SILVER OAK UNIVERSITY

EDUCATION TO INNOVATION

SILVER OAK COLLEGE OF COMPUTER APPLICATION

Subject: Core Java

Topics: Introduction of Java (UNIT 1)

By Prof. Ankit Patel

History of Java

- Java was initially developed in 1991 named as “oak” but was renamed “Java” in 1995.
- Originally designed for small, embedded systems in electronic appliances like set-top boxes.
- The primary motivation was the need for a platform-independent language that could be used to create software to be embedded in various consumer electronic devices.
- Java programming language was originally developed by **Sun Microsystems** which was initiated by **James Gosling** and released in 1995 as core component of Sun Microsystems' Java platform (Java 1.0 [J2SE]).
- It promised **Write Once, Run Anywhere** (WORA), providing no-cost run-times on popular platforms.
- Java 2, new versions had multiple configurations built for different types of platforms. J2EE included technologies and APIs for enterprise applications typically run in server environments, while J2ME featured APIs optimized for mobile applications.
- The desktop version was renamed J2SE. In **2006**, for marketing purposes, Sun renamed new J2 versions as Java EE, Java ME, and Java SE, respectively.
- On 13 November **2006**, Sun released much of Java as **free and open-source software** (FOSS), under the terms of the GNU **General Public License** (GPL).
- On 8 May **2007**, Sun finished the process, making all of Java's core code free and open-source, aside from a small portion of code to which Sun did not hold the copyright.

Write and explain Features of JAVA. OR Explain advantages of JAVA.

- Java promised “Write Once, Run Anywhere”, providing no-cost run-time on popular platform. Fairly secure and featuring configurable security, it allowed network and file access restriction.

1) Simple

- ✓ It’s simple because it contains many features of other languages like C and C++ and java **removes complexities** because it doesn’t use **pointers, Storage classes and Go To statement** and it also does not support **multiple Inheritance**.

2) Secure

- ✓ When we transfer the code from one machine to another machine, it will first check the code it is affected by the virus or not, it checks the safety of the code, if it contains virus then it will never execute that code.

3) Object-Oriented

- ✓ We know that all pure object oriented language, in them all of the code is in the form of classes and objects.
- ✓ This feature of java is most important and it also supports code reusability and maintainability etc.

4) Robust

- ✓ Two main reasons for program failures are:
  1. **Memory management mistake**

2. Mishandled exception or Run time errors

- ✓ Java does not support direct pointer manipulation. This resolves the java program to overwrite memory.
- ✓ Java manages the memory allocation and de-allocation itself. De-allocation is completely automatic, because Java provides garbage collection for unused objects.
- ✓ Java provides object-oriented exception handling. In a well written Java program all run-time errors can be managed by the program.

5) Multithreaded

- ✓ 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 multithreading is that it shares the same memory.
- ✓ Threads are important for multi-media, web application etc.

6) Distributed

- ✓ Java is designed for the distributed environment of the Internet, because it **handles TCP / IP** protocols.
- ✓ The widely used protocol like HTTP and FTP are developed in Java.
- ✓ Internet programmers can call functions on these protocols and can get access the files from any remote machine on the internet rather than writing code on their local system.

7) Architecture-Neutral

- ✓ It means that the programs written in one platform can run on any other platform without rewrite or recompile them. In other words it follows "**write once, run anywhere, any time, forever**" approach.
- ✓ Java program are compiled into bytecode format which does not depend on any machine architecture but can be easily translated into a specific machine by a **JVM** for that machine.
- ✓ This will be very helpful when applets or applications are developed which are download by any machine & run anywhere in any system.

8) Platform Independent

- ✓ It means when we compile a program in java, it will create a byte code of that program and that byte code will be executed when we run the program.
- ✓ It's not compulsory in java, that in which operating system we create java program, in the same operating system we have to execute the program.

9) Interpreted

- ✓ Most of the programming languages either compiled or interpreted, java is both compiled and interpreted.
- ✓ Java **compiler** translates a **java source file to byte code** and the java **interpreter** executes the **translated byte codes** directly on the system that implements the JVM.

10) High Performance

- ✓ Java programs are complied in to intermediate representation called **bytecode**, rather than to native machine level instructions and JVM executes Java bytecode on any machine on which JVM is installed.
- ✓ Java bytecode then translate directly into native machine code for very high performance by using a Just-In-Time compiler.
- ✓ So, Java programs are faster than program or scripts written in purely interpreted languages but slower than C and C++ programs that compiled to native machine languages.

11) Dynamic

- ✓ At the run time, java environment can extend itself by linking in classes that may be located on remote server on a network.
- ✓ At the run time, java interpreter performs name resolution while linking in the necessary classes.
- ✓ The java interpreter is responsible for determining the placement of object in the memory.

JDK (Java Development Kit)

- Java Developer Kit contains tools needed to develop the Java programs, and JRE to run the programs.
- The tools include compiler (javac.exe), Java application launcher (java.exe), Appletviewer, etc… Compiler converts java code into byte code.
- **Java application launcher** opens a **JRE**, loads the class, and invokes its main method.
- For running java programs, JRE is sufficient. JRE is targeted for execution of Java files i.e. **JRE = JVM + Java Packages Classes(like util, math, lang, awt,swing etc)+runtime libraries**.

JRE (Java Runtime Environment)

- Java Runtime Environment contains **JVM, class libraries, and other supporting files**.
- It does not contain any development tools such as compiler, debugger, etc.
- Actually, **JVM** runs the program, and it uses the class libraries, and other supporting files provided in JRE.

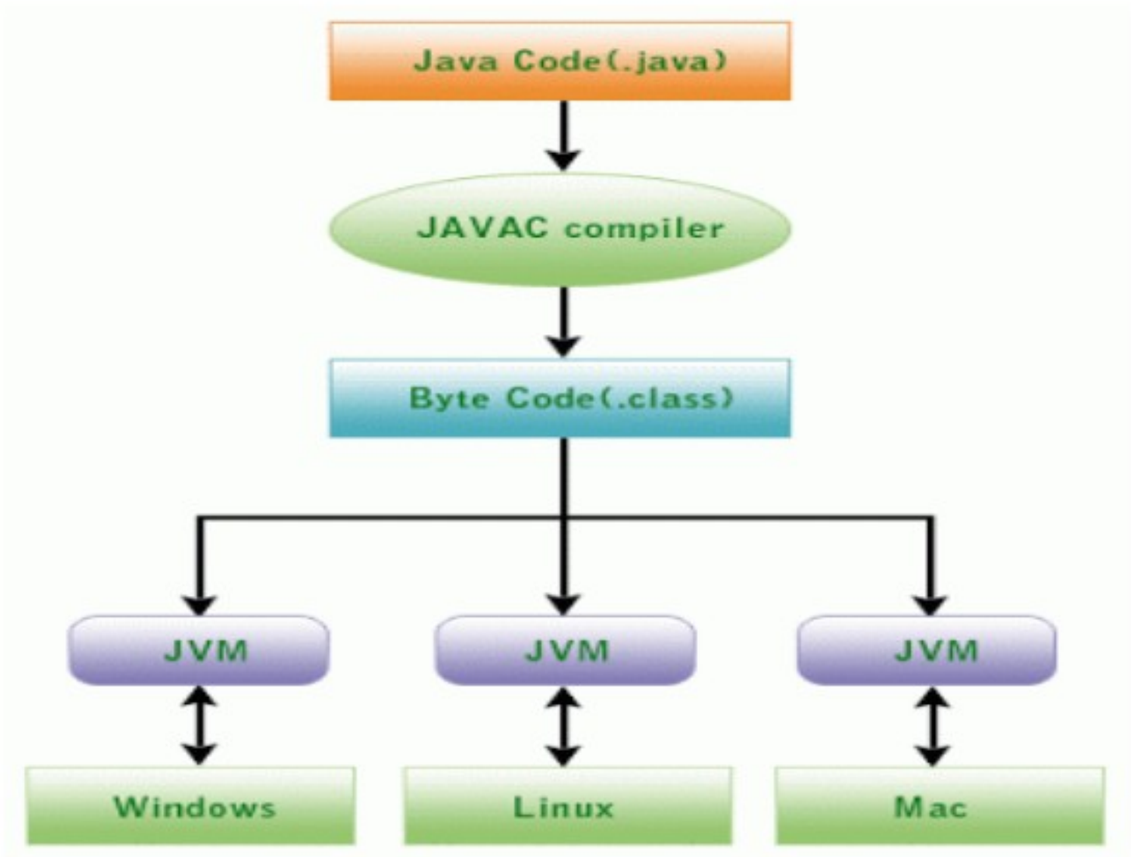
JVM (Java Virtual Machine)

- The JVM is called **virtual** because it provides a machine interface that does not depend on the operating system and machine hardware architecture.
- This independence from hardware and operating system is a cornerstone of the **write-once, run-anywhere** java programs.
- When we compile a Java file, output is not an **'.exe'** but it is a **'.class'** file.
- '.**class'** file consists of Java byte codes which are understandable by **JVM**.
- Java Virtual Machine **interprets** the byte code into the machine code depending upon the operating system and hardware combination.
- It is responsible for all the things like garbage collection, array bounds checking, etc…
- **JVM** itself is platform **dependent**.

- As of 2014 most JVMs use **JIT** (**Just in Time**) compiling, not interpreting, to achieve greater speed.

BYTE CODE

- **Bytecode** is nothing but the intermediate representation of Java source code which is produced by the Java compiler by compiling that source code.
- This byte code is a machine independent code. It is not completely a compiled code but it is an intermediate code somewhere in the middle which is **later interpreted and executed** by **JVM**.
- Bytecode is a machine code for JVM. But the machine code is platform specific whereas bytecode is platform independent that is the main difference between them.
- It is stored in .**class** file which is created after compiling the source code.



JAVA Environment Setup

- Setting up the path for windows: Assuming you have installed Java in **c:\Program Files\java\jdk** directory
- Right-click on **'My Computer'** and select **'Properties'**.
- Click on the **'Environment variables'** button under the **'Advanced'** tab.
- Now, Under **'System variables'** alter the **'Path'** variable so that it also contains the path to the Java executable.
- **Example**, if the path is currently set to 'C:\WINDOWS\SYSTEM32', then change your path to read 'C:\WINDOWS\SYSTEM32;c:\Program Files\java\jdk\bin'.

Describe the sample structure of JAVA program

- A Java Program may contain many **classes** of which only one class defines a **main** method.
- **Classes** contain **data members** and **methods** that operate on the data members of the class.
- **Methods** may contain data type declarations and executable statements.
- A Java Program may contain one or more sections as shown.

Documentation Section	Suggested
Package Statement	Optional
Import Statements	Optional

Interface Statement	Optional
Class Definitions	Optional

Main Method class
{
Main Method Definition
}
	Essential

Documentation Section

- The documentation section comprises a set of comment lines giving the name of the program, the author and other details, which the programmers would like to refer to at a later stage.
- Java also uses a comment such as /* …. */ know as documentation comment.
- This form is used for documentation automatically.

Package Statement

- The first statement allowed in Java file is a **package** statement.
- This statement declares a package name and informs the compiler that the classes defined here belong to this package.
  Example: Package Student;

Import Statements

- The next thing after a package statement may be a number of import statements. This is similar to the **#include** statement in C.
  Example: Import student.test;

Interface Statement

- An interface is like a **class** but includes a group of method declarations.
- This is also an optional section and is used only when we wish to implement the multiple inheritance features in the program.

Class Definition

- A **Java** Program may contain multiple class definitions.
- **Classes** are primary and essential elements of Java program.
- These classes are used to map the objects of real-world programs.

Main Method Class

- Since, every Java stand-alone program requires a **main** method as its starting point this is the essential part of Java program.
- A simple Java program may contain only this part.
- The main method creates objects of various classes and establishes communication between them.

Procedure-Oriented vs. Object-Oriented Programming

Procedure Oriented Programming (POP)	Object Oriented Programming (OOP)
• Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	• Importance is given to the data rather than procedures or functions.

Procedure Oriented Programming (POP)	Object Oriented Programming (OOP)
• **Top Down** approach in program design.	• **Bottom Up** approach in program design.
• Large programs are divided into smaller programs known as **functions.**	• Large programs are divided into **classes and objects**.
• POP does not have any access specifier.	• OOP has access specifier named **Public, Private, Protected**, etc..
• Most function uses **Global** data for sharing that can be accessed freely from function to function in the system.	• Data cannot move easily from **function to function**, it can be kept public or private so we can control the access of data.
• Adding of data and function is difficult.	• Adding of data and function is easy.
• Concepts like inheritance, polymorphism, data encapsulation, abstraction, access specifier are missing.	• Concepts like inheritance, polymorphism, data encapsulation, abstraction, access specifier are available and can be used easily.
• **Examples**: C, Fortran, Pascal, etc…	• **Examples**: C++, Java, C#, etc…

Basic Concepts of OOP

- Various concepts present in OOP to make it more **powerful, secure, reliable and easy**.

Object

- An **object** is an **instance of a class**.
- An **object** means anything from real world like as person, computer etc...
- Every object has at least one unique identity.
- An **object** is a component of a program that knows how to **interact** with other pieces of the program.
- An **object** is the **variable** of the type class.

Class

- A **class** is a template that specifies the attributes and behavior of objects.
- A class is a blueprint or prototype from which objects are created.
- Simply **class** is collection of objects.
- A class is the implementation of an **abstract data type** (ADT). It defines attributes and methods which implement the data structure and operations of the ADT, respectively.

Data Abstraction

- Just represent **essential** features without including the **background** details.
- Implemented in class to provide data security.
- We use **abstract** class and **interface** to achieve abstraction.

Encapsulation

- Wrapping up (Binding) of a data and functions into single unit is known as **encapsulation**.
- The data is not accessible to the outside world, only those functions can access it which is wrapped together within single unit.

Inheritance

- Inheritance is the process, by which class can acquire the properties and methods of another class.
- The mechanism of deriving a **new class** from an **old class** is called inheritance.
- The **new class** is called **derived** class and **old class** is called **base** class.
- The **derived** class may have all the features of the **base** class and the programmer can add new features to the derived class.

Polymorphism

- Polymorphism means the **ability to take more than one form**.
- It allows a single name to be used for more than one related purpose.
- It means ability of operators and functions to act differently in different situations.
- We use method overloading and method overriding to achieve polymorphism.

Message Passing

- A program contains set of object that communicates with each other.
- **Basic steps to communicate**
  1. Creating classes that define objects and their behavior.
  2. Creating objects from class definition
  3. Establishing communication among objects.

Dynamic and Static Binding

Static Binding	Dynamic Binding
• Type of the object is determined at **compiled** time (by the compiler), it is known as static binding.	• Type of the object is determined at **run-time**, it is known as dynamic binding.
• Static binding uses **Type** information for binding.	• Dynamic binding uses **Object** to resolve binding.
• **Static, private, final** methods and variables are resolved using static binding.	• **Virtual** methods are resolves during runtime based upon runtime object.
• **Overloaded** methods are resolve using static binding.	• **Overridden** methods are resolve using dynamic binding at runtime.

Write a Program to print "My First Program In Java" in java.

```
/* Write a Program to print "My First Program In Java" in java.*/
import  java.util.*;
class MyFirstProgram
{
        public static void main(String[] args)
        {
                // This statement will print the message.
                System.out.println("My First Program In Java");
        }
}
```

Primitive Data Types

- Primitive data types can be classified in four groups:

1) Integers :

- This group includes **byte**, **short**, **int**, and **long**.
- All of these are signed, **positive** and **negative** values.

DataType	Size	Example
byte	8-bit	byte b, c;
short	16-bit	short b,c;
int	32-bit	int b,c;
long	64-bit	long b,c;

2) Floating-point :

- This group includes **float** and **double**, which represent numbers with fractional precision.

DataType	Size	Example
float	32 bits	float a,b;
double	64 bits	double pi;

3) Characters :

- This group includes **char**, which represents symbols in a character set, like letters and numbers.
- Java **char** is a 16-bit type. The range of a **char** is 0 to 65,536. There are no negative **char**s.

  For example : char name = 'x';

4) Boolean :

- This group includes **boolean**, which is a special type for representing true/false values.
- It can have only one of two possible values, **true** or **false**.

  For example : boolean b = true;

Example:

```
class datatypeDemo
{
        public static void main(String args[])
        {
                int r=10;
                float pi=3.14f,a;                //You can also use here double data type
                char ch1=97,ch2='A' ;
                boolean x=true;
                a = pi*r*r;
                System.out.println("Areat of Circle is :: "+a);
                System.out.println("Ch1 and Ch2 are :: "+ch1+" "+ch2);
                System.out.println("Value of X is :: "+x);
        }
```

```
}
```

Output:

```
Area of Circle is :: 314.0
Ch1 and Ch2 are :: a A
Value of X is :: true
```

User Defined Data Type

Class

- A **class** is a template that specifies the attributes and behavior of things or objects.
- A **class** is a blueprint or prototype from which creates as many desired objects as required.

Example:

```
class Box
{
        double width=1;
        double height=2;
        double depth=3;
        void volume()
        {
                System.out.print("Volume is ");
                System.out.println(width * height * depth);
        }
}
class demo
{
        public static void main(String args[])
        {
                Box  b1 = new Box();
                b1.volume();
        }
}
```

Interface

- An **interface** is a collection of abstract methods.
- A **class** implements an interface, thereby inheriting the abstract methods of the interface.
- An **interface** is not a class. A class describes the attributes and behaviors of an object. An interface contains behaviors that a class implements.
- Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

Example:

```
interface  Animal
{
     public void eat();
     public void travel();
}
```

Identifiers and Literals

Identifiers

- Identifiers are used for class names, method names, and variable names.
- An identifier may be any descriptive sequence of **uppercase and lowercase letters, numbers, or the underscore characters and dollar-sign characters.**
- There are some rules to define identifiers as given below:
  1) Identifiers must start with a **letter or underscore** (_).
  2) Identifiers cannot start with a **number**.
  3) **White space(blank, tab, newline)** are not allowed.
  4) You can't use a **Java keyword** as an identifier.
  5) Identifiers in Java are **case-sensitive**; **foo** and **Foo** are two different identifiers.

Examples :

Valid Identifiers	Not Valid Identifiers
AvgNumber	2number
A1	int
$hello	-hello
First_Name	First-Name

Literals (Constants)

- A constant value in a program is denoted by a **literal**. Literals represent numerical (integer or floating-point), character, boolean or string values.

Integer	Floating-point	Character	Boolean	String
33, 0, -19	0.3, 3.14	(' 'R' 'r' '{'	(predefined values) true, false	“language”,”0.2” ,“r”

Variables

- A variable is defined by the combination of an **identifier**, a **type**, and an optional **initializer**.
- All variables have a scope, which is defines their visibility and lifetime.

Declaring of variable

- All variables must be declared before they can be used.

  Syntax:

  type identifier [= value][, identifier [= value] ...];
  - The **type** is one of Java’s **atomic types, or the name of a class or interface**.
  - The **identifier** is the name of the **variable**.
  - You can **initialize** the variable by specifying an **equal sign and a value**.
  - To declare **more than one variable** of the specified type, use a comma separated list.

  Example: int a, b, c = 10;

Dynamic Initialization

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

  Example:

```
int  a=2, b=3;        // Constants as initializer
int  c = a+b;         // Dynamic initialization
```

Scope of Variables

- Java allows variables to be declared within any **block**.
- A **block** is begun with an opening curly brace and ended by a closing curly brace.
- A block defines a scope. Thus, each time you start a new block, you are creating a new scope.
- A **scope** determines which **objects** are visible to other parts of your program.
- Java defines two general categories of scopes: **global** and **local**.
- Variables declared inside a scope is not **visible** to code that is defined outside that scope.
- Thus, when you declare a variable within a scope, then you can access it within that scope only and protecting it from **unauthorized access**.
- Scopes can be **nested**. So, outer scope encloses the inner scope. This means that objects declared in the **outer** scope will be **visible** to code within the **inner** scope.
- However, the reverse is not true. **Objects** declared within **inner** scope will not be **visible outside** it.

  Example:

```
class  scopeDemo
{
        public static void main(String args[])
        {
                int  x;             // visible to all code within main
                x = 10;
                if(x == 10) // start new scope
                {
                        int  y = 20;      // Visible only to this block
                                // x and y both are visible here.
                        System.out.println("x and y: " + x + " " + y);
                        x = y * 2;
                }
                // y = 100;                     // Error! y not visible here
                // x is still visible here.
                System.out.println("x is " + x);
        }

}
```

Default values of variables declared

- If you are not assigning value, then Java runtime assigns default value to variable and when you try to access the variable you get the default value of that variable.
- Following table shows variables types and their default values:

Data type	Default value
boolean	FALSE
char	\u0000
int,short,byte / long	0 / 0L
float /double	0.0f / 0.0d
any reference type	null

- Here, char primitive default value is \u0000, which means blank/space character.
- When you declare any local/block variable, they didn't get the default values.

Type Conversion and Casting

- It assigns a value of one type variable to a variable of another type. If the two types are **compatible**, then Java will perform the conversion automatically (Implicit conversion).
- **For example**, it is always possible to assign an **int** value to a **long** variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed.
- There is no automatic conversion defined from **double** to **byte**. Fortunately, it is possible conversion between **incompatible** types. For that you must perform type casting operation, which performs an **explicit conversion** between incompatible types.

Implicit Type Conversion (Widening Conversion)

- When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:
    1. **Two types are compatible.**
    2. **Destination type is larger than the source type.**
- When these two conditions are met, a **widening conversion** takes place.
- **For example**, the **int** type is always large enough to hold all valid **byte** values, so no explicit cast statement is required.
- **Following table shows compatibility of numeric data type:**

Status	Integer	Floating-Point	Char	Boolean
Compatible	✓	✓		
Incompatible			✓	✓

- Also, **Boolean** and **char** are not compatible with each other.

Explicit Type Conversion (Narrowing Conversion)

- Although the automatic type conversions are helpful, they will not fulfill all needs. **For example**, if we want to assign an **int** value to a **byte** variable then conversion will not be performed automatically, because a **byte** is smaller than an **int**.
- This kind of conversion is sometimes called a **narrowing** conversion.
- For, this type of conversion we need to make the value narrower explicitly. so that, it will fit into the target data type.
- To create a conversion between **two incompatible** types, you must use a **cast**.
- A cast is simply an explicit type conversion.

  Synatax: **(target-type) value**
- Here, **target-type** specifies the desired type to convert the specified value to.
- **For example**, the following casts an **int** to a **byte**.

```
int  a;
byte  b;
b = (byte) a;
```

- A different type of conversion will occur when a floating-point value is assigned to an integer type: **truncation**.
- As we know, integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost.
- **For example**, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated.

  Example:

```
class  conversionDemo
{
        public static void main(String args[])
        {
                byte b;
int i = 257;
double d = 323.142;

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

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

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

```
                System.out.println("d and b " + d + " " + b);
            }
        }
```

Output:

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

- When the value 257 is cast into a **byte** variable, the result is the remainder of the division of 257 by 256 (the range of a **byte**), which is 1 in this case.
- When the **d** is converted to an **int**, its fractional component is lost.
- When **d** is converted to a **byte**, its fractional component is lost, and the value is reduced modulo 256, which in this case is 67.

Wrapper Class

- **Wrapper** class wraps (encloses) around a data type and gives it an **object** appearance.
- Wrapper classes are used to convert any data type into an **object**.
- The primitive data types are not objects and they do not belong to any class.
- So, sometimes it is required to convert data types into objects in java.
- Wrapper classes include methods to unwrap the object and give back the data type.

  Example:

```
int  k = 100;
Integer  it1 = new Integer(k);
```

  - The **int** data type **k** is converted into an object, **it1** using **Integer** class.
  - The **it1** object can be used wherever **k** is required an object.
  - To unwrap (getting back int from Integer object) the object **it1**.

  Example:

```
int  m = it1.intValue();
System.out.println(m*m);       // prints 10000
```

  - **intValue()** is a method of **Integer** class that returns an **int** data type.
- Eight wrapper classes exist in **java.lang** package that represent 8 data types:

Primitive Data Type	Wrapper Class	Unwrap Methods
byte	Byte	byteValue()
short	Short	shortValue()
int	Integer	intValue()
long	Long	longValue()
float	Float	floatValue()
double	Double	doubleValue()

char	Character	charValue()
boolean	Boolean	booleanValue()

- There are mainly two uses with wrapper classes.
  1) To convert **simple data types** into **objects**.
  2) To convert **strings** into **data types** (known as parsing operations), here methods of type **parseX()** are used. (Ex. parseInt())
- The **wrapper classes** also provide methods which can be used to convert a **String** to any of the **primitive data types**, except **character**.
- These methods have the format **parsex()** where **x** refers to any of **the primitive data types** except **char**.
- To convert any of the primitive data type value to **a String**, we use the **valueOf()** methods of the String class.

  Example:

```
int x =  Integer.parseInt("34");          // x=34
double y = Double.parseDouble("34.7");       // y =34.7
String s1= String.valueOf('a');             // s1="a"
String s2=String.valueOf(true);          // s2="true"
```

Comment Syntax

- Comments are the statements which are never execute. (i.e. non-executable statements).
- Comments are often used to add notes between source code. So that it becomes easy to understand & explain the function or operation of the corresponding part of source code.
- Java Compiler doesn't read comments. **Comments are simply ignored during compilation**.
- There are three types of comments available in Java as follows;
  1. Single Line Comment
  2. Multi Line Comment
  3. Documentation Comment

Single Line Comment

- This comment is used whenever we need to write anything in single line.

  Syntax : //<write comment here>

  Example: //This is Single Line Comment.

Multi Line Comment

- These types of comments are used whenever we want to write detailed notes (i.e. more than one line or in multiple lines) related to source code.

  Syntax :

```
/*
   <Write comment here>
*/
```