Periyar Arts College ,Cuddalore

II Bsc Computer Science

JAVA(BCS 31)

UNIT I

What is Java technology?

Java is a programming language and computing platform first released by Sun Microsystems in 1995. There are lots of applications and websites that will not work unless you have Java installed, and more are created every day. Java is fast, secure, and reliable. From laptops to data enters, game consoles to scientific supercomputers, cell phones to the Internet, Java is everywhere.

Java is independent platform

Unit – I

What is a Computer language?

To communicate with the computers, we need some languages. These are computer languages.

There are mainly three different languages with the help of which we can develop computer programs. And they are –

- Machine Level language
- Assembly Level Language and
- High Level Language

Machine Level Language

Computer can understand only the language of Digital Electronics. Digital Electronics deals with presence and absence of voltages. Within the computer there are two logics can play their role.

Assembly Level Language and

1110011 Type the any letter is used to 1100101

High Level Language

High level language is the next development in the evolution of computer languages. Examples of some high-level languages are given below –

- PROLOG (for “PROgramming LOGic”);
- FORTRAN (for ‘FORrmula TRANslation’);
- LISP (for “LISt Processing”);
- Pascal (named after the French scientist Blaise Pascal).

Stages

What is the feature of Java?

Java is guaranteed to be write-once, run-anywhere language. On compilation Java program is compiled into bytecode. This bytecode is platform independent and can be run on any machine, plus this bytecode format also provide **security**. Any machine with Java Runtime Environment can run Java Programs.

What is meant by JDK in Java?

The **Java** Development Kit (**JDK**) is one of three core technology packages used in **Java** programming, along with the JVM (**Java** Virtual Machine) and the JRE (**Java** Runtime Environment).

Definition of OOP Concepts in Java

OOP concepts in Java are the main ideas behind Java's Object Oriented Programming. They are an abstraction**,** encapsulation, **inheritance**, and **polymorphism**. Grasping them is key to understanding how Java works. Basically, Java OOP concepts let us create working methods and variables, then re-use all or part of them without compromising security.

List of OOP Concepts in Java

There are four main OOP concepts in Java. These are:

- **Abstraction.** Abstraction means using simple things to represent complexity. We all know how to turn the TV on, but we don't need to know how it works in order to enjoy it. In Java, abstraction means simple things like **objects**, **classes**, and **variables** represent more complex underlying code and data. This is important because it lets avoid repeating the same work multiple times.
- **Encapsulation.** This is the practice of keeping fields within a class private, then providing access to them via public methods. It's a protective barrier that keeps the data and code safe within the class itself. This way, we can re-use objects like code components or variables without allowing open access to the data system-wide.
- **Inheritance.** This is a special feature of Object Oriented Programming in Java. It lets programmers create new classes that share some of the attributes of existing classes. This lets us build on previous work without reinventing the wheel.
- **Polymorphism.** This Java OOP concept lets programmers use the same word to mean different things in different contexts. One form of polymorphism in Java is **method overloading**. That's when different meanings are implied by the code itself. The other form is **method overriding**. That's when the different meanings are implied by the values of the supplied variables. See more on this below.

Oop example program

```
//save as Student.java

package com.javatpoint;
```

```
public class Student {

 private String name;

 public String getName() {

  return name; }

 public void setName(String name) {

  this.name = name

 } }

//save as Test.java

package com.javatpoint;

class Test {

 public static void main(String[] args) {

  Student s = new Student();

  s.setName(“vijay”);

  System.out.println(s.getName());

 } }

Compile By: javac -d . Test.java

Run By: java com.javatpoint.Test

Output: vijay
```

What is API in Java with example?

REST endpoint exposed onto network is another **example** of **API**. ... **Java application programming interface** (**API**) is a list of all classes that are part of the **Java** development kit (JDK). It includes all **Java** packages, classes, and interfaces, along with their methods, fields, and constructors.

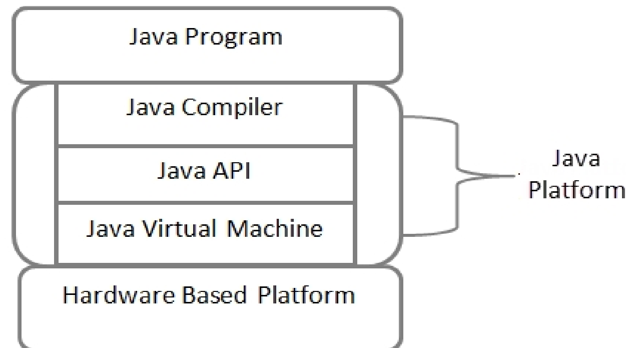
API Examples

As you continue learning programming, and specific languages in particular, you'll interact frequently with APIs. The Java programming language is loaded with APIs that you can use. In fact, many functions are nested within APIs and it's up to you to import them into your code.

For example, if you're dealing with dates, you'll need to import the Java API for dates, like the kind you can see below.

```
import java.util.Date;
```

Understanding API in Java



Data Types in Java

Data type defines the values that a variable can take, for example if a variable has int data type, it can only take integer values. In java we have two categories of data type: 1) Primitive data types 2) Non-primitive data types – Arrays and Strings are non-primitive data types, we will discuss them later in the coming tutorials. Here we will discuss primitive data types and literals in Java.

Java is a statically typed language. A language is statically typed, if the data type of a variable is known at compile time. This means that you must specify the type of the variable (Declare the variable) before you can use it.
In the last tutorial about Java Variables, we learned how to declare a variable, lets recall it:

```
int num;
```

So in order to use the variable num in our program, we must declare it first as shown above. It is a good programming practice to declare all the variables (that you are going to use) in the beginning of the program.

1) Primitive data types

In Java, we have eight primitive data types: boolean, char, byte, short, int, long, float and double. Java developers included these data types to maintain the portability of java as the size of these primitive data types do not change from one operating system to another.

byte, **short**, **int** and **long** data types are used for storing whole numbers.

float and **double** are used for fractional numbers.

char is used for storing characters(letters).

boolean data type is used for variables that holds either true or false.

byte:

This can hold whole number between -128 and 127. Mostly used to save memory and when you are certain that the numbers would be in the limit specified by byte data type.
Default size of this data type: 1 byte.
Default value: 0

- Byte data type is an 8-bit signed two's complement integer
- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive)(2^7 -1)
- Default value is 0
- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an integer.
- Example: byte a = 100, byte b = -50

Example:

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

            byte num;

            num = 113;
            System.out.println(num);
   }
}
```

Output: 113

Try the same program by assigning value assigning 150 value to variable num, you would get **type mismatch** error because the value 150 is out of the range of byte data type. The range of byte as I mentioned above is -128 to 127.

short:

This is greater than byte in terms of size and less than integer. Its range is -32,768 to 32767.
Default size of this data type: 2 byte

- Short data type is a 16-bit signed two's complement integer
- Minimum value is -32,768 (-2^15)
- Maximum value is 32,767 (inclusive) (2^15 -1)

- Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an integer
- Default value is 0.
- Example: short s = 10000, short r = -20000

```
short num = 45678;
```

int:

Used when short is not large enough to hold the number, it has a wider range: -2,147,483,648 to 2,147,483,647
Default size: 4 byte
Default value: 0

- Int data type is a 32-bit signed two's complement integer.
- Minimum value is - 2,147,483,648 (-2^31)
- Maximum value is 2,147,483,647(inclusive) (2^31 -1)
- Integer is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0
- Example: int a = 100000, int b = -200000

Example:

```
class JavaExample {
   public static void main(String[] args) {
            short num;
            num = 150;
            System.out.println(num);
   } }
```

Output: 150

The byte data type couldn’t hold the value 150 but a short data type can because it has a wider range.

long:

Used when int is not large enough to hold the value, it has wider range than int data type, ranging from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
size: 8 bytes
Default value: 0

- Long data type is a 64-bit signed two's complement integer
- Minimum value is -9,223,372,036,854,775,808(-2^63)

- Maximum value is 9,223,372,036,854,775,807 (inclusive)(2^63 -1)
- This type is used when a wider range than int is needed
- Default value is 0L
- Example: long a = 100000L, long b = -200000L

Example:

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

            long num = -12332252626L;
            System.out.println(num);
   }
}
```

Output: -12332252626

double:

Sufficient for holding 15 decimal digits
size: 8 bytes

- double data type is a double-precision 64-bit IEEE 754 floating point
- This data type is generally used as the default data type for decimal values, generally the default choice
- Double data type should never be used for precise values such as currency
- Default value is 0.0d
- Example: double d1 = 123.4

Example:

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

            double num = -42937737.9d;
            System.out.println(num);
   }
}
```

Output: -4.29377379E7

float:

Sufficient for holding 6 to 7 decimal digits
size: 4 bytes

- Float data type is a single-precision 32-bit IEEE 754 floating point
- Float is mainly used to save memory in large arrays of floating point numbers
- Default value is 0.0f
- Float data type is never used for precise values such as currency
- Example: float f1 = 234.5f

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

            float num = 19.98f;
            System.out.println(num);
   }
}
```

Output: 19.98

boolean:

holds either true of false.

- boolean data type represents one bit of information
- There are only two possible values: true and false
- This data type is used for simple flags that track true/false conditions
- Default value is false

Example: boolean one = true

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

            boolean b = false;
            System.out.println(b);
   }
}
```

Output: false

char:

holds characters.
size: 2 bytes

- char data type is a single 16-bit Unicode character
- Minimum value is '\u0000' (or 0)
- Maximum value is '\uffff' (or 65,535 inclusive)
- Char data type is used to store any character
- Example: char letterA = 'A'

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

            char ch = 'Z';
            System.out.println(ch);
   }
}
```

Output: Z

String class in Java

The String class represents character strings. All string literals in Java programs, such as "abc" , are implemented as instances of this class. Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared.

Creating Strings

The most direct way to create a string is to write −

String greeting = "Hello world!";

Whenever it encounters a string literal in your code, the compiler creates a String object with its value in this case, "Hello world!'.

As with any other object, you can create String objects by using the new keyword and a constructor. The String class has 11 constructors that allow you to provide the initial value of the string using different sources, such as an array of characters.

Example

```
public class StringDemo {

   public static void main(String args[]) {
      char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };
      String helloString = new String(helloArray);
      System.out.println( helloString );
   }
```

}

This will produce the following result −

Output

hello.

Note − The String class is immutable, so that once it is created a String object cannot be changed. If there is a necessity to make a lot of modifications to Strings of characters, then you should use String Buffer & String Builder Classes.

String Operations

Introduction

Simply put, a `String` is used to store text, i.e. a sequence of characters. Java's most used class is the `String` class, without a doubt, and with such high usage, it's mandatory for Java developers to be thoroughly acquainted with the class and its common operations.

String

There's a lot to say about `String`s, from the ways you can initialize them to the *String Literal Pool*, however in this article we'll focus on common operations, rather than the class itself.

Although, if you'd like to read more about various ways of creating strings in Java you should check out String vs StringBuilder vs StringBuffer.

Here, we're assuming that you're familiar with the fact that `String`s are *immutable*, as it's a very important thing to know before handling them. If not, refer to the previously linked article where it's explained in detail.

The `String` class comes with many helper methods that help us process our textual data:

- Determine String Length
- Finding Characters and Substrings
- Comparing Strings
- Extracting Substrings
- Changing String Case
- Removing Whitespace
- Formatting Strings

- Regex and Checking for Substrings
- Replacing Characters and Substrings
- Splitting and Joining Strings
- Creating Character Arrays
- String Equality

Type conversion in Java with Examples

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.

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.

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

Example:

```
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
```

What is a constant and how to define constants in Java?

A constant is a variable whose value **cannot change once it has been assigned**. Java doesn't have built-in support for constants.

A constant can make our program more easily read and understood by others. In addition, a constant is cached by the JVM as well as our application, so using a constant can improve performance.

To define a variable as a constant, we just need to add the keyword "**final**" in front of the variable declaration.

Syntax

```
final float pi = 3.14f;
```

The above statement declares the float variable "pi" as a constant with a value of 3.14f. We cannot change the value of "pi" at any point in time in the program. Later if we try to do that by using a statement like "pi=5.25f", Java will throw errors at compile time itself. It is not mandatory that we need to assign values of constants during initialization itself.

In the below example, we can define the primitive data type (byte, short, int, long, float, double, boolean and char) variables as constants by just adding the keyword "**final**" when we declare the variable.

Example

```
public class ConstantsDemo {

   public static void main(String args[]) {

      final byte var1 = 2;

      final byte var2;

      var2 = -3;

      final short var3 = 32;

      final short var4;

      var4 = -22;

      final int var5 = 100;

      final int var6;

      var6 = -112;

      final long var7 = 20000;
```

```
      final long var8;
      var8 = -11223;
      final float var9 = 21.23f;
      final float var10;
      var10 = -121.23f;
      final double var11 = 20000.3223;
      final double var12;
      var12 = -11223.222;
      final boolean var13 = true;
      final boolean var14;
      var14 = false;
      final char var15 = 'e';
      final char var16;
      var16 = 't';
     // Displaying values of all variables
      System.out.println("value of var1 : "+var1);
      System.out.println("value of var2 : "+var2);
      System.out.println("value of var3 : "+var3);
      System.out.println("value of var4 : "+var4);
      System.out.println("value of var5 : "+var5);
      System.out.println("value of var6 : "+var6);
      System.out.println("value of var7 : "+var7);
      System.out.println("value of var8 : "+var8);
      System.out.println("value of var9 : "+var9);
      System.out.println("value of var10 : "+var10);
      System.out.println("value of var11 : "+var11);
      System.out.println("value of var12 : "+var12);
      System.out.println("value of var13 : "+var13);
      System.out.println("value of var14 : "+var14);
      System.out.println("value of var15 : "+var15);
      System.out.println("value of var16 : "+var16);
   }
}
```

Output

```
value of var1 : 2
value of var2 : -3
value of var3 : 32
value of var4 : -22
value of var5 : 100
value of var6 : -112
value of var7 : 20000
value of var8 : -11223
value of var9 : 21.23
value of var10 : -121.23
value of var11 : 20000.3223
value of var12 : -11223.222
value of var13 : true
value of var14 : false
value of var15 : e
value of var16 : t
```

Java Scope

In Java, variables are only accessible inside the region they are created. This is called **scope**.

Method Scope

Variables declared directly inside a method are available anywhere in the method following the line of code in which they were declared.

Example

```
public class MyClass {
  public static void main(String[] args) {
    // Code here cannot use x
    int x = 100;
    System.out.println(x);
  }
}
```

Input from the keyboard

1. import **java**.util.Scanner; - imports the class Scanner from the library **java**.util.
2. Scanner scanner = new Scanner(System.in); - creates a new Scanner object, that is connected to standard **input** (the **keyboard**)
3. String inputString = scanner. nextLine();

Input from the keyboard

In Java, there are many ways to read strings from input. The simplest one is to make use of

the class Scanner, which is part of the java.util library and has been newly introduced in Java 5.0. Using this class, we can create an object to read input from the standard input channel System.in (typically, the keyboard), as follows:

Scanner scanner = new Scanner(System.in);
Then, we can use the nextLine() method of the Scanner class to get from standard input the next line (a sequence of characters delimited by a newline character), according to the following schema:

```
import java.util.Scanner;

public class KeyboardInput {
  public static void main (String[] args) {
    ...
    Scanner scanner = new Scanner(System.in);
    String inputString = scanner.nextLine();
    ...
    System.out.println(inputString);
    ...
  }
}
```

- import java.util.Scanner; - imports the class Scanner from the library java.util
- Scanner scanner = new Scanner(System.in); - creates a new Scanner object, that is connected to standard input (the keyboard)
- String inputString = scanner.nextLine();
    1. temporarily suspends the execution of the program, waiting for input from the keyboard;
    2. the input is accepted as soon as the user digits a carriage return;
    3. (a reference to) the string entered by the user (up to the first newline) is returned by the nextLine() method;
    4. (the reference to) the string is assigned to the variable inputString.

We can also read in a single word (i.e., a sequence of characters delimited by a whitespace character) by making use of the next() method of the Scanner class. We will see later that the Scanner class is also equipped with other methods to directly read various types of numbers from standard input.

index

Control statements

A **control statement** is a statement that determines whether other statements will be executed.

- An if statement decides whether to execute another statement, or decides which of two statements to execute.
- A loop decides how many times to execute another statement. There are three kinds of loops:
    - while loops test whether a condition is true before executing the

controlled statement.

  - do-while loops test whether a condition is true after executing the controlled statement.
  - for loops are (typically) used to execute the controlled statement a given number of times.

- A switch statement decides which of several statements to execute.

Java Conditions and If Statements

Java supports the usual logical conditions from mathematics:

- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`
- Equal to `a == b`
- Not Equal to: `a != b`

You can use these conditions to perform different actions for different decisions.

Java has the following conditional statements:

- Use `if` to specify a block of code to be executed, if a specified condition is true
- Use `else` to specify a block of code to be executed, if the same condition is false
- Use `else if` to specify a new condition to test, if the first condition is false
- Use `switch` to specify many alternative blocks of code to be executed

Looping Statement

A loop statement is a series of steps or sequence of statements executed repeatedly zero or more times satisfying the given condition is satisfied. Loop statements in programming languages, such as assembly languages or PERL make use of LABEL's to execute the statement repeatedly. In programming languages, such as C, C++, Java, and Python, come with “For, While and Do loop” statements.

For Loop:
For loop is used to repeat the statements that execute for specific number of times.
Syntax:

```
for(initialisation; Condition; Increment/Decrement)
{
    Statements;
}
```

While loop
Initially checks for the condition statement, if true then set of statements is executed repeatedly.
Syntax:

```
while(Condition)
{
    Statements;
}
```

while(i==0)

Do while loop

Do while loop is flow of control where
• Statement of blocks is executed atleast once whether condition is true or false.
• checks for the condition at the end of loop body.
• if condition is true, then executes set of statements.

Syntax:

```
do
{
  Statements;
}
while (Condition);
```

Java continue Statement

While working with loops, it is sometimes desirable to skip some statements inside the loop or terminate the loop immediately without checking the test expression.

In such cases, `break` and `continue` statements are used. To learn about the `break` statement, visit Java break.

The `continue` statement in Java skips the current iteration of a loop (`for`, `while`, `do...while`, etc) and the control of the program moves to the end of the loop. And, the test expression of a loop is evaluated.
In the case of `for` loop, the update statement is executed before the test expression.
The `continue` statement is almost always used in decision-making statements (if...else Statement). It's syntax is:

Java Break

You have already seen the `break` statement used in an earlier chapter of this tutorial. It was used to "jump out" of a `switch` statement.

The `break` statement can also be used to jump out of a **loop**.

This example jumps out of the loop when i is equal to 4:

Example

```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    break;   }
  System.out.println(i); }
```

Java Continue

The `continue` statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

This example skips the value of 4:

Example

```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    continue;
                      }   System.out.println(i);  }
```

Classes and Objects in Java

Classes and Objects are basic concepts of Object Oriented Programming which revolve around the real life entities.

Class

A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. **Modifiers** : A class can be public or has default access

2. **Class name:** The name should begin with a initial letter (capitalized by convention).
3. **Superclass(if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
4. **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
5. **Body:** The class body surrounded by braces, { }.

Constructors are used for initializing new objects. Fields are variables that provides the state of the class and its objects, and methods are used to implement the behavior of the class and its objects.

There are various types of classes that are used in real time applications such as nested classes, anonymous classes, lambda expressions.

Object

It is a basic unit of Object Oriented Programming and represents the real life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

1. **State** : It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behavior** : It is represented by methods of an object. It also reflects the response of an object with other objects.
3. **Identity** : It gives a unique name to an object and enables one object to interact with other objects.

Modifiers in Java

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

Understanding Java Access Modifiers

Let's understand the access modifiers in Java by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

1) Private

The private access modifier is accessible only within the class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

```
1.  class A{
2.  private int data=40;
3.  private void msg(){System.out.println("Hello java");}
4.  }
5.  
6.  public class Simple{
7.   public static void main(String args[]){
8.     A obj=new A();
9.     System.out.println(obj.data);//Compile Time Error
10.    obj.msg();//Compile Time Error
11.    }
12. }
```

2) Default

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from

outside the package.

```java
//save by A.java
package pack;
class A{
  void msg(){System.out.println("Hello");}
}
```

```java
//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();//Compile Time Error
   obj.msg();//Compile Time Error
  }
}
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

3) Protected

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifer.

Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```java
//save by A.java
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}
```

```java
//save by B.java
package mypack;
import pack.*;

class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }
}
```

```
Output:Hello
```

4) Public

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

Example of public access modifier

```
1. //save by A.java
2.
3. package pack;
4. public class A{
5. public void msg(){System.out.println("Hello");}
6. }
```

```
1. //save by B.java
2.
3. package mypack;
4. import pack.*;
5.
6. class B{
7.   public static void main(String args[]){
8.    A obj = new A();
9.    obj.msg();
10.  }
11. }
```

Output:Hello

Java Access Modifiers with Method Overriding

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

```
1. class A{
2. protected void msg(){System.out.println("Hello java");}
3. }
4.
5. public class Simple extends A{
6. void msg(){System.out.println("Hello java");}//C.T.Error
7.  public static void main(String args[]){
8.    Simple obj=new Simple();
9.    obj.msg();
10.   }
11. }
```

The default modifier is more restrictive than protected. That is why, there is a compile-time error.

Arguments

To run this java program, you must pass at least one argument from the command prompt.

class CommandLineExample{

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

System.out.println("Your first **argument** is: "+**args**[0]);

}
}

All arguments to methods in Java are *pass-by-value*.

What does this mean? How do I change a parameter? This short tutorial will help you figure out how parameter passing in Java works and will help you avoid some common mistakes.

First let's get the terminology straight. The terms "arguments" and "parameters" are used interchangeably; they mean the same thing. We use the term *formal parameters* to refer to the parameters in the definition of the method. In the example that follows, x and y are the formal parameters.

We use the term *actual parameters* to refer to the variables we use in the method call. In the following example, length and width are actual parameters.

```
// Method definition
public int mult(int x, int y)
{
  return x * y;
}

// Where the method mult is used
int length = 10;
int width = 5;
int area = mult(length, width);
```

Pass-by-value means that when you call a method, a copy of each actual parameter (argument) is passed. You can change that copy inside the method, but this will have no effect on the actual parameter. Unlike many other languages, Java has no mechanism to change the value of an actual parameter. Isn't this very restrictive? Not really. In Java, we can pass a reference to an object (also called a "handle")as a parameter. We can then change something inside the object; we just can't change what object the handle refers to.

Constructor with example?

When a class or struct is created, its constructor is called. Constructors have the same name as the class or struct, and they usually initialize the data members of the new **object**. In the following example, a class named Taxi is defined by using a simple constructor.

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.

It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

Note: It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

Rules for creating Java constructor

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax of default constructor:

```
1. <class_name>(){}
```

Example of default constructor

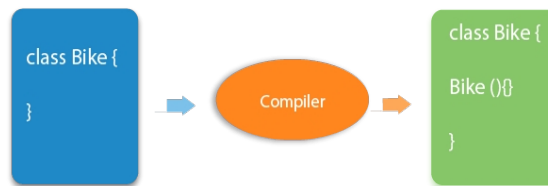
In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of o

```
1. //Java Program to create and call a default constructor
2. class Bike1{
3. //creating a default constructor
4. Bike1(){System.out.println("Bike is created");}
5. //main method
6. public static void main(String args[]){
7. //calling a default constructor
8. Bike1 b=new Bike1();
9. }
10. }
```

Test it Now

Output:

```
Bike is created
```



Q) What is the purpose of a default constructor?

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

Example of default constructor that displays the default values

```
1. //Let us see another example of default constructor
2. //which displays the default values
3. class Student3{
4. int id;
5. String name;
6. //method to display the value of id and name
7. void display(){System.out.println(id+" "+name);}
8.
9. public static void main(String args[]){
10. //creating objects
11. Student3 s1=new Student3();
12. Student3 s2=new Student3();
13. //displaying values of the object
14. s1.display();
15. s2.display();
16. }
17. }
```

Test it Now

Output:

```
0 null
0 null
```

Explanation: In the above class,you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

Package and import Java

Package is a collection of related classes. Java uses package to group related classes, interfaces and sub-packages.

We can assume package as a folder or a directory that is used to store similar files.

In Java, packages are used to avoid name conflicts and to control access of class, interface and enumeration etc. Using package it becomes easier to locate the related classes and it also provides a good structure for projects with hundreds of classes and other files.

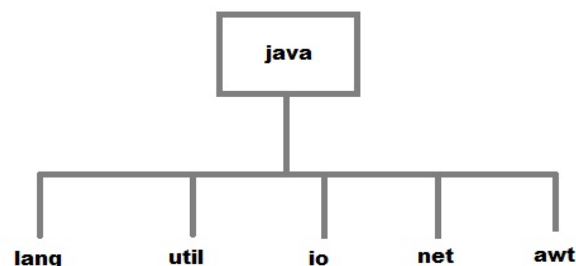
Lets understand it by a simple example, Suppose, we have some math related classes and interfaces then to collect them into a simple place, we have to create a package.

Types Of Package

Package can be built-in and user-defined, Java provides rich set of built-in packages in form of API that stores related classes and sub-packages.

Built-in Package: math, util, lang, i/o etc are the example of built-in packages.

- **User-defined-package:** Java package created by user to categorize their project's classes and interface are known as user-defined packages.



utility classes(**util**)

package **java.net(net)**

Abstract Window Toolkit (**AWT**)

How to Create a Package

Creating a package in java is quite easy, simply include a package command followed by name of the package as the first statement in java source file.

```
package mypack;
public class employee
{
    String empId;
    String name;
}
```

The above statement will create a package woth name **mypack** in the project directory.

Java uses file system directories to store packages. For example the `.java` file for any class you define to be part of **mypack** package must be stored in a **directory** called **mypack**.

Additional points about package:

- Package statement must be first statement in the program even before the import statement.
- A package is always defined as a separate folder having the same name as the package name.
- Store all the classes in that package folder.
- All classes of the package which we wish to access outside the package must be declared public.
- All classes within the package must have the package statement as its first line.
- All classes of the package must be compiled before use.

Example of Java packages

Now lets understand package creation by an example, here we created a **leanjava** package that stores the FirstProgram class file.

```
//save as FirstProgram.java

package learnjava;

public class FirstProgram{

    public static void main(String args[]) {

        System.out.println("Welcome to package example");

    }  }
```

Static class

You cannot use the static keyword with a class unless it is an inner class. A static inner class is a nested class which is a static member of the outer class. It can be accessed without instantiating the outer class, using other static members. Just like static members, a static nested class does not have access to the instance variables and methods of the outer class.

Syntax

```
class MyOuter {
  static class Nested_Demo {
  }
}
```

Instantiating a static nested class is a bit different from instantiating an inner class. The following program shows how to use a static nested class.

Example

Live Demo

```
public class Outer {
   Java Arrays with Answers
   static class Nested_Demo {
      public void my_method() {
         System.out.println("This is my nested class");
      }   }
   public static void main(String args[]) {
      Outer.Nested_Demo nested = new Outer.Nested_Demo();
      nested.my_method();
   } }
```

Output

If you compile and execute the above program, you will get the following result −

```
This is my nested class
```

Overload!

The good news is that this overload concept won't break up the highway. In Java, the term **overload** means that there are multiple versions of a constructor or method. They will each have a different number of **arguments** or values that they take in to work with.

For example, a payroll program could have an Employee class, and constructors that create Employee objects of varying types. One constructor might take in employee name, ID, and state code; another might not need any arguments and just create an Employee.

So what does overloading really look like in code?

Overloaded Methods

A method is a set of code that can take arguments, that is values, and do something with those values. For example, a method in a payroll program could multiply wages by hours worked. The method then returns the value it comes up with and this value is sent back to the line of code that called the method.

You can overload a method in the same way as a constructor. That is, multiple methods can exist, each with the same name. Remember, though, that they can't have the same number of arguments or the same type of arguments! Overloading isn't cloning.

In the following code example, we build upon our program and add in methods for performing the conversion. One takes a single value, the other two values:

```
1. public double setRate(double c) {
2.   conversionRate = 352.222;
3.   return conversionRate;
4. }
5. public double setRate(double c, double m) {
6.   conversionRate = 352.22;
7.   modifier = .0035;
8.   double newRate = conversionRate * modifier;
9.   return newRate;
10. }
```

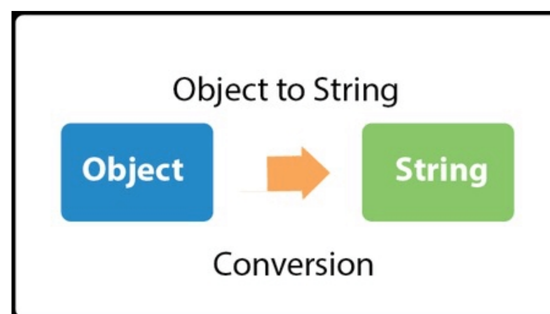
Overloaded Constructors

Here is an example of an overloaded constructor. The class Conversion has TWO constructors, one that takes a conversion rate and the other also has a modifier option.

```
1. public Conversion {
2.   public double conversionRate;
3.   public double modifier;
4.   // constructor here:
5.   public Conversion(double c) {
6.    conversionRate = c;
7.   }
8.   //another constructor: the overload
9.   public Conversion(double c, double m) {
10.   conversionRate = c;
11.   modifier = .00587;
12.  }
13. }
```

Java Convert Object to String

We can convert **Object to String in java** using toString() method of Object class or String.valueOf(object) method.



You can convert any object to String in java whether it is user-defined class, StringBuilder,

StringBuffer or anything else.

Here, we are going to see two examples of converting Object into String. In the first example, we are going to convert Emp class object into String which is an user-defined class. In second example, we are going to convert StringBuilder to String.