distributed double value between 0.0 and 1.0 from this random number generator's sequence
  Syntax:
- **public double nextDouble()**
- **Returns:** the next pseudo random, uniformly distributed double value between 0.0 and 1.0 from this random number generator's sequence

JAVA.UTIL.RANDOM.NEXTFLOAT():

- **java.util.Random.nextFloat():**
- Returns the next pseudo random, uniformly distributed float value between 0.0 and 1.0 from this random number generator's sequence
  Syntax:
- **public float nextFloat()**
- **Returns:** the next pseudorandom, uniformly distributed float value between 0.0 and 1.0 from this random number generator's sequence

JAVA.UTIL.RANDOM.NEXTINT():

- java.util.Random.nextInt():

  Returns the next pseudorandom, uniformly distributed int value from this random number generator's sequence

  Syntax:

- **public int nextInt()**

  Returns: the next pseudorandom, uniformly distributed int value from this random number generator's sequence

JAVA.UTIL.RANDOM.NEXTINT(INT BOUND):

- **java.util.Random.nextInt(int bound):**

  Returns a pseudo random, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence
  Syntax:
- **public int nextInt(int bound) Parameters:**
  bound - the upper bound (exclusive). Must be positive.

JAVA.UTIL.RANDOM.NEXTLONG():

- **java.util.Random.nextLong():**
- Returns the next pseudorandom, uniformly distributed long value from this random number generator’s sequence
  Syntax:public long nextLong()
- **Returns:** the next pseudorandom, uniformly distributed long value from this random number generator's sequence


Thank you!
shutterstock.com · 567687052

Exploring the String Class

By
M. BABY ANUSHA,
ASST.PROF IN CSE DEPT.,
RGUKT,NUZVID

STRINGS

- Strings represent a sequence of characters.
- The easiest way to represent a sequence of characters in JAVA is by using a character array.

```
char Array[ ] = new char [5];
```

Character arrays are not good enough to support the range of operations we want to perform on strings.

In JAVA strings are class objects and implemented using two classes:-

- ✓String
- ✓StringBuffer.

In JAVA strings are not a character array and is not NULL terminated.

DECLARING & INITIALISING

- Normally, objects in Java are created with the *new* keyword.

```
String name;
  name = new String("Craig");
```

OR

```
String name= new String("Craig");
```

- However, String objects can be created "implicitly":

```
String name;
name = "Craig";
```

The String Class

- String objects are handled specially by the compiler.
  - String is the only class which has "implicit" instantiation.
- The String class is defined in the **java.lang package.**
- Strings are immutable.
- The value of a String object can never be changed.
  - For mutable Strings, use the StringBuffer class.

DYNAMIC INITIALIZATION OF STRINGS

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

    String city = br.readLine();
```

Give throws IOException beside function name

```
Scanner sc=new Scanner(System.in);
    String state=sc.nextLine();
   String state1=sc.next();
```

STRING CONCATENATION

- JAVA string can be concatenated using + operator.

```
String name="Ankita";
String surname="Karia";
System.out.println(name+"  "+surname);
```

STRING Arrays

- An array of strings can also be created

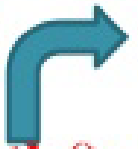
```
String cities [ ] = new String[5];
```

- Will create an array of CITIES of size 5 to hold string constants

String Methods

- The **String** class contains many useful methods for string-processing applications.
  - A **String** method is called by writing a **String** object, a dot, the name of the method, and a pair of parentheses to enclose any arguments
  - If a **String** method returns a value, then it can be placed anywhere that a value of its type can be used

```
String greeting = "Hello";
int count = greeting.length();
System.out.println("Length is " + greeting.length());
```



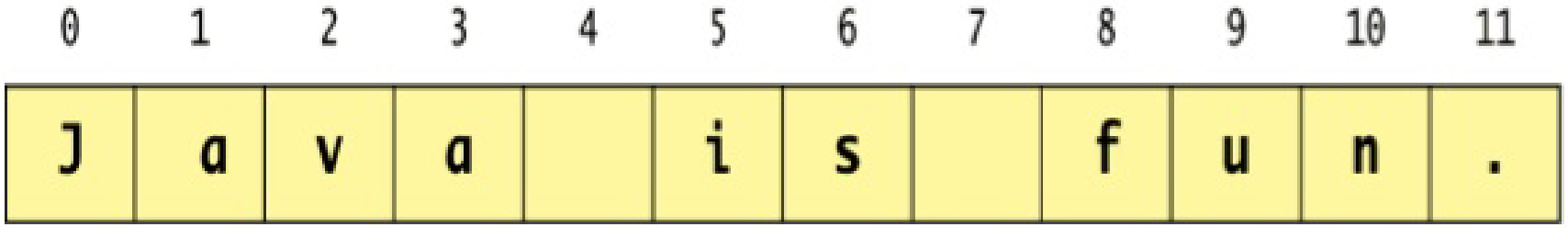
String method

  - Always count from zero when referring to the *position* or *index* of a character in a string

String Indexes

Display 1.5 String Indexes

The 12 characters in the string `"Java is fun."` have indexes 0 through 11.



Notice that the blanks and the period count as characters in the string.

Some Methods in the Class `String` (Part 1 of 8)

Display 1.4 Some Methods in the Class `String`

```
int length()
```

Returns the length of the calling object (which is a string) as a value of type `int`.

EXAMPLE

After program executes `String greeting = "Hello!";`
`greeting.length()` returns 6.

```
boolean equals(Other_String)
```

Returns `true` if the calling object string and the *Other_String* are equal. Otherwise, returns `false`.

EXAMPLE

After program executes `String greeting = "Hello";`
`greeting.equals("Hello")` returns `true`
`greeting.equals("Good-Bye")` returns `false`
`greeting.equals("hello")` returns `false`

Note that case matters. `"Hello"` and `"hello"` are not equal because one starts with an uppercase letter and the other starts with a lowercase letter.

(continued)

Some Methods in the Class `String` (Part 2 of 8)

Display 1.4 Some Methods in the Class `String`

`boolean equalsIgnoreCase(`*Other_String*`)`

Returns `true` if the calling object string and the *Other_String* are equal, considering uppercase and lowercase versions of a letter to be the same. Otherwise, returns `false`.

EXAMPLE

After program executes `String name = "mary!";`
`greeting.equalsIgnoreCase("Mary!")` returns `true`

`String toLowerCase()`

Returns a string with the same characters as the calling object string, but with all letter characters converted to lowercase.

EXAMPLE

After program executes `String greeting = "Hi Mary!";`
`greeting.toLowerCase()` returns `"hi mary!"`.

(continued)

Some Methods in the Class `String` (Part 3 of 8)

Display 1.4 Some Methods in the Class `String`

`String toUpperCase()`

Returns a string with the same characters as the calling object string, but with all letter characters converted to uppercase.

EXAMPLE

After program executes `String greeting = "Hi Mary!";`
`greeting.toUpperCase()` returns `"HI MARY!"`.

`String trim()`

Returns a string with the same characters as the calling object string, but with leading and trailing white space removed. Whitespace characters are the characters that print as white space on paper, such as the blank (space) character, the tab character, and the new-line character `'\n'`.

EXAMPLE

After program executes `String pause = "  Hmm  ";`
`pause.trim()` returns `"Hmm"`.

(continued)

Some Methods in the Class `String` (Part 4 of 8)

Display 1.4 Some Methods in the Class `String`

`char charAt(`*Position*`)`

Returns the character in the calling object string at the *Position*. Positions are counted 0, 1, 2, etc.

EXAMPLE

After program executes `String greeting = "Hello!";`
`greeting.charAt(0)` returns `'H'`, and
`greeting.charAt(1)` returns `'e'`.

`String substring(`*Start*`)`

Returns the substring of the calling object string starting from *Start* through to the end of the calling object. Positions are counted 0, 1, 2, etc. Be sure to notice that the character at position *Start* is included in the value returned.

EXAMPLE

After program executes `String sample = "AbcdefG";`
`sample.substring(2)` returns `"cdefG"`.

(continued)

Some Methods in the Class `String` (Part 5 o

Display 1.4 **Some Methods in the Class `String`**

`String substring(`*Start*`, `*End*`)`

Returns the substring of the calling object string starting from position *Start* through, but not including, position *End* of the calling object. Positions are counted 0, 1, 2, etc. Be sure to notice that the character at position *Start* is included in the value returned, but the character at position *End* is not included.

EXAMPLE

After program executes `String sample = "AbcdefG";`
`sample.substring(2, 5)` returns `"cde"`.

`int indexOf(`*A_String*`)`

Returns the index (position) of the first occurrence of the string *A_String* in the calling object string. Positions are counted 0, 1, 2, etc. Returns `-1` if *A_String* is not found.

EXAMPLE

After program executes `String greeting = "Hi Mary!";`
`greeting.indexOf("Mary")` returns 3, and
`greeting.indexOf("Sally")` returns `-1`.

(continued)

Some Methods in the Class `String` (Part 8 of 8)

Display 1.4 Some Methods in the Class `String`

```
int indexOf(A_String, Start)
```

Returns the index (position) of the first occurrence of the string *A_String* in the calling object string that occurs at or after position *Start*. Positions are counted 0, 1, 2, etc. Returns −1 if *A_String* is not found.

EXAMPLE

After program executes `String name = "Mary, Mary quite contrary";`
`name.indexOf("Mary", 1)` returns 6.
The same value is returned if 1 is replaced by any number up to and including 6.
`name.indexOf("Mary", 0)` returns 0.
`name.indexOf("Mary", 8)` returns −1.

```
int lastIndexOf(A_String)
```

Returns the index (position) of the last occurrence of the string *A_String* in the calling object string. Positions are counted 0, 1, 2, etc. Returns −1, if *A_String* is not found.

EXAMPLE

After program executes `String name = "Mary, Mary, Mary quite so";`
`greeting.indexOf("Mary")` returns 0, and
`name.lastIndexOf("Mary")` returns 12.

(continued)

Display 1.4 **Some Methods in the Class `String`**

`int compareTo(A_String)`

Compares the calling object string and the string argument to see which comes first in the lexicographic ordering. Lexicographic order is the same as alphabetical order but with the characters ordered as in Appendix 3. Note that in Appendix 3 all the uppercase letters are in regular alphabetical order and all the lowercase letters are in alphabetical order, but all the uppercase letters precede all the lowercase letters. So, lexicographic ordering is the same as alphabetical ordering provided both strings are either all uppercase letters or both strings are all lowercase letters. If the calling string is first, it returns a negative value. If the two strings are equal, it returns zero. If the argument is first, it returns a positive number.

EXAMPLE

After program executes `String entry = "adventure";`
`entry.compareTo("zoo")` returns a negative number,
`entry.compareTo("adventure")` returns `0`, and
`entry.compareTo("above")` returns a positive number.

(continu

```
int compareToIgnoreCase(A_String)
```

Compares the calling object string and the string argument to see which comes first in the lexicographic ordering, treating uppercase and lowercase letters as being the same. (To be precise, all uppercase letters are treated as if they were their lowercase versions in doing the comparison.) Thus, if both strings consist entirely of letters, the comparison is for ordinary alphabetical order. If the calling string is first, it returns a negative value. If the two strings are equal ignoring case, it returns zero. If the argument is first, it returns a positive number.

EXAMPLE

After program executes `String entry = "adventure";`
`entry.compareToIgnoreCase("Zoo")` returns a negative number,
`entry.compareToIgnoreCase("Adventure")` returns `0`, and
`"Zoo".compareToIgnoreCase(entry)` returns a positive number.

STRING BUFFER CLASS

- STRINGBUFFER class creates strings flexible length that can be modified in terms of both length and content.
- STRINGBUFFER may have characters and substrings inserted in the middle or appended to the end.
- STRINGBUFFER automatically grows to make room for such additions

Actually STRINGBUFFER has more characters pre allocated than are actually needed, to allow room for growth

STRING BUFFER CONSTRUCTORS

- **String Buffer():-** Reserves room fro 16 characters without reallocation

- **StringBuffer(int size):-** Accepts an integer argunent that explicilty sets the size of the buffer

- **StringBuffer(String str):-** Accepts STRING argument that sets the initial contents of the STRINGBUFFER and allocated room for 16 additional characters.

STRING BUFFER FUNCTIONS

- **length():-**Returns the current length of the string.
- **capacity():-** Returns the total allocated capacity.
- **void ensureCapacity():-** Preallocates room for a certain number of characters.
- **void setLength(int *len*):-** Sets the length of the string s1 to len.
  If len<s1.length(), s1 is truncated.
  If len>s1.length(), zeros are added to s1.
- **charAt(int *where*):-** Extracts value of a single character.
- **setCharAt(int *where*, char *ch*):-** Sets the value of character at specified position.

StringBuffer capacity() method

The capacity() method of StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

StringBuffer ensureCapacity() method

The ensureCapacity() method of StringBuffer class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

STRING BUFFER FUNCTIONS

- **append*(s2)*:-** Appends string s2 to s1 at the end.
- **insert*(n,s2)*:-** Inserts the string s2 at the position n of the string s1
- **reverse()**:- Returns the reversed object on when it is called.
- **delete*(int n1,int n2)*:-** Deletes a sequence of characters from the invoking object.

n1 → Specifies index of first character to remove

n2 → Specifies index one past the lastcharacter to remove

- **deleteCharAt*(int loc)*:-** Deletes the character at the index

STRING BUFFER FUNCTIONS

- *replace(int n1,int n2,String s1):-* Replaces one set of characters with another set.
- substring(*int startIndex*):- Returns the substring that starts at starts at *startIndex* and runs to the end.
- substring(*int startIndex, int endIndex*):- Returns the substring that starts at starts at *startIndex* and runs to the *endIndex-1*

Difference between String and StringBuffer :

No	String	StringBuffer
1)	String class is immutable.	StringBuffer class is mutable.
2)	String is slow and consumes more memory when you concat too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when you concat strings.
3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.


Thank you!
shutterstock.com · 567687052

STRING TOKENIZER IN JAVA

By
M. BABY ANUSHA,
ASST.PROF IN CSE DEPT.,
RGUKT,NUZVID

STRINGTOKENIZER CLASS :

- The **java.util.StringTokenizer** class allows you to break a string into tokens. It is simple way to break string.
- This class is a legacy class that is retained for compatibility reasons although its use is discouraged in new code.
- Its methods do not distinguish among identifiers, numbers, and quoted strings.
- This class methods do not even recognize and skip comments.

CLASS DECLARATION :

- Following is the declaration for **java.util.StringTokenizer** class −

```
public class StringTokenizer extends Object
{
    //code to be implemented;
}
```

CLASS CONSTRUCTORS :

- **StringTokenizer(String str)**

This constructor a string tokenizer for the specified string.

- **StringTokenizer(String str, String delim)**

This constructor constructs string tokenizer for the specified string.

- **StringTokenizer(String str, String delim, boolean returnDelims)**

This constructor constructs a string tokenizer for the specified string.

CLASS METHODS

- int countTokens() :This method calculates the number of times that this tokenizer's nextToken method can be called i.e., returns the total number of tokens.
- boolean hasMoreTokens() :This method tests if there are more tokens available from this tokenizer's string.
- boolean hasMoreElements() :This method returns the same value as the hasMoreTokens method.

CLASS METHODS

- String nextToken() :This method returns the next token from this string tokenizer.
- Object nextElement() :This method returns the same value as the nextToken method, except that its declared return value is Object rather than String.
- String nextToken(String delim) :This method returns the next token in this string tokenizer's string.

SAMPLE PROGRAM:

```
import java.util.*;
 public class STDemo
 {
   public static void main(String[] args)
   {
  // creating stringtokenizer
   StringTokenizer st = new StringTokenizer("Come to learn");
  // checking next token
   System.out.println("Next token is : " + st.nextToken());
    }
}
```

Output :

Next token is : Come


Thank you!
shutterstock.com · 567687052

Java static keyword

By
M. BABY ANUSHA,
ASST.PROF IN CSE DEPT.,
RGUKT,NUZVID

Why main method (only) static in java?

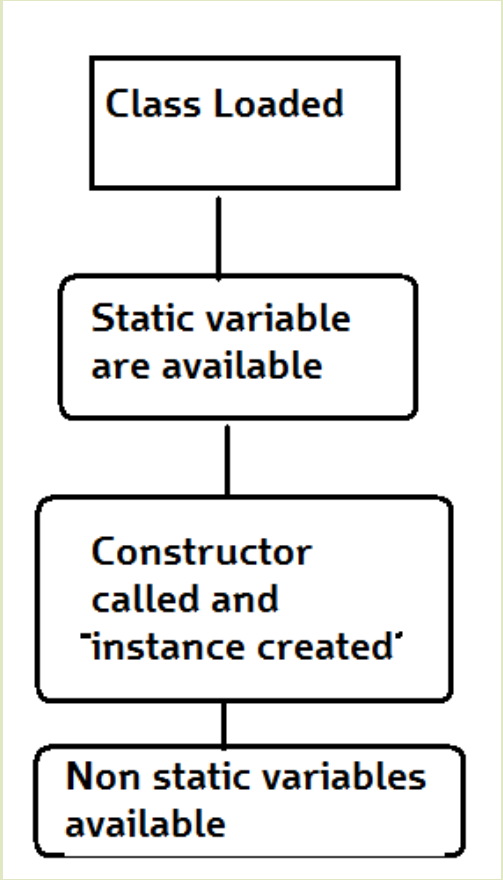
Guess 

How objects are created?

- Basic Steps of how objects are created

1. Class is loaded by JVM
2. Static variable and methods are loaded and initialized and available for use
3. Constructor is called to instantiate the non static variables
4. Non static variables and methods are now available

- As all the non static variable are available only after the constructor is called, there is a restriction on using non static variable in static methods.



what's the use of static Keyword and Why?

- The static keyword is used when a member variable of a class has to be shared between all the instances of the class.
- All static variables and methods belong to the class and not to any instance of the class

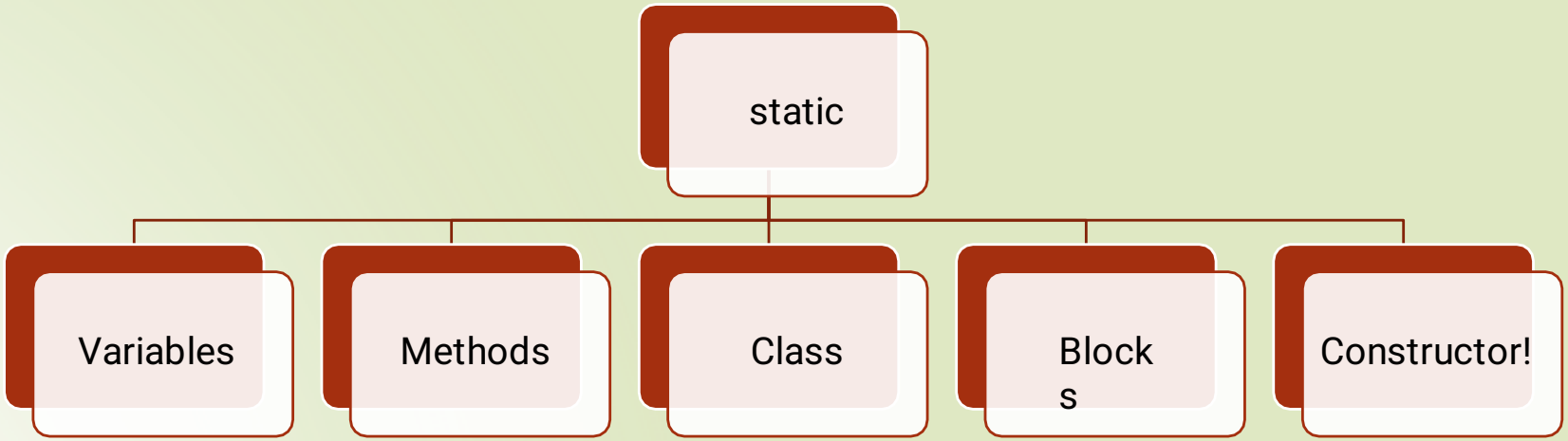
Why ?

- The **static keyword** in java is used for memory management mainly.
- Static keyword is used to save your time and typing .

Static and Non-static

- We can access static variables without creating an object of the class
- As they are already available at class loading time.
- We cannot use non static methods and variables without creating an object of the class as they are bound to the instance of the class.
- They are initialized by the constructor when we create the object using new operator.

The static can be:

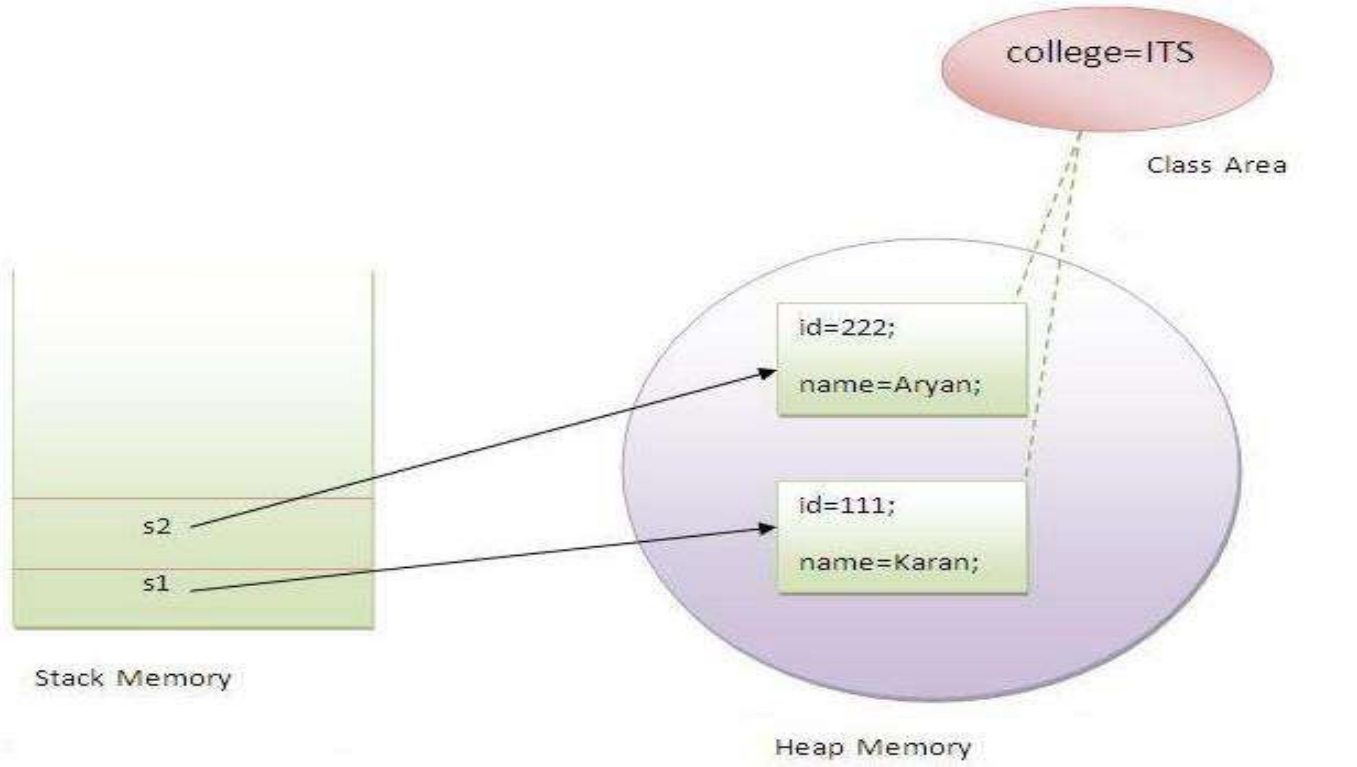


Static Variables

- The static variable can be used to refer the common property of all objects (that is not unique for each object).
- The static variable gets memory only once in class area at the time of class loading.
- **Advantage of static variable :-**

  It makes your program **memory efficient** (i.e. it saves memory).

Static Variables



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.

Static Classes (Only Nested Class)

- A Class can be made **static** only if it is a nested Class.
- The nested static class can be accessed without having an object of outer class.

Static Blocks

- Static block is mostly used for changing the default values of static variables.
- This block gets executed when the class is loaded in the memory.
- A class can have multiple Static blocks, which will execute in the same
sequence in which they have been written into the program.

Static Constructors

- They are not allowed in Java.

Thank You...

this keyword in Java

By
M. BABY ANUSHA,
ASST.PROF IN CSE DEPT.,
RGUKT,NUZVID

Introduction

- The this **keyword** refers to the current object in a method or constructor.
- The most common use of the this **keyword** is to eliminate the confusion between class attributes and parameters with the same name (because a class attribute is shadowed by a method or constructor parameter).

Continued…

- Following are the usage of this keyword.
- this can be used to get the current object.
- this can be used to invoke current object's method.
- this() can be used to invoke current class constructor
- this can be passed as a parameter to a method call.
- this can be passed as a parameter to a constructor.
- this can be used to return the current object from the method.



shutterstock.com · 567687052

Type conversion & casting in Java

By
M. BABY ANUSHA,
ASST.PROF IN CSE DEPT.,
RGUKT,NUZVID.

Type conversion :

- When you assign value of one data type to another, the two types might not be compatible with each other.
- If the data types are compatible, then Java will perform the conversion automatically known as Automatic Type Conversion and if not then they need to be casted or converted explicitly.
- For example, assigning an int value to a long variable.

Continued…

- In Java, there are two types of casting:

1. **Widening Casting** (automatically) - converting a smaller type to a larger type size
   byte -> short -> char -> int -> long -> float -> double

2. **Narrowing Casting** (manually) – converting a larger type to a smaller size type
   double -> float -> long -> int -> char -> short ->

Widening or Automatic Type Conversion

Widening conversion takes place when two data types are automatically converted. This happens when:

- The two data types are compatible.
- When we assign value of a smaller data type to a bigger data type.

Continued…

- For Example, in java the numeric data types are compatible with each other but no automatic conversion is supported from numeric type to char or boolean.
- Also, char and boolean are not compatible with each other.

Byte –> Short –> Int –> Long – > Float –> Double

Widening or Automatic Conversion

Widening or Automatic Type Conversion- Example 1

```
class Test
{
    public static void main(String[] args)
    {
        int i = 100;

        // automatic type conversion
        long l = i;

        // automatic type conversion
        float f = l;
        System.out.println("Int value "+i);
        System.out.println("Long value "+l);
        System.out.println("Float value "+f);
    }
}
```

Output :
Int value 100
Long value 100
Float value 100.0

Widening or Automatic Type Conversion- Example 2

```
public class Main
{
   public static void main(String[] args)
   {
       int myInt = 9;
       double myDouble = myInt;
      // Automatic casting: int to double
  System.out.println(myInt); // Outputs 9          System.out.
println(myDouble);  // Outputs 9.0
 }
}
```

Output:

9

9.0

Narrowing or Explicit Conversion :

- If we want to assign a value of larger data type to a smaller data type we perform explicit type casting or narrowing.
- This is useful for incompatible data types where automatic conversion cannot be done.
- Here, target-type specifies the desired type to convert the specified value to.

Continued…

- Narrowing casting must be done manually by placing the type in parentheses in front of the value.
- For Example, char and number are not compatible with each other. Let's see when we try to convert one into other.

Double –> Float –> Long –> Int –> Short –> Byte

Narrowing or Explicit Conversion

Example Without Explicit Conversion:

```
// Java program to illustrate incompatible data
// type for explicit type conversion
public class Test
{
  public static void main(String[] argv)
  {
    char ch = 'c';
    int num = 88;
    ch = num;
  }
}
```

Output:

Error:

7: error: incompatible types: possible lossy conversion from int to char ch = num; ^ 1 error

Narrowing or Explicit Conversion : Example 1

```
// Java program to illustrate explicit type conversion
class Test
{
    public static void main(String[] args)
    {
        double d = 100.04;
        //explicit type casting
        long l = (long)d;
        //explicit type casting
        int i = (int)l;
        System.out.println("Double value "+d);
        //fractional part lost
        System.out.println("Long value "+l);
        //fractional part lost
        System.out.println("Int value "+i);
    }
}
```

Output:
Double value 100.04
Long value 100
Int value 100

Narrowing or Explicit Conversion : Example 2

```
//Java program to illustrate Conversion of int and double to byte
class Test
{
    public static void main(String args[])
    {
        byte b;
        int i = 257;
        double d = 323.142;
        System.out.println("Conversion of int to byte.");
        //i%256
        b = (byte) i;
        System.out.println("i = " + i + " b = " + b);
        System.out.println("\nConversion of double to byte.");
        //d%256
        b = (byte) d;
        System.out.println("d = " + d + " b= " + b);
    }
}
```

Output:

```
Conversion of int to byte.
i = 257
b = 1
Conversion of double to byte.
d = 323.142
b = 67
```

Narrowing or Explicit Conversion : Example 2

```
public class Main
 {
public static void main(String[] args)
{
double myDouble = 9.78d;
 int myInt = (int) myDouble; // Manual casting: double to int
System.out.println(myDouble); // Outputs 9.78 System.out.println(myInt); // Outputs 9
}
}
```

Output:

9.78
9


Thank you!
shutterstock.com · 567687052

Wrapper Classes

By
M. BABY ANUSHA,
ASST.PROF IN CSE DEPT.,
RGUKT,NUZVID

INTRODUCTION

Java uses primitive types, such as int, char, double to hold the basic data types supported by the language.

Sometimes it is required to create an object representation of these primitive types.

These are collection classes that deal only with such objects. One needs to wrap the primitive type in a class.

Java2all http://www.java2all.com

To satisfy this need, java provides classes that correspond to each of the primitive types. Basically, these classes encapsulate, or wrap, the primitive types within a class.

Thus, they are commonly referred to as type wrapper. Type wrapper are classes that encapsulate a primitive type within an object.

The wrapper types are **Byte, Short, Integer, Long, Character, Boolean, Double, Float.**

These classes offer a wide array of methods that allow to fully integrate the primitive types into Java's object hierarchy.

Java2all http://www.java2all.com

WRAPPER CLASS

- Wrapper classes are classes that allow primitive types to be accessed as objects.

- Wrapper class is wrapper around a primitive data type because they "wrap" the primitive data type into an object of that class.

WHAT IS WRAPPER CLASS?

- Each of Java's eight primitive data types has a class dedicated to it.
- They are one per primitive type: Boolean, Byte, Character, Double, Float, Integer, Long and Short.
- Wrapper classes make the primitive type data to act as objects.

PRIMITIVE DATA TYPES AND WRAPPER CLASSES

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

Converting primitive numbers to Object numbers using constructor methods

Constructor calling	Conversion Action
Integer IntVal = new Integer(i);	Primitive integer to Integer object
Float FloatVal = new Float(f);	Primitive float to Float object
Double DoubleVal = new Double(d);	Primitive double to Double object
Long LongVal = new Long(l);	Primitive long to Long object

Java2all http://www.java2all.com

Converting Numeric Strings to Primitive numbers using Parsing method

Method calling	Conversion Action
int i = Integer.parseInt(str);	Converts String str into primitive integer i
long l = Long.parseLong(str);	Converts String str into primitive long l

Java2all http://www.java2all.com

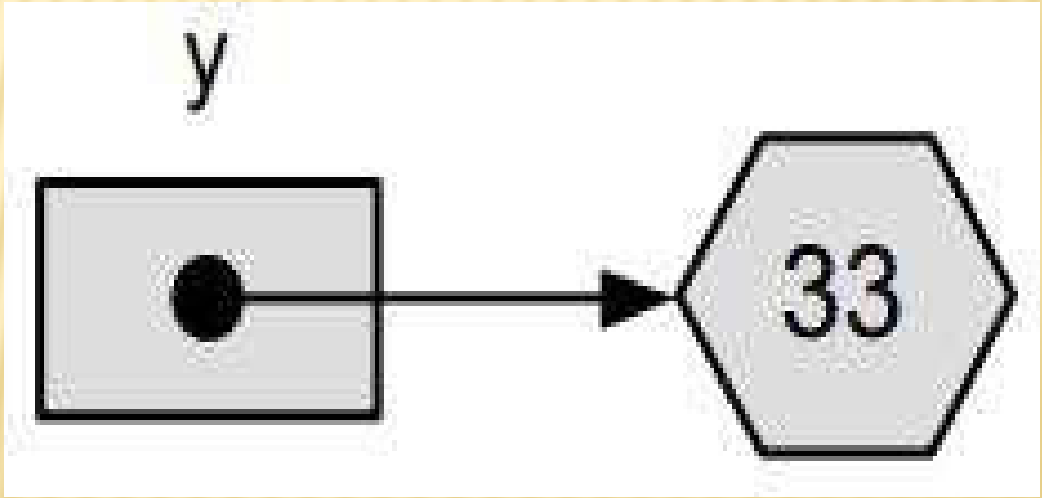
Difference b/w Primitive Data Type and Object of a Wrapper Class

- The following two statements illustrate the difference between a primitive data type and an object of a wrapper class:

```
int x = 25;

Integer y = new Integer(33);
```

- The second statement instantiates an Integer object.
- The object is initialized with the value 33 and a reference to the object is assigned to the object variable y.



- Clearly x and y differ by more than their values:
- x is a variable that holds a value.
- y is an object variable that holds a reference to an object.

WHAT IS THE NEED OF WRAPPER CLASSES

- Wrapper classes are used to be able to use the primitive data- types as objects.
- Many utility methods are provided by wrapper classes.

To get these advantages we need to use wrapper classes.

BOXING AND UNBOXING :

- The wrapping is done by the compiler.
- if we use a primitive where an object is expected, the compiler boxes the primitive in its wrapper class.
- Similarly, if we use a number object when a primitive is expected, the compiler un-boxes the object.

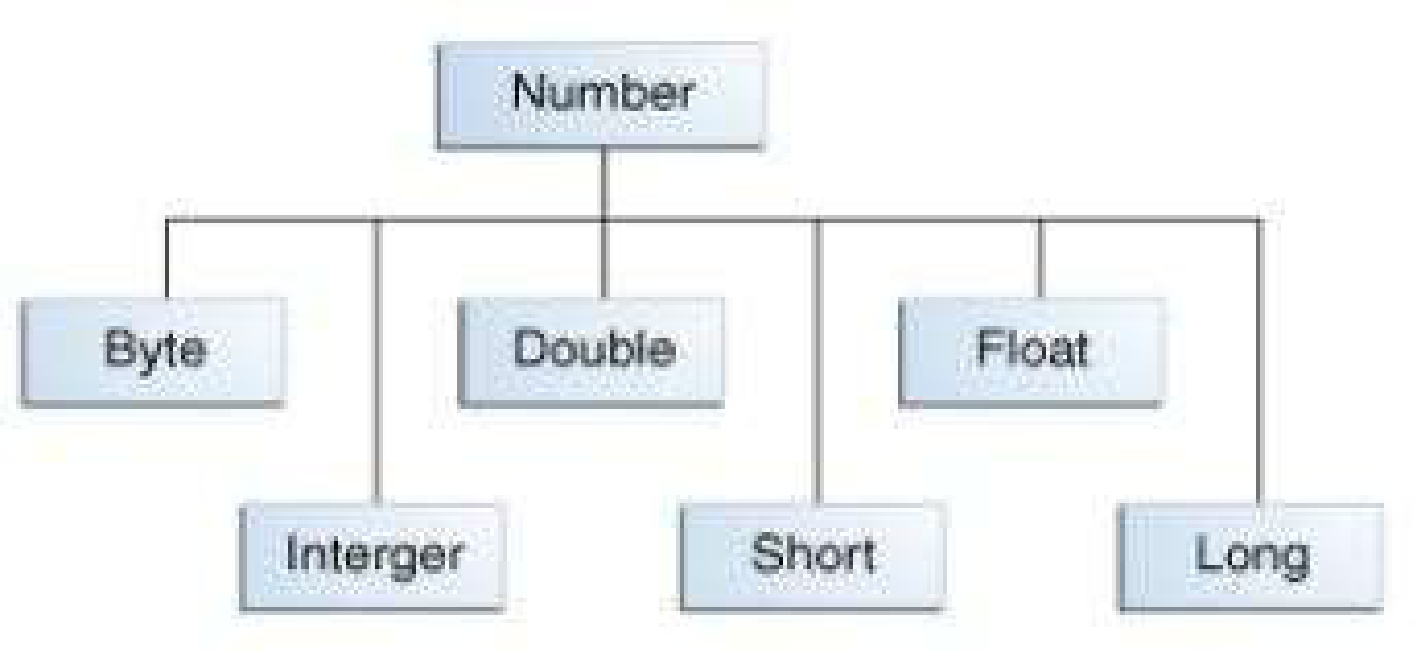
Example of boxing and unboxing:

- Integer x, y; x = 12; y = 15;System.out.println(x+y);
- When x and y are assigned integer values, the compiler boxes the integers
  because x and y are integer objects.
- In the println() statement, x and y are unboxed so that they can be added as integers.

NUMERIC WRAPPER CLASSES :

- All of the numeric wrapper classes are subclasses of the abstract class Number .

- Short, Integer, Double and Long implement Comparable interface.



The Byte class encapsulates a byte value. It defines the constants MAX_VALUE and MIN_VALUE and provides these constructors:

Byte(**byte b**)
Byte(**String str**)

Here, b is a byte value and str is the string equivalent of a byte value.

Java2all http://www.java2all.com

EX :

```
import java.util.*;
public class Byte_Demo
{
   public static void main(String args[])
   {
      Byte b1 = new Byte((byte)120);
      for(int i = 125; i<=135; i++)
      {
         Byte b2 = new Byte((byte)i);
         System.out.println("b2 = " + b2);
      }
      System.out.println("b1 Object = " + b1);
      System.out.println("Minimum Value of Byte = " + Byte.MIN_VALUE);
      System.out.println("Maximum Value of Byte = " + Byte.MAX_VALUE);
      System.out.println("b1* 2 = " + b1*2);
      System.out.println("b1* 2 = " + b1.byteValue()*2);
      Byte b3 = new Byte("120");
      System.out.println("b3 Object = " + b3);
      System.out.println("(b1==b3)? " + b1.equals(b3));
      System.out.println("(b1.compareTo(b3)? " + b1.compareTo(b3));
       /*Returns 0 if equal. Returns -1 if b1 is less than b3 and 1 if b1 is
      greater than 1*/
   }
}
```

Java2all http://www.java2all.com

Output :

```
b2 = 125
b2 = 126
b2 = 127
b2 = -128
b2 = -127
b2 = -126
b2 = -125
b2 = -124
b2 = -123
b2 = -122
b2 = -121
b1 Object = 120
Minimum Value of Byte = -128
Maximum Value of Byte = 127
b1* 2 = 240
b1* 2 = 240
b3 Object = 120
(b1==b3)? true
(b1.compareTo(b3)? 0
```

Java2all http://www.java2all.com

The Short class encapsulates a short value. It defines the constants **MAX_VALUE** and **MIN_VALUE** and provides the following constructors:

Short(**short s**)
Short(**String str**)

Java2all http://www.java2all.com

EX :

```
import java.util.*;
public class Short_Demo
{
   public static void main(String args[])
   {
      Short s1 = new Short((short)2345);
      for(int i = 32765; i<=32775; i++)
      {
         Short s2 = new Short((short)i);
         System.out.println("s2 = " + s2);
      }
      System.out.println("s1 Object = " + s1);
      System.out.println("Minimum Value of Short = " + Short.MIN_VALUE);
      System.out.println("Maximum Value of Short = " + Short.MAX_VALUE);
      System.out.println("s1* 2 = " + s1.shortValue()*2);
      Short s3 = new Short("2345");
      System.out.println("s3 Object = " + s3);
      System.out.println("(s1==s3)? " + s1.equals(s3));
      Short s4 = Short.valueOf("10", 16);
      System.out.println("s4 Object = " + s4);
   }
}
```

Java2all http://www.java2all.com

Output :

```
s2 = 32765
s2 = 32766
s2 = 32767
s2 = -32768
s2 = -32767
s2 = -32766
s2 = -32765
s2 = -32764
s2 = -32763
s2 = -32762
s2 = -32761
s1 Object = 2345
Minimum Value of Short = -32768
Maximum Value of Short = 32767
s1* 2 = 4690
s3 Object = 2345
(s1==s3)? true
s4 Object = 16
```

Java2all http://www.java2all.com

Integer class :

The Integer class encapsulates an integer value. This class provides following constructors:

Integer(**int i**)
Integer(**String str**)

Here, i is a simple int value and str is a String object.

Java2all http://www.java2all.com

EX :

```
import java.util.*;
public class Int_Demo
{
   public static void main(String args[])
   {
      Integer i1 = new Integer(12);
      System.out.println("I1 = " + i1);
      System.out.println("Binary Equivalent = " + Integer.toBinaryString(i1));
      System.out.println("Hexadecimal Equivalent = " + Integer.toHexString(i1));
      System.out.println("Minimum Value of Integer = " + Integer.MIN_VALUE);
      System.out.println("Maximum Value of Integer = " + Integer.MAX_VALUE);
      System.out.println("Byte Value of Integer = " + i1.byteValue());
      System.out.println("Double Value of Integer = " + i1.doubleValue());
      Integer i2 = new Integer(12);
      System.out.println("i1==i2 " + i1.equals(i2));
      System.out.println("i1.compareTo(i2) = " + i2.compareTo(i1));
      // Compareto - if it is less than it returns -1 else 1, if equal it return 0.
      Integer i3 = Integer.valueOf("11", 16);
      System.out.println("i3 = " + i3);
   }
}
```

Java2all http://www.java2all.com

Output :

```
I1 = 12
Binary Equivalent = 1100
Hexadecimal Equivalent = c
Minimum Value of Integer = -2147483648
Maximum Value of Integer = 2147483647
Byte Value of Integer = 12
Double Value of Integer = 12.0
i1==i2 true
i1.compareTo(i2) = 0
i3 = 17
```

Java2all http://www.java2all.com

Long :

The Long class encapsulates a long value. It defines the constants MAX_VALUE and MIN_VALUE and provides the following constructors:

Long(**long l**)
Long(**String str**)

Java2all http://www.java2all.com

EX :

```
import java.util.*;
 public class Long_Demo
{
   public static void main(String args[])
   {
      Long L1 = new Long(68764);
      Long L2 = new Long("686748");
      System.out.println("Object L1 = " + L1);
      System.out.println("Object L2 = " + L2);
      System.out.println("Minimum Value of Long = " + Long.MIN_VALUE);
      System.out.println("Maximum Value of Long = " + Long.MAX_VALUE);
      System.out.println("L1 * 2 = " + L1 * 2);
      System.out.println("L1.longValue() * 2 = " + L1.longValue() * 2);
      System.out.println("L1.compareTo(l2) = " + L1.compareTo(L2));
      System.out.println("L1==L2 ? = " + L1.equals(L2));
      Long L3 = Long.valueOf("10", 16);
      System.out.println("Object L3 = " + L3);
      System.out.println("Byte value of Long = " + L1.byteValue());
      System.out.println("int value of Long = " + L1.intValue());
      System.out.println("Double value of Long = " + L1.doubleValue());
      int i = 12;
      System.out.println("Binary equivalent of decimal " + i + "=" + Long.toBinaryString(i));
      System.out.println("Hexadecimal equivalent of decimal " + i + "=" + Long.toHexString(i));

   }
}
```

Java2all http://www.java2all.com

Output :

```
Object L1 = 68764
Object L2 = 686748
Minimum Value of Long = -9223372036854775808
Maximum Value of Long = 9223372036854775807
L1 * 2 = 137528
L1.longValue() * 2 = 137528
L1.compareTo(l2) = -1
L1==L2 ? = false
Object L3 = 16
Byte value of Long = -100
int value of Long = 68764
Double value of Long = 68764.0
Binary equivalent of decimal 12=1100
Hexadecimal equivalent of decimal 12=c
```

Java2all http://www.java2all.com

Double class :

The Double class encapsulates a double value. It defines several constants.

The largest and smallest values are saved in MAX_VALUE and MIN_VALUE.

The constant NaN (Not a Number) indicates that a value is not a number.

Java2all http://www.java2all.com

If you divide a double number by zero, the result is NaN. This class defines these constructors:

Double(**double d**)
Double(**String str**)

Here, d is a double value to be encapsulated in a Double object. In the last form, str is the string representation of a double value.

Java2all http://www.java2all.com

EX :

```
import java.util.*;
class Double_Demo
{
   public static void main(String args[])
   {
      Double d1 = new Double(687642365.4563);
      Double d2 = new Double("686748");
      System.out.println("Object d1 = " + d1);
      System.out.println("Object d2 = " + d2);
      System.out.println("Minimum Value of Double = " + Double.MIN_VALUE);
      System.out.println("Maximum Value of Double = " + Double.MAX_VALUE);
      System.out.println("Byte value of Double = " + d1.byteValue());
      System.out.println("int value of Double = " + d1.intValue());
      System.out.println("Float value of Double = " + d1.floatValue());
      Double d3 = Double.parseDouble("765.89");
      System.out.println("Double value from the string \"765.89\"="+d3);

   }
}
```

Java2all http://www.java2all.com

Output :

```
Object d1 = 6.876423654563E8
Object d2 = 686748.0
Minimum Value of Double = 4.9E-324
Maximum Value of Double = 1.79769313486231 57E308
Byte value of Double = -3
int value of Double = 687642365
Float value of Double = 6.8764237E8
Double value from the string "765.89"=765.89
```

Java2all http://www.java2all.com

Float class :

The float class encapsulates a float value. It defines several constants the largest and smallest values are stored in MAX_VALUE and MIN_VALUE.

The constant NaN indicates that a value is not a number. If you divide a floating – point number by zero, the result is NaN.

Java2all http://www.java2all.com

This class defines these constructors:

Float(**float f**)
Float(**double d**)
Float(**String str**)

Here, f and d are float and double types to be encapsulated in a Float object.

str is the string representation of a float value.

Java2all http://www.java2all.com

EX :

```
import java.util.*;

public class Float_Demo
{
   public static void main(String args[])
   {
      Float f1 = new Float(123.5626);
      Float f2 = new Float(854.32f);
      Float i = new Float(10);
      System.out.println("Object f1 = " + f1);
      System.out.println("Object f2 = " + f2);
      System.out.println("Minimum Value of Float = " + Float.MIN_VALUE);
      System.out.println("Maximum Value of Float = " + Float.MAX_VALUE);
      System.out.println("Byte value of Float = " + f1.byteValue());
      System.out.println("Short value of Float = " + f1.shortValue());
      System.out.println("Integer value of Float = " + f1.intValue());
      System.out.println("Double value of Float = " + f1.doubleValue());
      System.out.println("(f1==f2) ?= " + f1.equals(f2));
      System.out.println("f1.compareTo(f2) = " + f1.compareTo(f2));
      System.out.println("f1 is not a number = " + i.isNaN());

   }
}
```

Java2all http://www.java2all.com

Output :

```
Object f1 = 123.5626
Object f2 = 854.32
Minimum Value of Float = 1.4E-45
Maximum Value of Float = 3.4028235E38
Byte value of Float = 123
Short value of Float = 123
Integer value of Float = 123
Double value of Float = 123.5625991821289
(f1==f2) ?= false
f1.compareTo(f2) = -1
f1 is not a number = false
```

Java2all http://www.java2all.com

Character class :

The Character class encapsulates a char value. This class provides the following constructor.

Character(**char ch**)

Here, c is a char value. charValue() method returns the char value that is encapsulated by a Character object and has the following form:

char charValue()

Java2all http://www.java2all.com

EX :

```
import java.util.*;

public class Char_Demo
{
   public static void main(String args[])
   {
      Character c1 = new Character('m');
      char c2 = 'O';
      System.out.println("Object C1 = " + c1);
      System.out.println("char value of Character Object = " + c1.charValue());
      System.out.println(c2 + " is defined character set ? " + Character.isDefined(c2));
      System.out.println("c2 is digit = " + Character.isDigit(c2));
      System.out.println("c2 is lowercase character = " + Character.isLowerCase(c2));
      System.out.println("c2 is uppercase character = " + Character.isUpperCase(c2));


   }
}
```

Java2all http://www.java2all.com

Output :

Object C1 = m
char value of Character Object = m
O is defined character set ? true
c2 is digit = false
c2 is lowercase character = false
c2 is uppercase character = true

Java2all http://www.java2all.com

Boolean class :

The Boolean class encapsulates a Boolean value. It defines FALSE and TRUE constants.

This class provides following constructors:

Boolean(**Boolean b**)
Boolean(**String str**)

Here, b is a Boolean value and str is the string equivalent of a Boolean value.

Java2all http://www.java2all.com

The methods associated with Boolean Class are as follows:

1. **Boolean booleanValue()**
2. **Boolean equals(Boolean b)**
3. **String toString(Boolean b)**

Java2all http://www.java2all.com

EX :

```
import java.util.*;

public class Boolean_Demo
{
   public static void main(String args[])
   {
      Boolean b1 = new Boolean(true);
      Boolean b2 = new Boolean(false);
      System.out.println("Object B1 = " + b1);
      System.out.println("Object B2 = " + b2);
      Boolean b3 = new Boolean("true");
      Boolean b4 = new Boolean("false");
      System.out.println("Object B3 = " + b3);
      System.out.println("Object B4 = " + b4);
      System.out.println("Boolean Value = " + b1.booleanValue());
      System.out.println("(b1==b2)? " + b1.equals(b2));
      String S1 = b1.toString();
      System.out.println("String S1 " + S1);

   }
}
```

Java2all http://www.java2all.com

Output :

Object B1 = true
Object B2 = false
Object B3 = true
Object B4 = false
Boolean Value = true
(b1==b2)? false
String S1 true


Thank you!
shutterstock.com · 567687052