

ECE/EEE 225
OBJECT ORIENTED
PROGRAMMING THROUGH JAVA

UNIT – I:
OOPS CONCEPTS AND JAVA
PROGRAMMING

History of c, c++, java

- “C is a programming language which born at ‘AT & T’s Bell Laboratories’ of USA in 1972. It was written by Dennis Ritchie.
- purpose: to design the UNIX operating system.
- The committee formed by the American National Standards Institute (ANSI) approved a version of C in 1989 which is known as ANSI C.
- ANSI C was then approved by the International Standards Organization (ISO) in 1990.
- It was named C because its predecessor was called B which was developed by Ken Thompson of Bell Labs.

- “C++ was written by Bjarne Stroustrup at Bell Labs during 1983-1985.
- C++ is an extension of C. It is superset of C.
- Bjarne Stroustrup added feature of **OOP (Object Oriented Programming)** in C and formed what he called ‘C with Classes’.
- Java started to be developed in 1991 by James Gosling from Sun Microsystems and his team.
- The original name of Java is Oak. But it had to change its original name because Oak had been used by another programming language.
- Java is viewed as a programming language to design applications for the Internet.

- 1951 – [Regional Assembly Language](#)
- 1952 – [Autocode](#)
- 1954 – [IPL](#) (forerunner to LISP)
- 1955 – [FLOW-MATIC](#) (led to COBOL)
- 1957 – [FORTRAN](#) (first compiler)
- 1957 – [COMTRAN](#) (precursor to COBOL)
- 1958 – [LISP](#)
- 1958 – [ALGOL 58](#)
- 1959 – [FACT](#) (forerunner to COBOL)
- 1959 – [COBOL](#)
- 1959 – [RPG](#)
- 1962 – [APL](#)
- 1962 – [Simula](#)
- 1962 – [SNOBOL](#)
- 1963 – [CPL](#) (forerunner to C)
- 1964 – [Speakeasy](#)
- 1964 – [BASIC](#)
- 1964 – [PL/I](#)
- 1966 – [JOSS](#)
- 1966 - [MUMPS](#)
- 1967 – [BCPL](#) (forerunner to C)
- 1967 – [BCPL](#) (forerunner to B)
- 1968 – [Logo](#)
- 1969 – [B](#) (forerunner to C)
- 1970 – [Pascal](#)
- 1970 – [Forth](#)
- 1972 – [C](#)
- 1972 – [Smalltalk](#)
- 1972 – [Prolog](#)
- 1973 – [ML](#)
- 1975 – [Scheme](#)
- 1978 – [SQL](#) (a query language, later extended)

1980 – C++ (as C with classes, renamed in 1983)

1983 – Ada

1984 – Common Lisp

1984 – MATLAB

1984 – dBase III, dBase III Plus (Clipper and FoxPro as FoxBASE, later developing into Visual FoxPro)

1985 – Eiffel

1986 – Objective-C

- 1986 – LabVIEW (Visual Programming Language)

- 1986 – Erlang

- 1987 – Perl

- 1988 – Tcl

- 1988 – Wolfram Language (as part of Mathematica, only got a separate name in June 2013)

- 1989 – FL (Backus)

- 1990 – Haskell

- 1990 – Python

- 1991 – Visual Basic

- 1993 – Lua

- 1993 – R

- 1994 – CLOS (part of ANSI Common Lisp)

- 1995 – Ruby

- 1995 – Ada 95

- 1995 – Java

- 1995 – Delphi (Object Pascal)

- 1995 – JavaScript

- 1995 – PHP

- 1997 – Rebol

Software

- Software is a set of instructions or programs used to operate computer and to execute specific tasks.
- Two main categories of software:
- System software: to run computer and to provide platform for running applications.
- Application software: to run user applications.
- Programming languages: It is defined as a set of keywords and syntaxes used to perform a specific task.
- Two common types of low-level programming languages are [assembly language](#) and [machine language](#).
- Low level programming languages: machine friendly, difficult to understand.

- Machine language, or machine code, is the lowest level of computer languages.
- Assembly language is one step closer to a high-level language than machine language. It includes commands such as MOV (move), ADD (add), and SUB (subtract).
- Assembly language can be converted to the machine language using an [assembler](#).
- High level languages: the code is not recognized directly by the [CPU](#). Instead, it must be [compiled](#) into a low-level language.
- High level programming languages: closed to human languages. Programmer friendly, easy to understand, debug and maintain

Programming paradigms

- Programming paradigms are a way to classify **programming languages** based on their features.
- Monolithic programming
- Procedural programming
- Structured programming
- Object Oriented Programming

Monolithic programming

The Monolithic programming paradigm is the oldest. It has the following characteristics.

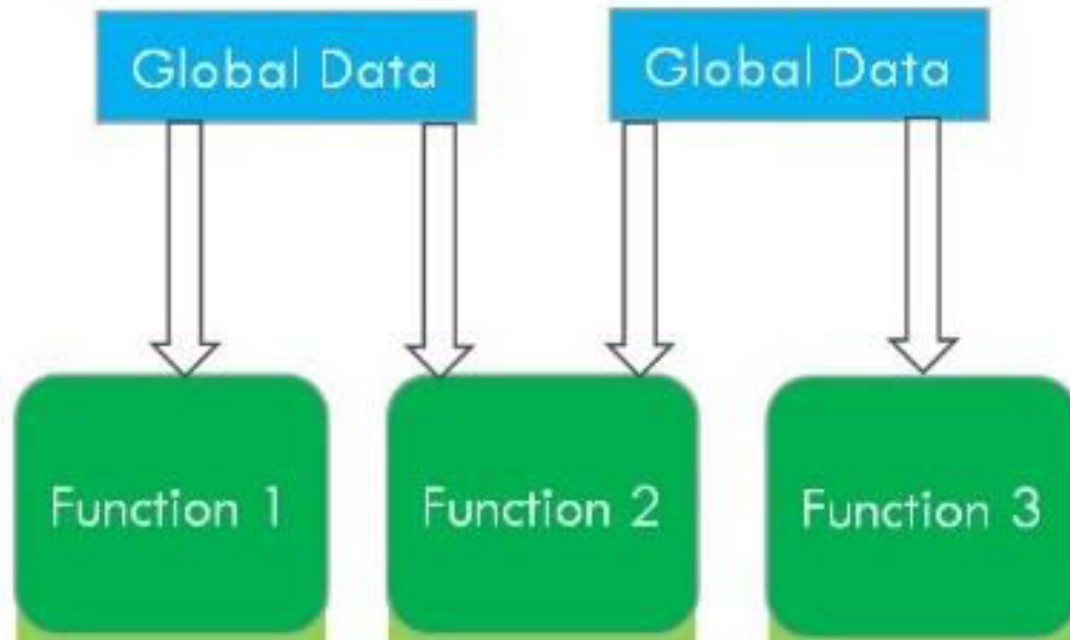
- In this programming paradigm, the whole program is written in a single block.
- It uses all data as global data which leads to data insecurity.
- There are no flow control statements like if, switch, for, and while statements in this paradigm.
- We use the **goto** statement to jump from one statement to another statement.
- There is no concept of data types.
- An **example** of a Monolithic programming paradigm is **Assembly language**.
- Ex: BASIC, ASSEMBLY

Procedural Programming Paradigm

The procedure-oriented programming paradigm is the advanced paradigm of the monolithic paradigm. It has the following characteristics.

- This paradigm introduces a modular programming concept where a larger program is divided into smaller modules.
- It provides the concept of code reusability.
- It is introduced with the concept of data types.
- The control of the program is transferred using unsafe **goto** statement.
- In this paradigm, all the data is used as global data which leads to data insecurity.
- Procedural programming languages are known as top-down languages
- **Examples** of a procedural-oriented programming paradigm is **ALGOL, FORTRON, COBOL, PL/I and Ada.**

Procedural Oriented Programming

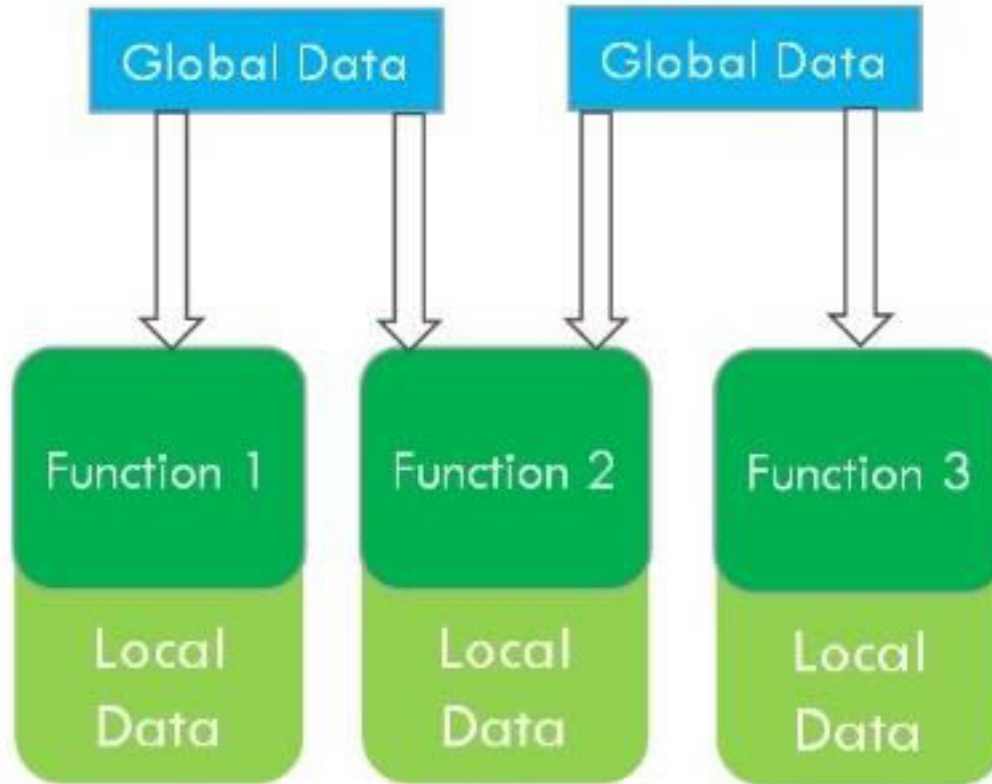


Structured Programming Paradigm

The structured-oriented programming paradigm is the advanced paradigm of a procedural-oriented paradigm. It has the following characteristics.

- This paradigm introduces a modular programming concept where a larger program is divided into smaller modules.
- It provides the concept of code reusability.
- It is introduced with the concept of data types.
- It provides flow control statements that provide more control to the user.
- It follows all the concepts of procedural-oriented programming paradigm but the data is defined as global data, and also local data to the individual modules.
- User defined data types are introduced.
- In this paradigm, functions may transform data from one form to another.
- Procedural programming languages are known as top-down languages
- **Examples** of structured-oriented programming paradigm is **C, visual basic, PASCAL**, etc.

Structured Programming Paradigm

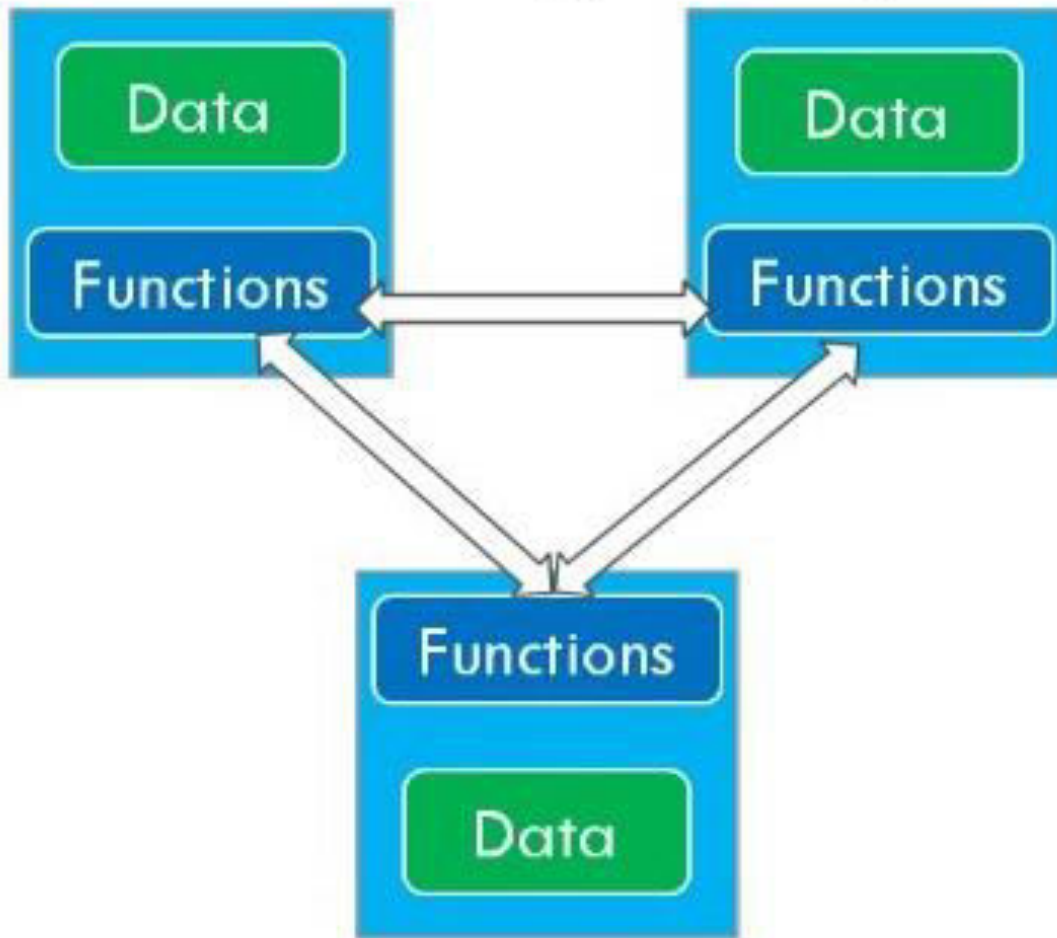


Object-oriented Programming Paradigm

The object-oriented programming paradigm is the most popular. It has the following characteristics.

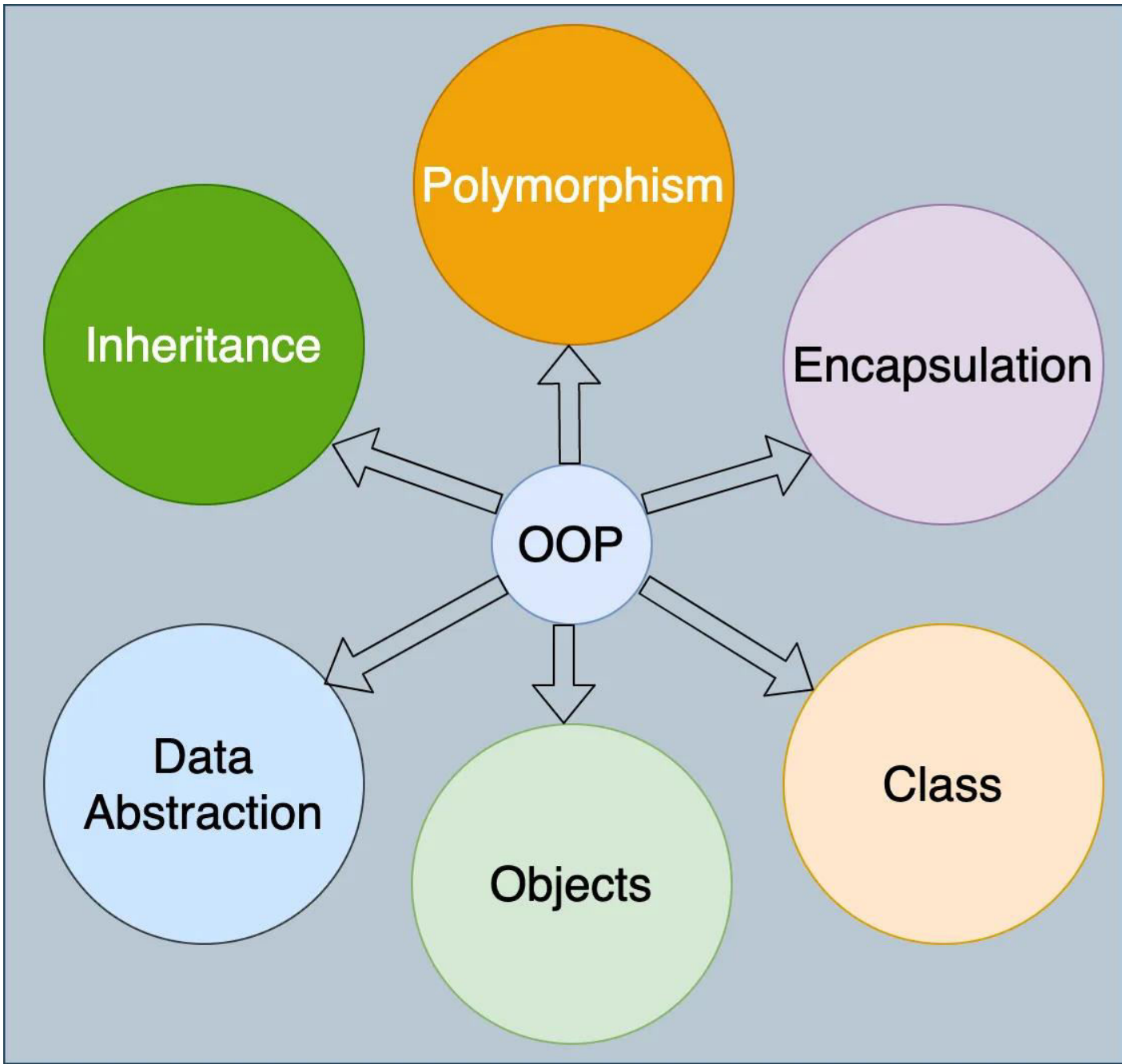
- In this paradigm, the whole program is created on the concept of objects.
- In this paradigm, objects may communicate with each other through function.
- This paradigm mainly focuses on data rather than functionality.
- In this paradigm, programs are divided into what are known as objects.
- It follows the bottom-up flow of execution.
- It introduces concepts like data abstraction, inheritance, and overloading of functions and operators overloading.
- In this paradigm, data is hidden and cannot be accessed by an external function.
- It has the concept of friend functions and virtual functions.
- In this paradigm, everything belongs to objects.
- **Examples** of object-oriented programming paradigm : **C++**, **Java**, **C#**, **Python**, etc.

Object Oriented Programming



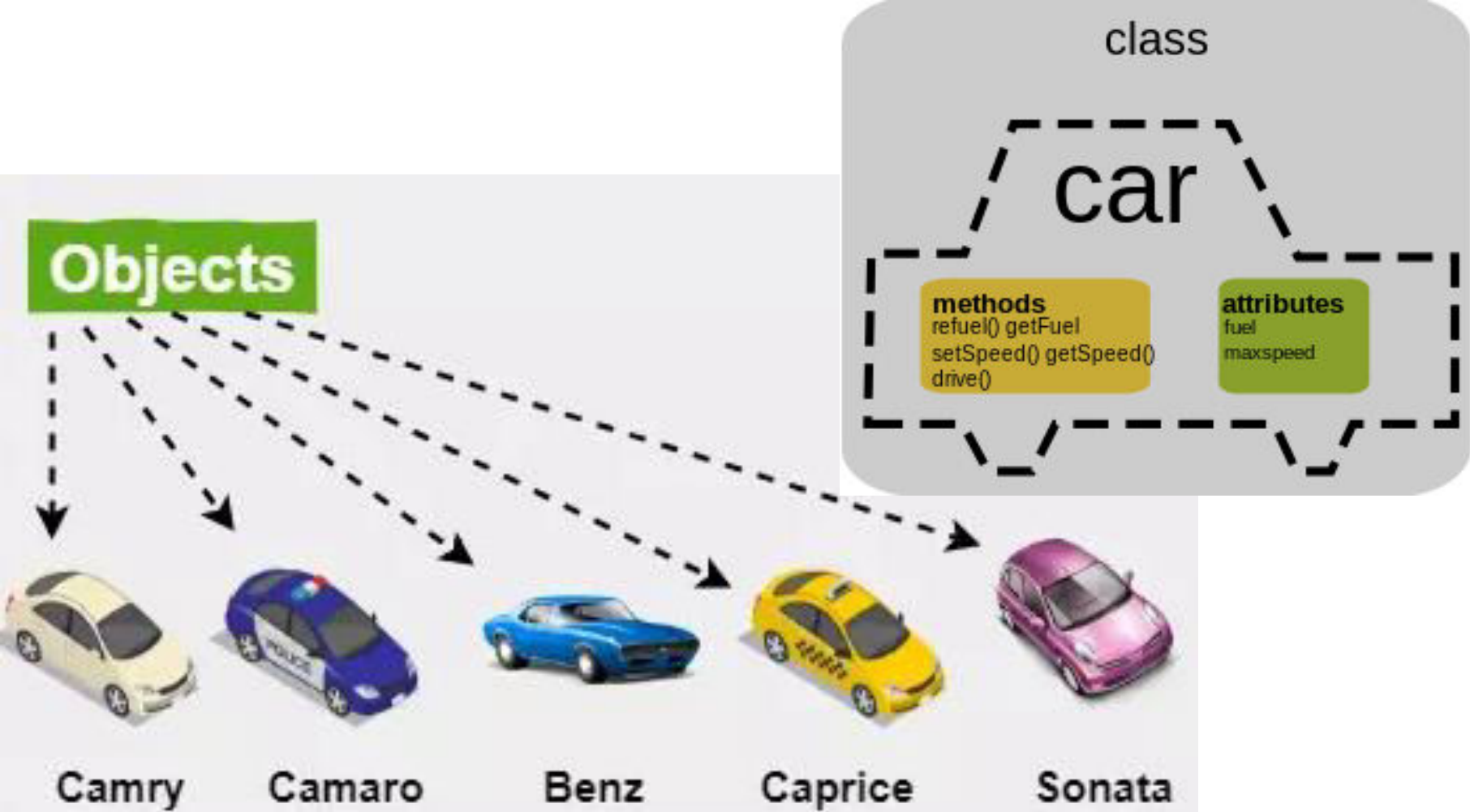
OOPs concepts

- Object-oriented programming uses objects in programming.
- The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.
- So that security will be provided to the data.



Class

- The class is a user-defined data type which defines its properties(attributes or variables) and its behaviour (functions or methods).
- A class is grouping of objects having identical properties, common behavior and shared relationship.
- It supports a template for creating objects which bind code and data.
- The class does not occupy any memory space.
- The class is the only logical representation of the data.
- For example, Human being is a class. The body parts of a human being are its properties, and the actions performed by the body parts are known as functions.



The syntax of declaring the class:

```
class student
{
//data members;
//Member functions
}
```

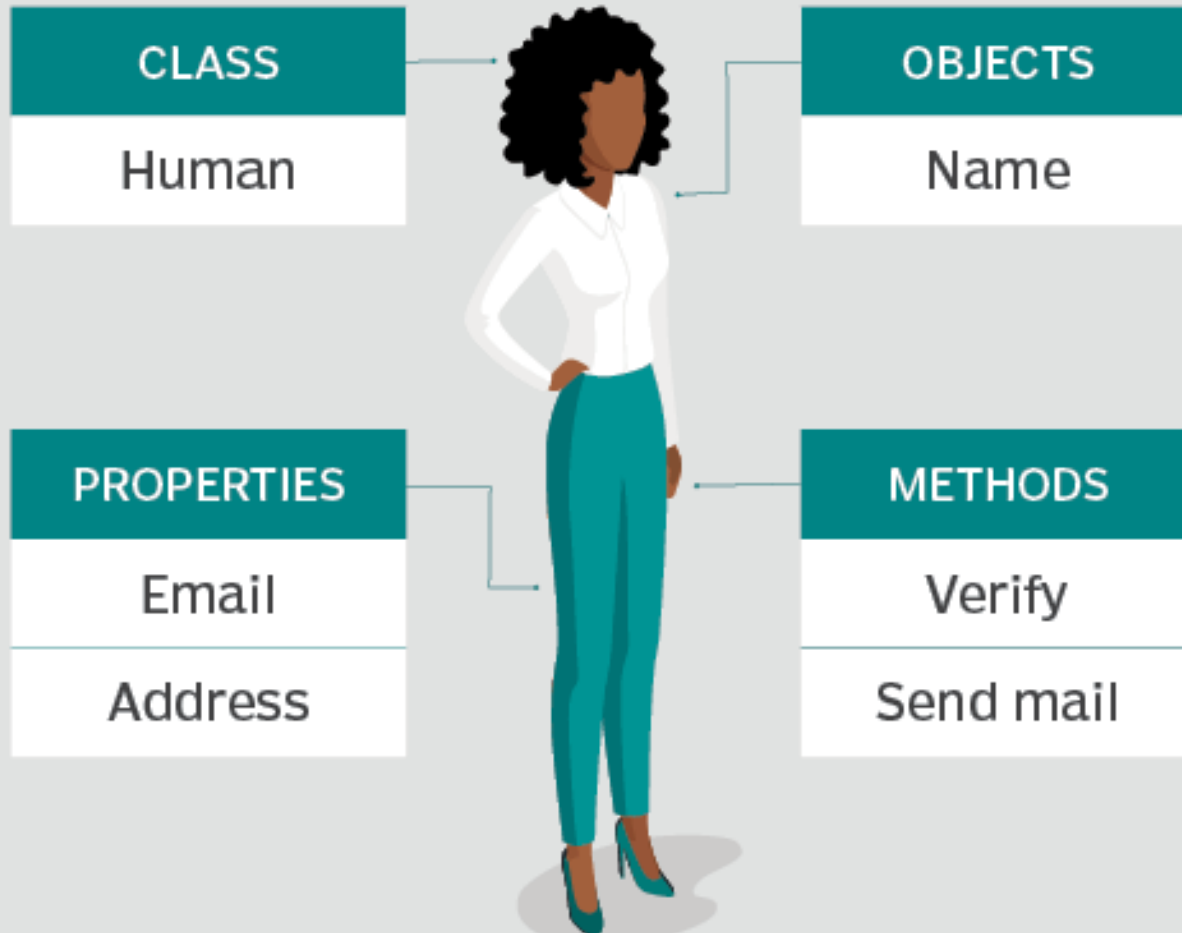
Object

- An object is the instance of the class. An object is a run-time entity.
- An object can represent a person, place or any other item.
- An object can operate on both data members and member functions.
- When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

The syntax for declaring the object:

```
Student s = new Student();
```

Object-oriented programming



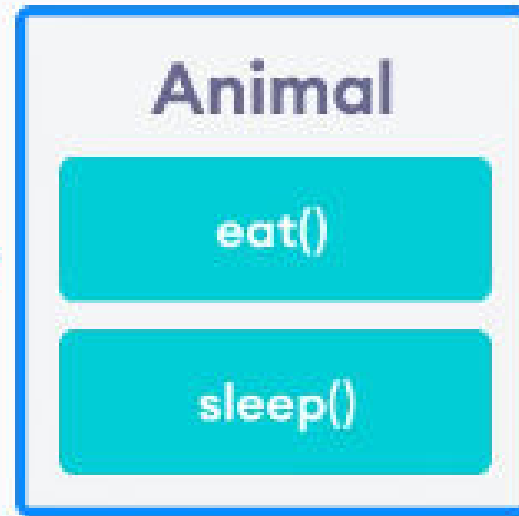
- **Encapsulation:** wrapping the data members and member functions in a single unit. It binds the data within a class, and no outside method can access the data. If the data member is private, then the member function can only access the data.
- This characteristic of [data hiding](#) provides greater program security and avoids unintended [data corruption](#).
- **Abstraction:** Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

- Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name “sales section”.
- Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

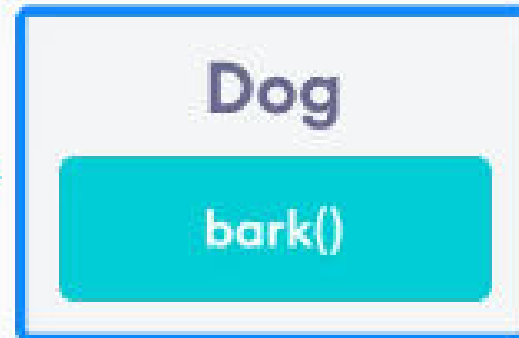
- Polymorphism: Polymorphism means multiple forms. It means having more than one function with the same function name but with different functionalities.
- Ex: A person at the same time can have different characteristics. Like a man at the same time is a father, an employee. So the same person possesses different behaviour in different situations. This is called polymorphism.
- Operator overloading(only in C++)
- Function overloading

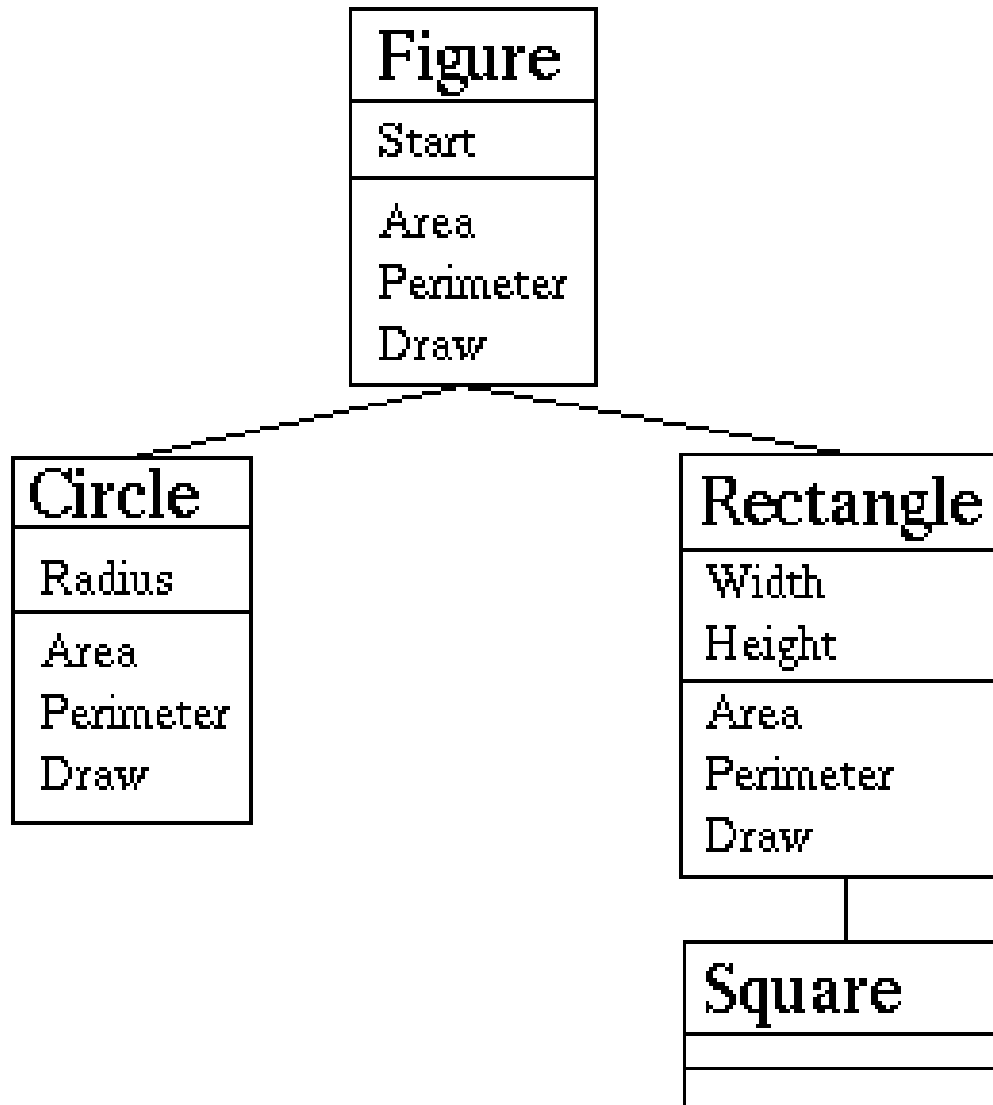
- **Inheritance:** The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.
 - **Sub Class:** The class that inherits properties from another class is called Sub class or Derived Class.
 - **Super Class:** The class whose properties are inherited by sub class is called Base Class or Super class.
 - **Reusability:** Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

**Base
Class**



**Derived
Class**





Java Programming

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) Initially designed for small, [embedded systems](#) in electronic appliances like set-top boxes.
- 3) Firstly, it was called "**Greentalk**" by James Gosling, and the file extension was .gt.
- 4) After that, it was called **Oak** and was developed as a part of the Green project.
- 5) **Why Oak?** Oak is a symbol of strength and chosen as a national tree of many countries like the U.S.A., France, Germany, Romania, etc.
- 6) In 1995, Oak was renamed as "**Java**" because it was already a trademark by Oak Technologies.

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

The principles for creating Java programming were "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted, and Dynamic".

initiated this project to develop a language for digital devices such as set-top boxes, televisions, etc.

However, it was suited for internet programming.

Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc.

Version	Release date
JDK Beta	1995
JDK 1.0	January 1996
JDK 1.1	February 1997
J2SE 1.2	December 1998
J2SE 1.3	May 2000
J2SE 1.4	February 2002
J2SE 5.0	September 2004
Java SE 6	December 2006
Java SE 7	July 2011
Java SE 8 (LTS)	March 2014
Java SE 9	September 2017
Java SE 10	March 2018
Java SE 11 (LTS)	September 2018
Java SE 12	March 2019
Java SE 13	September 2019
Java SE 14	March 2020
Java SE 15	September 2020
Java SE 16	March 2021

Features of Java

Features of



Java

→ **Simple**

→ **Object-oriented**

→ **Distributed**

→ **Robust**

→ **Secure**

→ **System independence**

→ **Portability**

→ **Interpreted**

→ **High Performance**

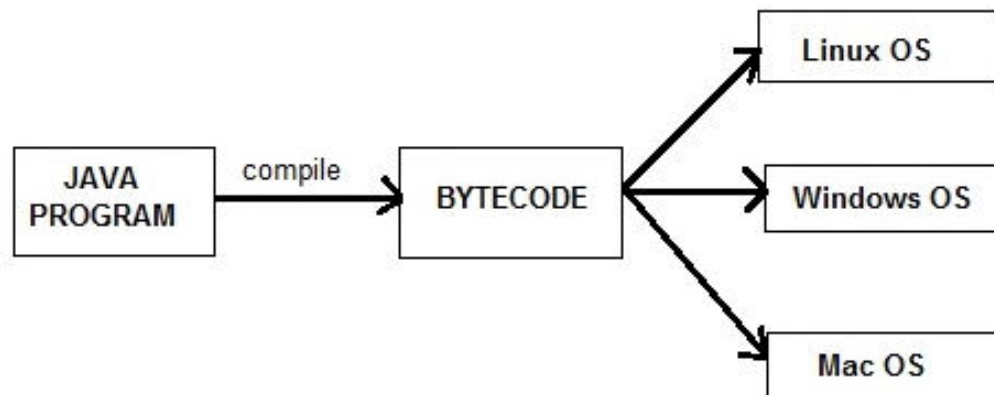
→ **Multithreaded**

→ **Dynamic**

- The features of Java are also known as *java buzzwords*.
- Simple:
 - Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun, Java language is a simple programming language because:
 - Java syntax is based on C,C++ (so easier for programmers to learn it after C++).
 - Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
 - There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.
- Object-oriented
 - Java is an [object-oriented](#) programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior.

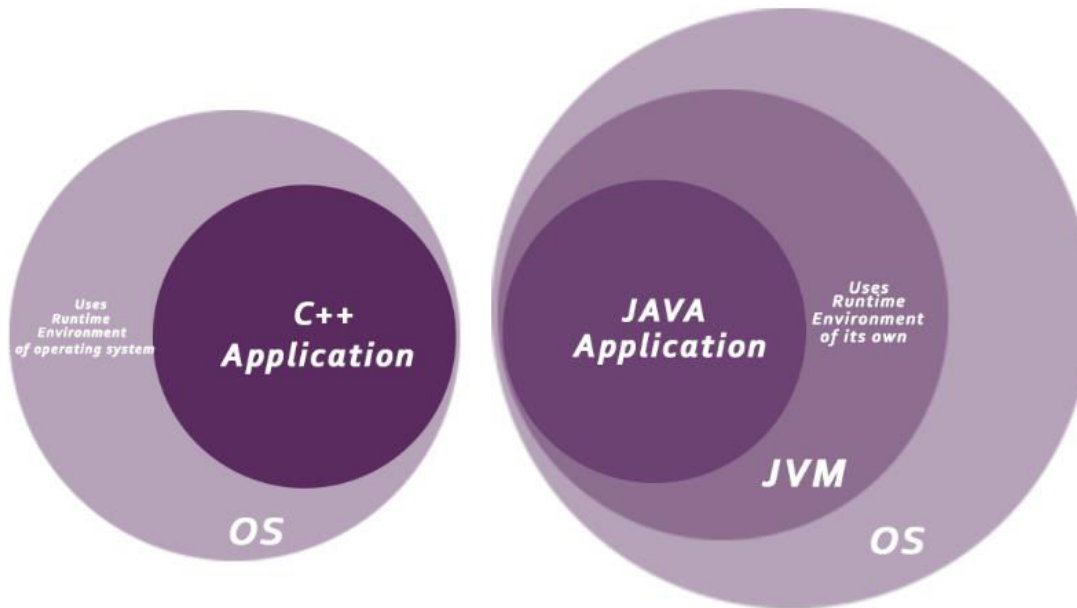
• Platform Independent:

- A platform is the hardware or software environment in which a program runs.
- Java is a write once, run anywhere language.
- Java code can be run on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc.
- Java code is compiled and converted into bytecode.
- This code is not understood by any platform, but only a virtual platform called the **Java Virtual Machine**.
- This Virtual Machine resides in the RAM of your operating system. When the Virtual Machine is fed with this bytecode, it identifies the platform it is working on and converts the bytecode into the native machine code.



- **Secure:**

- Java program always runs in Java runtime environment with almost null interaction with system OS, hence it is more secure.
- Java has no explicit pointer & Java Programs run inside a virtual machine
- With Java, we can develop virus-free systems.



- **Robust:**

- Robust means strong.
- It uses strong memory management.
- There is a lack of pointers that avoids security problems.
- There is automatic garbage collection in java which runs on the Java Virtual Machine.
- There are exception handling and the type checking mechanism in Java. So it provides compile time error checking and runtime checking.

- **Architecture-neutral:**

- Compiler generates bytecodes, which have nothing to do with a particular computer architecture, hence a Java program is easy to interpret on any machine.

- **Distributed:**

- Java is distributed because it facilitates users to create distributed applications in Java.
- RMI and EJB are used for creating distributed applications.
- This feature of Java makes us able to access files by calling the methods from any machine on the internet by using TCP/IP protocols.

- **Multi-threaded:**
 - We can write Java programs that deal with many tasks at once by defining multiple threads.
 - Benefit of multithreading is that it utilizes same memory and other resources to execute multiple threads at the same time.
 - Ex: While typing, grammatical errors are checked along.
- **Dynamic:**
 - Java is a dynamic language.
 - It supports dynamic loading of classes i.e., classes are loaded on demand.
 - Java supports automatic memory management (garbage collection).
- **High Performance:**
 - Java architecture is designed to reduce overheads during run-time.
 - The concept of multithreading in Java also increases the execution speed of Java programs.
 - Java is an interpreted language, so it will never be as fast as a compiled language like C or C++.
 - Java enables high performance with the use of just-in-time compiler.
- **Interpreted:**
 - Once the java program is created, it is compiled by Java Compiler. This compiled code (Byte code) can be executed using Java Interpreter.

Java Comments

- Comments are mainly used to help programmers to understand the code.
- proper use of comments makes maintenance easier and finding bugs easily.
- Comments are ignored by the compiler while compiling a code. These are optional.
- In Java there are three types of comments:
 1. **Single – line comments.**
 2. **Multi – line comments.**
 3. **Documentation comments.**
- **Single-line comments** start with two forward slashes (//). we use // for short comments.

Ex: single-line comment before a line of code:

```
// This is a comment  
System.out.println("Hello World");
```

Ex: single-line comment at the end of a line of code:

```
System.out.println("Hello World"); // This is a  
comment
```

- **Multi-line comments** start with `/*` and ends with `*/`. We use `/* */` for longer comments.
- Any text between `/*` and `*/` will be ignored by Java.

Ex:

```
/* The code below will print the words Hello World to the  
screen, and it is amazing */
```

```
System.out.println("Hello World");
```

- **Documentation Comments:**

This type of comments are used generally when writing code for a project/software package, since it helps to generate a documentation page for reference, which can be used for getting information about methods present, its parameters, etc.

The documentation comment is used to create documentation API. To create documentation API, you need to use [javadoc tool](#).

Ex:

```
/** The Calculator class provides methods to get addition and subtraction of given 2 numbers.*/
```

```
public class Calculator {
```

```
/** The add() method returns addition of given numbers.*/
```

```
public static int add(int a, int b){return a+b;}
```

```
/** The sub() method returns subtraction of given numbers.*/
```

```
public static int sub(int a, int b){return a-b;}
```

```
}
```

Create Documentation API by javadoc tool: `javadoc Calculator.java`

Java tokens

- A **token** is the smallest element of a program that is meaningful to the compiler.
- The Java compiler breaks the line of code into text (words) is called **Java tokens**.

```
public class Demo
{
    public static void main(String args[])
    {
        System.out.println("javatpoint");
    }
}
```

In the above code snippet, **public**, **class**, **Demo**, **{**, **static**, **void**, **main**, **(**, **String**, **args**, **[**, **]**, **)**, **System**, **.**, **out**, **println**, **javatpoint**, etc. are the Java tokens. The Java compiler translates these tokens into [Java bytecode](#). Further, these bytecodes are executed inside the interpreted Java environment.

- Java tokens can be classified as follows:
 - Keywords
 - Identifiers
 - Constants
 - Special Symbols/separators
 - Operators
 - Comments
- Keywords:
 - Keywords are pre-defined or reserved words in a programming language.
 - Each keyword is meant to perform a specific function in a program.
 - keywords are referred names for a compiler, they can't be used as variable names because by doing so, we are trying to assign a new meaning to the keyword which is not allowed.
 - Keywords are always written in lower case.

Java language supports following keywords:

abstract	assert	boolean
break	byte	case
catch	char	class
const	continue	default
do	double	else
enum	exports	extends
final	finally	float
for	goto	if
implements	import	instanceof
int	interface	long
module	native	new
open	opens	package
private	protected	provides
public	requires	return
short	static	strictfp
super	switch	synchronized
this	throw	throws
to	transient	transitive
try	uses	void
volatile	while	with

- **Identifier:**

- Identifiers are used to name a variable, constant, function, class, and array.
- Identifiers usually defined by the user.
- The identifier name must be different from the reserved keywords.
- There are some rules to declare identifiers are:
 - Identifiers must begin with a letter, dollar sign or underscore.
 - Apart from the first character, an identifier can have any combination of characters.
 - Identifiers in Java are case sensitive.
 - Java Identifiers can be of any length.
 - Identifier name cannot contain white spaces.
 - Any identifier name must not begin with a digit but can contain digits within.
 - Most importantly, **keywords** can't be used as identifiers in Java.

- **Examples of valid and invalid identifiers :**

- \$myvariable //correct

- _variable //correct

- Variable //correct

- &variable //error

- 23identifier //error

- switch //error

- Literals:
 - literal is a notation that represents a fixed value (constant) in the source code.
 - It can be categorized as an integer literal, string literal, Boolean literal, etc.
 - Java provides five types of literals are as follows:

Literal	Type
23	int
9.86	double
false, true	boolean
'K', '7', '-'	char
"javatpoint"	String
null	any reference type

- All values that we write in a program are literals.
- Each literal belongs to one of java's 4 primitive data types: int, double, boolean, char.
 - integer literals: represents countable and discrete quantities.
 - decimal/octal/hexa-decimal
 - double Literals: represents measurable quantities like real numbers, fractions and numbers with decimal places.
 - E or e
 - boolean Literals: represent for calculating truth values.
 - char literal: It is a type of text literal. It represents one character inside single quotes.
 - String Literal: It is a type of text literal. Enclosed in double quotes.

Escape Sequences

- Each escape sequence is translated into a character that prints in some special way.

Escape Sequences

Escape Sequence	Description
<code>\t</code>	Insert a tab in the text at this point.
<code>\b</code>	Insert a backspace in the text at this point.
<code>\n</code>	Insert a newline in the text at this point.
<code>\r</code>	Insert a carriage return in the text at this point.
<code>\f</code>	Insert a formfeed in the text at this point.
<code>\'</code>	Insert a single quote character in the text at this point.
<code>\"</code>	Insert a double quote character in the text at this point.
<code>\\</code>	Insert a backslash character in the text at this point.

```
public class Test {
    public static void main(String[] args)
    {
        System.out.println("Hi geek, welcome to \"GeeksforGeeks\".");
    }
}
```

Output: Hi geek, welcome to "GeeksforGeeks".

```
public class Test {
    public static void main(String[] args)
    {
        System.out.println("Good Morning\t Geeks! ");
    }
}
```

Good Morning Geeks!

```
public class Test {
    public static void main(String[] args)
    {
        System.out.println("Good Morning\bG Geeks! ");
    }
}
```

Good Morning Geeks!

```
public class Gfg {
    public static void main(String[] args)
    {
        System.out.println("\\- this is a backslash. ");
    }
}
```

\\- this is a backslash.

Separators

- The separators in Java is also known as **punctuators**.
- have special meaning known to Java compiler and cannot be used for any other purpose.
- separator <= ; | , | . | (|) | { | } | [|]
- **Square Brackets []**: It is used to define array elements. A pair of square brackets represents the single-dimensional array, two pairs of square brackets represent the two-dimensional array.
- **Parentheses ()**: It is used to call the functions and parsing the parameters.
- **Curly Braces {}**: The curly braces denote the starting and ending of a code block.
- **Comma (,)**: It is used to separate two values, statements, and parameters.
- **Assignment Operator (=)**: It is used to assign a variable and constant.
- **Semicolon (;)**: It is the symbol that can be found at end of the statements. It separates the two statements.
- **Period (.)**: It separates the package name form the sub-packages and class. It also separates a variable or method from a reference variable.

Operators

- Operators are the special symbols that tells the compiler to perform a special operation.
- Java provides different types of operators that can be classified according to the functionality they provide.
- There are eight types of operators in Java, are as follows:
 - Arithmetic Operators
 - Assignment Operators
 - Relational Operators
 - Unary Operators
 - Logical Operators
 - Ternary Operators
 - Bitwise Operators
 - Shift Operators

Operator	Symbols
Arithmetic	<code>+, -, /, *, %</code>
Unary	<code>++, --, !</code>
Assignment	<code>=, +=, -=, *=, /=, %=, ^=</code>
Relational	<code>==, !=, <, >, <=, >=</code>
Logical	<code>&&, </code>
Ternary	<code>(Condition) ? (Statement1) : (Statement2);</code>
Bitwise	<code>&, , ^, ~</code>
Shift	<code><<, >></code>

Data types

- Data types are for identifying and assessing the type of data. Java is rich in data types which allows the programmer to select the appropriate type needed to build variables of an application.

Data Types available in Java are:

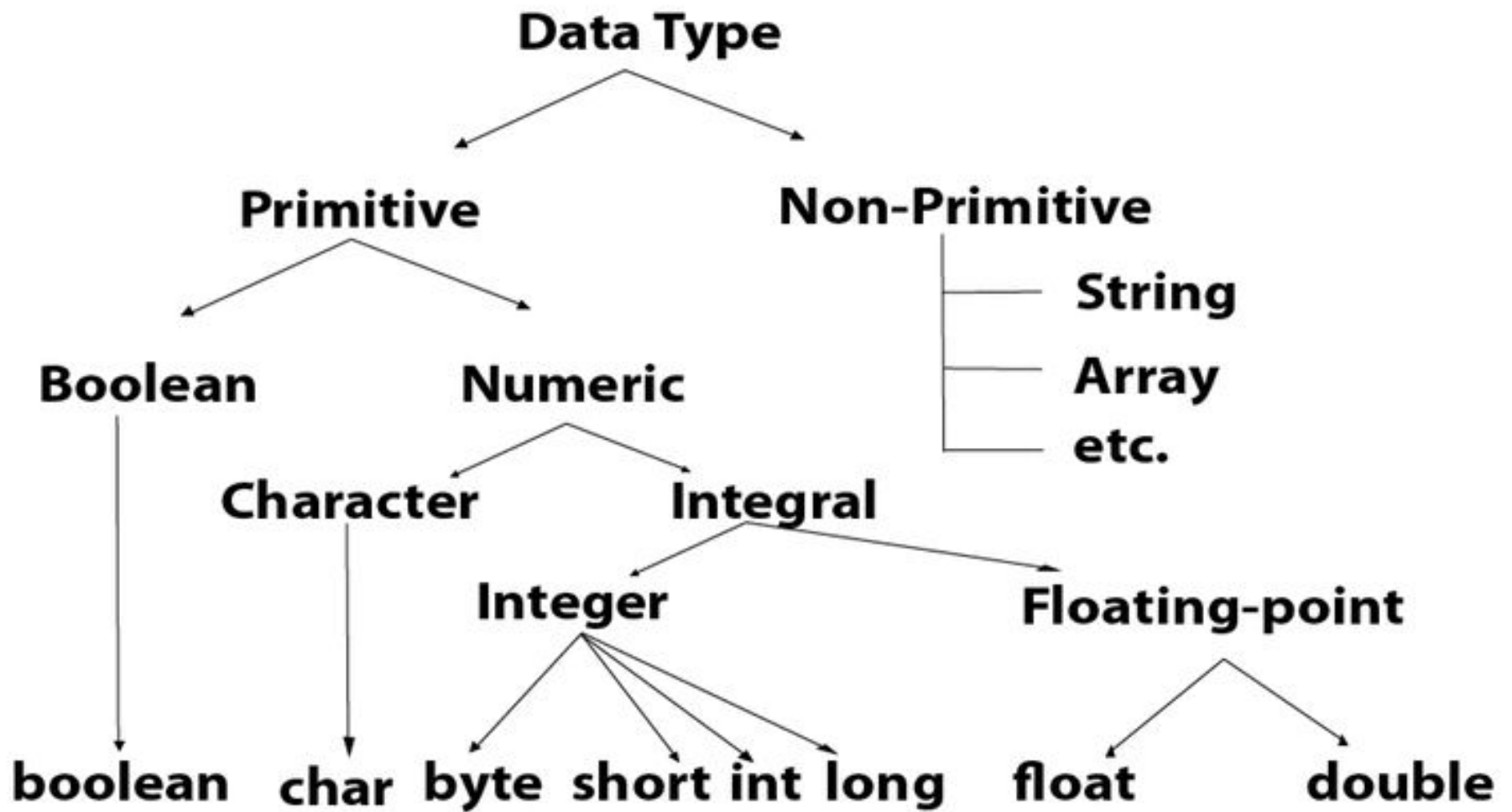
1. Primary Data Type

Java supports eight primitive data types: `byte`, `short`, `int`, `long`, `float`, `double`, `char` and `boolean`. These eight data types are further classified into four groups:

1. Integer,
2. Relational Numbers(Floating point)
3. Characters
4. Boolean(Conditional).

2. Non-Primitive Data Types

`Classes`, `Interface`, `Arrays`, etc.



Integer Types

Integer is the whole number without any fractional point. It can hold whole numbers such as 196, -52, 4036, etc. Java supports four different types of integers, these are:

Type	Contains	Default	Size	Range
byte	Signed integer	0	8 bit or 1 byte	-2 ⁷ to 2 ⁷ -1 or -128 to 127
short	Signed integer	0	16 bit or 2 bytes	-2 ¹⁵ to 2 ¹⁵ -1 or -32,768 to 32767
int	Signed integer	0	32 bit or 4 bytes	-2 ³¹ to 2 ³¹ -1 or -2147,483,648 to 2147,483,647
long	Signed integer	0	64 bit or 8 bytes	-2 ⁶³ to 2 ⁶³ -1 or -9223,372,036,854,755,808 to 9223,372,036,854,755,807

Conditional

Boolean type is used to test a particular condition during program execution. Boolean variables can take either `true` or `false` and is denoted by the keyword `boolean` and usually consumes one byte of storage.

Type	Contains	Default	Size	Range
boolean	true or false	false	1 bit	true or false

Rational Numbers

It is used to hold whole numbers containing fractional part such as 36.74, or -23.95 (which are known as floating point constants). There are two types of floating point storage in java. These are:

Type	Contains	Default	Size	Range
float	IEEE 754 floating point single-precision	0.0f	32 bit or 4 bytes	$\pm 1.4E-45$ to $\pm 3.40282347E+38F$
double	IEEE 754 floating point double-precision	0.0	64 bit or 8 bytes	$\pm 439E-324$ to $\pm 1.7976931348623157E+308$

Characters

It is used to store character constants in memory. Java provides a character data type called char whose type consumes a size of two bytes but can hold only a single character.

Type	Contains	Default	Size	Range
char	Unicode character unsigned	\u0000	16 bits or 2 bytes	0 to $2^{16}-1$ or \u0000 to \uFFFF

```
class DataTypes{
    public static void main(String args[]){
        byte byteVar = 5;
        short shortVar = 20;
        int intVar = 30;
        long longVar = 60;
        float floatVar = 20;
        double doubleVar = 20.123;
        boolean booleanVar = true;
        char charVar = 'W';

        System.out.println("Value Of byte Variable is " + byteVar);
        System.out.println("Value Of short Variable is " + shortVar);
        System.out.println("Value Of int Variable is " + intVar);
        System.out.println("Value Of long Variable is " + longVar);
        System.out.println("Value Of float Variable is " + floatVar);
        System.out.println("Value Of double Variable is " + doubleVar);
        System.out.println("Value Of boolean Variable is " + booleanVar);
        System.out.println("Value Of char Variable is " + charVar);
    }
}
```

Program Output:

```
Value Of byte Variable is 5
Value Of short Variable is 20
Value Of int Variable is 30
Value Of long Variable is 60
Value Of float Variable is 20.0
Value Of double Variable is 20.123
Value Of boolean Variable is true
Value Of char Variable is W
```


Constants

- A constant is a variable whose value **cannot change once it has been assigned**.
- 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;
```

Java will throw errors at compile time itself if we change the value of constant variable.

Expressions

- A Java expression consists of variables, operators, literals, and method calls.

- Examples:

```
int score;
```

```
score = 90;// Expression
```

```
Double a = 2.2, b = 3.4, result;
```

```
result = a + b - 3.4;//Expression
```

```
if (number1 == number2) //Expression
```

```
System.out.println("Number 1 is larger than number 2");
```

- Expression evaluation in Java is based upon the following concepts:
 - Operator precedence
 - Associativity rules
 - Type promotion rules

Operator Precedence

- Operator precedence determines the order in which the operators in an expression are evaluated.
- All the operators in Java are divided into several groups and are assigned a precedence level.
- Ex: $10 - 2 * 5$
- Based on the operator precedence chart, $*$ has higher precedence than $+$. So, $2 * 5$ is evaluated first which gives 10 and then $10 - 10$ is evaluated which gives 0.

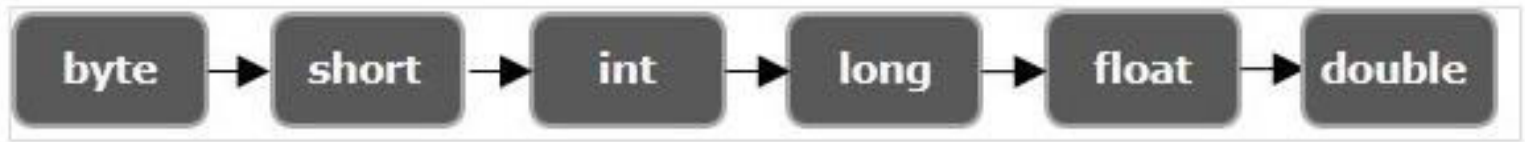
Precedence	Operator	Type	Associativity
15	() [] .	Parentheses Array subscript Member selection	Left to Right
14	++ --	Unary post-increment Unary post-decrement	Right to left
13	++ -- + - ! ~ (type)	Unary pre-increment Unary pre-decrement Unary plus Unary minus Unary logical negation Unary bitwise complement Unary type cast	Right to left
12	* / %	Multiplication Division Modulus	Left to right
11	+ -	Addition Subtraction	Left to right
10	<<< >>> >>>	Bitwise left shift Bitwise right shift with sign extension Bitwise right shift with zero extension	Left to right
9	< <= > >= instanceof	Relational less than Relational less than or equal Relational greater than Relational greater than or equal Type comparison (objects only)	Left to right
8	== !=	Relational is equal to Relational is not equal to	Left to right
7	&	Bitwise AND	Left to right
6	^	Bitwise exclusive OR	Left to right
5		Bitwise inclusive OR	Left to right
4	&&	Logical AND	Left to right
3		Logical OR	Left to right
2	? :	Ternary conditional	Right to left
1	= += -= *= /= % =	Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment	Right to left

Associativity Rules

- If the expression contains two or more operators from the same group then such ambiguities are solved using the associativity rules.
- When an expression contains operators from the same group, associativity rules are applied to determine which operation should be performed first.
- Ex: $10-6+2$
- The operators $+$ and $-$ both belong to the same group. So, we have to check the associativity rules for evaluating the above expression. Associativity rule for $+$ and $-$ group is left-to-right i.e, evaluate the expression from left to right. So, $10-6=4$ and $4+2=6$.

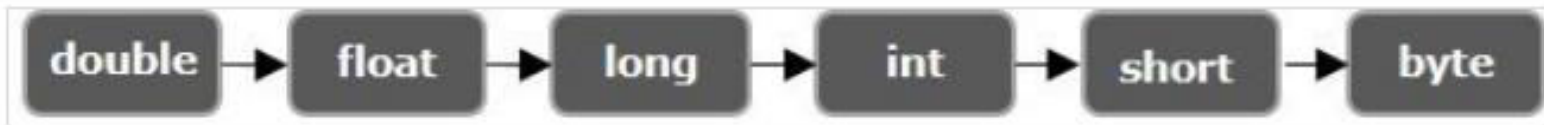
Type conversion

- Converting a value from one type to another type (data type) is known as type conversion.
- Type conversion is of two types based on how the conversion is performed:
 - *Implicit conversion* (automatic conversion or coercion or widening conversion)
 - *Explicit conversion* (type casting or narrowing conversion.)
- Implicit Conversion:
 - Implicit casting is performed to convert a lower data type into a higher data type. It is also known as **automatic type promotion in Java**.
 - In this case both datatypes should be compatible with each other.



- **Explicit Conversion:**

- Converting a higher datatype to a lower datatype is known as narrowing. In this case the casting/conversion is not done automatically, you need to convert explicitly using the cast operator “()” .
- In this case both datatypes need not be compatible with each other.



- **Syntax for type casting:**

(destination-type) value

- 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.
- Ex for implicit type conversion:

```
int myInt = 9; o/p: 9
```

```
double myDouble = myInt; // implicit conversion int to double o/p:9.0
```

```
int i = 100; 100
```

```
long l = i; 100
```

```
float f = l; 100.0
```


- Ex for explicit type conversion:
- `double myDouble = 9.78;`
- `int myInt = (int) myDouble; // Manual casting:`
double to int 9
- **float** `b = 3.0;`
- **int** `a = (int) b; // converting a float value into integer`
3
- `double d = 100.04; 100.04`
- `long l = (long)d; 100`
- `int i = (int)l; 100`

Type Promotion in Expressions

- In addition to assignment statements, type conversion can occur in expressions also.
- An expression is a collection of variables, values, operators and method calls which evaluate to a single value.
- Type promotion rules of Java for expressions are listed below:
 - All *char*, *short* and *byte* values are automatically promoted to *int* type.
 - If at least one operand in an expression is a *long* type, then the entire expression will be promoted to *long*.
 - If at least one operand in an expression is a *float* type, then the entire expression will be promoted to *float*.
 - If at least one operand in an expression is a *double* type, then the entire expression will be promoted to *double*.
 - Boolean values cannot be converted to another type.

```
class Sample
{
    public static void main(String[] args)
    {
        int i = 1000000;
        char c = 'z';
        short s = 200;
        byte b = 120;
        float f = 3.45f;
        double d = 1.6789;
        double result = (f * b) + (i / c) - (d * s);
        System.out.println("Result = "+result);
    }
}
```

Output of the above program is: Result = 8274.22

In the above program the expression is $(f * b) + (i / c) - (d * s)$. In the first sub expression $(f * b)$, as one operand is float, the result of the expression will be a float. In the second sub expression (i / c) , char type will be promoted to int and the result of the expression will be an int. In the third sub expression $(d * s)$, as one operand is double, the result of the expression is a double.

Type casting in expressions

- While evaluating expressions, the result is automatically updated to larger data type of the operand.
- But if we store that result in any smaller data type it generates compile time error, due to which we need to type cast the result.

EX:

```
float x = 3.5, y = 4.5; // the size of float variable is 4 byte.
```

```
int area; // the size of the int variable is 4 bytes.
```

```
area = (int) x * y; // after conversion the product converts into int
```

Ex:

```
byte b = 50;
```

```
b = (byte)(b * 2); //type casting int to byte
```

The differences between implicit casting and explicit casting in Java are as follows:

1. Implicit type casting is done internally by java compiler whereas, explicit type casting is done by the programmer. Java compiler does not perform it automatically.
2. In explicit casting, cast operator is needed whereas, no need for any operator in the case of implicit type casting.
3. If we perform explicit type casting in a program, we can lose information or data but in the case of implicit type casting, there is no loss of data.
4. Accuracy is not maintained in explicit type casting whereas, there is no issue of accuracy in implicit type conversion.
5. Implicit type conversion is safe but explicit type casting is not safe.

Control statements

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

```
switch(expression){  
  case value1:  
    //code to be executed;  
    break; //optional  
  case value2:  
    //code to be executed;  
    break; //optional  
  .....
```

default:

```
  code to be executed if all cases are not match  
  ed;  
}
```

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

```
int i=1;  
while(i<=10){  
  System.out.println(i);  
  i++;  
}
```

```
int i=1;  
do{  
  System.out.println(i);  
  i++;  
}while(i<=10);
```

```
int arr[]={12,23,44,56,78};  
  //Printing array using for-each loop  
  for(int i:arr){  
    System.out.println(i);  
  }
```

Java Labeled For Loop:

We can have a name of each Java for loop. To do so, we use label before the for loop. It is useful if we have nested for loop so that we can break/continue specific for loop.

Usually, break and continue keywords breaks/continues the innermost for loop only.

aa:

```
for(int i=1;i<=3;i++){
    bb:
    for(int j=1;j<=3;j++){
        if(i==2&&j==2){
            break aa;
        }
        System.out.println(i+" "+j);
    }
}
```

Infinite for loop

```
for(;;){
//code to be executed
}
```

Infinite while loop

```
while(true){
    System.out.println("infinite while loop");
}

do{
//code to be executed
}while(true);
```


Break:

```
for(int i=1;i<=10;i++){  
    if(i==5){  
        //breaking the loop  
        break;  
    }  
    System.out.println(i);  
}
```

Continue:

```
for(int i=1;i<=10;i++){  
    if(i==5){  
        //using continue statement  
        continue;//it will skip the rest statement  
    }  
    System.out.println(i);  
}  
}
```

aa:

```
for(int i=1;i<=3;i++){  
    bb:  
    for(int j=1;j<=3;j++){  
        if(i==2&&j==2){  
            //using continue with label  
            continue aa;  
        }  
        System.out.println(i+" "+j);  
    }  
}
```

First Java Program | Hello World Example

- Requirements to run the java program:
 - Install the JDK, [download the JDK](#) from oracle.com and install it.
 - Set path of the jdk/bin directory.
 - Create the java program
 - Compile and run the java program

How to set Permanent Path of JDK in Windows

1) Go to MyComputer properties



2) Click on the advanced tab

The screenshot shows a Windows 7 desktop with a blue background. A calendar in the top right corner displays 'Wednesday 25'. The desktop contains several icons including 'suhail', 'Get Started With Oracle...', 'Google Mail', 'Computer', 'NetBeans IDE 6.9', 'MyEclipse 6.0', 'Recycle Bin', 'Nokia Ovi Suite', 'New folder', 'Virtual CloneDrive', 'Acer.com', 'Virtual Workstation', 'Microsoft Office', 'VirtualBox', 'Adobe Reader 9', 'msn Messenger', 'AVC 2011', and 'Google Chrome'. A window titled 'Control Panel > System and Security > System' is open, displaying system information. The window's address bar shows 'Control Panel > System and Security > System' and a search box for 'Control Panel'. The main content area is titled 'View basic information about your computer' and includes a 'Windows edition' section showing 'Windows 7 Ultimate' and a copyright notice. Below this is a 'System' section with a 'Rating' of 2.7, 'Processor' as 'Intel(R) Core(TM)2 Duo CPU E4600 @ 2.40GHz', 'Installed memory (RAM)' as '1.00 GB', 'System type' as '32-bit Operating System', and 'Pen and Touch' as 'No Pen or Touch Input is available for this Display'. A 'See also' section lists 'Action Center', 'Windows Update', and 'Performance Information and Tools'. A 'Computer name, domain, and workgroup settings' section shows 'Computer name' as 'suhail-PC', 'Full computer name' as 'suhail-PC', 'Computer description' as 'suhail-PC', and 'Workgroup' as 'WORKGROUP'. A 'Change settings' link is visible next to the computer name.

Control Panel > System and Security > System

View basic information about your computer

Windows edition

Windows 7 Ultimate
Copyright © 2009 Microsoft Corporation. All rights reserved.

System

Rating: **2.7** Windows Experience Index

Processor: Intel(R) Core(TM)2 Duo CPU E4600 @ 2.40GHz 2.40 GHz

Installed memory (RAM): 1.00 GB

System type: 32-bit Operating System

Pen and Touch: No Pen or Touch Input is available for this Display

See also

- Action Center
- Windows Update
- Performance Information and Tools

Computer name, domain, and workgroup settings

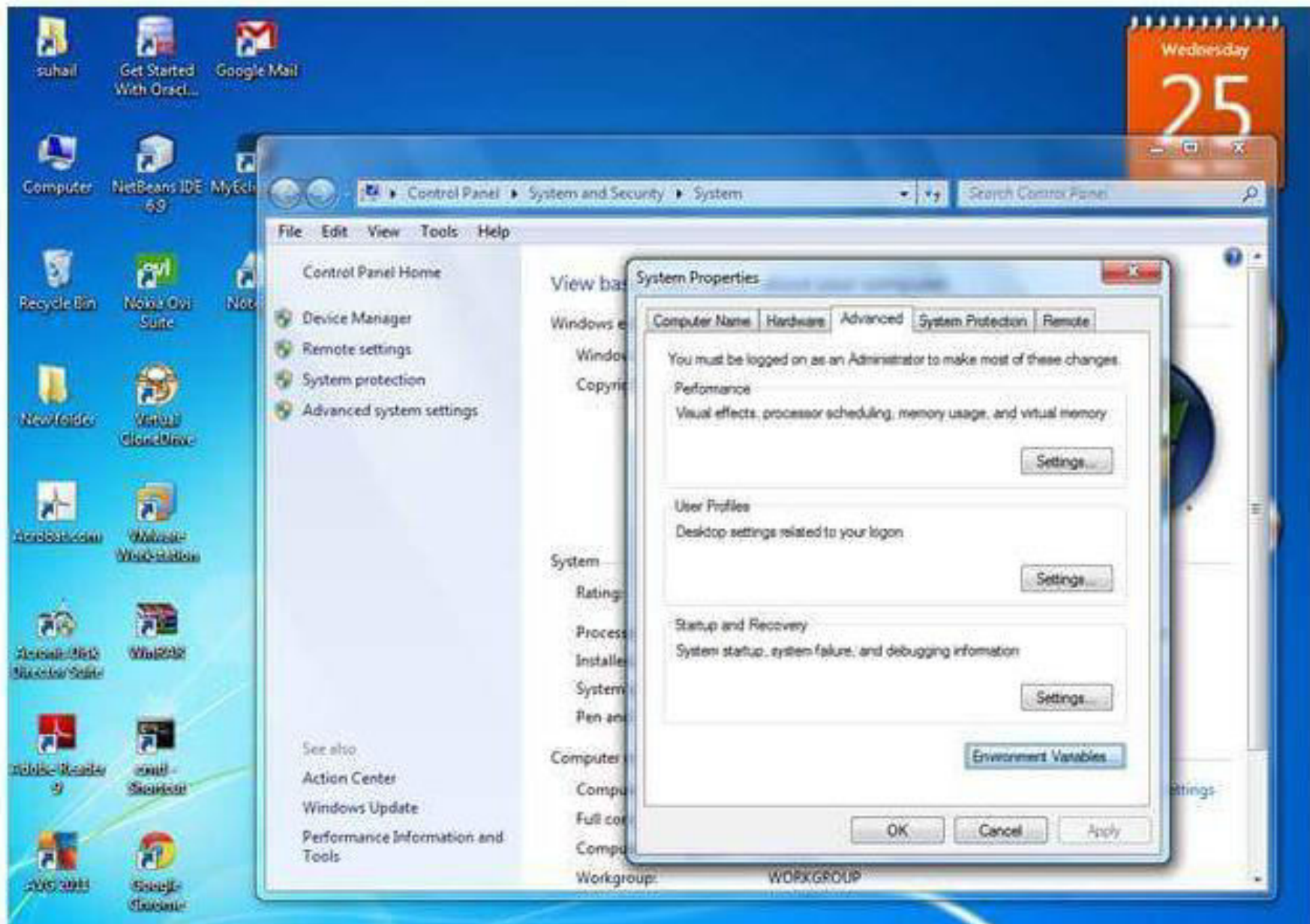
Computer name: suhail-PC [Change settings](#)

Full computer name: suhail-PC

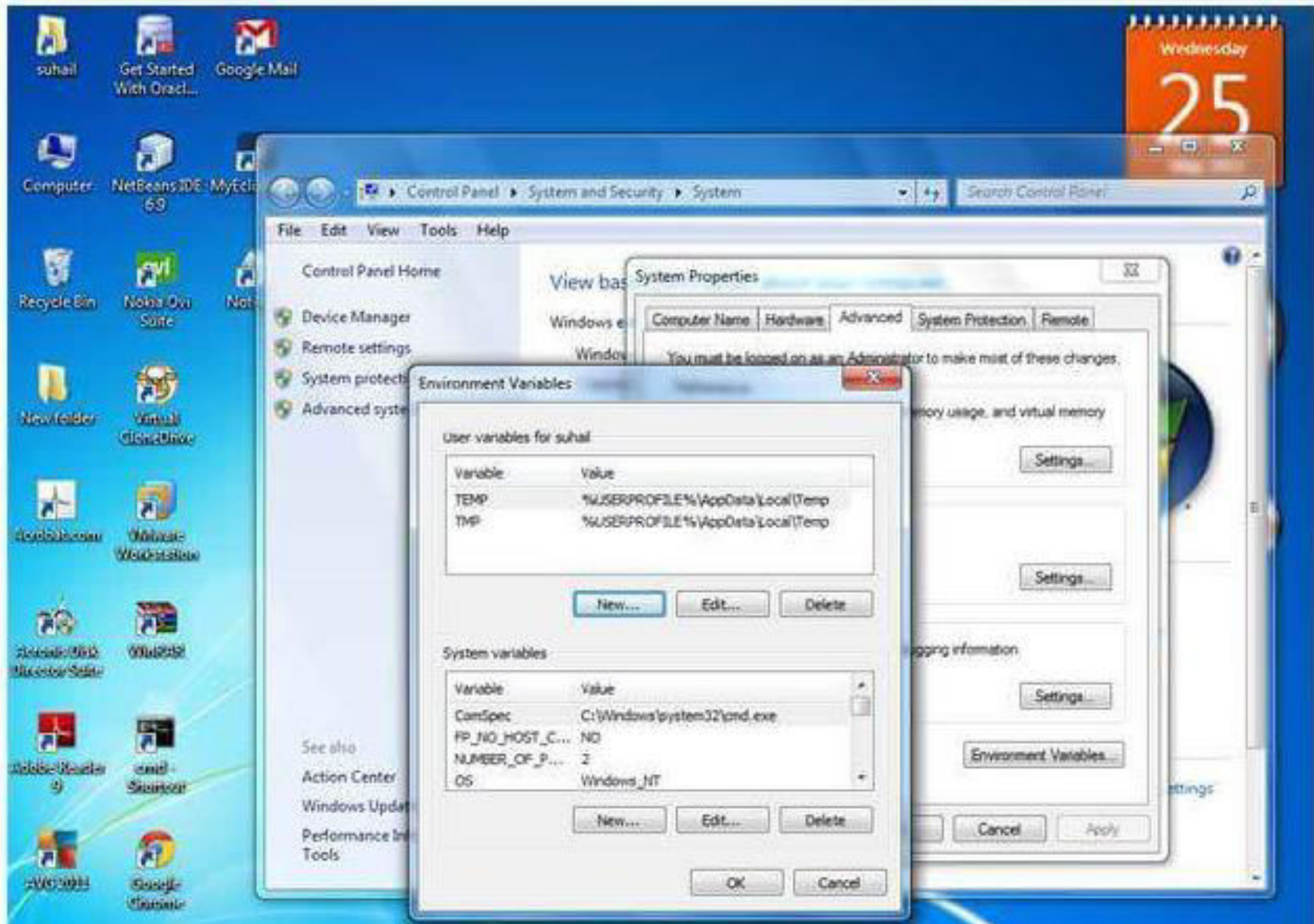
Computer description: suhail-PC

Workgroup: WORKGROUP

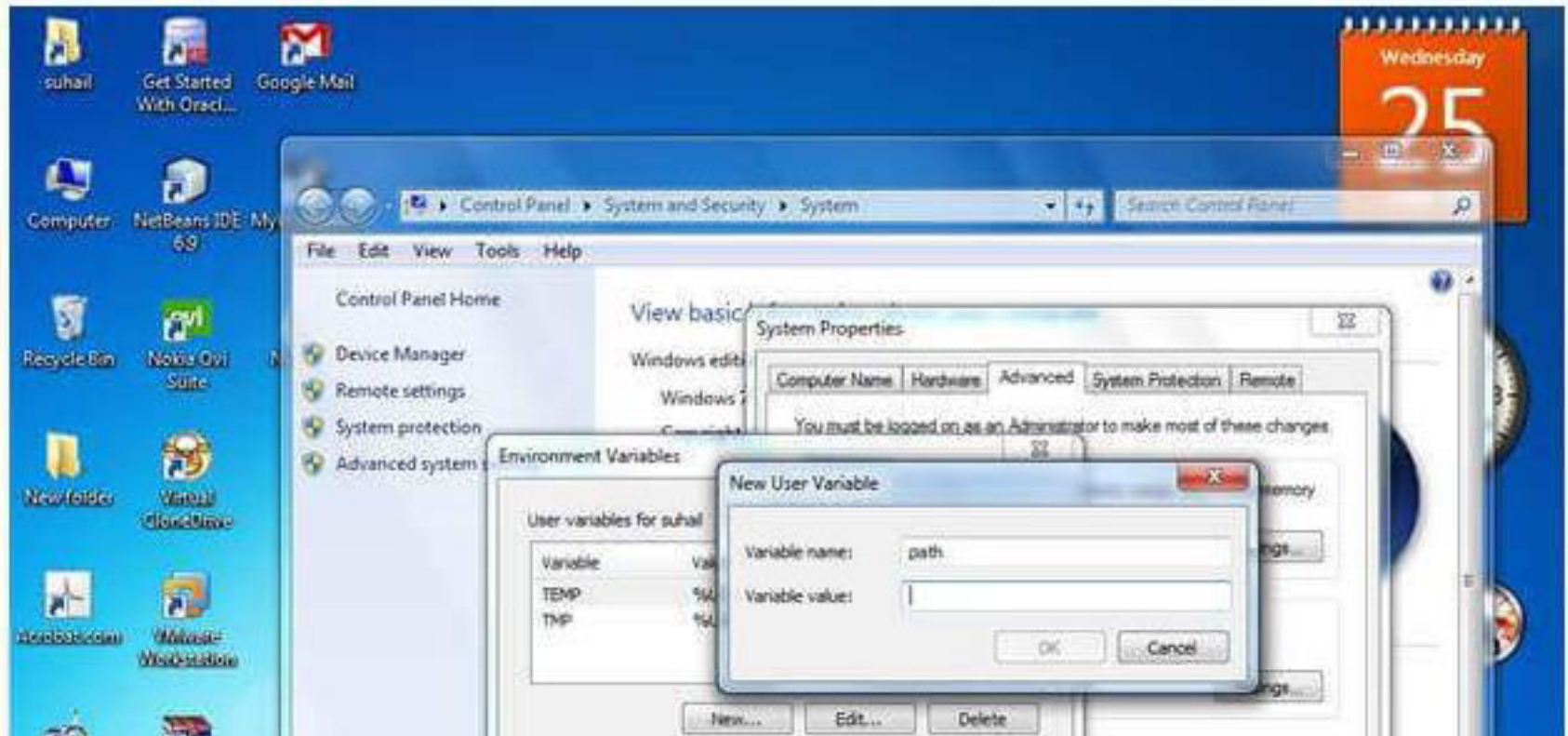
3) Click on environment variables



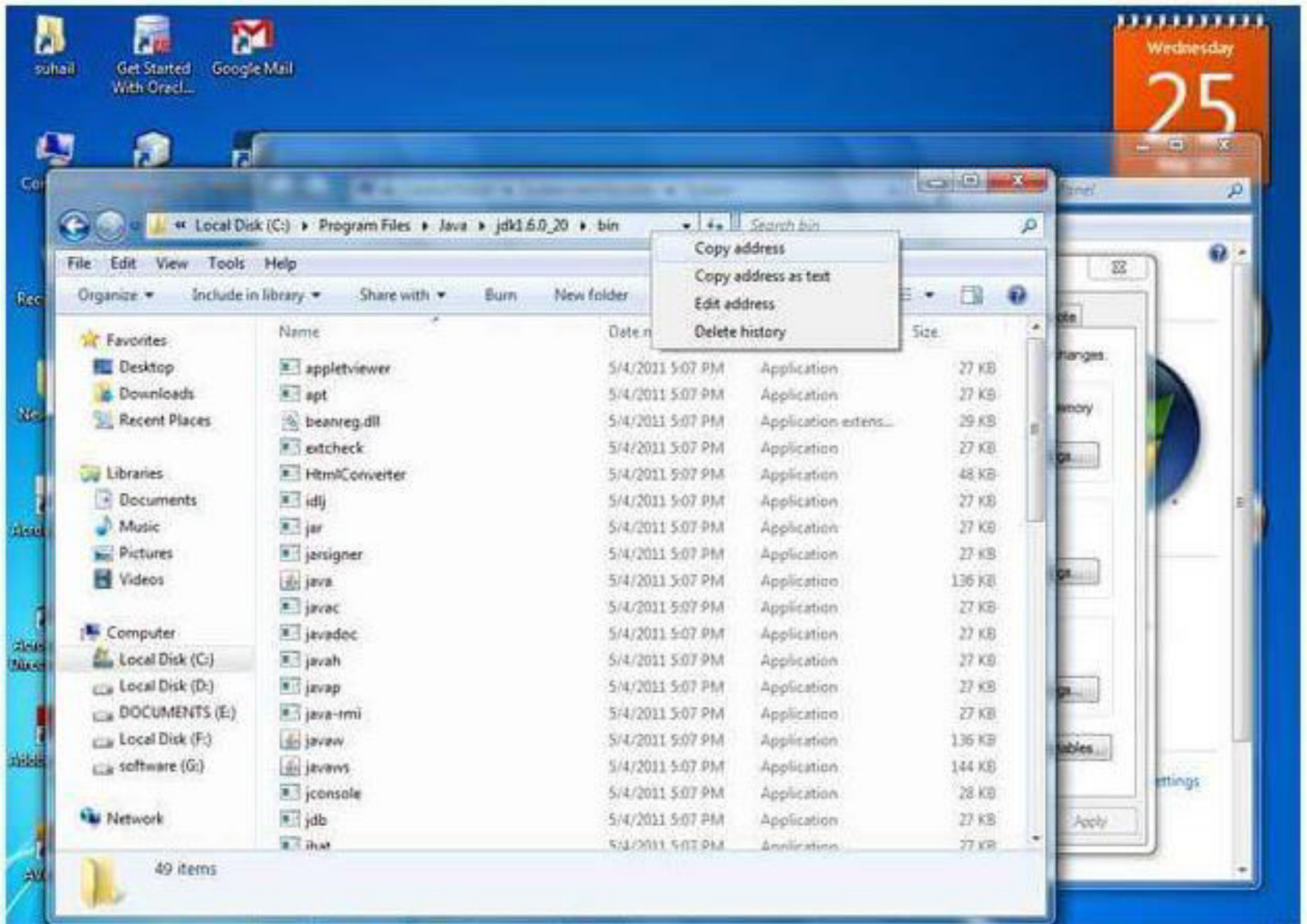
4) Click on the new tab of user variables



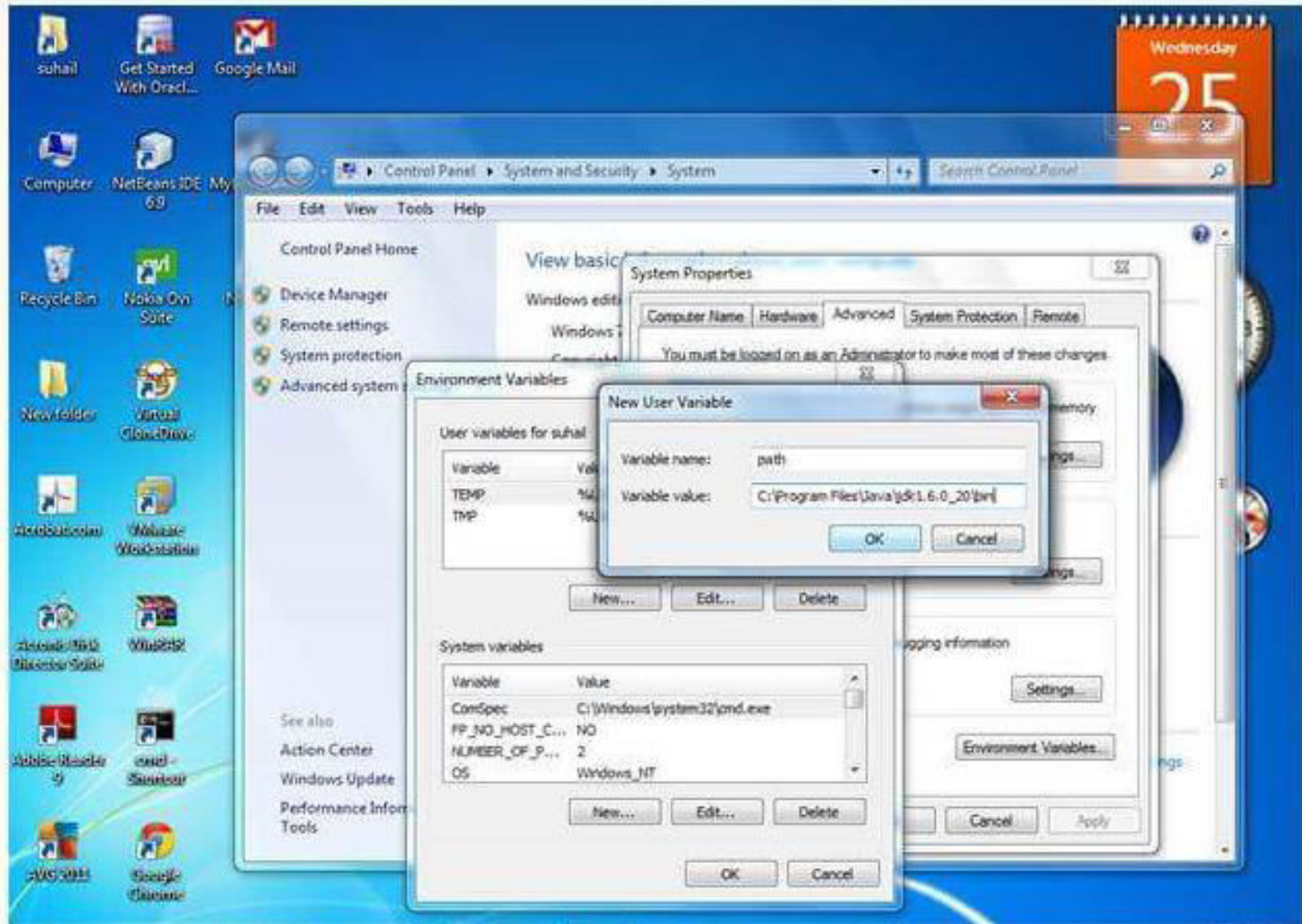
5) Write the path in the variable name



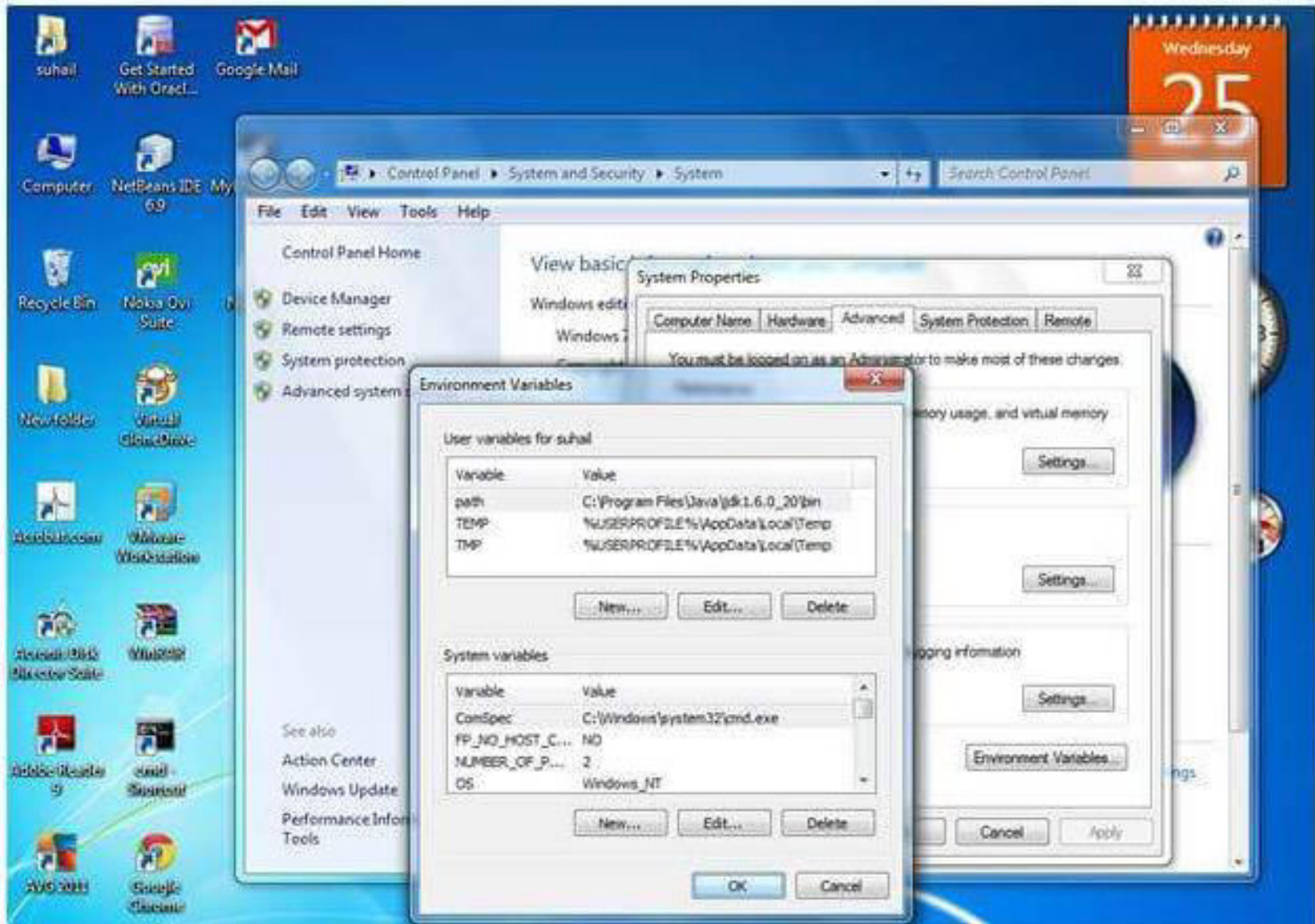
6) Copy the path of bin folder



7) Paste path of bin folder in the variable value



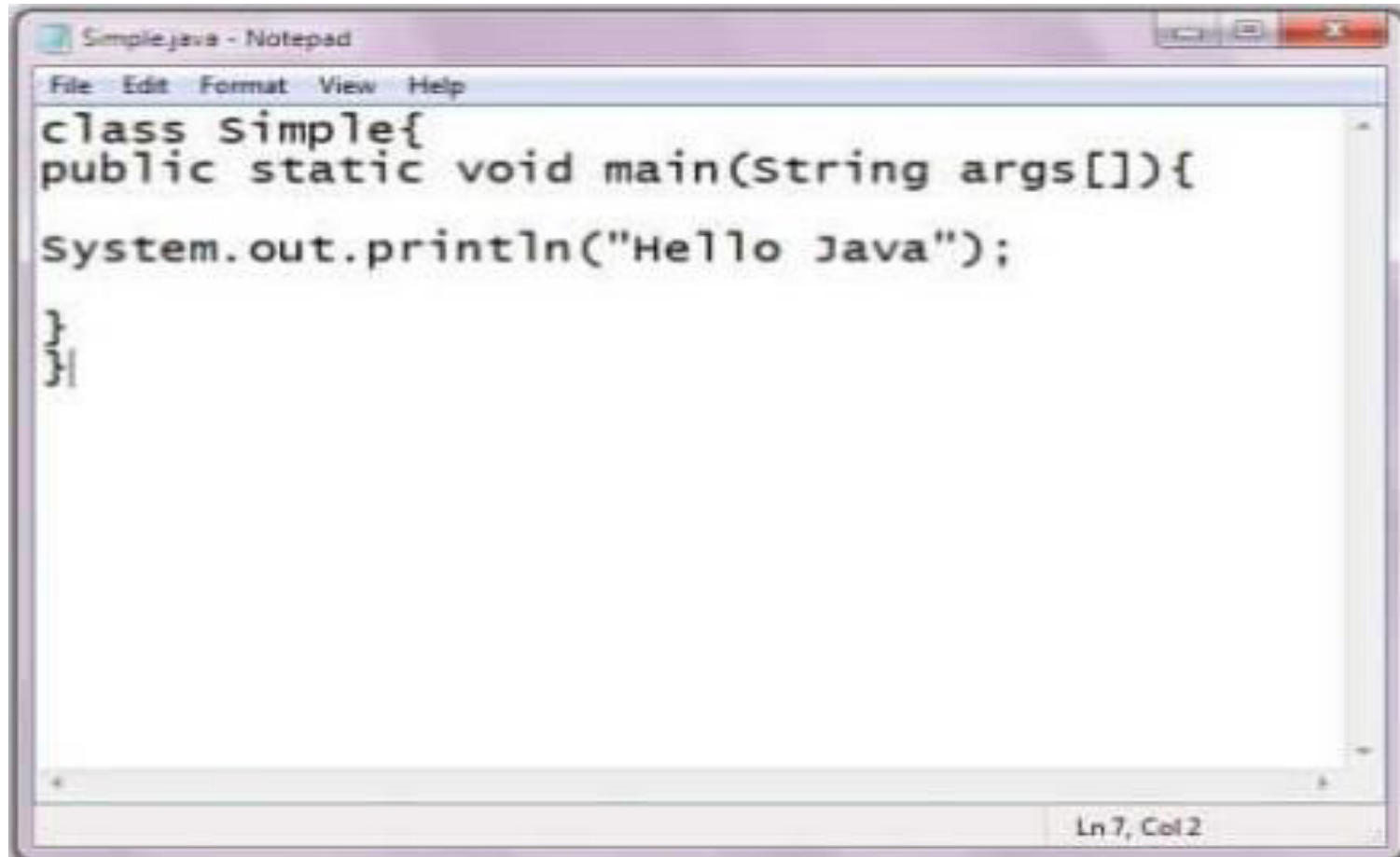
8) Click on ok button



Now your permanent path is set. You can now execute any program of java from any drive.

Create the java program

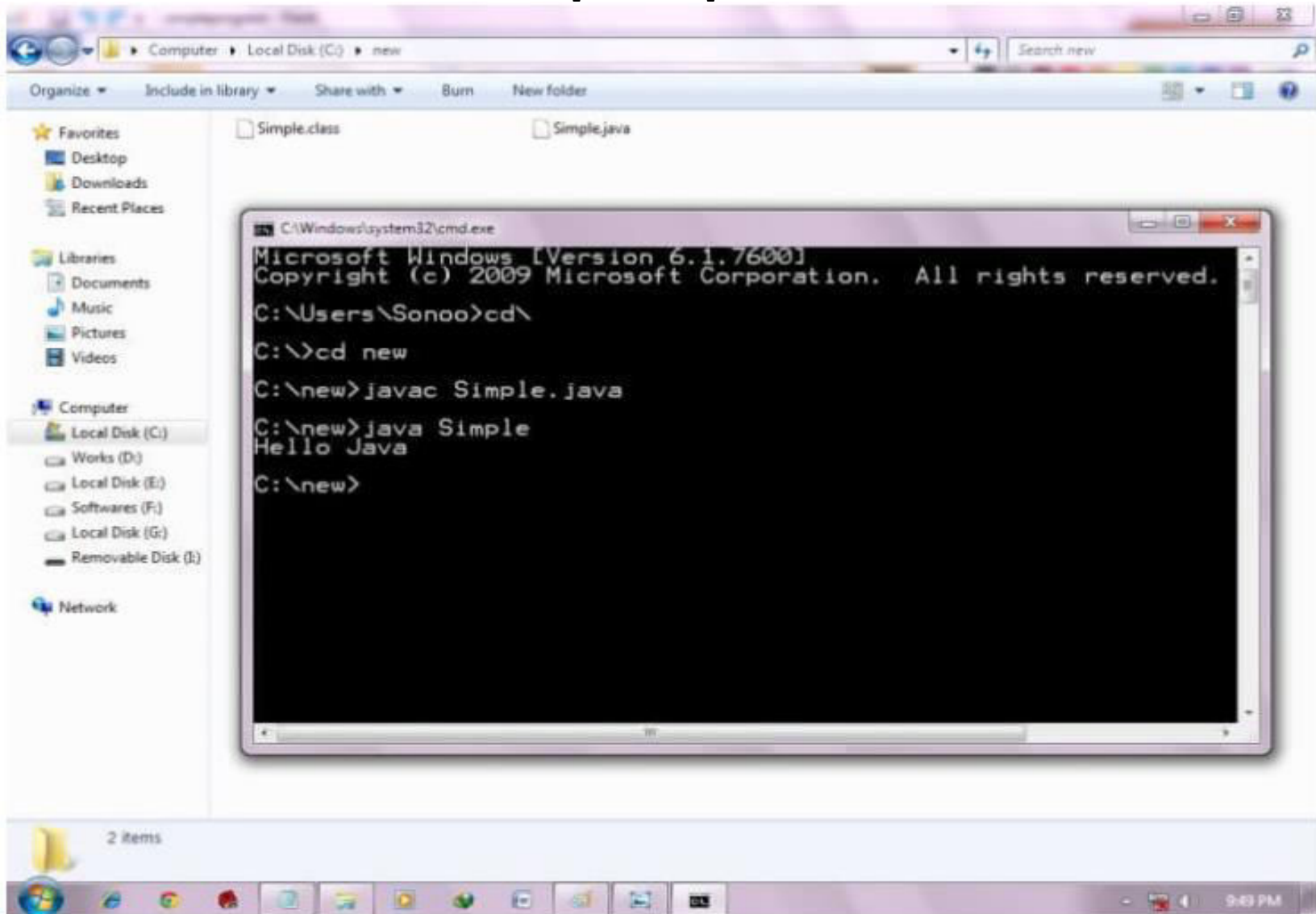
To write the simple program, you need to open notepad by **start menu -> All Programs -> Accessories -> notepad.** write the simple program of java in notepad and saved it as Simple.java.

A screenshot of a Notepad window titled "Simple.java - Notepad". The window contains the following Java code:

```
File Edit Format View Help
class Simple{
public static void main(String args[]){
System.out.println("Hello Java");
}
}
```

The status bar at the bottom right of the window shows "Ln 7, Col 2".

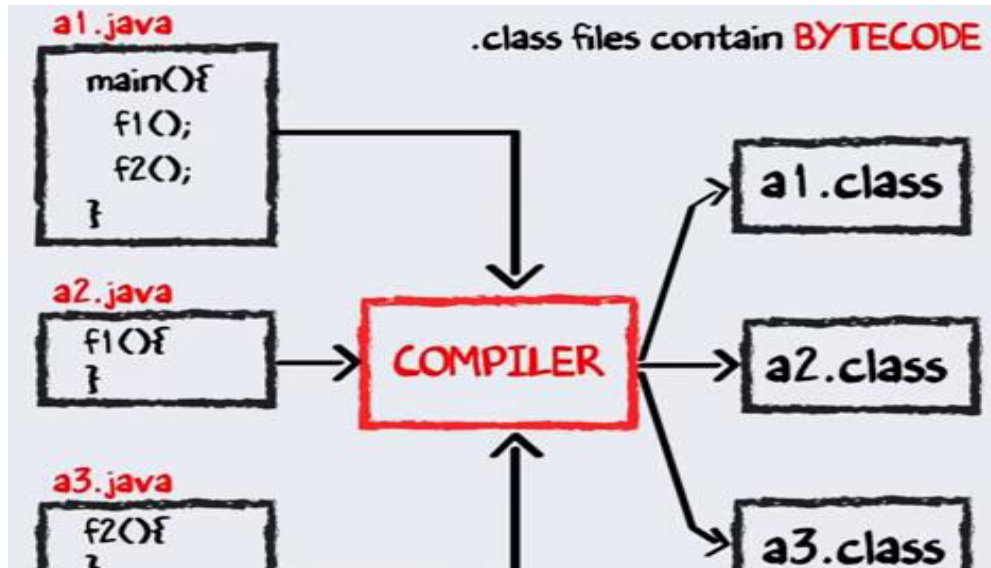
To compile and run this program, you need to open the command prompt by **start menu -> All Programs -> Accessories -> command prompt.**



To compile and run the above program, go to your current directory first. my current directory is c:\new. Write here:

To compile: `javac Simple.java`

When we compile Java program using javac tool, java compiler converts the source code into byte code.



To execute: `java Simple`

Output: Hello Java

Parameters used in First Java Program

```
class Simple
```

```
{
```

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

```
    {
```

```
        System.out.println("Hello Java");
```

```
    }
```

```
}
```

- **class** keyword is used to declare a class in java.
- **public** keyword is an access modifier which represents visibility. It means it is visible to all.

- **static** is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method. The main method is executed by the JVM, so it doesn't require to create an object to invoke the main method. So it saves memory.
- **void** is the return type of the method. It means it doesn't return any value.
- **main** represents the starting point of the program.
- **String[] args** is used for command line argument.
- **System.out.println()** is used to print statement. Here, System is a class, out is the object of PrintStream class, println() is the method of PrintStream class.

Java Virtual Machine

- **Java Virtual Machine (JVM)** is an engine that provides runtime environment to drive the Java Code or applications. It converts Java bytecode into machines language. However, Java compiler produces code for a Virtual Machine known as Java Virtual Machine.

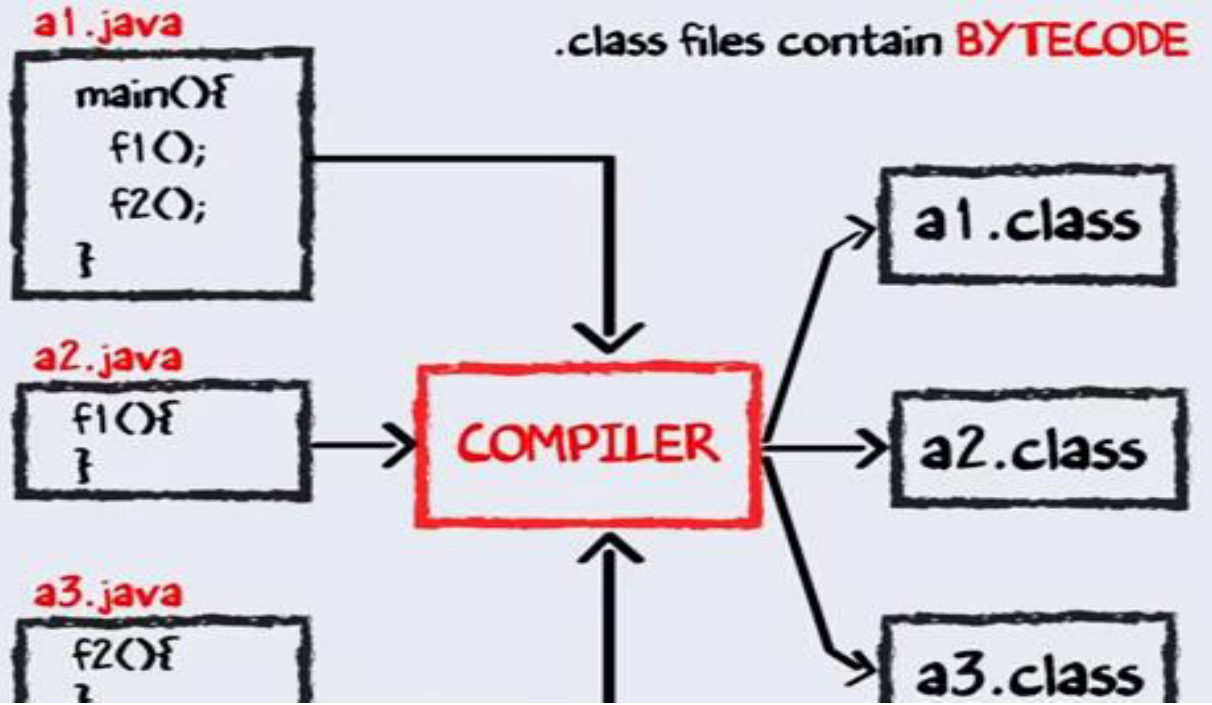
Loading and executing class files in the JVM

- JVM's main role is running Java applications. In order to run Java applications, the JVM depends on the Java class loader and a Java execution engine.
- **The Java class loader in the JVM**
 - Everything in Java is a class, and all Java applications are built from classes.
 - An application could consist of one class or thousands.
 - The [Java class loader](#) is the part of the JVM that loads classes into memory and makes them available for execution.
 - Every Java Virtual Machine includes a class loader.

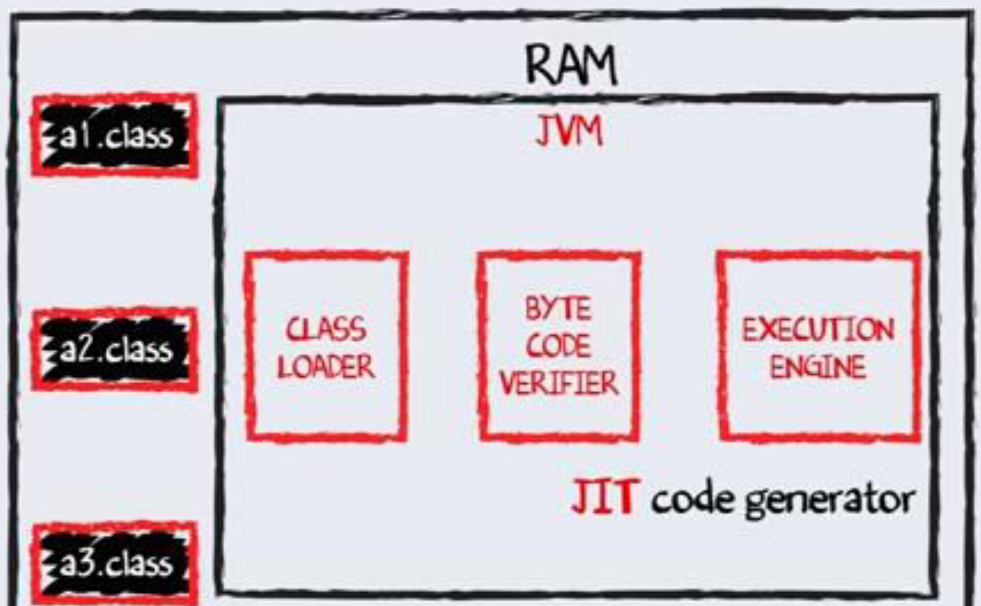
- **The execution engine in the JVM**

- The execution engine is essential to the running JVM.
- Once the class loader has done its work of loading classes, the JVM begins executing the code in each class.
- The *execution engine* is the JVM component that handles this function.
- Executing code involves managing access to system resources. The JVM execution engine stands between the running program--with its demands for file, network and memory resources--and the operating system, which supplies those resources.

.class files contain **BYTECODE**



JIT converts **BYTECODE** into machine code



Console input and output

- **We have 3 ways for reading user's input from console(text entry and display device) in Java:**
 1. **Reading User's Input using Scanner class**
 2. **Reading User's Input using Console class**
 3. **Reading User's Input using BufferedReader class**
- **We have 2 ways for printing output to console in Java:**
 1. **write to console with System.out.println**
 2. **write to console with printf()**

Reading User's Input using Scanner class

- Scanner class can be used to read input from the user in the command line. available since Java 1.5.

Syntax:

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

```
System.out.print("What's your name? ");  
String name = scanner.next();
```

```
System.out.print("How old are you? ");  
int age = scanner.nextInt();
```

```
System.out.print("What is value of PI? ");  
float pi = scanner.nextFloat();
```

```
System.out.println("Your name is: " + name);  
System.out.println("Your age is: " + age);  
System.out.println("Your PI is: " + pi);
```

```
scanner.close();
```

```
Scanner scanner = new Scanner(new File("numbers.txt"));
```

The following import is needed for this example:

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

Scanner also provides utility methods for reading data types other than String. These include:

```
scanner.nextByte();  
scanner.nextShort();  
scanner.nextInt();  
scanner.nextLong();  
scanner.nextFloat();  
scanner.nextDouble();  
scanner.nextBigInteger();  
scanner.nextBigDecimal();
```

BigInteger class is used for mathematical operation which involves very big integer calculations that are outside the limit of all available primitive data types.

For example factorial of 100 contains 158 digits

Reading User's Input using Console class

- The **Console** class was introduced in Java 1.6, and it has been becoming a preferred way for reading user's input from the command line.

```
Console console = System.console();  
String name = console.readLine("What's your name? ");  
String age = console.readLine("How old are you? ");  
String city = console.readLine("Where do you live? ");  
  
console.format("%s, a %s year-old man is living in %s", name, age, city);
```

```
What's your name? John  
How old are you? 40  
Where do you live? California
```

```
John, a 40 year-old man is living in California
```

- it can be used for reading password-like input without echoing the characters entered by the user.

```
char[] password = console.readPassword("Enter your password: ");
```

Reading User's Input using BufferedReader class

- By wrapping the **System.in** (standard input stream) in an **InputStreamReader** which is wrapped in a **BufferedReader**, we can read input from the user in the command line.

```
BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
System.out.print("Enter your name: ");

String name = reader.readLine();
System.out.println("Your name is: " + name);
```

the `readLine()` method reads a line of text from the command line.

write to console with System.out.println

```
System.out.println("Hello, world!");
```

write to console with printf()

The `printf(String format, Object... args)` method takes an output string and multiple parameters which are substituted in the given string to produce the formatted output content. This formatted output is written in the console.

```
String name = "Lokesh";  
int age = 38;  
  
console.printf("My name is %s and my age is %d", name, age);
```

```
My name is Lokesh and my age is 38
```

formatting output

- `System.out.printf()` method can be used to print formatted output in java.
 - `printf()` comes from the C programming language and stands for *print formatted*.
 - `printf()` uses `java.util.Formatter`
 - `System.out.printf(String format, String... arguments);`
 - the method expects a format and a vararg arguments.

2.1. Format Rules

Let's have a look at format string more closely. It consists of literals and format specifiers. **Format specifiers include flags, width, precision, and conversion characters** in this sequence:

```
%[flags][width][.precision]conversion-character
```

Specifiers in the brackets are optional.

Internally, *printf()* uses the `java.util.Formatter` class to parse the format string and generate the output. Additional format string options can be found in the `Formatter` Javadoc.

2.2. Conversion Characters

The *conversion-character* is required and determines how the argument is formatted.

Conversion characters are only valid for certain data types. Here are some common ones:

- *s* formats strings.
- *d* formats decimal integers.
- *f* formats floating-point numbers.
- *t* formats date/time values.

2.3. Optional Modifiers

The *[flags]* define standard ways to modify the output and are most common for formatting integers and floating-point numbers.

The *[width]* specifies the field width for outputting the argument. It represents the minimum number of characters written to the output.

The *[.precision]* specifies the number of digits of precision when outputting floating-point values. Additionally, we can use it to define the length of a substring to extract from a *String*.

Boolean Formatting

To format Boolean values, we use the *%b* format. It works the following way: If the input value is *true*, the output is *true*. Otherwise, the output is *false*.

So, if we do the following:

```
System.out.printf("%b%n", null);  
System.out.printf("%B%n", false);  
System.out.printf("%B%n", 5.3);  
System.out.printf("%b%n", "random text");
```

then we'll see:

```
false  
FALSE  
TRUE  
true
```

Format Specifiers

- With `printf()`, you can print values such as numbers, Strings, dates, etc. To let the method know what exactly you're trying to print, you need to provide a *format specifier* for each of the values.

There are many format specifiers we can use. Here are some common ones:

- **%c** - Character
- **%d** - Decimal number (base 10)
- **%e** - Exponential floating-point number
- **%f** - Floating-point number
- **%i** - Integer (base 10)
- **%o** - Octal number (base 8)
- **%s** - String
- **%u** - Unsigned decimal (integer) number
- **%x** - Hexadecimal number (base 16)
- **%t** - Date/time
- **%n** - Newline

```
System.out.printf("Hello, %s!", "Michael Scott");  
System.out.printf("Hello, %s!", "Jim");  
System.out.printf("Hello, %s!", "Dwight");
```

Hello, Michael Scott!Hello, Jim!Hello, Dwight!

```
System.out.printf("Hello, %s!\n", "Michael Scott");  
System.out.printf("Hello, %s!\n", "Jim");  
System.out.printf("Hello, %s!\n", "Dwight");
```

Hello, Michael Scott!
Hello, Jim!
Hello, Dwight!

```
System.out.printf("%10s\n", "stack");
```

stack

Here, after the `%` character, we've passed a number and a format specifier. Specifically, we want a String with `10` characters, followed by a newline. Since `stack` only contains 5 characters, 5 more are added as padding to "fill up" the String to the character target

```
System.out.printf("%-10s\n", "stack");
```

right-padding

Argument Index

If no argument index is provided, the arguments will simply follow the order of presence in the method call:

```
System.out.printf("First argument is %d, second argument is %d", 2, 1);
```

This would result in:

```
First argument is 2, argument number is 1
```

However, after the `%` escape character and before the format specifier, we can add another command. `$_n` will specify the argument index:

```
System.out.printf("First argument is %2$d, second argument is %1$d", 2, 1);
```

Here, `2$` is located between `%` and `d`. `2$` specifies that we'd like to attach the *second* argument from the list of arguments to *this* specifier. Similarly, the `1$` specifies that we'd like to attach the first argument from the list to the other specifier.

Running this code results in:

```
First argument is 1, second argument is 2
```

```
class JavaFormatter1
{
    public static void main(String args[])
    {
        int x = 100;
        System.out.printf("Printing simple integer: x = %d\n", x);

        // this will print it upto 2 decimal places
        System.out.printf("Formatted with precison: PI = %.2f\n", Math.PI);

        float n = 5.2f;

        // automatically appends zero to the rightmost part of decimal
        System.out.printf("Formatted to specific width: n = %.4f\n", n);

        n = 2324435.3f;

        // here number is formatted from right margin and occupies a
        // width of 20 characters
        System.out.printf("Formatted to right margin: n = %20.4f\n", n);
    }
}
```

```
Printing simple integer: x = 100
Formatted with precison: PI = 3.14
Formatted to specific width: n = 5.2000
Formatted to right margin: n =                2324435.2500
```


- String provides `format()` method and it can be used to print formatted output in java.
 - The **java string format()** method returns a formatted string using the given **locale**, specified **format string** and **arguments**.
 - *public static String **format**(String form, Object... args)*
 - **form**– *format of the output string*
 - **args**– *It specifies the number of arguments for the format string. It may be zero or more.*
- Syntax:

```
public static String format(String format, Object... args)
public static String format(Locale l, String format, Object... args)
```

```
class Gfg1 {
    public static void main(String args[])
    {
        String str = "GeeksforGeeks.";

        // Concatenation of two strings
        String gfg1 = String.format("My Company name is %s", str);

        // Output is given upto 8 decimal places
        String str2 = String.format("My answer is %.8f", 47.65734);

        // between "My answer is" and "47.65734000" there are 15 spaces
        String str3 = String.format("My answer is %15.8f", 47.65734);

        System.out.println(gfg1);
        System.out.println(str2);
        System.out.println(str3);
    }
}
```

Output:

```
My Company name is GeeksforGeeks.
My answer is 47.65734000
My answer is      47.65734000
```

```
public class FormatExample{
public static void main(String args[]){
String name="sonoo";
String sf1=String.format("name is %s",name);
String sf2=String.format("value is %f",32.33434);
String sf3=String.format("value is %32.12f",32.33434);//returns 12 char fractional part filling with 0

System.out.println(sf1);
System.out.println(sf2);
System.out.println(sf3);
}}
```

```
name is sonoo
value is 32.334340
value is          32.334340000000
```

Concatenation of Strings using format() method

```
class Gfg2 {  
    public static void main(String args[])  
    {  
        String str1 = "GFG";  
        String str2 = "GeeksforGeeks";  
  
        //%1$ represents first argument, %2$ second argument  
        String gfg2 = String.format("My Company name" +  
            " is: %1$s, %1$s and %2$s", str1, str2);  
  
        System.out.println(gfg2);  
    }  
}
```

Output:

```
My Company name is: GFG, GFG and GeeksforGeeks
```

Left padding using format() method

```
class Gfg3 {  
    public static void main(String args[])  
    {  
        int num = 7044;  
  
        // Output is 3 zero's("000") + "7044",  
        // in total 7 digits  
        String gfg3 = String.format("%07d", num);  
  
        System.out.println(gfg3);  
    }  
}
```

Output:

0007044

```
public class FormatExample3 {  
    public static void main(String[] args) {  
        String str1 = String.format("%d", 101);  
        String str2 = String.format("|%10d|", 101); // Specifying length of integer  
        String str3 = String.format("|%-10d|", 101); // Left-justifying within the specified width  
        String str4 = String.format("|% d|", 101);  
        String str5 = String.format("|%010d|", 101); // Filling with zeroes  
        System.out.println(str1);  
        System.out.println(str2);  
        System.out.println(str3);  
        System.out.println(str4);  
        System.out.println(str5);  
    }  
}
```

```
101  
|           101 |  
|101        |  
| 101 |  
|00000000101|
```

Displaying String, int, hexadecimal, float, char, octal value using format() method

```
public class JavaExample {  
    public static void main(String[] args) {  
        String str1 = String.format("%d", 15); // Integer value  
        String str2 = String.format("%s", "BeginnersBook.com"); // String  
        String str3 = String.format("%f", 16.10); // Float value  
        String str4 = String.format("%x", 189); // Hexadecimal value  
        String str5 = String.format("%c", 'P'); // Char value  
        String str6 = String.format("%o", 189); // Octal value  
        System.out.println(str1);  
        System.out.println(str2);  
        System.out.println(str3);  
        System.out.println(str4);  
        System.out.println(str5);  
        System.out.println(str6);  
    }  
}
```

```
15  
BeginnersBook.com  
16.100000  
bd  
P  
275
```

Java String format() with Locale

The String `format()` method also has another syntax if you have to work with the specified locale.

```
String.format(Locale l,  
              String format,  
              Object... args)
```

```
// to use Locale  
import java.util.Locale;  
  
class Main {  
    public static void main(String[] args) {  
        int number = 8652145;  
        String result;  
  
        // using the current locale  
        result = String.format("Number: %,d", number);  
        System.out.println(result);  
  
        // using the GERMAN locale as the first argument  
        result = String.format(Locale.GERMAN, "Number in German: %,d", number);  
        System.out.println(result);  
    }  
}
```

Number: 8,652,145
Number in German: 8.652.145

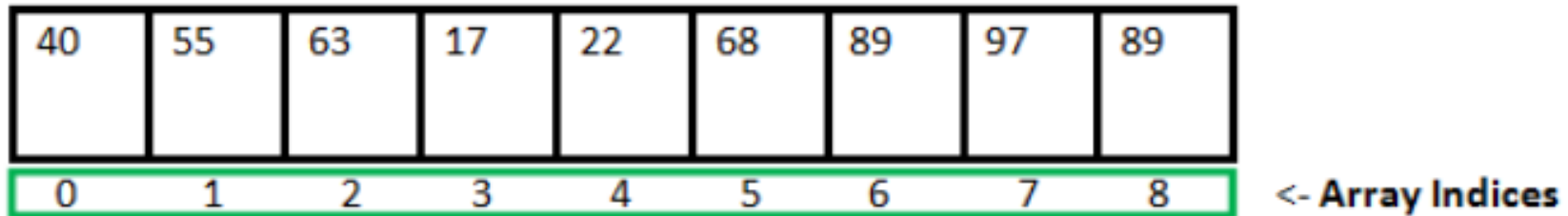
Arrays in Java

- An array is a group of like-typed variables that are referred to by a common name. Arrays in Java work differently than they do in C/C++.
- important points about Java arrays:
 - In Java all arrays are dynamically allocated.(discussed below)
 - Since arrays are objects in Java, we can find their length using the object property *length*.
 - A Java array variable can also be declared like other variables with [] after the data type.
 - The variables in the array are ordered and each have an index beginning from 0.
 - Java array can be also be used as a static field, a local variable or a method parameter.
 - The **size** of an array must be specified by an int or short value and not long

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array



Array Length = 9

First Index = 0

Last Index = 8

Creating, Initializing one dimensional array

- An array declaration has two components: the type and the name.
- the element type for the array determines what type of data the array will hold.
- Like an array of integers, we can also create an array of other primitive data types like char, float, double, etc. or user-defined data types (objects of a class).
- The general form of a one-dimensional array declaration is

```
type var-name[];
```

OR

```
type[] var-name;
```

```
// both are valid declarations
```

```
int intArray[];
```

```
or int[] intArray;
```

Creating Arrays

- You can create an array by using the new operator with the following syntax –

```
dataType[] arrayRefVar;  
arrayRefVar = new dataType[arraySize];
```

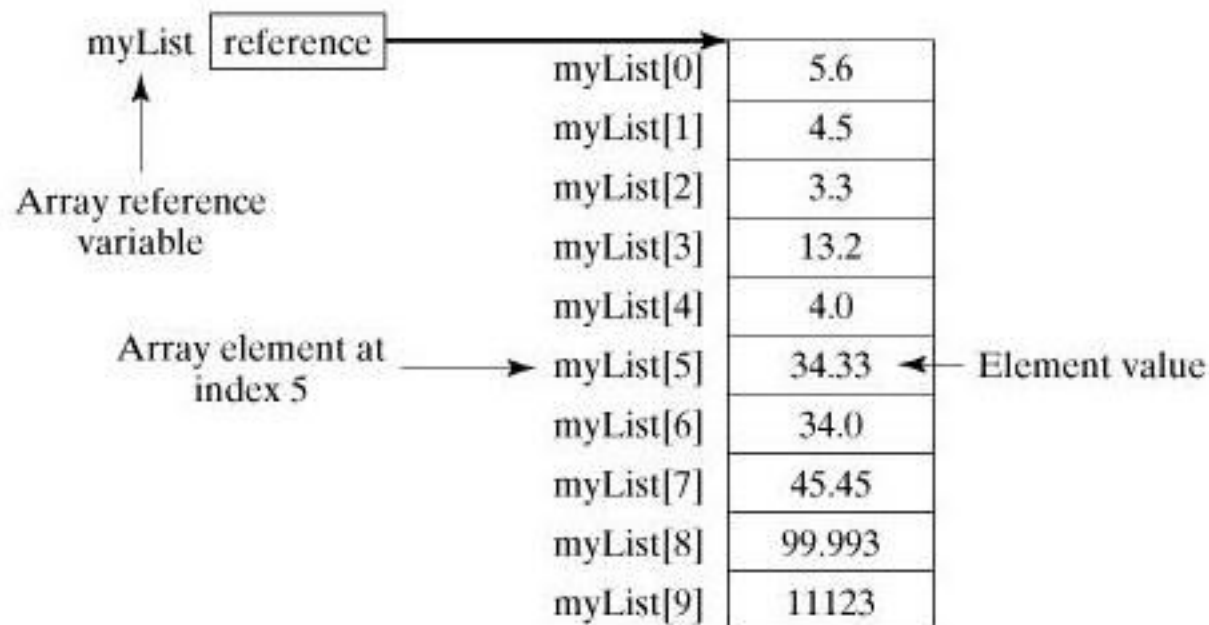
- The above statement does two things –
 - It creates an array using new dataType[arraySize].
 - It assigns the reference of the newly created array to the variable arrayRefVar.

- Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement:

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

- The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.
- Ex: `double[] myList = new double[10];`

Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.



```
class Testarray{
public static void main(String args[]){
int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
//traversing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

10

20

70

40

50

```
class Testarray1{  
public static void main(String args[]){  
int a[]={33,3,4,5};//declaration, instantiation and initialization  
//printing array  
for(int i=0;i<a.length;i++)//length is the property of array  
System.out.println(a[i]);  
}}
```

33

3

4

5

For each loop is used to traverse the complete array sequentially without using an index variable.

The syntax of the for-each loop is given below:

```
for(data_type variable:array){  
    //body of the loop  
}
```

```
public class TestArray {  
  
    public static void main(String[] args) {  
        double[] myList = {1.9, 2.9, 3.4, 3.5};  
  
        // Print all the array elements  
        for (double element: myList) {  
            System.out.println(element);  
        }  
    }  
}
```

```
1.9  
2.9  
3.4  
3.5
```



```
public class MyExample
{
    public static void main(String[] args)
    {
        // Creating a int Array with with values
        int[] number = new int[] { 11, 22, 33, 44, 55, 66, 77, 88, 99};
        System.out.println("Print int Array:");
        for(int i = 0; i < number.length; i++)
        {
            System.out.println("Values on index "+ i + ": " + number[i]);
        }

        // Creating a String Array with with values
        String[] names = new String[] { "Java", "Goal", "Learning", "Website", "for",
"Java", "Concepts"};
        System.out.println("Print String Array:");
        for(int i = 0; i < names.length; i++)
        {
            System.out.println("Values on index "+ i + ": " + names[i]);
        }
    }
}
```

Output: *Print int Array:*

int Array values on index 0: 11

int Array values on index 1: 22

int Array values on index 2: 33

int Array values on index 3: 44

int Array values on index 4: 55

int Array values on index 5: 66

int Array values on index 6: 77

int Array values on index 7: 88

int Array values on index 8: 99

Print String Array:

String Array values on index 0: Java

String Array values on index 1: Goal

String Array values on index 2: Learning

String Array values on index 3: Website

String Array values on index 4: for

String Array values on index 5: Java

String Array values on index 6: Concepts

Default values of the array

- After the array is declared, (before to initialize) array will be filled every array index with default values in memory locations.
- the default value is **0** for numeric type, null for String and false for boolean type.

```
class Array_Default{
public static void main(String args[]){
String[] names=new String[5];//array is declared, but not initilized
System.out.println(names[0]);
System.out.println(names[1]);
System.out.println(names[2]);
System.out.println(names[3]);
System.out.println(names[4]);
}
}
```

Output:

- null
- **null**
- **null**
- **null**
- null

Passing Array to a Method in Java

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

```
class Testarray2{
//creating a method which receives an array as a parameter
static void min(int arr[]){
int min=arr[0];
for(int i=1;i<arr.length;i++)
if(min>arr[i])
min=arr[i];

System.out.println(min);
}

public static void main(String args[]){
int a[]={33,3,4,5};//declaring and initializing an array
min(a);//passing array to method
}}
```

Output: 3

Returning Array from the Method

- We can also return an array from the method in Java.

```
class TestReturnArray{  
    //creating method which returns an array  
    static int[] get(){  
        return new int[]{10,30,50,90,60};  
    }  
  
    public static void main(String args[]){  
        //calling method which returns an array  
        int arr[]=get();  
        //printing the values of an array  
        for(int i=0;i<arr.length;i++)  
            System.out.println(arr[i]);  
    }  
}
```

10

30

50

90

60

Anonymous Array in Java

- Java supports the feature of an anonymous array, so you don't need to declare the array while passing an array to the method.

```
public class TestAnonymousArray{  
    //creating a method which receives an array as a parameter  
    static void printArray(int arr[]){  
        for(int i=0;i<arr.length;i++)  
            System.out.println(arr[i]);  
    }  
  
    public static void main(String args[]){  
        printArray(new int[]{10,22,44,66});//passing anonymous array to method  
    }  
}
```

10

22

44

66

Multidimensional Array in Java

- In multidimensional arrays, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in Java

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

Example to instantiate Multidimensional Array in Java

```
int[][] arr=new int[3][3];//3 row and 3 column
```


Example to initialize Multidimensional Array in Java

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

```
arr[0][1]=2;
```

```
arr[0][2]=3;
```

```
arr[1][0]=4;
```

```
arr[1][1]=5;
```

```
arr[1][2]=6;
```

```
arr[2][0]=7;
```

```
arr[2][1]=8;
```

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

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

1	2	3
2	4	5
4	4	5

Jagged Array in Java

- If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.

```
class TestJaggedArray{
    public static void main(String[] args){
        //declaring a 2D array with odd columns
        int arr[][] = new int[3][];
        arr[0] = new int[3];
        arr[1] = new int[4];
        arr[2] = new int[2];
        //initializing a jagged array
        int count = 0;
        for (int i=0; i<arr.length; i++)
            for(int j=0; j<arr[i].length; j++)
                arr[i][j] = count++;
    }
}
```

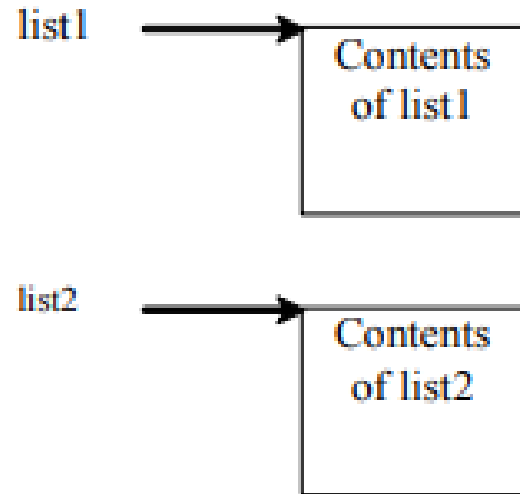
```
//printing the data of a jagged array
for (int i=0; i<arr.length; i++){
    for (int j=0; j<arr[i].length; j++){
        System.out.print(arr[i][j]+" ");
    }
    System.out.println();//new line
}
}
```

```
0 1 2
3 4 5 6
7 8
```

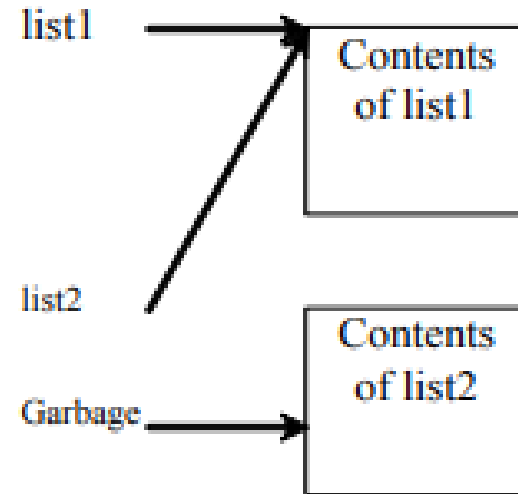
Copying Arrays

- `list2 = list1;`
- This statement does not copy the contents of the array referenced by `list1` to `list2`, but merely copies the reference value from `list1` to `list2`.
- After this statement, `list1` and `list2` reference to the same array.

Before the assignment
list2 = list1;



After the assignment
list2 = list1;



- The array previously referenced by list2 is no longer referenced; it becomes garbage, which will be automatically collected by the Java Virtual Machine.
- You can use assignment statements to copy primitive data type variables, but not arrays.
- Assigning one array variable to another variable actually copies one reference to another and makes both variables point to the same memory location.
- We use a **loop** or **arraycopy()** to copy arrays.

- Using a **loop**:

```
int[] sourceArray = {2, 3, 1, 5, 10};  
int[] targetArray = new int[sourceArray.length];  
  
for (int i = 0; i < sourceArray.length; i++)  
    targetArray[i] = sourceArray[i];
```

- The **arraycopy** method:

```
arraycopy(sourceArray, src_pos, targetArray, tar_pos, length);
```

Example:

```
System.arraycopy(sourceArray, 0, targetArray, 0, sourceArray.length);
```

- The number of elements copied from `sourceArray` to `targetArray` is indicated by `length`.
- The `arraycopy` does **not** allocate memory space for the target array. The target array must have already been created with its memory space allocated.
- After the copying take place, `targetArray` and `sourceArray` have the same content but independent memory locations.

```
class TestArrayCopyDemo {  
    public static void main(String[] args) {  
        //declaring a source array  
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',  
            'i', 'n', 'a', 't', 'e', 'd' };  
        //declaring a destination array  
        char[] copyTo = new char[7];  
        //copying array using System.arraycopy() method  
        System.arraycopy(copyFrom, 2, copyTo, 0, 7);  
        //printing the destination array  
        System.out.println(String.valueOf(copyTo));  
    }  
}
```

Output: caffein

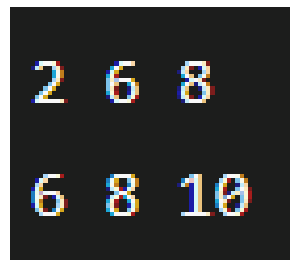
Addition of 2 Matrices

```
class Testarray5{
public static void main(String args[]){
//creating two matrices
int a[][]={{1,3,4},{3,4,5}};
int b[][]={{1,3,4},{3,4,5}};

//creating another matrix to store the sum of two matrices
int c[][]=new int[2][3];

//adding and printing addition of 2 matrices
for(int i=0;i<2;i++){
for(int j=0;j<3;j++){
c[i][j]=a[i][j]+b[i][j];
System.out.print(c[i][j]+" ");
}
System.out.println();//new line
}

}}
```



2	6	8
6	8	10

Multiplication of 2 Matrices

```
public class MatrixMultiplicationExample{
public static void main(String args[]){
//creating two matrices
int a[][]={{1,1,1},{2,2,2},{3,3,3}};
int b[][]={{1,1,1},{2,2,2},{3,3,3}};
//creating another matrix to store the multiplication of two matrices
int c[][]=new int[3][3]; //3 rows and 3 columns
//multiplying and printing multiplication of 2 matrices
for(int i=0;i<3;i++){
for(int j=0;j<3;j++){
c[i][j]=0;
for(int k=0;k<3;k++)
{
c[i][j]+=a[i][k]*b[k][j];
}
}
}
System.out.print(c[i][j]+" "); //printing matrix element
}
}
//end of j loop
System.out.println();//new line
}
}
```

```
6 6 6
12 12 12
18 18 18
```

Reading array using Scanner class

```
import java.util.*;
class OnedimensionalScanner
{
    public static void main(String args[])
    {
        int len;
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter Array length : ");
        len=sc.nextInt();
        int a[]=new int[len];//declaration
        System.out.print("Enter " + len + " Element to Store in Array :\n");
        for(int i=0; i<len; i++)
        {
            a[i] = sc.nextInt();
        }
        System.out.print("Elements in Array are :\n");
        for(int i=0; i<len; i++)
        {
            System.out.print(a[i] + " ");
        }
    }
}
```

```
Enter Array length :
4
Enter 4 Element to Store in Array :
1
2
3
4
Elements in Array are :
1 2 3 4
```

Class and object

- Java is an object-oriented programming language.
- Everything in Java is associated with classes and objects.
- A class is a user defined blueprint or prototype from which objects are created.
- Class represents the set of properties or methods that are common to all objects of one type.
- Example: a car is an object. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.

Creating a class in java

- We can create a class in Java using the **class** keyword.
- Class contains variables and methods.
- Syntax:

```
class classname
```

```
{
```

```
    variable declaration(class attributes);
```

```
    method declaration;
```

```
}
```

Create a Class:

```
public class Main {  
    int x = 5;  
}
```

Create an Object (for accessing the class attributes) :

In Java, an object is created from a class.

In Java, the new keyword is used to create new objects.

When we create an instance of the class by using the new keyword, it allocates memory (heap) for the newly created **object** and also returns the **reference** of that object.

You can access attributes by creating an object of the class, and by using the dot syntax (.)

Syntax:

```
ClassName object = new ClassName();
```

```
Main myObj = new Main();
```

```
public class Main
{
    int x = 5;

    public static void main(String[] args)
    {
        Main myObj = new Main();
        System.out.println(myObj.x);
    }
}
```

Output: 5

Modify Attributes

```
public class Main {  
    int x;  
  
    public static void main(String[] args) {  
        Main myObj = new Main();  
        System.out.println(myObj.x);  
    }  
}
```

Output: 0

```
public class Main {  
    int x;  
  
    public static void main(String[] args) {  
        Main myObj = new Main();  
        myObj.x = 40;  
        System.out.println(myObj.x);  
    }  
}
```

Output: 40


```
public class Main {
    int x = 10;

    public static void main(String[] args) {
        Main myObj = new Main();
        myObj.x = 25; // x is now 25
        System.out.println(myObj.x);
    }
}
```

Output: 25

If you don't want to override existing values, declare the attribute as **final**. The final keyword is useful when you want a variable to always store the same value, like PI (3.14159...).

```
public class Main {
    final int x = 10;

    public static void main(String[] args) {
        Main myObj = new Main();
        myObj.x = 25; // will generate an error
        System.out.println(myObj.x);
    }
}
```

```
Main.java:6: error: cannot assign a value to final variable x
        myObj.x = 25;
                ^
1 error
```

Multiple objects

If you create multiple objects of one class, you can change the attribute values in one object, without affecting the attribute values in the other.

```
public class Main {  
    int x = 5;  
  
    public static void main(String[] args) {  
        Main myObj1 = new Main();  
        Main myObj2 = new Main();  
        System.out.println(myObj1.x);  
        System.out.println(myObj2.x);  
    }  
}
```

Output: 5
5

```
public class Main {  
    int x = 5;  
  
    public static void main(String[] args) {  
        Main myObj1 = new Main();  
        Main myObj2 = new Main();  
        myObj2.x = 25;  
        System.out.println(myObj1.x);  
        System.out.println(myObj2.x);  
    }  
}
```

5

25

Multiple Attributes

- We can specify as many attributes as we want in a class.

```
public class Main {  
    String name = "Rama";  
    int age = 10;  
  
    public static void main(String[] args) {  
        Main myObj = new Main();  
        System.out.println("Name: " + myObj.name);  
        System.out.println("Age: " + myObj.age);  
    }  
}
```

Name: Rama

Age: 10

Using Multiple Classes

- You can also create an object of a class and access it in another class.
- This is often used for better organization of classes. one class has all the attributes and methods, while the other class holds the main() method.
- the name of the java file should match the class name.

```
public class Main {  
    int x = 5;  
}  
  
class Second {  
    public static void main(String[] args) {  
        Main myObj = new Main();  
        System.out.println(myObj.x);  
    }  
}
```

Save the above program with Second.java because the main method is available at Second class.

```
C:\Users\Your Name>javac Second.java
```

```
C:\Users\Your Name>java Second
```

Class methods

```
public class Example1
{
void show()
{
System.out.println("Welcome to ANU");
}
public static void main(String[] args)
{
//creating an object using new keyword
Example1 obj = new Example1();
//invoking method using the object
obj.show();
}
}
```

Output: Welcome to ANU

initialize object

- There are 3 ways to initialize object in Java.
 - By reference variable
 - By method
 - By constructor

Initialization through reference

- Initializing an object means storing data into the object.

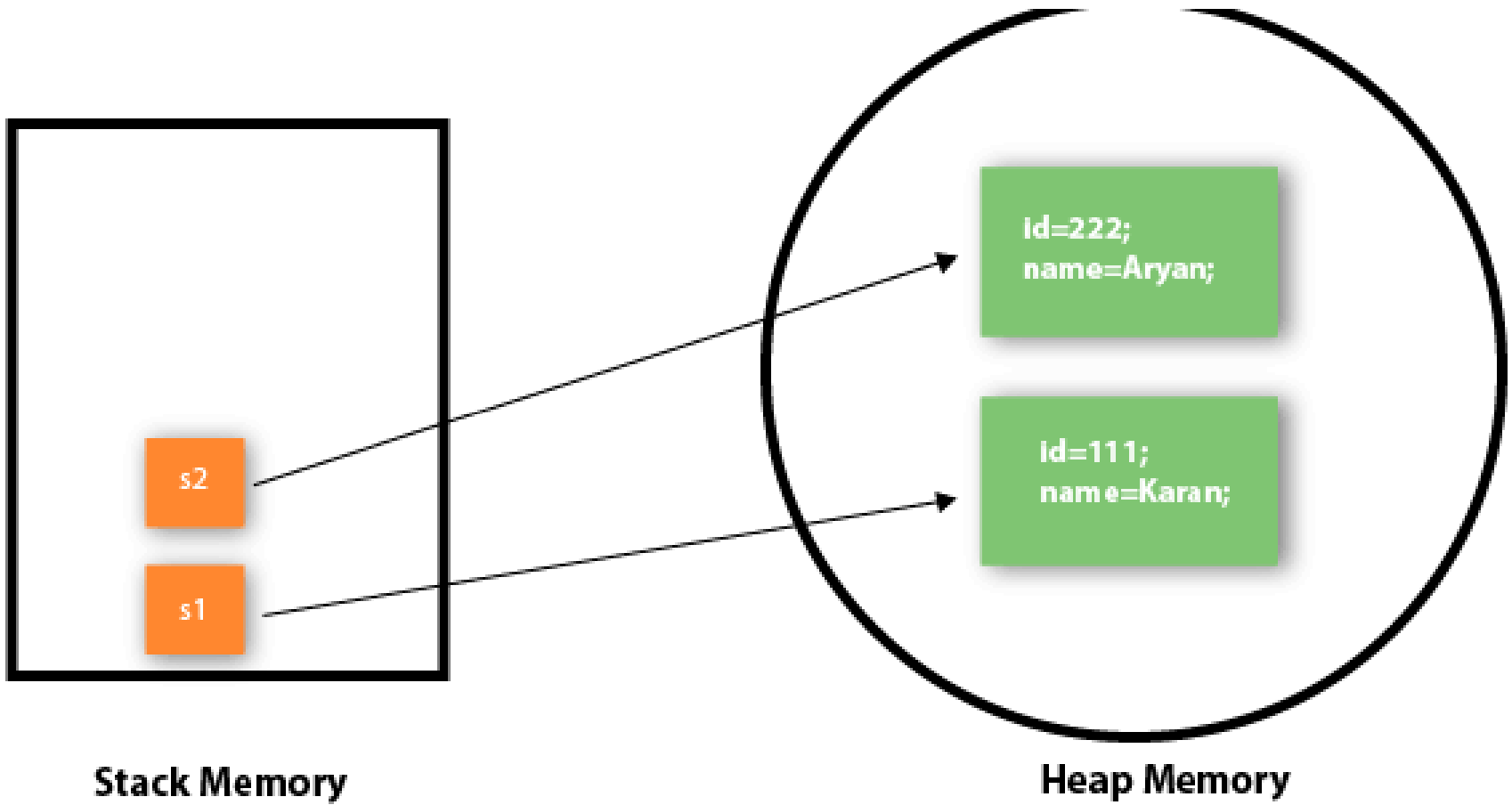
```
class Student{
    int id;
    String name;
}
class TestStudent2{
    public static void main(String args[]){
        Student s1=new Student();
        s1.id=101;
        s1.name="Sonoo";
        System.out.println(s1.id+" "+s1.name);//printing members with a white space
    }
}
```

Initialization through method

In this, we create methods for initialising the objects.

```
class Student{
    int rollno;
    String name;
    void insertRecord(int r, String n){
        rollno=r;
        name=n;
    }
    void displayInformation(){System.out.println(rollno+" "+name);}
}
class TestStudent4{
    public static void main(String args[]){
        Student s1=new Student();
        Student s2=new Student();
        s1.insertRecord(111,"Karan");
        s2.insertRecord(222,"Aryan");
        s1.displayInformation();
        s2.displayInformation();
    }
}
```

111 Karan
222 Aryan



```
class Employee{
    int id;
    String name;
    float salary;
    void insert(int i, String n, float s) {
        id=i;
        name=n;
        salary=s;
    }
    void display(){System.out.println(id+" "+name+" "+salary);}
}

public class TestEmployee {
    public static void main(String[] args) {
        Employee e1=new Employee();
        Employee e2=new Employee();
        Employee e3=new Employee();
        e1.insert(101,"ajeet",45000);
        e2.insert(102,"irfan",25000);
        e3.insert(103,"nakul",55000);
        e1.display();
        e2.display();
        e3.display();
    }
}
```

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

```
class Rectangle{  
  int length;  
  int width;  
  void insert(int l, int w){  
    length=l;  
    width=w;  
  }  
  void calculateArea(){System.out.println(length*width);}  
}
```

```
class TestRectangle1{  
  public static void main(String args[]){  
    Rectangle r1=new Rectangle();  
    Rectangle r2=new Rectangle();  
    r1.insert(11,5);  
    r2.insert(3,15);  
    r1.calculateArea();  
    r2.calculateArea();  
  }  
}
```

55

45

```
class Employee{
    int id;
    String name;
    float salary;
    void insert(int i, String n, float s) {
        id=i;
        name=n;
        salary=s;
    }
    void display(){System.out.println(id+ " "+name+ " "+salary);}
}
```

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

```
    Employee e1=new Employee();
    Employee e2=new Employee();
    Employee e3=new Employee();
    e1.insert(101,"ajeet",45000);
    e2.insert(102,"irfan",25000);
    e3.insert(103,"nakul",55000);
    e1.display();
    e2.display();
    e3.display();
```

```
}
```

```
}
```

```
101 ajeet 45000.0
```

```
102 irfan 25000.0
```

```
103 nakul 55000.0
```

Constructors in java

- A constructor in Java is a **special method** that is used to initialize the newly created object before it is used.
- The constructor is called when an object of a class is created.
- It can be used to set initial values for object attributes.
- It has the same name as its class and is syntactically similar to a method.
- All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero.
- However, once you define your own constructor, the default constructor is no longer used.

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

```
class Student3{  
  
int id;  
  
String name;  
  
//method to display the value of id and name  
void display(){System.out.println(id+ " "+name);}  
  
  
public static void main(String args[]){  
  
//creating objects  
Student3 s1=new Student3();  
Student3 s2=new Student3();  
  
//displaying values of the object  
s1.display();  
s2.display();  
}  
}
```

```
0 null
```

```
0 null
```

```
public class Main {
    int x; // Create a class attribute

    // Create a class constructor for the Main class
    public Main() {
        x = 5; // Set the initial value for the class attribute x
    }

    public static void main(String[] args) {
        Main myObj = new Main(); // Create an object of class Main (This will call the constructor)
        System.out.println(myObj.x); // Print the value of x
    }
}

// Outputs 5
```

Java Parameterized Constructor

- A constructor which has a specific number of parameters is called a parameterized constructor.
- The parameterized constructor is used to provide different values to distinct objects.

```
class Student4{
    int id;
    String name;
    //creating a parameterized constructor
    Student4(int i,String n){
        id = i;
        name = n;
    }
    //method to display the values
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
        //creating objects and passing values
        Student4 s1 = new Student4(111,"Karan");
        Student4 s2 = new Student4(222,"Aryan");
        //calling method to display the values of object
        s1.display();
        s2.display();
    }
}
```

111 Karan

222 Aryan

Constructor Overloading - Multiple Constructors for a Java Class

- 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.

```
class Student5{
    int id;
    String name;
    int age;
    //creating two arg constructor
    Student5(int i,String n){
        id = i;
        name = n;
    }
    //creating three arg constructor
    Student5(int i,String n,int a){
        id = i;
        name = n;
        age=a;
    }
    void display(){System.out.println(id+" "+name+" "+age);}
```

```
public static void main(String args[]){
    Student5 s1 = new Student5(111,"Karan");
    Student5 s2 = new Student5(222,"Aryan",25);
    s1.display();
    s2.display();
}
```

```
111 Karan 0
222 Aryan 25
```

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

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

Java static keyword

The **static keyword** in **Java** is used for memory management mainly. We can apply static keyword with **variables**, methods, blocks and **nested classes**. The static keyword belongs to the class than an instance of the class.

The static can be:

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class

1) Java static variable

If you declare any variable as static, it is known as a static variable.

- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

Advantages of static variable

It makes your program **memory efficient** (i.e., it saves memory)

Understanding the problem without static variable

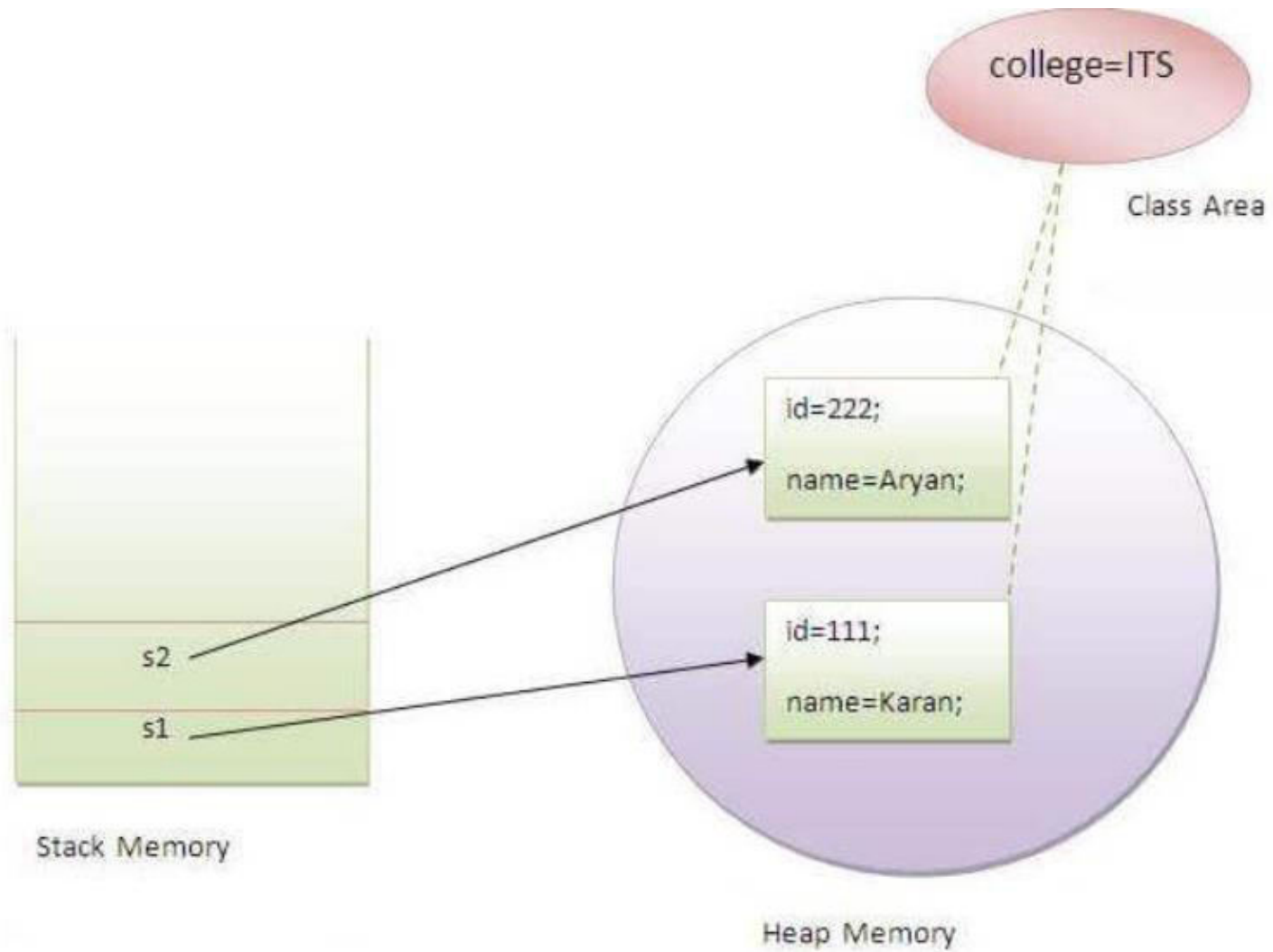
```
class Student{  
    int rollno;  
    String name;  
    String college="ITS";  
}
```

Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all **objects**. If we make it static, this field will get the memory only once.

```
//Java Program to demonstrate the use of static variable
class Student{
    int rollNo;//instance variable
    String name;
    static String college ="ITS";//static variable
    //constructor
    Student(int r, String n){
        rollno = r;
        name = n;
    }
    //method to display the values
    void display (){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to show the values of objects
public class TestStaticVariable1{
    public static void main(String args[]){
        Student s1 = new Student(111,"Karan");
        Student s2 = new Student(222,"Aryan");
        //we can change the college of all objects by the single line of code
        //Student.college="BBDIT";
        s1.display();
        s2.display();
    }
}
```

Output:

```
111 Karan ITS
222 Aryan ITS
```



```
//Java Program to illustrate the use of static variable which
//is shared with all objects.
class Counter2{
static int count=0;//will get memory only once and retain its value

Counter2(){
count++;//incrementing the value of static variable
System.out.println(count);
}

public static void main(String args[]){
//creating objects
Counter2 c1=new Counter2();
Counter2 c2=new Counter2();
Counter2 c3=new Counter2();
}
}
```

Output:

```
1
2
3
```

2) Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

```
//Java Program to demonstrate the use of a static method.
class Student{
    int rollno;
    String name;
    static String college = "ITS";
    //static method to change the value of static variable
    static void change(){
        college = "BBDIT";
    }
    //constructor to initialize the variable
    Student(int r, String n){
        rollno = r;
        name = n;
    }
    //method to display values
    void display(){System.out.println(rollno+" "+name+" "+college);}
}
```

```
//Test class to create and display the values of object
```

```
public class TestStaticMethod{  
    public static void main(String args[]){  
        Student.change();//calling change method  
        //creating objects  
        Student s1 = new Student(111,"Karan");  
        Student s2 = new Student(222,"Aryan");  
        Student s3 = new Student(333,"Sonoo");  
        //calling display method  
        s1.display();  
        s2.display();  
        s3.display();  
    }  
}
```

```
Output:111 Karan BBDIT  
        222 Aryan BBDIT  
        333 Sonoo BBDIT
```

Another example of a static method that performs a normal calculation

```
//Java Program to get the cube of a given number using the static method
```

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

Output:125

Restrictions for the static method

There are two main restrictions for the static method. They are:

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

```
class A{  
    int a=40;//non static  
  
    public static void main(String args[]){  
        System.out.println(a);  
    }  
}
```

Output:Compile Time Error

3) Java static block

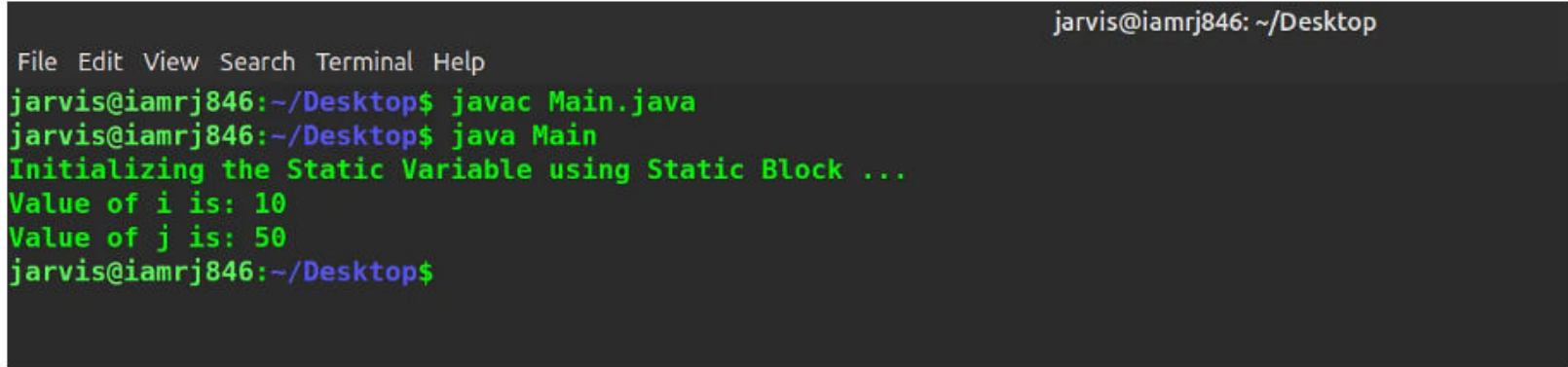
- Is used to initialize the static data member.
- It is executed before the main method at the time of classloading.

Example of static block

```
class A2{  
    static{System.out.println("static block is invoked");}  
    public static void main(String args[]){  
        System.out.println("Hello main");  
    }  
}
```

```
Output:static block is invoked  
        Hello main
```

```
class Test{
    static int i = 10;
    static int j;
    static{
        System.out.println("Initializing the Static Variable using Static Block ...");
        j = i * 5;
    }
}
class Main{
    public static void main(String args[]){
        System.out.println("Value of i is: " + Test.i);
        System.out.println("Value of j is: " + Test.j);
    }
}
```



```
jarvis@iamrj846: ~/Desktop
File Edit View Search Terminal Help
jarvis@iamrj846:~/Desktop$ javac Main.java
jarvis@iamrj846:~/Desktop$ java Main
Initializing the Static Variable using Static Block ...
Value of i is: 10
Value of j is: 50
jarvis@iamrj846:~/Desktop$
```

Static Nested Classes in Java

In Java, you can use static keywords for nested classes as well. However, it isn't possible to use the static keyword for outer classes or top-level classes. Please note that when you use nested classes, they don't need any sort of reference for outer classes in Java. Also, a nested static class cannot access the members of the outer class that are non-static. Let's consider the below example for better understanding.

```
class Test{
    static int i = 10;
    static class NestedTest{
        public void printer(){
            System.out.println("The value of i is: " + i);
        }
    }
}
class Main{
    public static void main(String args[]){
        Test.NestedTest t = new Test.NestedTest();
        t.printer();
    }
}
```

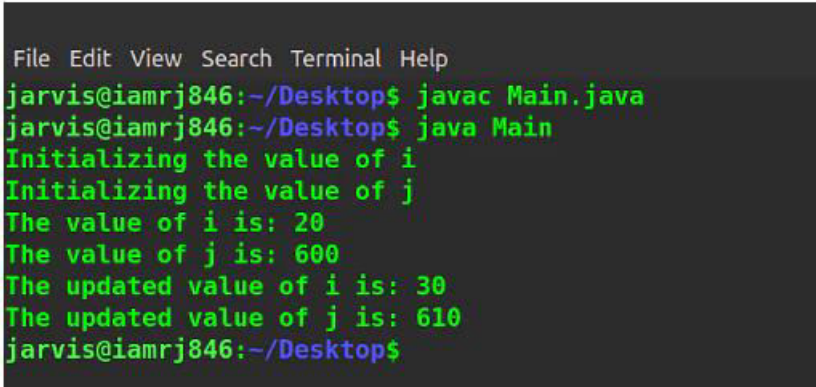


```
jarvis@iamrj846: ~/Desktop
File Edit View Search Terminal Help
jarvis@iamrj846:~/Desktop$ javac Main.java
jarvis@iamrj846:~/Desktop$ java Main
The value of i is: 10
jarvis@iamrj846:~/Desktop$
```

Let's consider a final example that uses all the types of static members that were discussed above.

```
class Test{
    //Static variable
    static int i;
    static int j;
    //Multiple Static Blocks
    static{
        System.out.println("Initializing the value of i");
        i = 20;
    }
    static{
        System.out.println("Initializing the value of j");
        j = i * 30;
    }
    //Static Method
    public static void display(){
        System.out.println("The value of i is: " + i);
        System.out.println("The value of j is: " + j);
    }
    //Static Nested Class
    static class NestedTest{
        public void changer(){
            i = i + 10;
            j = j + 10;
            System.out.println("The updated value of i is: " + i);
            System.out.println("The updated value of j is: " + j);
        }
    }
}

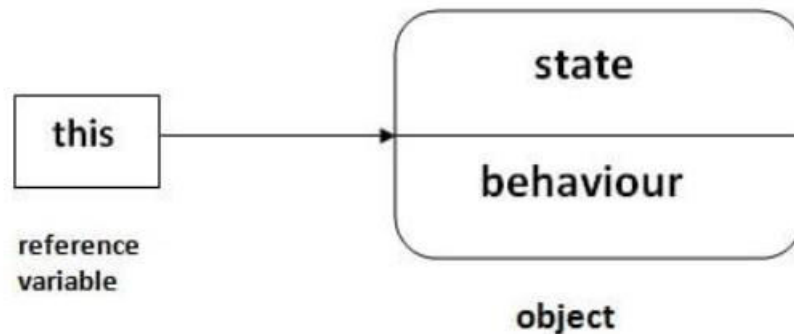
class Main{
    public static void main(String args[]){
        Test.display();
        Test.NestedTest t = new Test.NestedTest();
        t.changer();
    }
}
```



```
File Edit View Search Terminal Help
jarvis@iamrj846:~/Desktop$ javac Main.java
jarvis@iamrj846:~/Desktop$ java Main
Initializing the value of i
Initializing the value of j
The value of i is: 20
The value of j is: 600
The updated value of i is: 30
The updated value of j is: 610
jarvis@iamrj846:~/Desktop$
```

this keyword in Java

- There can be a lot of usage of **Java this keyword**. In Java, this is a **reference variable** that refers to the current object.
- Usage of Java this keyword
- Here is given the 6 usage of java this keyword.
 1. [this can be used to refer current class instance variable.](#)
 2. [this can be used to invoke current class method \(implicitly\)](#)
 3. [this\(\) can be used to invoke current class constructor.](#)



1) this: to refer current class instance variable

The `this` keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

```
class Student{
    int rollNo;
    String name;
    float fee;
    Student(int rollNo,String name,float fee){
        rollNo=rollNo;
        name=name;
        fee=fee;
    }
    void display(){System.out.println(rollNo+" "+name+" "+fee);}
}

class TestThis1{
    public static void main(String args[]){
        Student s1=new Student(111,"ankit",5000f);
        Student s2=new Student(112,"sumit",6000f);
        s1.display();
        s2.display();
    }
}
```

```
0 null 0.0
```

```
0 null 0.0
```

```
class Student{
    int rollNo;
    String name;
    float fee;
    Student(int rollNo,String name,float fee){
        this.rollNo=rollNo;
        this.name=name;
        this.fee=fee;
    }
    void display(){System.out.println(rollNo+" "+name+" "+fee);}
}
```

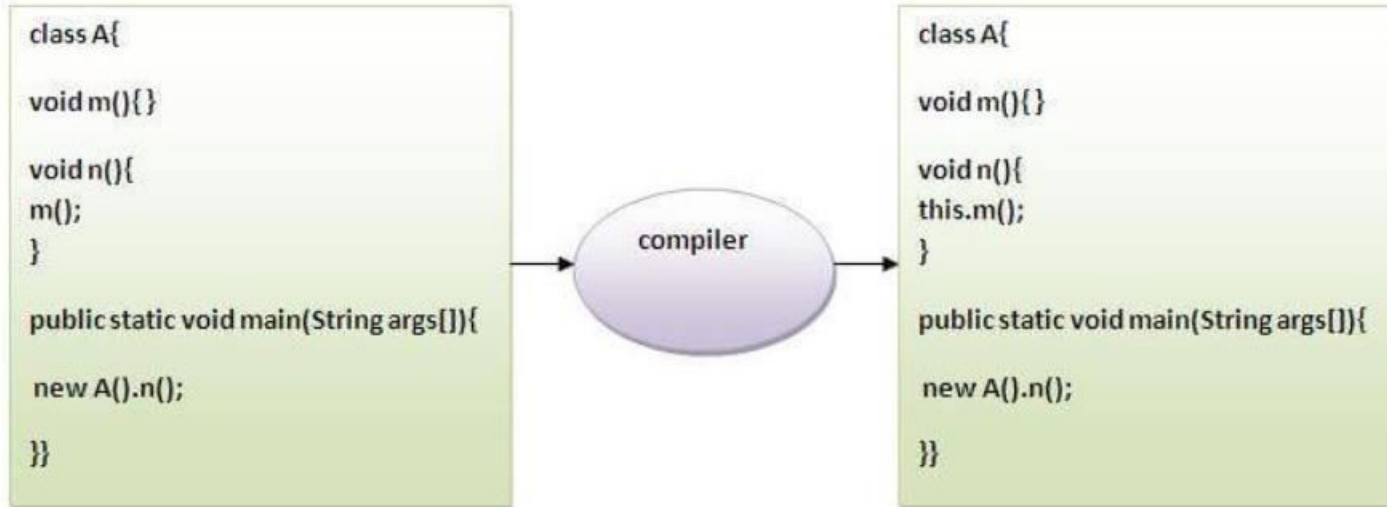
```
class TestThis2{
    public static void main(String args[]){
        Student s1=new Student(111,"ankit",5000f);
        Student s2=new Student(112,"sumit",6000f);
        s1.display();
        s2.display();
    }}
}
```

```
111 ankit 5000.0
112 sumit 6000.0
```

If local variables(formal arguments) and instance variables are different, there is no need to use this keyword

2) this: to invoke current class method

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example



```
class A{
void m(){System.out.println("hello m");}
void n(){
System.out.println("hello n");
//m();//same as this.m()
this.m();
}
}
class TestThis4{
public static void main(String args[]){
A a=new A();
a.n();
}}
```

```
hello n
hello m
```

3) this() : to invoke current class constructor

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

Calling default constructor from parameterized constructor:

```
class A{
A(){System.out.println("hello a");}
A(int x){
this();
System.out.println(x);
}
}

class TestThis5{
public static void main(String args[]){
A a=new A(10);
}}
```

```
hello a
10
```

Calling parameterized constructor from default constructor:

```
class A{
A(){
this(5);
System.out.println("hello a");
}
A(int x){
System.out.println(x);
}
}
class TestThis6{
public static void main(String args[]){
A a=new A();
}}
```

5

hello a

Real usage of this() constructor call

The this() constructor call should be used to reuse the constructor from the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining. Let's see the example given below that displays the actual use of this keyword.

```
class Student{
    int rollNo;
    String name,course;
    float fee;
    Student(int rollNo,String name,String course){
        this.rollNo=rollNo;
        this.name=name;
        this.course=course;
    }
    Student(int rollNo,String name,String course,float fee){
        this(rollNo,name,course);//reusing constructor
        this.fee=fee;
    }
    void display(){System.out.println(rollNo+" "+name+" "+course+" "+fee);}
}
class TestThis7{
    public static void main(String args[]){
        Student s1=new Student(111,"ankit","java");
        Student s2=new Student(112,"sumit","java",6000f);
        s1.display();
```

```
111 ankit java 0.0
112 sumit java 6000.0
```

```
Student(int rollNo,String name,String course){
    this.rollNo=rollNo;
    this.name=name;
    this.course=course;
}
Student(int rollNo,String name,String course,float fee){
    this.fee=fee;
    this(rollNo,name,course);//C.T.Error
}
```

Java String

In **Java**, string is basically an object that represents sequence of char values. An **array** of characters works same as Java string. For example:

```
char[] ch={'j','a','v','a','t','p','o','i','n','t'};  
String s=new String(ch);
```

is same as:

```
String s="javatpoint";
```

Java String class provides a lot of methods to perform operations on strings such as `compare()`, `concat()`, `equals()`, `split()`, `length()`, `replace()`, `compareTo()`, `intern()`, `substring()` etc.

The `java.lang.String` class implements *Serializable*, *Comparable* and *CharSequence* **interfaces**.

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 a string object.

How to create a string object?

There are two ways to create String object:

1. By string literal
2. By new keyword

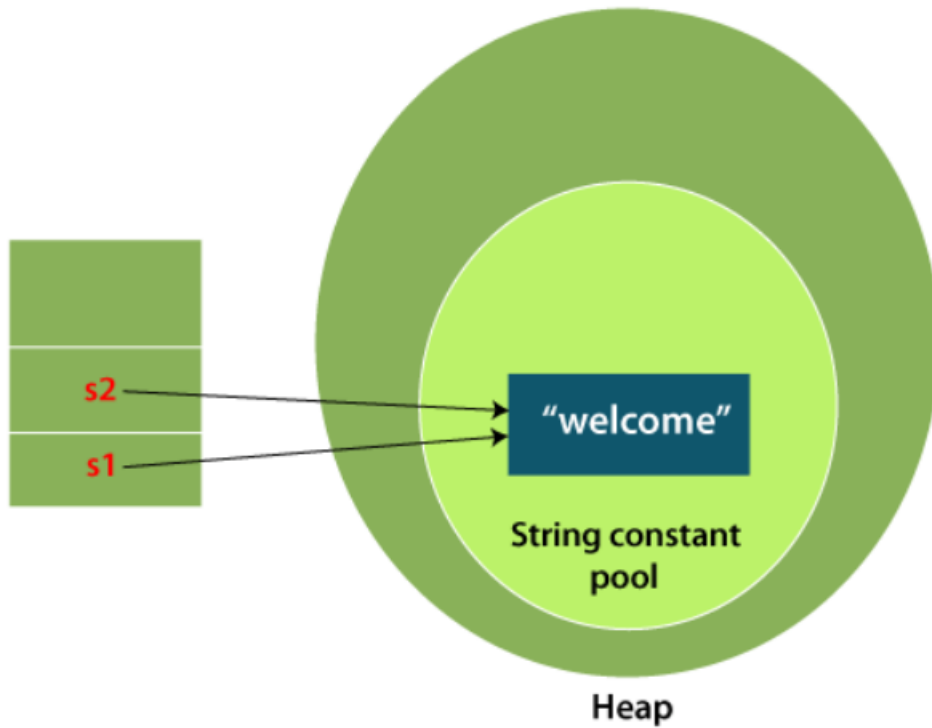
1) String Literal

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

```
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 the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

```
String s1="Welcome";  
String s2="Welcome";//It doesn't create a 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 that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

2) By new keyword

```
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 a heap (non-pool).

Java String Example

StringExample.java

```
public class StringExample{
    public static void main(String args[]){
        String s1="java";//creating string by Java string literal
        char ch[]={ 's','t','r','i','n','g','s'};
        String s2=new String(ch);//converting char array to string
        String s3=new String("example");//creating Java string by new keyword
        System.out.println(s1);
        System.out.println(s2);
        System.out.println(s3);
    }
}
```

java
strings
example

Constructors

String(byte[] byte_arr) – Construct a new String by decoding the *byte array*. It uses the platform's default character set for decoding.

Example:

```
byte[] b_arr = {71, 101, 101, 107, 115};  
String s_byte = new String(b_arr); //Geeks
```

String(byte[] byte_arr, String char_set_name) – Construct a new String by decoding the *byte array*. It uses the *char_set_name* for decoding.

It looks similar to the above constructs and they appear before similar functions but it takes the *String(which contains char_set_name)* as parameter while the above constructor takes *CharSet*.

Example:

```
byte[] b_arr = {71, 101, 101, 107, 115};  
String s = new String(b_arr, "US-ASCII"); //Geeks
```

String(byte[] byte_arr, int start_index, int length) – Construct a new string from the *bytes array* depending on the *start_index(Starting location)* and *length(number of characters from starting location)*.

Example:

```
byte[] b_arr = {71, 101, 101, 107, 115};  
String s = new String(b_arr, 1, 3); // eek
```

String(char[] char_arr) – Allocates a new String from the given *Character array*

Example:

```
char char_arr[] = {'G', 'e', 'e', 'k', 's'};
String s = new String(char_arr); //Geeks
```

String(char[] char_array, int start_index, int count) – Allocates a String from a given *character array* but choose *count* characters from the *start_index*.

Example:

```
char char_arr[] = {'G', 'e', 'e', 'k', 's'};
String s = new String(char_arr , 1, 3); //eek
```

String(int[] uni_code_points, int offset, int count) – Allocates a String from a *uni_code_array* but choose *count* characters from the *start_index*.

Example:

```
int[] uni_code = {71, 101, 101, 107, 115};
String s = new String(uni_code, 1, 3); //eek
```

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	<code>char charAt(int index)</code>	It returns char value for the particular index
2	<code>int length()</code>	It returns string length
3	<code>static String format(String format, Object... args)</code>	It returns a formatted string.
4	<code>static String format(Locale l, String format, Object... args)</code>	It returns formatted string with given locale.
5	<code>String substring(int beginIndex)</code>	It returns substring for given begin index.
6	<code>String substring(int beginIndex, int endIndex)</code>	It returns substring for given begin index and end index.
7	<code>boolean contains(CharSequence s)</code>	It returns true or false after matching the sequence of char value.
8	<code>static String join(CharSequence delimiter, CharSequence... elements)</code>	It returns a joined string.

9	<code>static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)</code>	It returns a joined string.
10	<code>boolean equals(Object another)</code>	It checks the equality of string with the given object.
11	<code>boolean isEmpty()</code>	It checks if string is empty.
12	<code>String concat(String str)</code>	It concatenates the specified string.
13	<code>String replace(char old, char new)</code>	It replaces all occurrences of the specified char value.
14	<code>String replace(CharSequence old, CharSequence new)</code>	It replaces all occurrences of the specified CharSequence.
15	<code>static String equalsIgnoreCase(String another)</code>	It compares another string. It doesn't check case.
16	<code>String[] split(String regex)</code>	It returns a split string matching regex.
17	<code>String[] split(String regex, int limit)</code>	It returns a split string matching regex and limit.
18	<code>String intern()</code>	It returns an interned string.

19	<code>int indexOf(int ch)</code>	It returns the specified char value index.
20	<code>int indexOf(int ch, int fromIndex)</code>	It returns the specified char value index starting with given index.
21	<code>int indexOf(String substring)</code>	It returns the specified substring index.
22	<code>int indexOf(String substring, int fromIndex)</code>	It returns the specified substring index starting with given index.
23	<code>String toLowerCase()</code>	It returns a string in lowercase.
24	<code>String toLowerCase(Locale l)</code>	It returns a string in lowercase using specified locale.
25	<code>String toUpperCase()</code>	It returns a string in uppercase.
26	<code>String toUpperCase(Locale l)</code>	It returns a string in uppercase using specified locale.
27	<code>String trim()</code>	It removes beginning and ending spaces of this string.
28	<code>static String valueOf(int value)</code>	It converts given type into string. It is an overloaded method.

String Length

Methods used to obtain information about an object are known as **accessor methods**. One accessor method that you can use with strings is the `length()` method, which returns the number of characters contained in the string object.

The following program is an example of **`length()`**, method String class.

Example

```
public class StringDemo {  
  
    public static void main(String args[]) {  
        String palindrome = "Dot saw I was Tod";  
        int len = palindrome.length();  
        System.out.println( "String Length is : " + len );  
    }  
}
```

This will produce the following result –

Output

```
String Length is : 17
```

Concatenating Strings

The String class includes a method for concatenating two strings –

```
string1.concat(string2);
```

This returns a new string that is string1 with string2 added to it at the end. You can also use the concat() method with string literals, as in –

```
"My name is ".concat("Zara");
```

Strings are more commonly concatenated with the + operator, as in –

```
"Hello," + " world" + "!"
```

which results in –

```
"Hello, world!"
```

String Methods

1. **int length():** Returns the number of characters in the String.

```
"GeeksforGeeks".length(); // returns 13
```

2. **Char charAt(int i):** Returns the character at i^{th} index.

```
"GeeksforGeeks".charAt(3); // returns 'k'
```

3. **String substring(int i):** Return the substring from the i^{th} index character to end.

```
"GeeksforGeeks".substring(3); // returns "ksforGeeks"
```

4. **String substring(int i, int j):** Returns the substring from i to $j-1$ index.

```
"GeeksforGeeks".substring(2, 5); // returns "eks"
```

5. **String concat(String str):** Concatenates specified string to the end of this string.

```
String s1 = "Geeks";  
String s2 = "forGeeks";  
String output = s1.concat(s2); // returns "GeeksforGeeks"
```


6. [int indexOf \(String s\)](#): Returns the index within the string of the first occurrence of the specified string.

```
String s = "Learn Share Learn";  
int output = s.indexOf("Share"); // returns 6
```

7. [int indexOf \(String s, int i\)](#): Returns the index within the string of the first occurrence of the specified string, starting at the specified index.

```
String s = "Learn Share Learn";  
int output = s.indexOf("ea",3);// returns 13
```

8. [Int lastIndexOf\(String s\)](#): Returns the index within the string of the last occurrence of the specified string.

```
String s = "Learn Share Learn";  
int output = s.lastIndexOf("a"); // returns 14
```

9. **boolean equals(Object otherObj)**: Compares this string to the specified object.

```
Boolean out = "Geeks".equals("Geeks"); // returns true  
Boolean out = "Geeks".equals("geeks"); // returns false
```

10. **`boolean equalsIgnoreCase (String anotherString)`**: Compares string to another string, ignoring case considerations.

```
Boolean out= "Geeks".equalsIgnoreCase("Geeks"); // returns true
Boolean out = "Geeks".equalsIgnoreCase("geeks"); // returns true
```

11. **`int compareTo(String anotherString)`**: Compares two string lexicographically.

```
int out = s1.compareTo(s2); // where s1 ans s2 are
                          // strings to be compared
```

This returns difference s1-s2. If :

```
out < 0 // s1 comes before s2
out = 0 // s1 and s2 are equal.
out > 0 // s1 comes after s2.
```

12. **`int compareToIgnoreCase(String anotherString)`**: Compares two string lexicographically, ignoring case considerations.

```
int out = s1.compareToIgnoreCase(s2);
// where s1 ans s2 are
// strings to be compared
```

This returns difference s1-s2. If :

```
out < 0 // s1 comes before s2
out = 0 // s1 and s2 are equal.
out > 0 // s1 comes after s2.
```

13. [String toLowerCase\(\)](#): Converts all the characters in the String to lower case.

```
String word1 = "HeLlLo";  
String word3 = word1.toLowerCase(); // returns "hello"
```

14. [String toUpperCase\(\)](#): Converts all the characters in the String to upper case.

```
String word1 = "HeLlLo";  
String word2 = word1.toUpperCase(); // returns "HELLO"
```

15. [String trim\(\)](#): Returns the copy of the String, by removing whitespaces at both ends. It does not affect whitespaces in the middle.

```
String word1 = " Learn Share Learn ";  
String word2 = word1.trim(); // returns "Learn Share Learn"
```

16. [String replace \(char oldChar, char newChar\)](#): Returns new string by replacing all occurrences of *oldChar* with *newChar*.

```
String s1 = "feeksforfeeks";  
String s2 = "feeksforfeeks".replace('f', 'g'); // returns "geeksgorgeeks"
```

Note:- s1 is still feeksforfeeks and s2 is geeksgorgeeks

StringBuffer class in Java

StringBuffer is a peer class of **String** that provides much of the functionality of strings. The string represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences. **StringBuffer** may have characters and substrings inserted in the middle or appended to the end. It will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.

- `java.lang.StringBuffer`

Constructors of StringBuffer class

1. StringBuffer(): It reserves room for 16 characters without reallocation

```
StringBuffer s = new StringBuffer();
```

2. StringBuffer(int size): It accepts an integer argument that explicitly sets the size of the buffer.

```
StringBuffer s = new StringBuffer(20);
```

3. StringBuffer(String str): It accepts a string argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.

```
StringBuffer s = new StringBuffer("GeeksforGeeks");
```

Important Constructors of StringBuffer Class

Constructor	Description
StringBuffer()	It creates an empty String buffer with the initial capacity of 16.
StringBuffer(String str)	It creates a String buffer with the specified string..
StringBuffer(int capacity)	It creates an empty String buffer with the specified capacity as length.

<code>append()</code>	Used to add text at the end of the existing text.
<code>length()</code>	The length of a StringBuffer can be found by the <code>length()</code> method
<code>capacity()</code>	the total allocated capacity can be found by the <code>capacity()</code> method
<code>charAt()</code>	
<code>delete()</code>	Deletes a sequence of characters from the invoking object
<code>deleteCharAt()</code>	Deletes the character at the index specified by <i>loc</i>
<code>ensureCapacity()</code>	Ensures capacity is at least equals to the given minimum.
<code>insert()</code>	Inserts text at the specified index position
<code>length()</code>	Returns length of the string
<code>reverse()</code>	Reverse the characters within a StringBuffer object
<code>replace()</code>	Replace one set of characters with another set inside a StringBuffer object

- [ensureCapacity\(\)](#)

It is used to increase the capacity of a StringBuffer object. The new capacity will be set to either the value we specify or twice the current capacity plus two (i.e. capacity+2), whichever is larger. Here, capacity specifies the size of the buffer.

Syntax:

```
void ensureCapacity(int capacity)
```

1) StringBuffer Class append() Method

The append() method concatenates the given argument with this String.

```
class StringBufferExample{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.append("Java");//now original string is changed
System.out.println(sb);//prints Hello Java
}
}
```

Output:

```
Hello Java
```

2) StringBuffer insert() Method

The insert() method inserts the given String with this string at the given position.

StringBufferExample2.java

```
class StringBufferExample2{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.insert(1,"Java");//now original string is changed
System.out.println(sb);//prints HJavaello
}
}
```

Output:

```
HJavaello
```


3) StringBuffer replace() Method

The replace() method replaces the given String from the specified beginIndex and endIndex.

StringBufferExample3.java

```
class StringBufferExample3{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.replace(1,3,"Java");
System.out.println(sb);//prints HJavao
}
}
```

Output:

```
HJavao
```

4) StringBuffer delete() Method

The delete() method of the StringBuffer class deletes the String from the specified beginIndex to endIndex.

```
class StringBufferExample4{  
    public static void main(String args[]){  
        StringBuffer sb=new StringBuffer("Hello");  
        sb.delete(1,3);  
        System.out.println(sb);//prints Hlo  
    }  
}
```

Output:

Hlo

5) StringBuffer reverse() Method

The reverse() method of the StringBuiler class reverses the current String.

StringBufferExample5.java

```
class StringBufferExample5{  
    public static void main(String args[]){  
        StringBuffer sb=new StringBuffer("Hello");  
        sb.reverse();  
        System.out.println(sb);//prints olleH  
    }  
}
```

olleH

6) StringBuffer capacity() Method

The capacity() method of the StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by $(oldcapacity*2)+2$. For example if your current capacity is 16, it will be $(16*2)+2=34$.

StringBufferExample6.java

```
class StringBufferExample6{
public static void main(String args[]){
StringBuffer sb=new StringBuffer();
System.out.println(sb.capacity());//default 16
sb.append("Hello");
System.out.println(sb.capacity());//now 16
sb.append("java is my favourite language");
System.out.println(sb.capacity());//now  $(16*2)+2=34$  i.e  $(oldcapacity*2)+2$ 
}
}
```

16

16

34

7) StringBuffer ensureCapacity() method

The ensureCapacity() method of the StringBuffer class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by $(oldcapacity*2)+2$. For example if your current capacity is 16, it will be $(16*2)+2=34$.

StringBufferExample7.java

```
class StringBufferExample7{
    public static void main(String args[]){
        StringBuffer sb=new StringBuffer();
        System.out.println(sb.capacity());//default 16
        sb.append("Hello");
        System.out.println(sb.capacity());//now 16
        sb.append("java is my favourite language");
        System.out.println(sb.capacity());//now  $(16*2)+2=34$  i.e  $(oldcapacity*2)+2$ 
        sb.ensureCapacity(10);//now no change
        System.out.println(sb.capacity());//now 34
        sb.ensureCapacity(50);//now  $(34*2)+2$ 
        System.out.println(sb.capacity());//now 70
    }
}
```

Output:

```
16
16
34
34
70
```