

History and evolution of java:

Java history is interesting to know. The history of java starts from Green Team. 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.

James Gosling - founder of java

Currently, Java is used in internet programming, mobile devices, games, e-business solutions etc. There are given the major points that describes the history of java.

- 1) James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991. The small team of sun engineers called Green Team.
- 2) Originally designed for small, embedded systems in electronic appliances like set-top boxes.
- 3) Firstly, it was called "Greentalk" by James Gosling and file extension was .gt.
- 4) After that, it was called Oak and was developed as a part of the Green project.

Java History from Oak to Java

Why "Oak" name

- 5) Why Oak? Oak is a symbol of strength and choosen as a national tree of many countries like 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" name

7) Why had they choosen 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.

- 8) Java is an island of Indonesia where first coffee was produced (called java coffee).
- 9) Notice that Java is just a name not an acronym.
- 10) Originally 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).

How Java Changed the Internet

The Internet helped catapult Java to the forefront of programming, and Java, in turn, had a profound effect on the Internet. In addition to simplifying web programming in general, Java innovated a new type of networked program called the applet that changed the way the online world thought about

content. Java also addressed some of the thorniest issues associated with the Internet: portability and security. Let's look more closely at each of these.

Java Applets

An applet is a special kind of Java program that is designed to be transmitted over the Internet and automatically executed by a Java-compatible web browser. Furthermore, an applet is downloaded on demand, without further interaction with the user. If the user clicks a link that contains an applet, the applet will be automatically downloaded and run in the browser. Applets are intended to be small programs. They are typically used to display data provided by the server, handle user input, or provide simple functions, such as a loan calculator, that execute locally, rather than on the server. In essence, the applet allows some functionality to be moved from the server to the client.

The creation of the applet changed Internet programming because it expanded the universe of objects that can move about freely in cyberspace. In general, there are two very broad categories of objects that are transmitted between the server and the client: passive information and dynamic, active programs. For example, when you read your e-mail, you are viewing passive data. Even when you download a program, the program's code is still only passive data until you execute it. By contrast, the applet is a dynamic, self-executing program. Such a program is an active agent on the client computer, yet it is initiated by the server. As desirable as dynamic, networked programs are, they also present serious problems in the areas of security and portability. Obviously, a program that downloads and executes automatically on the client computer must be prevented from doing harm. It must also be able to run in a variety of different environments and under different operating systems.

Security

As you are likely aware, every time you download a "normal" program, you are taking a risk, because the code you are downloading might contain a virus, Trojan horse, or other harmful code. At the core of the problem is the fact that malicious code can cause its damage because it has gained unauthorized access to system resources. For example, a virus program might gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer's local file system. In order for Java to enable applets to be downloaded and executed on the client computer safely, it was necessary to prevent an applet from launching such an attack.

Java achieved this protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer. The ability to download applets with confidence that no harm will be done and that no security will be breached is considered by many to be the single most innovative aspect of Java.

Portability

Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it. If a Java program were to be run on virtually any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems. For example, in the case of an applet, the same applet must be able to be downloaded and executed by the wide variety of CPUs, operating systems, and browsers connected to the Internet. It is not practical to have different versions of the applet for different computers. The same code must work on all computers. Therefore, some means of generating portable executable code was needed.

Java's Magic: The Byte Code:

The means that allows Java to solve both the security and the portability problems is that the output of a Java compiler is not executable code but it is the Bytecode. Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM). JVM is an interpreter for bytecode. The fact that a Java program is executed by JVM helps solve the major problems associated with downloading programs over the Internet. Translating a Java program into bytecode helps make it much easier to run a program in a wide variety of environments. This is because only the JVM needs to be implemented for each platform.

Once the run-time package exists for a given system, any Java program can run on it. If a Java program is compiled to its subject code, then different versions of the same program would exist for each type of CPU connected to the Internet. This is not a practical solution. Thus, the understanding of bytecode is the most effective way to create strictly portable programs. Now, if a Java program is interpreted, it also helps to make it secure. Since the execution of every Java program is under the control of the JVM, the JVM can contain the program, and prevent it from generating side effects outside of the system. The use of bytecode enables the Java run-time system to execute programs much faster.

Sun provides a facility called Just In Time (JIT) compiler for bytecode. It is not possible to compile an entire Java program into executable code all at once, because Java performs various run-time checks that can be done only at run time. JIT compiles the code as and when needed.

Object oriented programming:

What is Object Oriented Programming (OOP)?

OOP is a high-level programming language where a program is divided into small chunks called objects using the object-oriented model, hence the name. This paradigm is based on objects and classes.

Object – An object is basically a self-contained entity that accumulates both data and procedures to manipulate the data. Objects are merely instances of classes.

Class – A class, in simple terms, is a blueprint of an object which defines all the common properties of one or more objects that are associated with it. A class can be used to define multiple objects within a program.

The OOP paradigm mainly eyes on the data rather than the algorithm to create modules by dividing a program into data and functions that are bundled within the objects. The modules cannot be modified when a new object is added restricting any non-member function access to the data. Methods are the only way to access the data.

Objects can communicate with each other through same member functions. This process is known as message passing. This anonymity among the objects is what makes the program secure. A programmer can create a new object from the already existing objects by taking most of its features thus making the program easy to implement and modify.

An Overview Of Java:

Object-Oriented Programming

Object-oriented programming is at the core of Java. In fact, all Java programs are object-oriented—this isn't an option the way that it is in C++, for example. OOP is so integral to Java that you must understand its basic principles before you can write even simple Java programs. Therefore, this chapter begins with a discussion of the theoretical aspects of OOP.

Two Paradigms

There are the two paradigms that govern how a program is constructed. The first way is called the **process-oriented model**. This approach characterizes a program as a series of linear steps (that is, code). The process-oriented model can be thought of as code acting on data. Procedural languages such as C employ this model to considerable success.

To manage increasing complexity, the second approach, called **object-oriented programming**, was conceived. Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as data controlling access to code. As you will see, by switching the controlling entity to data, you can achieve several organizational benefits.

Difference between POP(Procedure Oriented Program) AND OOP(Object Oriented Program):

OOP vs. POP

OOP	POP
OOP takes a bottom-up approach in designing a program.	POP follows a top-down approach.
Program is divided into objects depending on the problem.	Program is divided into small chunks based on the functions.
Each object controls its own data.	Each function contains different data.
Focuses on security of the data irrespective of the algorithm.	Follows a systematic approach to solve the problem.
The main priority is data rather than functions in a program.	Functions are more important than data in a program.
The functions of the objects are linked via message passing.	Different parts of a program are interconnected via parameter passing.
Data hiding is possible in OOP.	No easy way for data hiding.
Inheritance is allowed in OOP.	No such concept of inheritance in POP.
Operator overloading is allowed.	Operator overloading is not allowed.
C++, Java.	Pascal, Fortran.

Abstraction.

Hiding internal details and showing functionality is known as 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.

Three OOP Principles:

1. Encapsulation
2. Inheritance
3. Polymorphism

Encapsulation:

Encapsulation is an OOP technique of wrapping the data and code. In this OOPS concept, the variables of a class are always hidden from other classes. It can only be accessed using the methods of their current class. For example - in school, a student cannot exist without a class.

2. 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.

3. Polymorphism:

The ability to take more than one form. The ability to define more than one function with the same name is called Polymorphism. In java there are two type of polymorphism.

- a) compile time polymorphism (overloading)
- b) runtime polymorphism (overriding).

A first Simple Program:

Now that the basic object-oriented underpinning of Java has been discussed, let's look at some actual Java programs. Let's start by compiling and running the short sample program shown here. As you will see, this involves a little more work than you might imagine.

```
*/  
This is a simple Java program.  
Call this file "Example.java".  
*/  
Imports java.io.*;  
Imports java.util.*;  
class Example {  
    // Your program begins with a call to main().  
    public static void main(String args[])  
    {  
        System.out.println("This is a simple Java program."); }  
}
```

Output: This is a simple Java program.

Lexical Issues:

Whitespace

Java is a free-form language. This means that you do not need to follow any special indentation rules. For example, the Example program could have been written all on one line or in any other strange way you felt like typing it, as long as there was at least one whitespace character between each token that was not already delineated by an operator or separator. In Java, whitespace is a space, tab, or newline.

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 and dollar-sign characters. They must not begin with a number, lest they be confused with a numeric literal. Again, Java is case-sensitive, so VALUE is a different identifier than Value. Some examples of valid identifiers are:

AvgTemp count a4 \$test this_is_ok

Invalid variable names include:

2count high-temp Not/ok

Literals

A constant value in Java is created by using a literal representation of it. For example, here are some literals:

100 98.6 'X' "This is a test"

Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string. A literal can be used anywhere a value of its type is allowed.

Comments

As mentioned, there are three types of comments defined by Java. You have already seen two: single-line and multiline. The third type is called a documentation comment. This type of comment is used to produce an HTML file that documents your program. The documentation comment begins with a `/**` and ends with a `*/`. Documentation comments are explained in Appendix A.

Separators

In Java, there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon. As you have seen, it is used to terminate statements. The separators are shown in the following table:

Symbol	Name	Purpose
()	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically

[] Brackets

initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.

Used to declare array types. Also used when dereferencing array values.

; Semicolon

Terminates statements.

, Comma

Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a **for** statement.

. Period

Used to separate package names from sub packages and classes. Also used to separate a variable or method from a reference variable.

Data Types, Variables and Arrays:

Primitive Types:

The eight primitive data types in Java are:

- Boolean: the type whose values are either true or false
- Char: the character type whose values are 16-bit Unicode characters
- the arithmetic types:
 - the integral types:
 - byte
 - short
 - int
 - long
 - the floating-point types:
 - float
 - double

Primitive data types in Java

Type	Description	Default	Size	Example Literals
boolean	true or false	false	1 bit	true, false
byte	two's complement integer	0	8 bits	(none)

char	Unicode character	\u0000 0	16 bits	'a', '\u0041', '\101', '\\', '\\', , '\n', '\b'
short	two's complement integer	0	16 bits	(none)
int	two's complement integer	0	32 bits	-2, -1, 0, 1, 2
long	two's complement integer	0	64 bits	-2L, -1L, 0L, 1L, 2L
float	IEEE 754 floating point	0.0	32 bits	1.23e100f, -1.23e-100f, .3f, 3.14F
double	IEEE 754 floating point	0.0	64 bits	1.23456e300d, -1.23456e-300d, 1e1d

Wrapper class in Java

Wrapper class in java provides the mechanism to convert primitive into object and object into primitive.

Since J2SE 5.0, autoboxing and unboxing feature converts primitive into object and object into primitive automatically. The automatic conversion of primitive into object is known as autoboxing and vice-versa unboxing.

The eight classes of java.lang package are known as wrapper classes in java. The list of eight wrapper classes are given below:

Primitive Type	Wrapper class
Boolean	Boolean

Char	Character
Byte	Byte
Short	Short
Int	Integer
Long	Long
Float	Float
double	Double

Literals:

A literal is the source code representation of a fixed value.

Literals in Java are a sequence of characters (digits, letters, and other characters) that represent constant values to be stored in variables. Java language specifies five major types of literals. Literals can be any number, text, or other information that represents a value. This means what you type is what you get. We will use literals in addition to variables in Java statement. While writing a source code as a character sequence, we can specify any value as a literal such as an integer. They are:

Integer literals

Floating literals

Character literals

String literals

Boolean literals

Each of them has a type associated with it. The type describes how the values behave and how they are stored.

Integer literals: Integer data types consist of the following primitive data types: int, long, byte, and short. byte, int, long, and short can be expressed in decimal(base 10), hexadecimal(base 16) or octal(base 8) number systems as well. Prefix 0 is used to indicate octal and prefix 0x indicates hexadecimal when using these number systems for literals.

Examples:

```
int decimal = 100;
```

```
int octal = 0144;
```

```
int hexa = 0x64;
```

Floating-point literals:

Floating-point numbers are like real numbers in mathematics, for example, 4.13179, -0.000001. Java has two kinds of floating-point numbers: float and double. The default type when you write a floating-point literal is double, but you can designate it explicitly by appending the D (or d) suffix. However, the suffix F (or f) is appended to designate the data type of a floating-point literal as float. We can also specify a floating-point literal in scientific notation using Exponent (short E ore), for instance: the double literal 0.0314E2 is interpreted as:

0.0314 *10² (i.e 3.14).

6.5E+32 (or 6.5E32) Double-precision floating-point literal

7D Double-precision floating-point literal

.01f Floating-point literal

Character literals:

char data type is a single 16-bit Unicode character. We can specify a character literal as a single printable character in a pair of single quote characters such as 'a', '#', and '3'. You must know about the ASCII character set. The ASCII character set includes 128 characters including letters, numerals, punctuation etc. Below table shows a set of these special characters.

Escape	Meaning
\n	New line
\t	Tab
\b	Backspace
\r	Carriage return
\f	Formfeed
\\	Backslash
\'	Single quotation mark
\"	Double quotation mark
\d	Octal
\xd	Hexadecimal
\ud	Unicode character

String Literals:

The set of characters is represented as String literals in Java. Always use "double quotes" for String literals. There are few methods provided in Java to combine strings, modify strings and to know whether two strings have the same values.

""	The empty string
----	------------------

"\""	A string containing
"This is a string"	A string containing 16 characters
"This is a " + "two-line string"	actually a string-valued constant expression, formed from two string literals

Null Literals

The final literal that we can use in Java programming is a Null literal. We specify the Null literal in the source code as 'null'. To reduce the number of references to an object, use null literal. The type of the null literal is always null. We typically assign null literals to object reference variables. For instance

```
s = null;
```

Boolean Literals:

The values true and false are treated as literals in Java programming. When we assign a value to a boolean variable, we can only use these two values. Unlike C, we can't presume that the value of 1 is equivalent to true and 0 is equivalent to false in Java. We have to use the values true and false to represent a Boolean value.

Example

```
boolean chosen = true;
```

Remember that the literal true is not represented by the quotation marks around it. The Java compiler will take it as a string of characters, if it is in quotation marks.

Type Casting

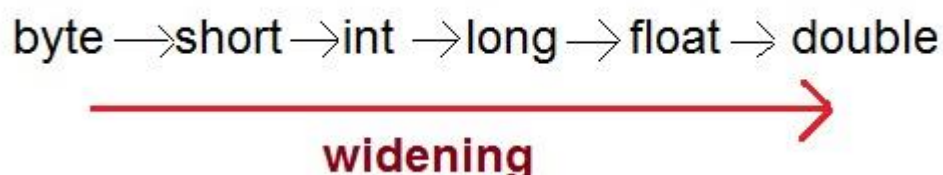
Assigning a value of one type to a variable of another type is known as **Type Casting**.

Example :

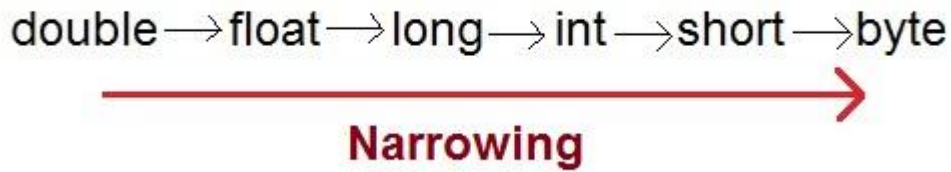
```
int x = 10;
byte y = (byte)x;
```

In Java, type casting is classified into two types,

- Widening Casting(Implicit)



- Narrowing Casting(Explicitly done)



Widening or Automatic type conversion

Automatic Type casting take place when,

- the two types are compatible
- the target type is larger than the source type

Example :

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

Narrowing or Explicit type conversion

When you are assigning a larger type value to a variable of smaller type, then you need to perform explicit type casting.

Example :

```
public class Test
{
    public static void main(String[] args)
    {
        double d = 100.04;
        long l = (long)d; //explicit type casting required
        int i = (int)l;    //explicit type casting required

        System.out.println("Double value "+d);
        System.out.println("Long value "+l);
        System.out.println("Int value "+i);
    }
}
```

Output

```
Double value 100.04
Long value 100
Int value 100
```

Arrays:

An array is a one of the data structure in Java, that can store a fixed-size sequential collection of elements of the same data type.

For many large applications, there may arise some situations that need a single name to store multiple values. To process such large amount of data, programmers need powerful data types that would facilitate the efficient contiguous bulk amount of storage facility, accessing and dealing with such data items. So, arrays are used in Java. In this tutorial, you will learn about what arrays are and what the types are and how they are used within a Java program.

Defining an array in java:

The syntax of declaring array variables is:

Syntax:

```
datatype[] identifier; //preferred way
```

or

datatype identifier[];

The syntax used for instantiating arrays within a Java program is:

Example:

char[] refVar;

int[] refVar;

short[] refVar;

long[] refVar;

int[][] refVar; //two-dimensional array

y

Initialize an Array in Java

By using new operator array can be initialized.

Example:

```
int[] age = new int[5];    //5 is the size of array.
```

Arrays can be initialized at declaration time also.

Example:

```
int age[5]={22,25,30,32,35};
```

Initializing each element separately in the loop.

Example:

```
public class Sample {  
  
    public static void main(String args[]) {  
  
        int[] newArray = new int[5];
```

```

// Initializing elements of array separately

for (int n = 0; n < newArray.length; n++) {

    newArray[n] = n;

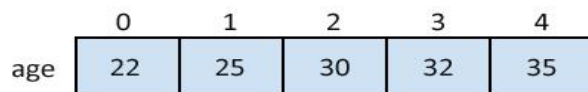
}

}

}

```

A Pictorial Representation of Array



	0	1	2	3	4
age	22	25	30	32	35

Accessing Array Elements in Java

Example:

```

public class Sample {

    public static void main(String args[]) {

        int[] newArray = new int[5];

        // Initializing elements of array separately

        for (int n = 0; n < newArray.length; n++) {

            newArray[n] = n;

        }

        System.out.println(newArray[2]); // Assigning 2nd element of array value

    }

}

```

Program Output:

```
run:
2
BUILD SUCCESSFUL (total time: 0 seconds)
```

Related Contents

Operators:

Operator in java is a symbol that is used to perform operations. For example: +, -, *, / etc.

There are many types of operators in java which are given below:

- Unary Operator,
- Arithmetic Operator,
- shift Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator and
- Assignment Operator.

Java Operator Precedence

Operator Type	Category	Precedence
Unary	postfix	<code>expr++ expr--</code>
	prefix	<code>++expr --expr +expr -expr ~ !</code>

Arithmetic	multiplicative	* / %
	additive	+ -
Shift	shift	<< >> >>>
Relational	comparison	< > <= >= instanceof
	equality	== !=
Bitwise	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	
Logical	logical AND	&&
	logical OR	
Ternary	ternary	? :
Assignment	Assignment	= += -= *= /= %= &= ^= = <<= >>= >>>=

A program executes from top to bottom except when we use control statements, we can control the order of execution of the program, based on logic and values.

In Java, control statements can be divided into the following three categories:

Selection Statements

Iteration Statements

Jump Statements

Selection Statements

Selection statements allow you to control the flow of program execution on the basis of the outcome of an expression or state of a variable known during runtime.

Selection statements can be divided into the following categories:

The if and if-else statements

The if-else statements

The if-else-if statements

The switch statements

The if statements

The first contained statement (that can be a block) of an if statement only executes when the specified condition is true. If the condition is false and there is not else keyword then the first contained statement will be skipped and execution continues with the rest of the program. The condition is an expression that returns a boolean value.

Example

```
import java.util.Scanner;
```

```
public class IfDemo
```

```
{
```

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

```
        int age;
```

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

```
        System.out.print("Please enter Age: ");
```

```
        age = inputDevice.nextInt();
```

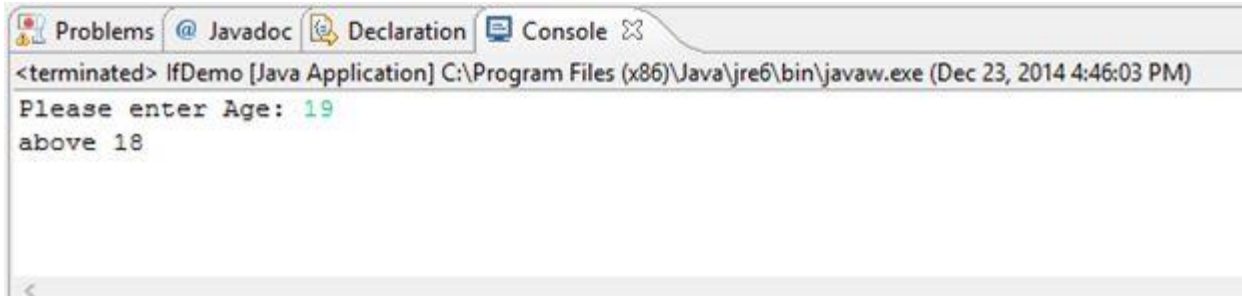
```
        if(age > 18)
```

```
            System.out.println("above 18 ");
```

```
    }
```

```
}
```

Output:



```
<terminated> IfDemo [Java Application] C:\Program Files (x86)\Java\jre6\bin\javaw.exe (Dec 23, 2014 4:46:03 PM)
Please enter Age: 19
above 18
```

The if-else statements

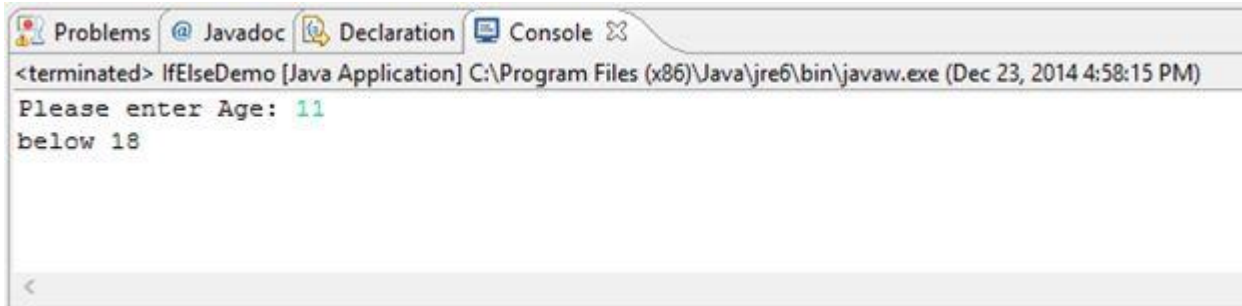
In if-else statements, if the specified condition in the if statement is false, then the statement after the else keyword (that can be a block) will execute.

Example

```
import java.util.Scanner;

public class IfElseDemo
{
    public static void main( String[] args )
    {
        int age;
        Scanner inputDevice = new Scanner( System.in );
        System.out.print( "Please enter Age: " );
        age = inputDevice.nextInt();
        if ( age >= 18 )
            System.out.println( "above 18 " );
        else
            System.out.println( "below 18" );
    }
}
```

Output



```
<terminated> IfElseDemo [Java Application] C:\Program Files (x86)\Java\jre6\bin\javaw.exe (Dec 23, 2014 4:58:15 PM)
Please enter Age: 11
below 18
```

The if-else-if statements

This statement following the else keyword can be another if or if-else statement.

That would look like this:

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

Whenever the condition is true, the associated statement will be executed and the remaining conditions will be bypassed. If none of the conditions are true then the else block will execute.

Example

```
import java.util.Scanner;

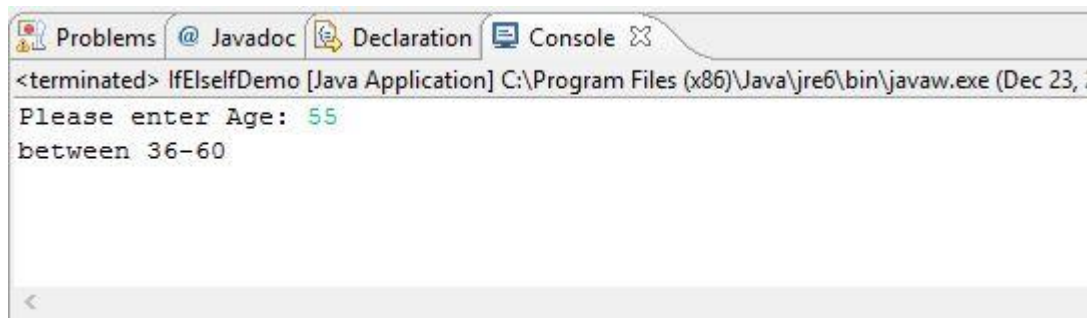
public class IfElseDemo
{
    public static void main( String[] args )
    {
        int age;
        Scanner inputDevice = new Scanner( System.in );
        System.out.print( "Please enter Age: " );
        age = inputDevice.nextInt();
        if ( age >= 18 && age <=35 )
            System.out.println( "between 18-35 " );
    }
}
```

```

else if(age >35 && age <=60)
    System.out.println("between 36-60");
else
    System.out.println( "not matched" );
}
}

```

Output



```

<terminated> IfElseIfDemo [Java Application] C:\Program Files (x86)\Java\jre6\bin\javaw.exe (Dec 23, 2011)
Please enter Age: 55
between 36-60

```

The Switch Statements

The switch statement is a multi-way branch statement. The switch statement of Java is another selection statement that defines multiple paths of execution of a program. It provides a better alternative than a large series of if-else-if statements.

Example

```

import java.util.Scanner;
public class SwitchDemo
{
    public static void main( String[] args )
    {
        int age;
        Scanner inputDevice = new Scanner( System.in );
        System.out.print( "Please enter Age: " );
        age = inputDevice.nextInt();
        switch ( age )
        {
            case 18:
                System.out.println( "age 18" );
                break;
            case 19:
                System.out.println( "age 19" );

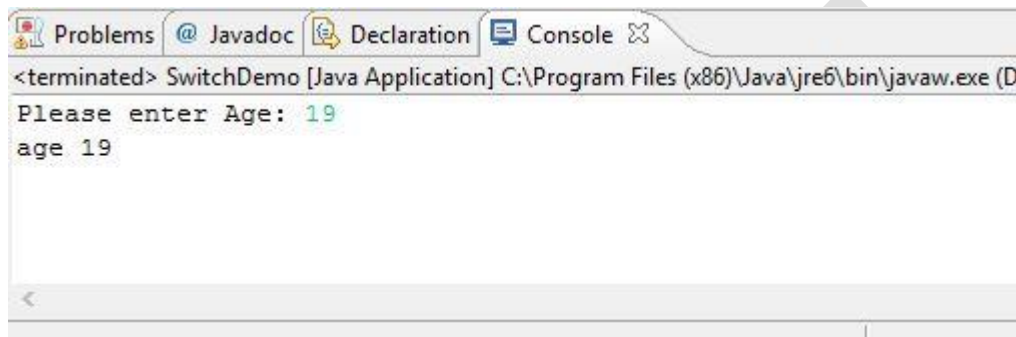
```

```

        break;
    default:
        System.out.println( "not matched" );
        break;
    }
}
}

```

Output



An expression must be of a type of byte, short, int or char. Each of the values specified in the case statement must be of a type compatible with the expression. Duplicate case values are not allowed. The break statement is used inside the switch to terminate a statement sequence. The break statement is optional in the switch statement

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 (Refer [this](#) for details).
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.

Object and Class Example: main within class

In this example, we have created a Student class that have two data members id and name. We are creating the object of the Student class by new keyword and printing the objects value.

Here, we are creating main() method inside the class.

File: Student.java

```
class Student{  
    int id;//field or data member or instance variable  
    String name;  
  
    public static void main(String args[]){  
        Student s1=new Student();//creating an object of Student  
        System.out.println(s1.id);//accessing member through reference variable  
        System.out.println(s1.name);  
    }  
}
```

Output:

0
Null

Object and Class Example: main outside class

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.

We can have multiple classes in different java files or single java file. If you define multiple classes in a single java source file, it is a good idea to save the file name with the class name which has main() method.

File: TestStudent1.java

```
1. class Student{
2.   int id;
3.   String name;
4. }
5. class TestStudent1{
6.   public static void main(String args[]){
7.     Student s1=new Student();
8.     System.out.println(s1.id);
9.     System.out.println(s1.name);
10.  }
11. }
```

Output:

```
0
null
```

3 Ways to initialize object

There are 3 ways to initialize object in java.

1. By reference variable
2. By method
3. By constructor

1) Object and Class Example: Initialization through reference

Initializing object simply means storing data into object. Let's see a simple example where we are going to initialize object through reference variable.

File: TestStudent2.java

```
1. class Student{
2.   int id;
3.   String name;
4. }
5. class TestStudent2{
6.   public static void main(String args[]){
7.     Student s1=new Student();
8.     s1.id=101;
9.     s1.name="Sonoo";
10.   System.out.println(s1.id+" "+s1.name);//printing members with a white space
```



```
11. }  
12. }
```

Output:

```
101 Sonoo
```

We can also create multiple objects and store information in it through reference variable.

File: TestStudent3.java

```
1. class Student{  
2.     int id;  
3.     String name;  
4. }  
5. class TestStudent3{  
6.     public static void main(String args[]){  
7.         //Creating objects  
8.         Student s1=new Student();  
9.         Student s2=new Student();  
10.        //Initializing objects  
11.        s1.id=101;  
12.        s1.name="Sonoo";  
13.        s2.id=102;  
14.        s2.name="Amit";  
15.        //Printing data  
16.        System.out.println(s1.id+ " "+s1.name);  
17.        System.out.println(s2.id+ " "+s2.name);  
18.    }  
19. }
```

Output:

```
101 Sonoo  
102 Amit
```

2) Object and Class Example: Initialization through method

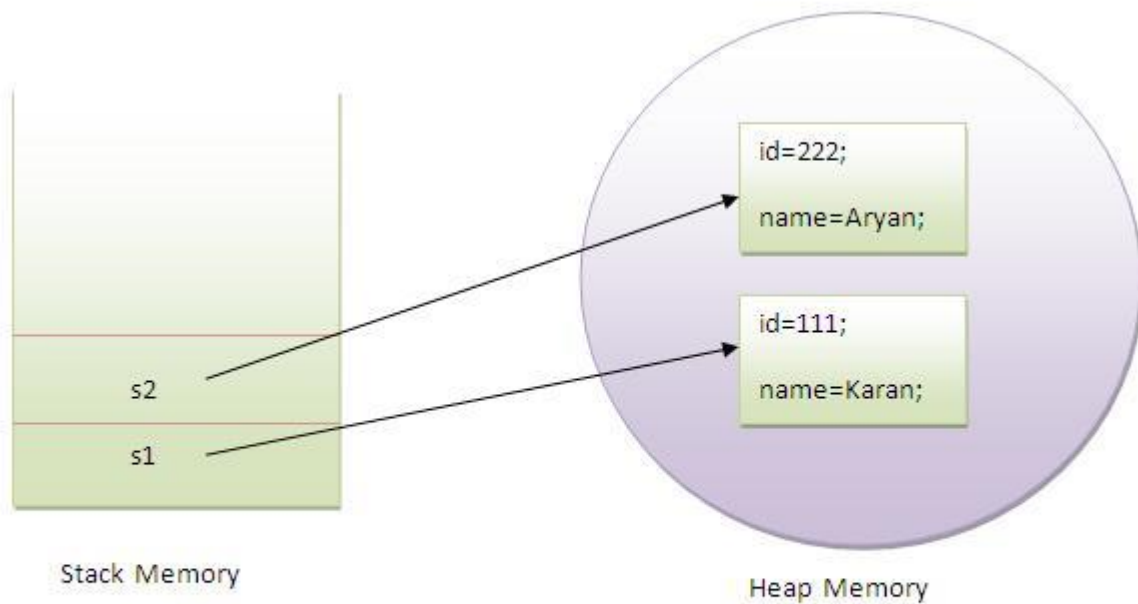
In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

File: TestStudent4.java

```
1. class Student{
2.   int rollno;
3.   String name;
4.   void insertRecord(int r, String n){
5.     rollno=r;
6.     name=n;
7.   }
8.   void displayInformation(){System.out.println(rollno+ " "+name);}
9. }
10. class TestStudent4{
11.   public static void main(String args[]){
12.     Student s1=new Student();
13.     Student s2=new Student();
14.     s1.insertRecord(111,"Karan");
15.     s2.insertRecord(222,"Aryan");
16.     s1.displayInformation();
17.     s2.displayInformation();
18.   }
19. }
```

Output:

```
111 Karan
222 Aryan
```



As you can see in the above figure, object gets the memory in heap memory area. The reference variable refers to the object allocated in the heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

Object and Class Example: Employee

Let's see an example where we are maintaining records of employees.

File: TestEmployee.java

```
1. class Employee{  
2.     int id;  
3.     String name;  
4.     float salary;  
5.     void insert(int i, String n, float s) {  
6.         id=i;  
7.         name=n;  
8.         salary=s;  
9.     }  
10.    void display(){System.out.println(id+" "+name+" "+salary);}
```

```
11. }
12. public class TestEmployee {
13. public static void main(String[] args) {
14.     Employee e1=new Employee();
15.     Employee e2=new Employee();
16.     Employee e3=new Employee();
17.     e1.insert(101,"ajeet",45000);
18.     e2.insert(102,"irfan",25000);
19.     e3.insert(103,"nakul",55000);
20.     e1.display();
21.     e2.display();
22.     e3.display();
23. }
24. }
```

Output:

```
101 ajeet 45000.0
102 irfan 25000.0
103 nakul 55000.0
```

Object and Class Example: Rectangle

There is given another example that maintains the records of Rectangle class.

File: TestRectangle1.java

```
1. class Rectangle{
2.     int length;
3.     int width;
4.     void insert(int l, int w){
5.         length=l;
6.         width=w;
7.     }
8.     void calculateArea(){System.out.println(length*width);}
9. }
10. class TestRectangle1{
11. public static void main(String args[]){
12.     Rectangle r1=new Rectangle();
13.     Rectangle r2=new Rectangle();
14.     r1.insert(11,5);
15.     r2.insert(3,15);
```

```
16. r1.calculateArea();
17. r2.calculateArea();
18. }
19. }
```

Output:

```
55
45
```

Constructor in java:

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

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

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

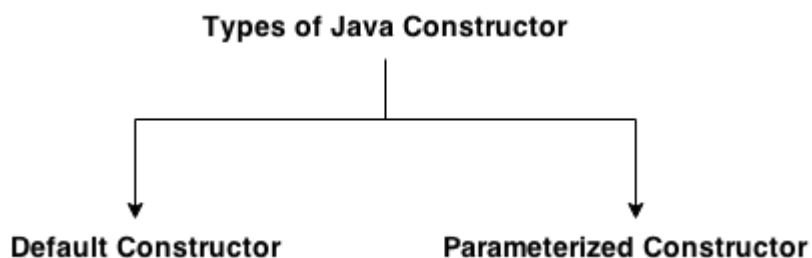
There are basically two rules defined for the constructor.

1. Constructor name must be same as its class name
2. Constructor must have no explicit return type

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

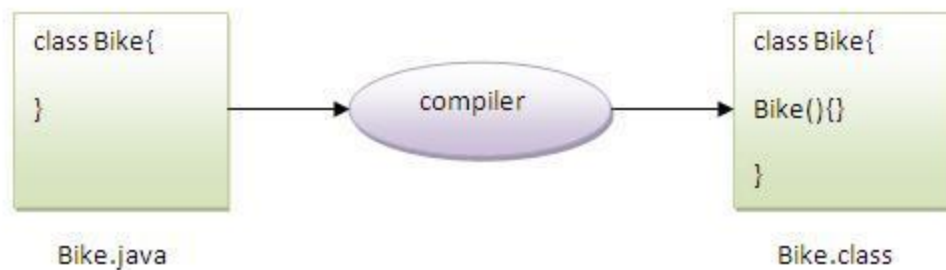
In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

```
1. class Bike1{  
2.   Bike1(){System.out.println("Bike is created");}  
3. public static void main(String args[]){  
4.   Bike1 b=new Bike1();  
5. }  
6. }
```

Output:

Bike is created

Rule: If there is no constructor in a class, compiler automatically creates a default constructor.



Q) What is the purpose of default constructor?

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. class Student3{  
2. int id;  
3. String name;  
4.   
5. void display(){System.out.println(id+" "+name);}  
6.   
7. public static void main(String args[]){  
8. Student3 s1=new Student3();  
9. Student3 s2=new Student3();  
10. s1.display();  
11. s2.display();  
12. }  
13. }
```

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.

Java parameterized constructor

A constructor which has a specific number of parameters is called parameterized constructor.

Why use parameterized constructor?

Parameterized constructor is used to provide different values to the distinct objects.

Example of parameterized constructor

In this example, we have created the constructor of Student class that have two

parameters. We can have any number of parameters in the constructor.

```
1. class Student4{
2.     int id;
3.     String name;
4.
5.     Student4(int i,String n){
6.         id = i;
7.         name = n;
8.     }
9.     void display(){System.out.println(id+" "+name);}
10.
11.     public static void main(String args[]){
12.         Student4 s1 = new Student4(111,"Karan");
13.         Student4 s2 = new Student4(222,"Aryan");
14.         s1.display();
15.         s2.display();
16.     }
17. }
```

Output:

```
111 Karan
222 Aryan
```

Constructor Overloading in Java

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

Example of Constructor Overloading

```
1. class Student5{
2.     int id;
3.     String name;
4.     int age;
5.     Student5(int i,String n){
6.         id = i;
```



```

7.   name = n;
8.   }
9.   Student5(int i,String n,int a){
10.  id = i;
11.  name = n;
12.  age=a;
13.  }
14.  void display(){System.out.println(id+" "+name+" "+age);}
15.
16.  public static void main(String args[]){
17.  Student5 s1 = new Student5(111,"Karan");
18.  Student5 s2 = new Student5(222,"Aryan",25);
19.  s1.display();
20.  s2.display();
21.  }
22. }

```

Output:

```

111 Karan 0
222 Aryan 25

```

Difference between constructor and method in java

There are many differences between constructors and methods. They are given below.

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	Method is not provided by

constructor.	compiler in any case.
Constructor name must be same as the class name.	Method name may or may not be same as class name.

What is THIS Keyword in Java?

Keyword THIS is a reference variable in Java that refers to the current object.

The various usages of 'THIS' keyword in Java are as follows:

- It can be used to refer instance variable of current class
- It can be used to invoke or initiate current class constructor
- It can be passed as an argument in the method call
- It can be passed as argument in the constructor call
- It can be used to return the current class instance

Instance Variable Hiding in Java

In Java, if there a local variable in a method with same name as instance variable, then the local variable hides the instance variable. If we want to reflect the change made over to the instance variable, this can be achieved with the help of [this reference](#).

```
class Test
{
    // Instance variable or member variable
    private int value = 10;

    void method()
    {
        // This local variable hides instance variable
        int value = 40;

        System.out.println("Value of Instance variable :")
    }
}
```

```

        + this.value);

        System.out.println("Value of Local variable : "

        + value);

    }

}

class UseTest

{

    public static void main(String args[])

    {

        Test obj1 = new Test();

        obj1.method();

    }

}

```

Output:

```

Value of Instance variable :10
Value of Local variable :40

```

Method Overloading:

Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. It is similar to constructor overloading in Java, that allows a class to have more than one constructor having different argument lists.

Three ways to overload a method

In order to overload a method, the argument lists of the methods must differ in either of these:

1. Number of parameters.

For example: This is a valid case of overloading

```
add(int, int)
```

```
add(int, int, int)
```

2. Data type of parameters.

For example:

```
add(int, int)
add(int, float)
```

3. Sequence of Data type of parameters.

For example:

```
add(int, float)
add(float, int)
```

Invalid case of method overloading:

When I say argument list, I am not talking about return type of the method, for example if two methods have same name, same parameters and have different return type, then this is not a valid method overloading example. This will throw compilation error.

```
int add(int, int)
float add(int, int)
```

Method overloading is an example of Static Polymorphism. We will discuss polymorphism and types of it in a separate tutorial.

Points to Note:

1. Static Polymorphism is also known as compile time binding or early binding.
2. Static binding happens at compile time. Method overloading is an example of static binding where binding of method call to its definition happens at Compile time.

Method Overloading examples

As discussed in the beginning of this guide, method overloading is done by declaring same method with different parameters. The parameters must be different in either of these: number, sequence or types of parameters (or arguments). Lets see examples of each of these cases.

Argument list is also known as parameter list

Example 1: Overloading – Different Number of parameters in argument list

This example shows how method overloading is done by having different number of parameters

```
class DisplayOverloading
{
    public void disp(char c)
    {
```

```
        System.out.println(c);
    }
    public void disp(char c, int num)
    {
        System.out.println(c + " "+num);
    }
}
class Sample
{
    public static void main(String args[])
    {
        DisplayOverloading obj = new DisplayOverloading();
        obj.disp('a');
        obj.disp('a',10);
    }
}
```

Output:

a
a 10

In the above example – method disp() is overloaded based on the number of parameters – We have two methods with the name disp but the parameters they have are different. Both are having different number of parameters.

Example 2: Overloading – Difference in data type of parameters

In this example, method disp() is overloaded based on the data type of parameters – We have two methods with the name disp(), one with parameter of char type and another method with the parameter of int type.

```
class DisplayOverloading2
{
    public void disp(char c)
    {
        System.out.println(c);
    }
}
```

```
public void disp(int c)
{
    System.out.println(c);
}
}
```

```
class Sample2
{
    public static void main(String args[])
    {
        DisplayOverloading2 obj = new DisplayOverloading2();
        obj.disp('a');
        obj.disp(5);
    }
}
```

Output:

a
5

Example3: Overloading – Sequence of data type of arguments

Here method disp() is overloaded based on sequence of data type of parameters – Both the methods have different sequence of data type in argument list. First method is having argument list as (char, int) and second is having (int, char). Since the sequence is different, the method can be overloaded without any issues.

```
class DisplayOverloading3
{
    public void disp(char c, int num)
    {
        System.out.println("I'm the first definition of method disp");
    }
    public void disp(int num, char c)
    {
        System.out.println("I'm the second definition of method disp" );
    }
}
```

```

    }
}
class Sample3
{
    public static void main(String args[])
    {
        DisplayOverloading3 obj = new DisplayOverloading3();
        obj.disp('x', 51 );
        obj.disp(52, 'y');
    }
}

```

Output:

I'm the first definition of method disp

I'm the second definition of method disp

Constructor overloading in java.

The process of defining more than one constructor with different parameters in a class is known as constructor overloading. Parameters can differ in number, type or order.

Example:

```

class StudentData
{
    private int stuID;
    private String stuName;
    private int stuAge;
    StudentData()
    {
        //Default constructor
        stuID = 100;
        stuName = "New Student";
        stuAge = 18;
    }
    StudentData(int num1, String str, int num2)
    {
        //Parameterized constructor
    }
}

```

```
    stuID = num1;
    stuName = str;
    stuAge = num2;
}
//Getter and setter methods
public int getStuID() {
    return stuID;
}
public void setStuID(int stuID) {
    this.stuID = stuID;
}
public String getStuName() {
    return stuName;
}
public void setStuName(String stuName) {
    this.stuName = stuName;
}
public int getStuAge() {
    return stuAge;
}
public void setStuAge(int stuAge) {
    this.stuAge = stuAge;
}

public static void main(String args[])
{
    //This object creation would call the default constructor
    StudentData myobj = new StudentData();
    System.out.println("Student Name is: "+myobj.getStuName());
    System.out.println("Student Age is: "+myobj.getStuAge());
    System.out.println("Student ID is: "+myobj.getStuID());

    /*This object creation would call the parameterized
```



```

    * constructor StudentData(int, String, int)*/
    StudentData myobj2 = new StudentData(555, "Chaitanya", 25);
    System.out.println("Student Name is: "+myobj2.getStuName());
    System.out.println("Student Age is: "+myobj2.getStuAge());
    System.out.println("Student ID is: "+myobj2.getStuID());
}
}

```

Output:

Student Name is: New Student

Student Age is: 18

Student ID is: 100

Student Name is: Sreekanth

Student Age is: 25

Student ID is: 555

Using objects as parameters:

- While creating a variable of a class type, we only create a reference to an object. Thus, when we pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument.
- This effectively means that objects act as if they are passed to methods by use of call-by-reference.
- Changes to the object inside the method do reflect in the object used as an argument.
- In Java we can pass objects to methods. For example, consider the following program :

```

// Java program to demonstrate objects
// passing to methods.
class ObjectPassDemo
{
    int a, b;

    ObjectPassDemo(int i, int j)
    {
        a = i;
        b = j;
    }

    // return true if o is equal to the invoking
    // object notice an object is passed as an
    // argument to method
    boolean equalTo(ObjectPassDemo o)
    {
        return (o.a == a && o.b == b);
    }
}

```

```

}

// Driver class
public class Test
{
    public static void main(String args[])
    {
        ObjectPassDemo ob1 = new ObjectPassDemo(100, 22);
        ObjectPassDemo ob2 = new ObjectPassDemo(100, 22);
        ObjectPassDemo ob3 = new ObjectPassDemo(-1, -1);

        System.out.println("ob1 == ob2: " + ob1.equals(ob2));
        System.out.println("ob1 == ob3: " + ob1.equals(ob3));
    }
}

```

- Run on IDE
- Output:

```

▪ ob1 == ob2: true
▪ ob1 == ob3: false

```

Recursion in Java:

Recursion in java is a process in which a method calls itself continuously. A method in java that calls itself is called recursive method.

It makes the code compact but complex to understand.

Syntax:

```

returntype methodname(){
//code to be executed
methodname();//calling same method
}

```

Java Recursion Example 4: Fibonacci Series

```

public class RecursionExample4 {
    static int n1=0,n2=1,n3=0;
    static void printFibo(int count){
        if(count>0){
            n3 = n1 + n2;
            n1 = n2;
            n2 = n3;

```

```
        System.out.print(" "+n3);  
        printFibo(count-1);  
    }  
}
```

```
public static void main(String[] args) {  
    int count=15;  
    System.out.print(n1+" "+n2);//printing 0 and 1  
    printFibo(count-2);//n-2 because 2 numbers are already printed  
}  
}
```

Output:

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377

Access control:

Encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: access control. Java's access specifiers are public, private, and protected. Java also defines a default access level protected applies only when inheritance is involved. When a member of a class is modified by the public specifier, then that member can be accessed by any other code.

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

Access specifiers

public:

A class, method, constructor, interface etc declared public can be accessed from any other class. Therefore fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.

private:

Methods, Variables and Constructors that are declared private can only be accessed within the declared class itself. Private access modifier is the most restrictive access level. Class and interfaces cannot be private. Variables that are declared private can be accessed outside the class if public getter methods are present in the class.

protect:

Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class. The protected access modifier cannot be applied to class and interfaces.

Default:

Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc. A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public.

Example for access control

```
/* This program demonstrates the difference between public and private.*/  
class Test  
{  
    int a; // default access  
    public int b; // public access  
    private int c; // private access  
    // methods to access c  
    void setc(int i) // set c's value  
    {  
        c = i;  
    }  
    int getc() // get c's value  
    {  
        return c;  
    }  
}  
class AccessTest  
{  
    public static void main(String args[])  
    {
```

```
Test ob = new Test();  
// These are OK, a and b may be accessed directly  
ob.a = 10;  
ob.b = 20;  
// This is not OK and will cause an error  
// ob.c = 100; // Error!  
// You must access c through its methods  
ob.setc(100); // OK  
System.out.println("a, b, and c: " + ob.a + " " +  
ob.b + " " + ob.getc());  
}  
}
```

Java static nested class

A static class i.e. created inside a class is called static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name.

- It can access static data members of outer class including private.
- Static nested class cannot access non-static (instance) data member or method.

Java static nested class example with instance method

```
class TestOuter1{  
    static int data=30;  
    static class Inner{  
        void msg(){System.out.println("data is "+data);}  
    }  
    public static void main(String args[]){  
        TestOuter1.Inner obj=new TestOuter1.Inner();  
        obj.msg();  
    }  
}
```

Output:

data is 30

Final :

The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

- variable
- method
- class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these

String classes:

What is String in java

Generally, string is a sequence of characters. But in java, string is an object that represents a sequence of characters. The java.lang.String class is used to create string object.

How to create String object?

There are two ways to create String object:

1. By string literal
2. By new keyword

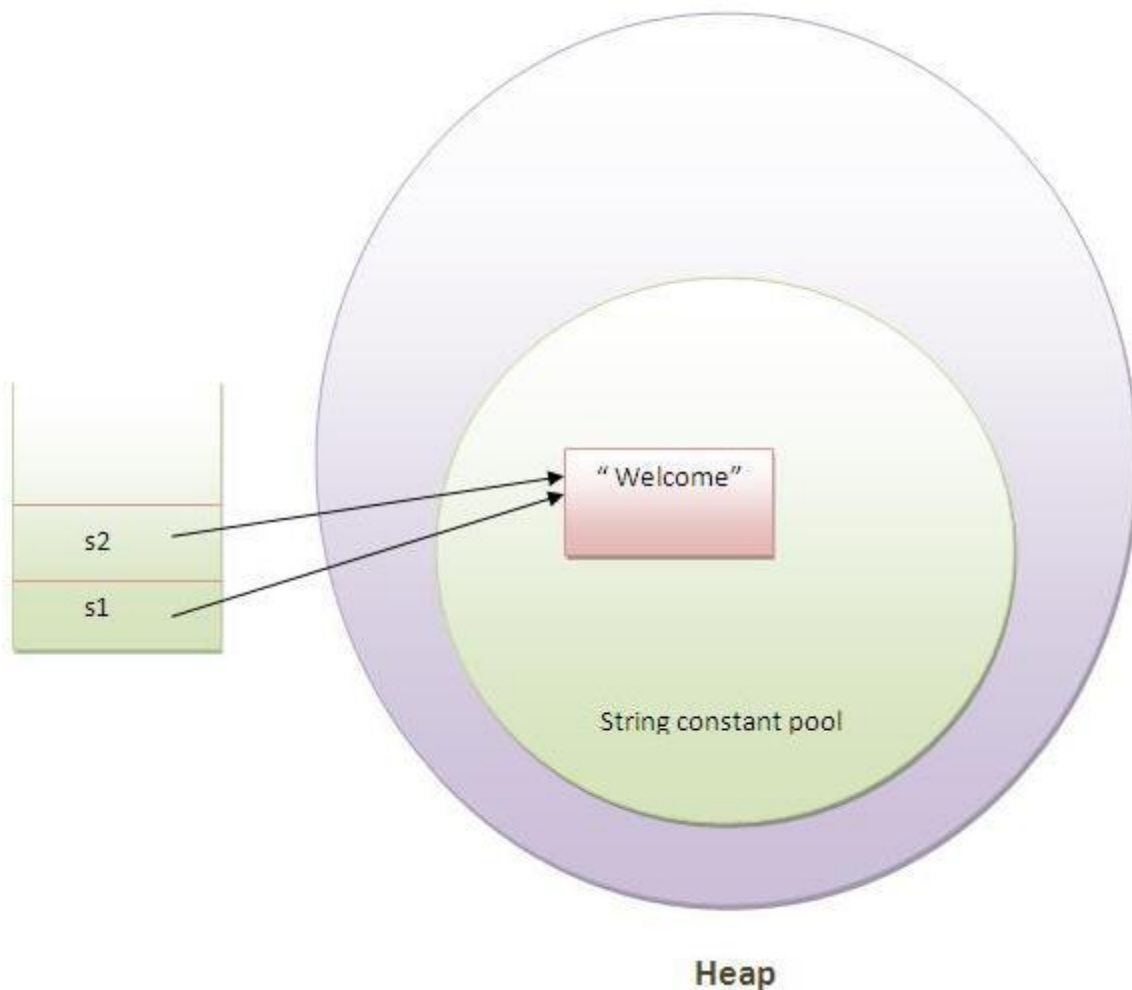
1) String Literal

Java String literal is created by using double quotes. For Example:

1. String s="welcome";

Each time you create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, a reference to the pooled instance is returned. If string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

1. String s1="Welcome";
2. String s2="Welcome";//will not create new instance



In the above example only one object will be created. Firstly JVM will not find any string object with the value "Welcome" in string constant pool, so it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create new object but will return the reference to the same instance.

Note: String objects are stored in a special memory area known as string constant pool.

Why java uses concept of string literal?

To make Java more memory efficient (because no new objects are created if it exists already in string constant pool).

2) By new keyword

1. String s=**new** String("Welcome");//creates two objects and one reference variable

In such case, JVM will create a new string object in normal(non pool) heap memory and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in heap(non pool).

Java String Example

1. **public class** StringExample{
2. **public static void** main(String args[]){
3. String s1="java";//creating string by java string literal
4. **char** ch[]={'s','t','r','i','n','g','s'};
5. String s2=**new** String(ch);//converting char array to string
6. String s3=**new** String("example");//creating java string by new keyword
7. System.out.println(s1);
8. System.out.println(s2);
9. System.out.println(s3);
- 10.}}

```
java
strings
example
```

Java String class methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

No.	Method	Description
1	<u>char charAt(int index)</u>	returns char value for the particular index
2	<u>int length()</u>	returns string length
3	<u>static String format(String format, Object... args)</u>	returns formatted string
4	<u>static String format(Locale l, String format, Object... args)</u>	returns formatted string with given locale

5	<u>String substring(int beginIndex)</u>	returns substring for given begin index
6	<u>String substring(int beginIndex, int endIndex)</u>	returns substring for given begin index and end index
7	<u>boolean contains(CharSequence s)</u>	returns true or false after matching the sequence of char value
8	<u>static String join(CharSequence delimiter, CharSequence... elements)</u>	returns a joined string
9	<u>static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)</u>	returns a joined string
10	<u>boolean equals(Object another)</u>	checks the equality of string with object
11	<u>boolean isEmpty()</u>	checks if string is empty
12	<u>String concat(String str)</u>	concatinates specified string
13	<u>String replace(char old, char new)</u>	replaces all occurrences of specified char value
14	<u>String replace(CharSequence old, CharSequence new)</u>	replaces all occurrences of specified CharSequence
15	<u>static String equalsIgnoreCase(String another)</u>	compares another string. It doesn't check case.

16	<u>String[] split(String regex)</u>	returns splitted string matching regex
17	<u>String[] split(String regex, int limit)</u>	returns splitted string matching regex and limit
18	<u>String intern()</u>	returns interned string
19	<u>int indexOf(int ch)</u>	returns specified char value index
20	<u>int indexOf(int ch, int fromIndex)</u>	returns specified char value index starting with given index
21	<u>int indexOf(String substring)</u>	returns specified substring index
22	<u>int indexOf(String substring, int fromIndex)</u>	returns specified substring index starting with given index
23	<u>String toLowerCase()</u>	returns string in lowercase.
24	<u>String toLowerCase(Locale l)</u>	returns string in lowercase using specified locale.
25	<u>String toUpperCase()</u>	returns string in uppercase.
26	<u>String toUpperCase(Locale l)</u>	returns string in uppercase using specified locale.

27	<u>String trim()</u>	removes beginning and ending spaces of this string.
28	<u>static String.valueOf(int value)</u>	converts given type into string. It is overloaded.

Command line argument in Java

The command line argument is the argument passed to a program at the time when you run it. To access the command-line argument inside a java program is quite easy, they are stored as string in **String** array passed to the args parameter of `main()` method.

Example

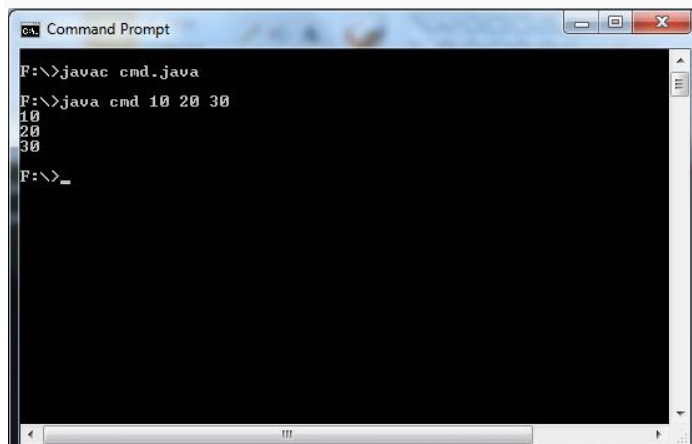
```
class cmd
{
    public static void main(String[] args)
    {
        for(int i=0;i< args.length;i++)
        {
            System.out.println(args[i]);
        }
    }
}
```

Execute this program as `java cmd 10 20 30`

10

20

30



Inheritance in Java

Inheritance in java is a mechanism in which one object acquires all the properties and behaviors of parent object.

The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.

Inheritance represents the IS-A relationship, also known as parent-child relationship.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
    //methods and fields
}
```

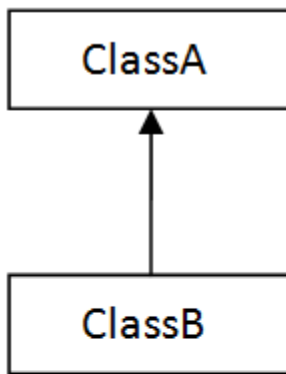
The extends keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called parent or super class and the new class is called child or subclass.

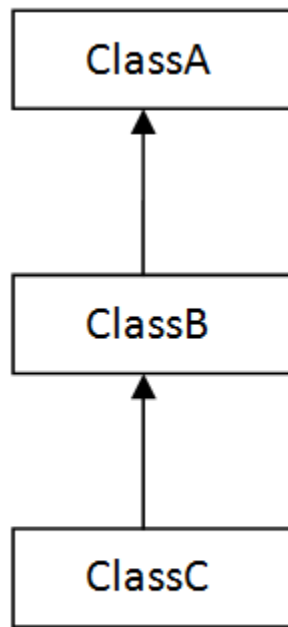
Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

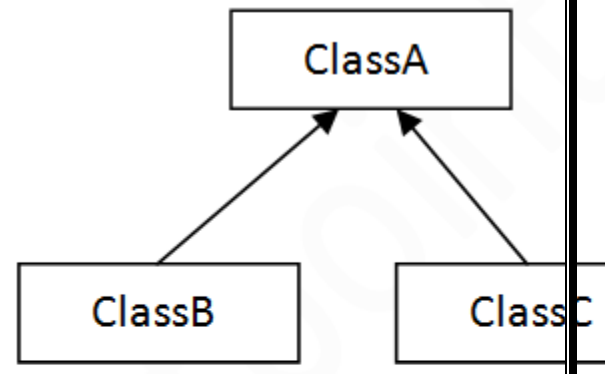
In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



1) Single



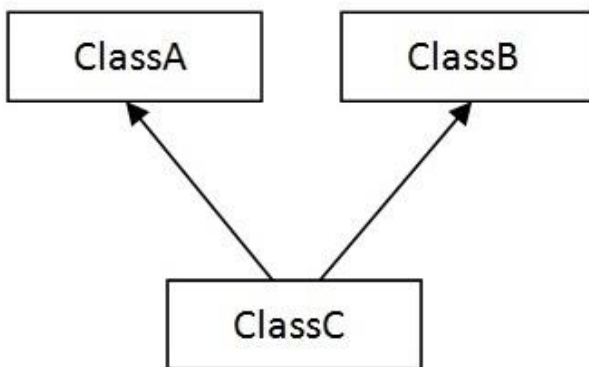
2) Multilevel



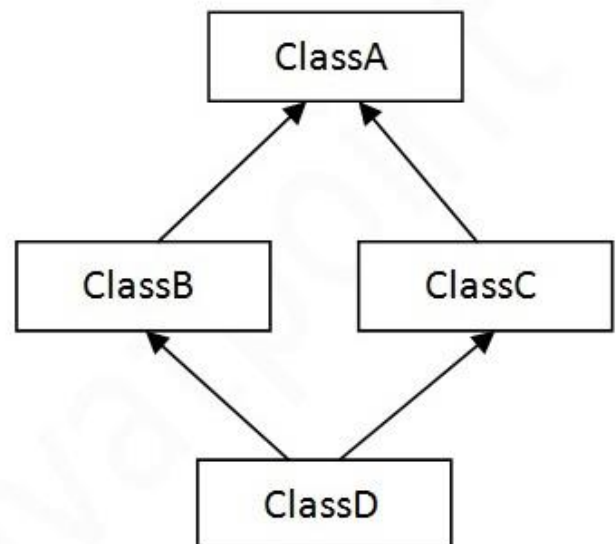
3) Hierarchical

Note: Multiple inheritance is not supported in java through class.

When a class extends multiple classes i.e. known as multiple inheritance. For Example:



4) Multiple



5) Hybrid

Single Inheritance Example

File: *TestInheritance.java*

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class TestInheritance{
8. public static void main(String args[]){
9. Dog d=new Dog();
10. d.bark();
11. d.eat();
12. }}
```

Output:

```
barking...
eating...
```

Multilevel Inheritance Example

File: *TestInheritance2.java*

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class BabyDog extends Dog{
8. void weep(){System.out.println("weeping...");}
9. }
10. class TestInheritance2{
11. public static void main(String args[]){
12. BabyDog d=new BabyDog();
13. d.weep();
14. d.bark();
15. d.eat();
}
```

16.}}

Output:

```
weeping...  
barking...  
eating...
```

Hierarchical Inheritance Example

File: TestInheritance3.java

```
1. class Animal{  
2. void eat(){System.out.println("eating...");}  
3. }  
4. class Dog extends Animal{  
5. void bark(){System.out.println("barking...");}  
6. }  
7. class Cat extends Animal{  
8. void meow(){System.out.println("meowing...");}  
9. }  
10. class TestInheritance3{  
11. public static void main(String args[]){  
12. Cat c=new Cat();  
13. c.meow();  
14. c.eat();  
15. //c.bark();//C.T.Error  
16. }}
```

Output:

```
meowing...  
eating...
```

Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and you call it from child class object, there will be ambiguity to call method of A or B class.

Since compile time errors are better than runtime errors, java renders compile time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error now.

```

1. class A{
2. void msg(){System.out.println("Hello");}
3. }
4. class B{
5. void msg(){System.out.println("Welcome");}
6. }
7. class C extends A,B{//suppose if it were
8.
9. Public Static void main(String args[]){
10. C obj=new C();
11. obj.msg();//Now which msg() method would be invoked?
12. }
13. }

```

Output:

Compile Time Error

Superclass Variable Can Reference a Subclass Object

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass. In the following program, obj is a reference to NewData object, Since NewData is a subclass of Data, it is permissible to assign obj a reference to the NewData object. When a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass. This is way obj can't access data3 and data4 even it refers to a NewData object.

Program:

```

class Data {

    int data1;
    int data2;
}

```

```

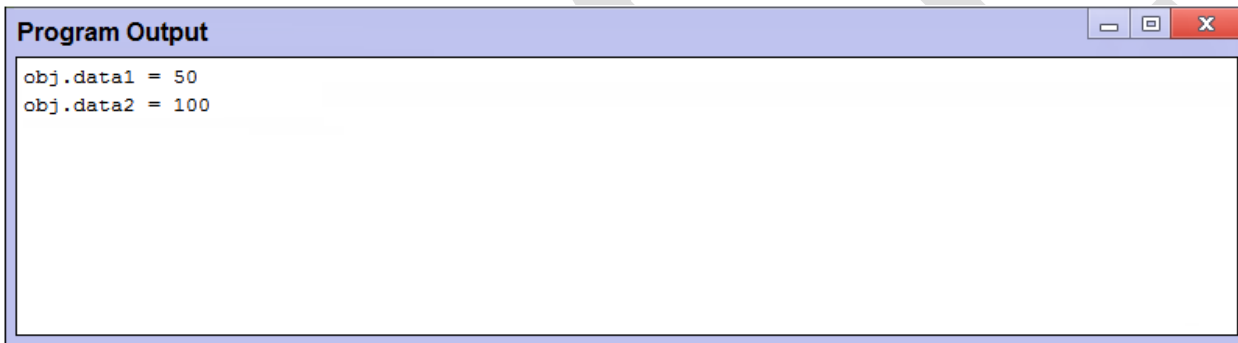
class NewData extends Data{

    int data3;
    int data4;
}

```



```
public class Javaapp {  
  
    public static void main(String[] args) {  
  
        Data obj = new NewData();  
        obj.data1 = 50;  
        obj.data2 = 100;  
        System.out.println("obj.data1 = "+obj.data1);  
        System.out.println("obj.data2 = "+obj.data2);  
    }  
}
```



The screenshot shows a window titled "Program Output" with a standard Windows-style title bar (minimize, maximize, close buttons). The output text inside the window is:

```
obj.data1 = 50  
obj.data2 = 100
```

Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in java.

In other words, If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

- method must have same name as in the parent class
- method must have same parameter as in the parent class.
- must be IS-A relationship (inheritance).

Example of method overriding

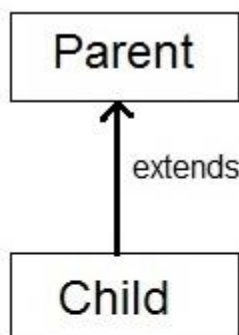
In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method is same and there is IS-A relationship between the classes, so there is method overriding.

```
1. class Vehicle{
2. void run(){System.out.println("Vehicle is running");}
3. }
4. class Bike2 extends Vehicle{
5. void run(){System.out.println("Bike is running safely");}
6.
7. public static void main(String args[]){
8. Bike2 obj = new Bike2();
9. obj.run();
10. }
```

Output:Bike is running safely

Runtime Polymorphism or Dynamic method dispatch

Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime. This is how java implements runtime polymorphism. When an overridden method is called by a reference, java determines which version of that method to execute based on the type of object it refer to. In simple words the type of object which it referred determines which version of overridden method will be called.



Parent p = new Parent();

Child c = new Child();

Parent p = new Child();

Upcasting

~~Child c = new Parent();~~

incompatible type

Upcasting

When **Parent** class reference variable refers to **Child** class object, it is known as **Upcasting**

Example

```
class Game
{
    public void type()
    { System.out.println("Indoor & outdoor"); }
}

Class Cricket extends Game
{
    public void type()
    { System.out.println("outdoor game"); }

    public static void main(String[] args)
    {
        Game gm = new Game();
        Cricket ck = new Cricket();
        gm.type();
        ck.type();
        gm=ck;           //gm refers to Cricket object
        gm.type();       //calls Cricket's version of type
    }
}
```

Indoor & outdoor

Outdoor game

Outdoor game

Abstract class in Java

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

Before learning java abstract class, let's understand the abstraction in java first.

Abstraction in Java

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

Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

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

Abstract class in Java

A class that is declared as abstract is known as abstract class. It needs to be extended and its method implemented. It cannot be instantiated.

Example abstract class

```
abstract class A{
```

abstract method

A method that is declared as abstract and does not have implementation is known as abstract method.

Example abstract method

```
abstract void printStatus();//no body and abstract
```

Example of abstract class that has abstract method

In this example, Bike the abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

```
abstract class Bike{
    abstract void run();
}

class Honda4 extends Bike{
    void run(){System.out.println("running safely..");}

    public static void main(String args[]){
        Bike obj = new Honda4();
        obj.run();
    }
}
```

```
}
```

Output

running safely..

Packages and Interfaces:

Java Package

A **java package** is a group of similar types of classes, interfaces and sub-packages.

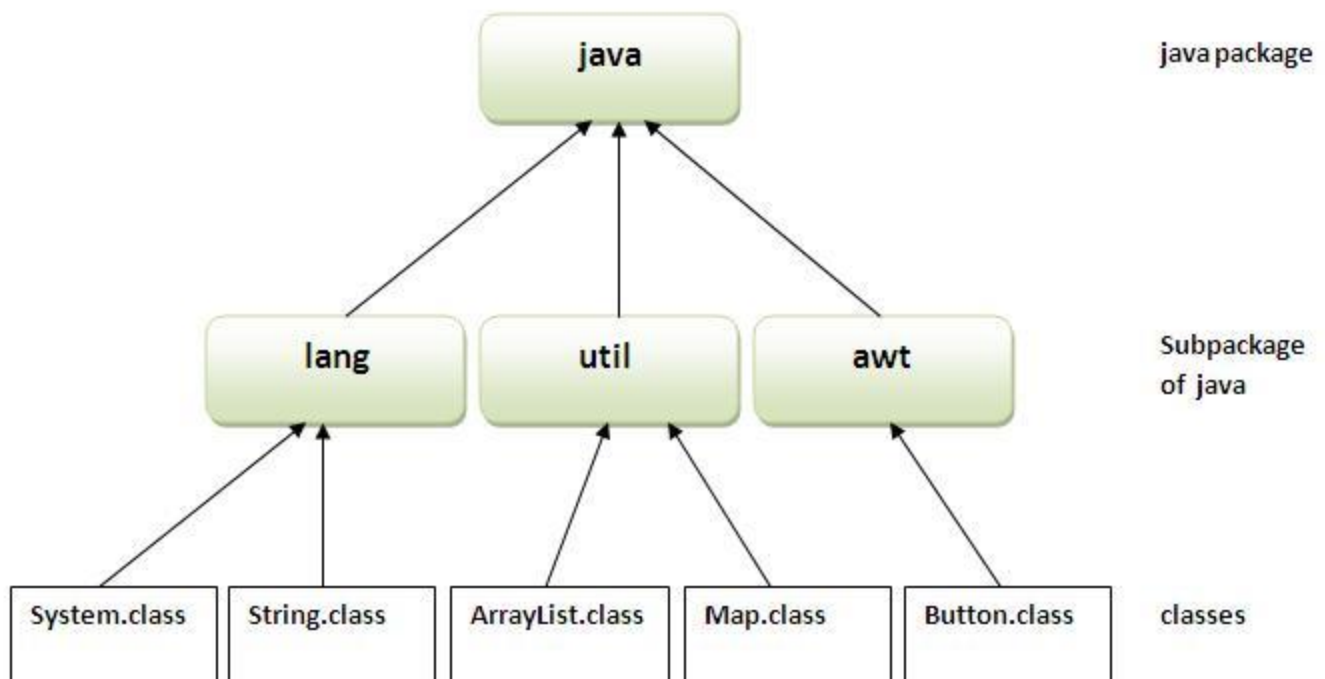
Package in java can be categorized in two form, built-in package and user-defined package.

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

Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

- 1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- 2) Java package provides access protection.
- 3) Java package removes naming collision.



Simple example of java package

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

```
//save as Simple.java
package mypack;
public class Simple{
    public static void main(String args[]){
        System.out.println("Welcome to package");
    }
}
```

How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

```
javac -d directory javafilename
```

For **example**

```
javac -d . Simple.java
```

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

mpile: javac -d . Simple.java

n: java mypack.Simple

Output:Welcome to package

l is a switch that tells the compiler where to put the class file i.e. it represents destination.
represents the current folder.

How to access package from another package?

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

- 1. import package.*;
- 2. import package.classname;
- 3. fully qualified name.

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

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

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

Output:Hello

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

```
1. //save by A.java
```

```

1. package pack;
2. public class A{
3.     public void msg(){System.out.println("Hello");}
4. }
5. //save by B.java
6. package mypack;
7. import pack.A;
8.
9. class B{
10.     public static void main(String args[]){
11.         A obj = new A();
12.         obj.msg();
13.     }
14. }

```

Output:Hello

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```

1. //save by A.java
2. package pack;
3. public class A{
4.     public void msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. package mypack;
8. class B{
9.     public static void main(String args[]){
10.         pack.A obj = new pack.A();//using fully qualified name
11.         obj.msg();
12.     }
13. }

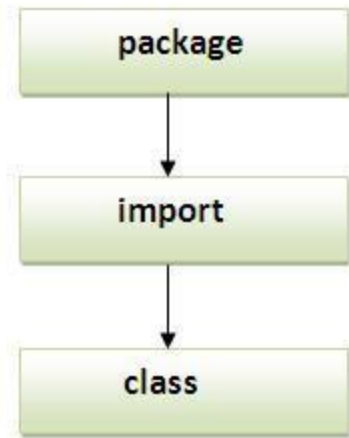
```

Output:Hello

Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

Note: Sequence of the program must be package then import then class.



Interface in Java

An **interface in java** is a blueprint of a class. It has static constants and abstract methods.

The interface in java is **a mechanism to achieve abstraction**. There can be only abstract methods in the java interface not method body. It is used to achieve abstraction and multiple inheritance in Java.

Java Interface also **represents IS-A relationship**.

It cannot be instantiated just like abstract class.

Why use Java interface?

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

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

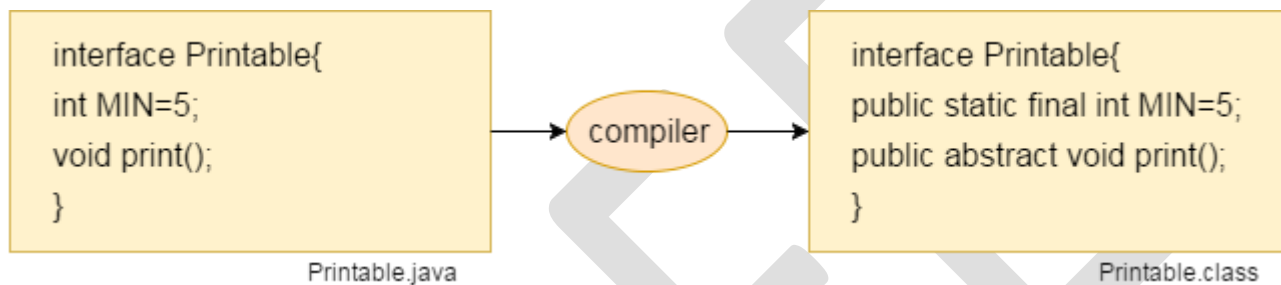
Java 8 Interface Improvement

Since Java 8, interface can have default and static methods which is discussed later.

Internal addition by compiler

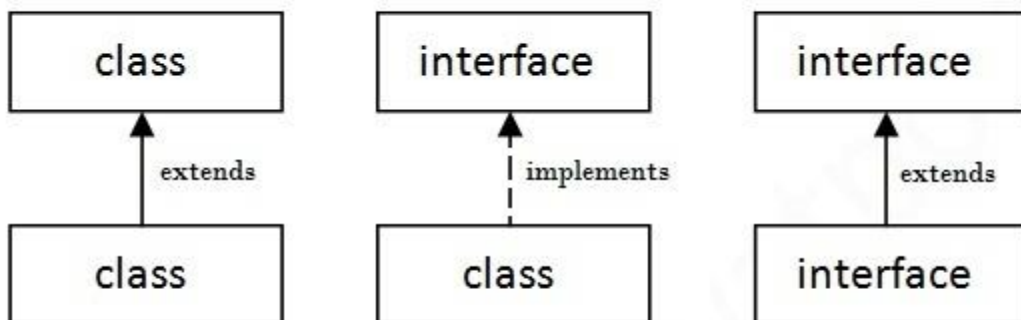
The java compiler adds **public** and **abstract** keywords before the interface method. More, it adds **public**, **static** and **final** keywords before data members.

In other words, Interface fields are public, static and final by default, and methods are public and abstract.



Understanding relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**.



Java Interface Example

In this example, Printable interface has only one method, its implementation is provided in the A class.

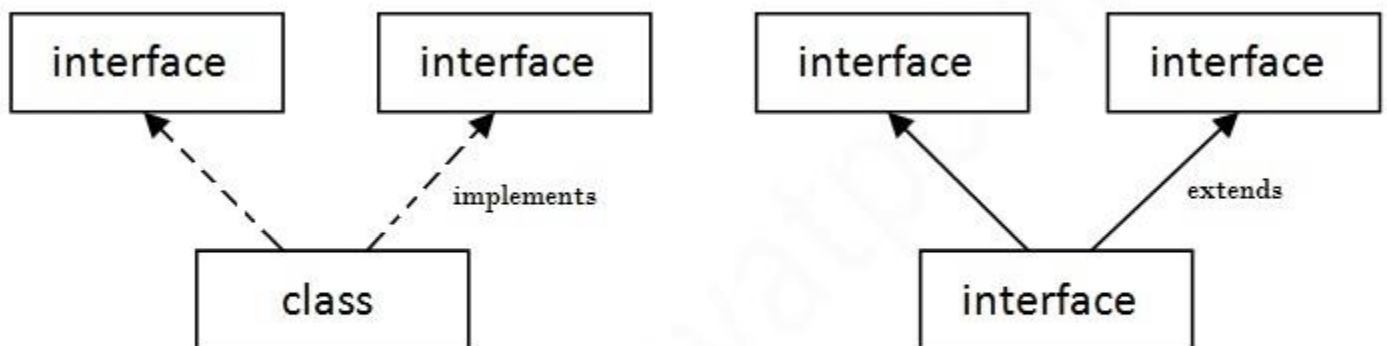
```
1. interface printable{
2. void print();
3. }
4. class A6 implements printable{
5. public void print(){System.out.println("Hello");}
6.
7. public static void main(String args[]){
8. A6 obj = new A6();
9. obj.print();
10. }
11. }
```

Output:

```
Hello
```

Multiple inheritance in Java by interface

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



Multiple Inheritance in Java

Interface inheritance

A class implements interface but one interface extends another interface .

```
1. interface Printable{
2. void print();
3. }
4. interface Showable extends Printable{
5. void show();
6. }
7. class TestInterface4 implements Showable{
8. public void print(){System.out.println("Hello");}
9. public void show(){System.out.println("Welcome");}
10.
11. public static void main(String args[]){
12. TestInterface4 obj = new TestInterface4();
13. obj.print();
14. obj.show();
15. }
16. }
```

Output:

```
Hello
Welcome
```