

**SIDDARTHA INSTITUTE OF SCIENCE AND  
TECHNOLOGY  
(AUTONOMOUS)**

**DEPARTMENT OF COMPUTER SCIENCE AND  
ENGINEERING**



**OBJECT ORIENTED PROGRAMMING THROUGH JAVA  
(20CS0506)**

# COURSE OBJECTIVES

1. Gain knowledge about basic Java language syntax and semantics to write Java programs and use concepts such as variables, conditional and iterative execution methods etc
2. Understand the basic object oriented programming concepts and apply them in problem solving.
3. Illustrate inheritance concepts for reusing the program.
4. Demonstrate on the multi-tasking by using multiple threads.
5. Understand the basics of java console and GUI based programming

# COURSE OUTCOMES

1. Understand simple abstract data types and design implementations using abstraction functions.
2. Recognize features of object-oriented design such as encapsulation, polymorphism, inheritance, and composition of systems based on object identity.
3. Implement Exception handling with synchronization.
4. Execute programs on Multithreading and String handling concepts.
5. Design applications with an event-driven graphical user interface.
6. Develop the Application using Programming Interfaces.

# **Unit-1**

The Java Language -Importance of Java - Programming Paradigms -The History and Evolution of Java -Java Byte Code -The Java Buzzwords. Introduction of OOP- Abstraction, Encapsulation, Inheritance, Polymorphism, Understanding static - Varargs -Data Types -Type Casting -Java Tokens - Java Statements - Arrays -Command line arguments.

# Importance of java

One of the most significant advantages of Java is **its ability to move easily from one computer system to another**. The ability to run the same program on many different systems is crucial to World Wide Web software, and Java succeeds at this by being platform-independent at both the source and binary levels.

# Programming Paradigms

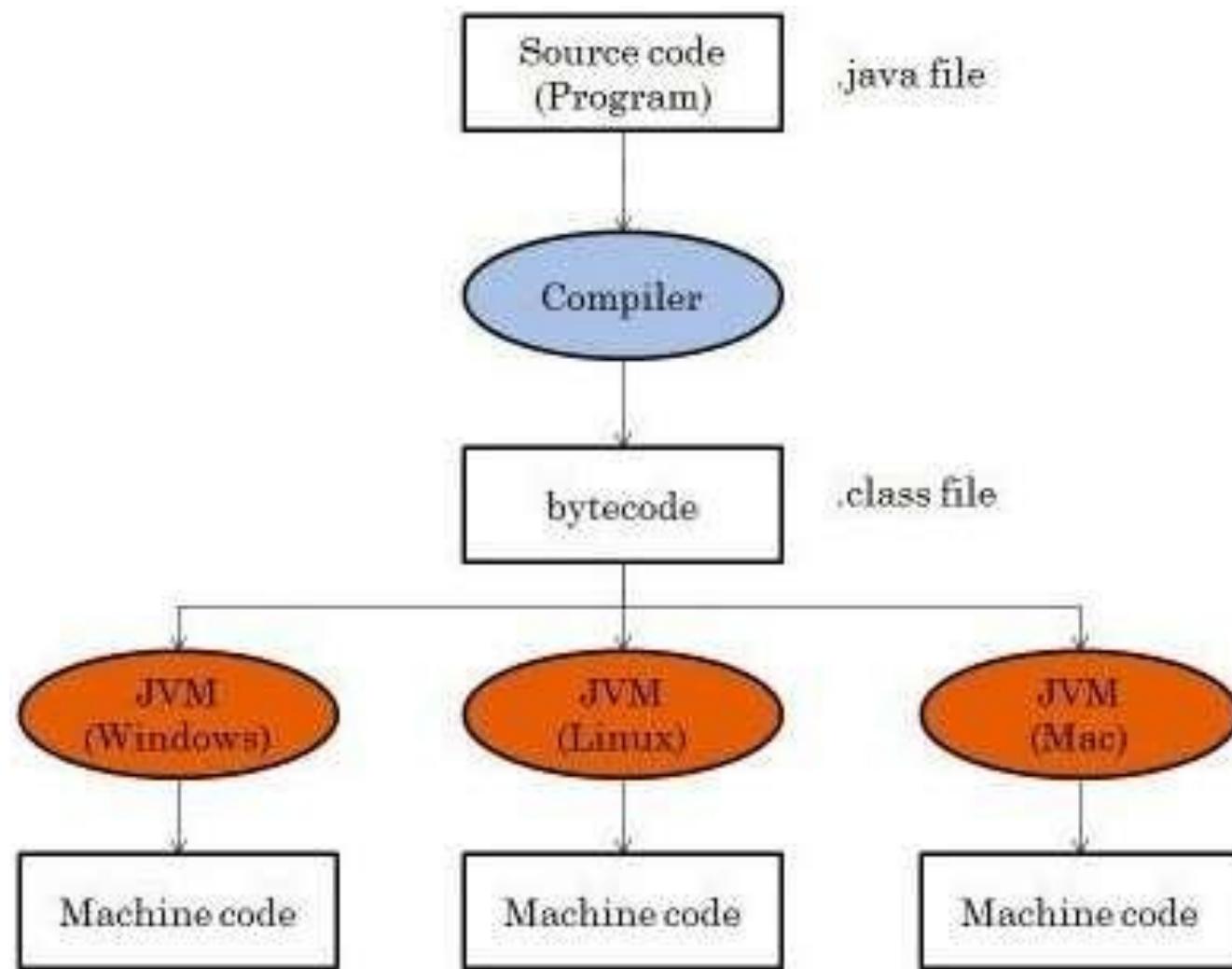
**Paradigm** can also be termed as method to solve some problem or do some task. Programming paradigm is an approach to solve problem using some programming language or also we can say it is a method to solve a problem using tools and techniques

# History of Java

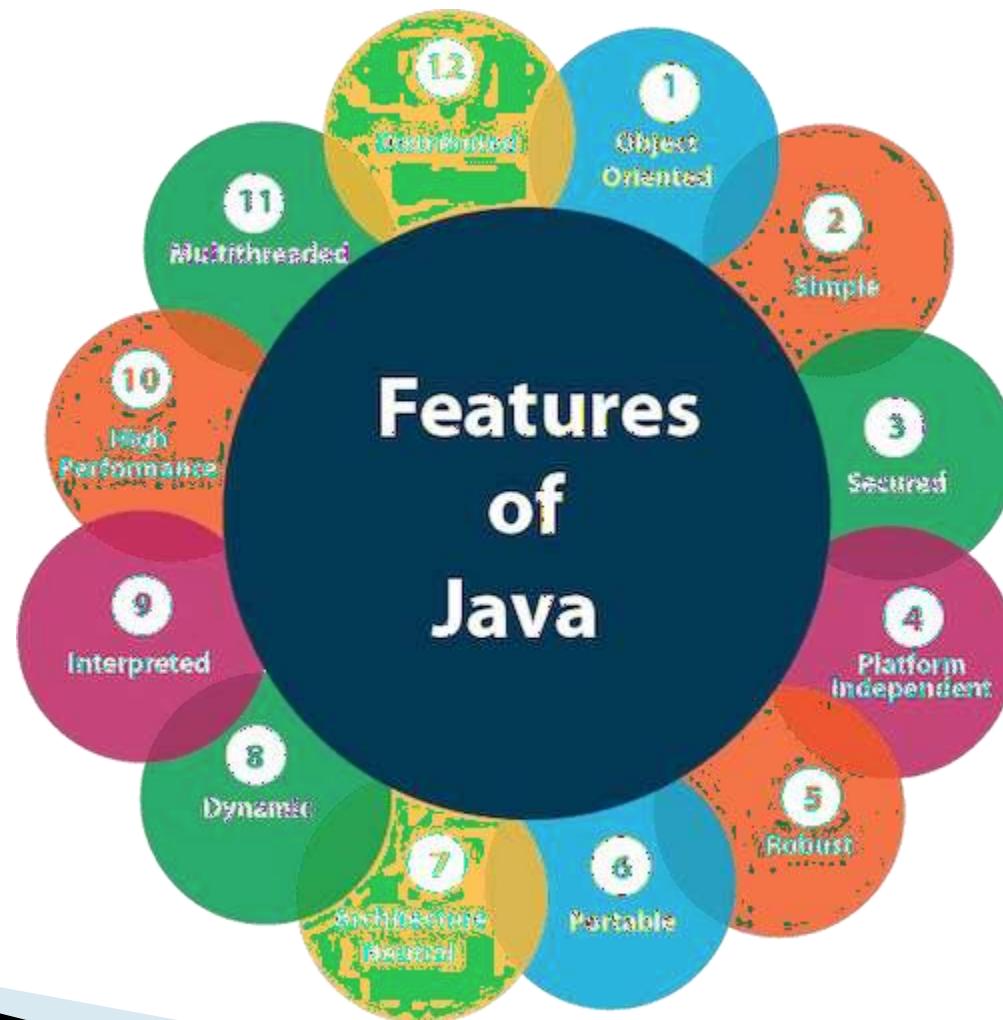
Java was developed by James Gosling, who is known as the father of Java, in 1995. James Gosling and his team members started the project in the early '90s.

- 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 it was 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.

# Java Byte Code



# The Java Buzzwords

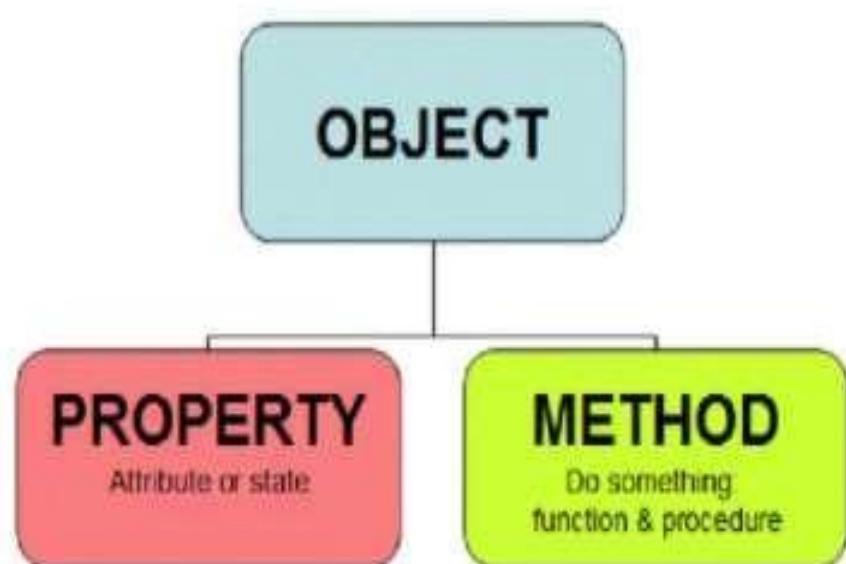


# Introduction of OOP

## OOP, What is it?

Something about objects and classes

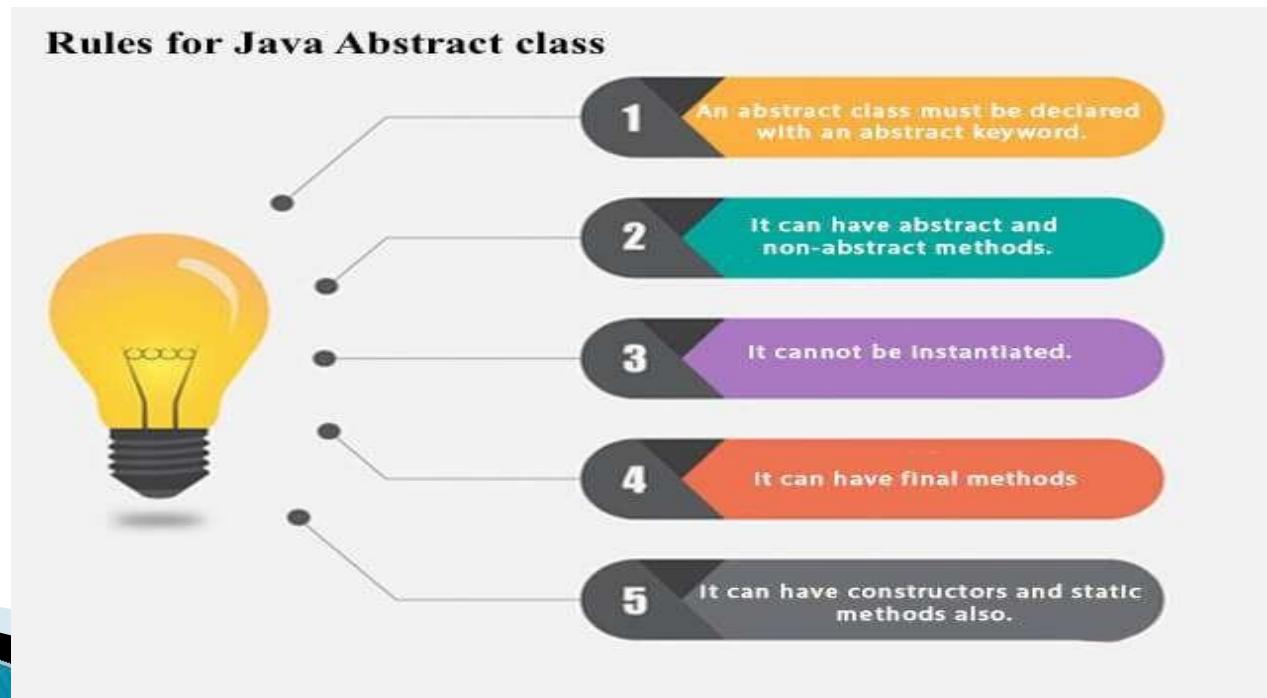
- Encapsulation
- Polymorphism
- Abstraction
- Inheritance



# Abstraction

**Abstraction** is the concept of object-oriented programming that "shows" only essential attributes and "hides" unnecessary information.

The main purpose of abstraction is hiding the unnecessary details from the users. Abstraction is selecting data from a larger pool to show only relevant details of the object to the user.



# Encapsulation

The process of binding data and corresponding methods (behavior) together into a single unit is called **encapsulation in Java**.

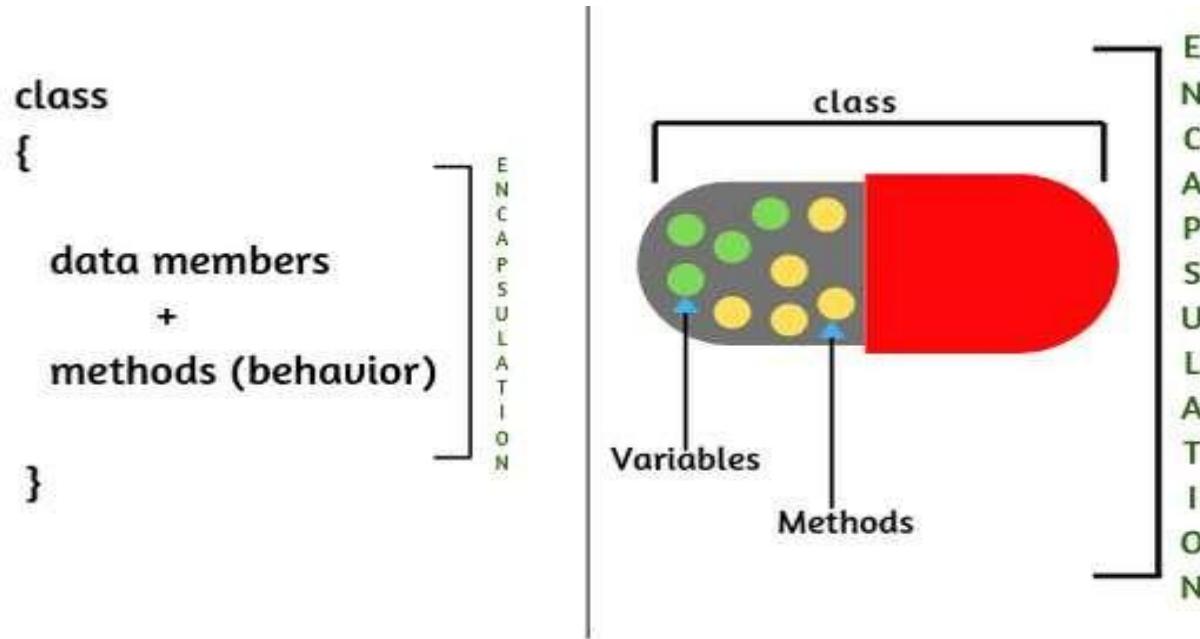
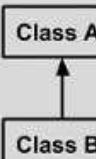
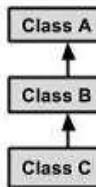
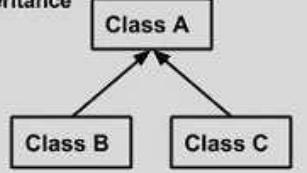
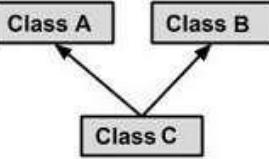


Fig: Encapsulation

# Inheritance

- Inheritance is the mechanism by which an object acquires the some/all properties of another object.
- It supports the concept of hierarchical classification.

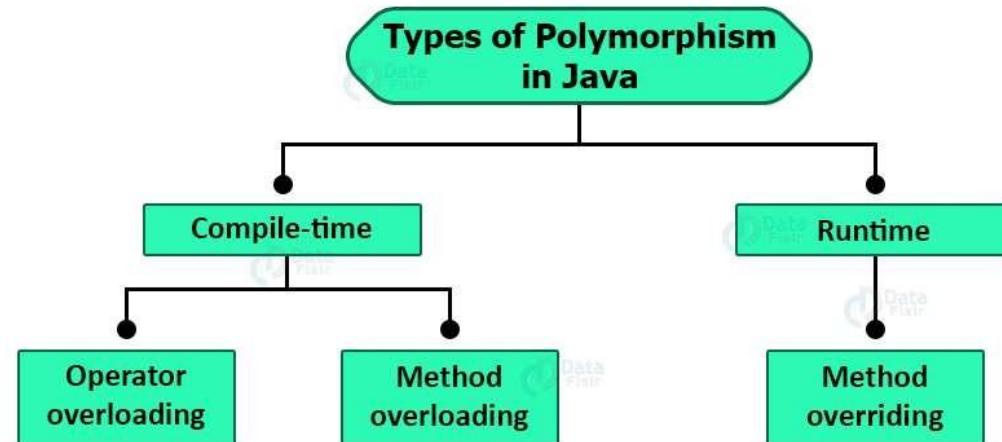
Single Inheritance		public class A { ..... } public class B extends A { ..... }
Multi Level Inheritance		public class A { ..... } public class B extends A { ..... } public class C extends B { ..... }
Hierarchical Inheritance		public class A { ..... } public class B extends A { ..... } public class C extends A { ..... }
Multiple Inheritance		public class A { ..... } public class B { ..... } public class C extends A,B { ..... } } // Java does not support multiple Inheritance

# Polymorphism

**Polymorphism** in Java is a concept by which we can perform a single action in different ways. Polymorphism is derived from 2 Greek words: **poly** and **morphs**. The word "poly" means **many** and "morphs" means **forms**. So polymorphism means many forms.

There are two types of polymorphism in Java: **compile-time polymorphism** and **runtime polymorphism**.

We can perform polymorphism in java by **method overloading** and **method overriding**.



# Understanding static keyword

- The static keyword in java is used for memory management mainly.
- We can apply static keyword with variables, methods, blocks and nested class
- 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

# Understanding static keyword

## 1. 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.

## Advantages of static variable:

- It makes your program memory efficient(it saves the memory)

# Understanding static keyword

## 2. 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.

# Understanding static keyword

## 3. Static Block:

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

Example:

```
class A2
```

```
{
```

```
    Static
```

```
{
```

```
        System.out.println("static block is invoked");
```

```
}
```

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

```
{
```

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

```
}
```

```
}
```

# Varargs

- The varargs allows the method to accept zero or multiple arguments.
- Variable length arguments are most useful when the number of arguments to be passed to the method is not known beforehand
- They also reduce the code as overloaded methods are not required
- A method that takes a variable number of arguments is a varargs method

## Syntax:

```
return_type method_name(data_type... variableName){ }
```

# Varargs

- A variable-length arguments is specified by three periods (...) for example

```
Public static void fun(int...a)
```

```
{  
    //method body
```

```
}
```

- This syntax tells the compiler that fun() can be called with zero or more arguments

# Varargs

## Important points of varargs:

- Varargs methods can also be overloaded but overloading may lead to ambiguity.
- There can be only one variable argument in a method
- Variable argument (varargs) must be the last argument

# Data types

**Data types** are different sizes and values that can be stored in the variable that is made as per convenience and circumstances to cover up all test cases. majorly two types of languages that are as follows:

First, one is a **Statically typed language** where each variable and expression type is already known at compile time.. For example C, C++, Java.

The other is **Dynamically typed languages**. These languages can receive different data types over time. For example Ruby, Python.

# Type casting

- **type casting** is a method or process that converts a data type into another data type in both ways manually and automatically.
- The automatic conversion is done by the compiler and manual conversion performed by the programmer.

Or

- Convert a value from one data type to another data type is known as **type casting**.

# Type casting

Type casting divided into two types:

1. Automatic type conversion(Implicit conversion/Widening ).

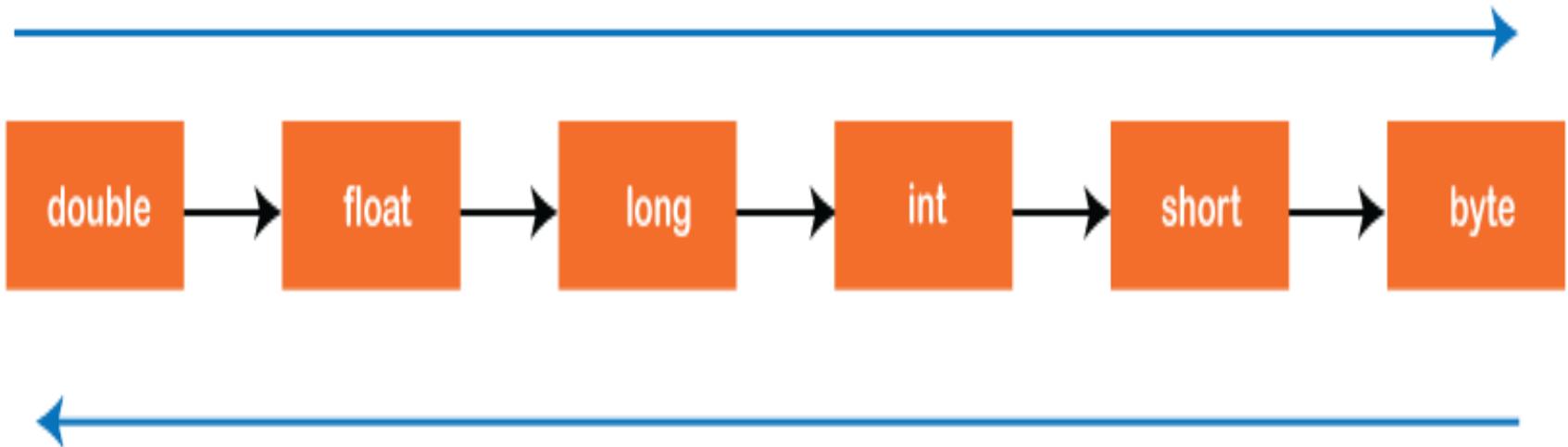
2. manual type conversion ( Explicit Conversion/Narrowing)

1. Automatic type conversion(Implicit conversion/Widening ).

- Converting a lower data type into a higher one is called **widening** type casting. It is also known as **implicit conversion or casting down**.
- It is done automatically. It is safe because there is no chance to lose data.
- Both data types must be compatible with each other.
- The target type must be larger than the source type.

# Type casting

Narrowing Type Casting



Widening Type Casting

Type Casting in Java

# Type casting

## Example program:

```
public class WideningTypeCastingExample
{
    public static void main(String[] args)
    {
        int x = 7;
        //automatically converts the integer type into long type
        long y = x;
        //automatically converts the long type into float type
        float z = y;
        System.out.println("Before conversion, int value "+x);
        System.out.println("After conversion, long value "+y);
        System.out.println("After conversion, float value "+z);
    }
}
```

# Type casting

Compile and executing program:

```
D:\>java>javac Implicit.java  
D:\>java>Implicit  
Before conversion, int value 7  
After conversion, long value 7  
After conversion, float value 7.0  
D:\>_
```

# Type casting

## 2. manual type conversion ( Explicit Conversion/Narrowing):

- If we want to assign a value of larger data type to a smaller data type we perform explicit type casting or narrowing.
- This is useful for incompatible data types where automatic conversion cannot be done.
- Here, target-type specifies the desired type to convert the specified value to.

**double -> float -> long -> int -> char -> short -> byte**

# Type casting

Compile and executing program:

```
[root@centos ~]# cat typecast.c  
#include <stdio.h>  
  
int main()  
{  
    float a = 1.2345;  
    float b = 2.3456;  
    printf("a = %f\n", a);  
    printf("b = %f\n", b);  
}  
[root@centos ~]# g++ typecast.c  
[root@centos ~]# ./typecast  
a = 1.2345  
b = 2.3456
```

Output

# Type casting

## Example program:

```
public class Explicit
{
    public static void main(String args[])
    {
        double d = 166.66;
        //converting double data type into long data type
        long l = (long)d;
        //converting long data type into int data type
        int i = (int)l;
        System.out.println("Before conversion: "+d);
        //fractional part lost
        System.out.println("After conversion into long type: "+l);
        //fractional part lost
        System.out.println("After conversion into int type: "+i);
    }
}
```

# Java statements:

- Selection statements

- If
  - If–else
  - Switch

- Iteration statements

- For
  - While
  - Do–while

- Jump statements

- Break
  - Continue
  - Return

# Selection statements:

## □ If statements:

- If statement is java's conditional statements
- If statement execution through two different paths
- Execute statement-block may be a single statements or group of statements.
- If the test expression is true the statements block will be executed
- Otherwise statement-x is executed
- Remember if the given expression/condition is true executes both block of statement and statement-x are executed sequence.

# If statement:

- Syntax/General form of if Statement“:

If(expression/condition)

{

    block of statement;

}

Statement-x;

- Example program:

# Selection statements:

## □ If-else statement:

- If condition is true block of statement will be executed.
- Condition is false otherwise else block will be executed.

### General form of if-else:

If(condition)

{

//executes this block of if

//condition is true

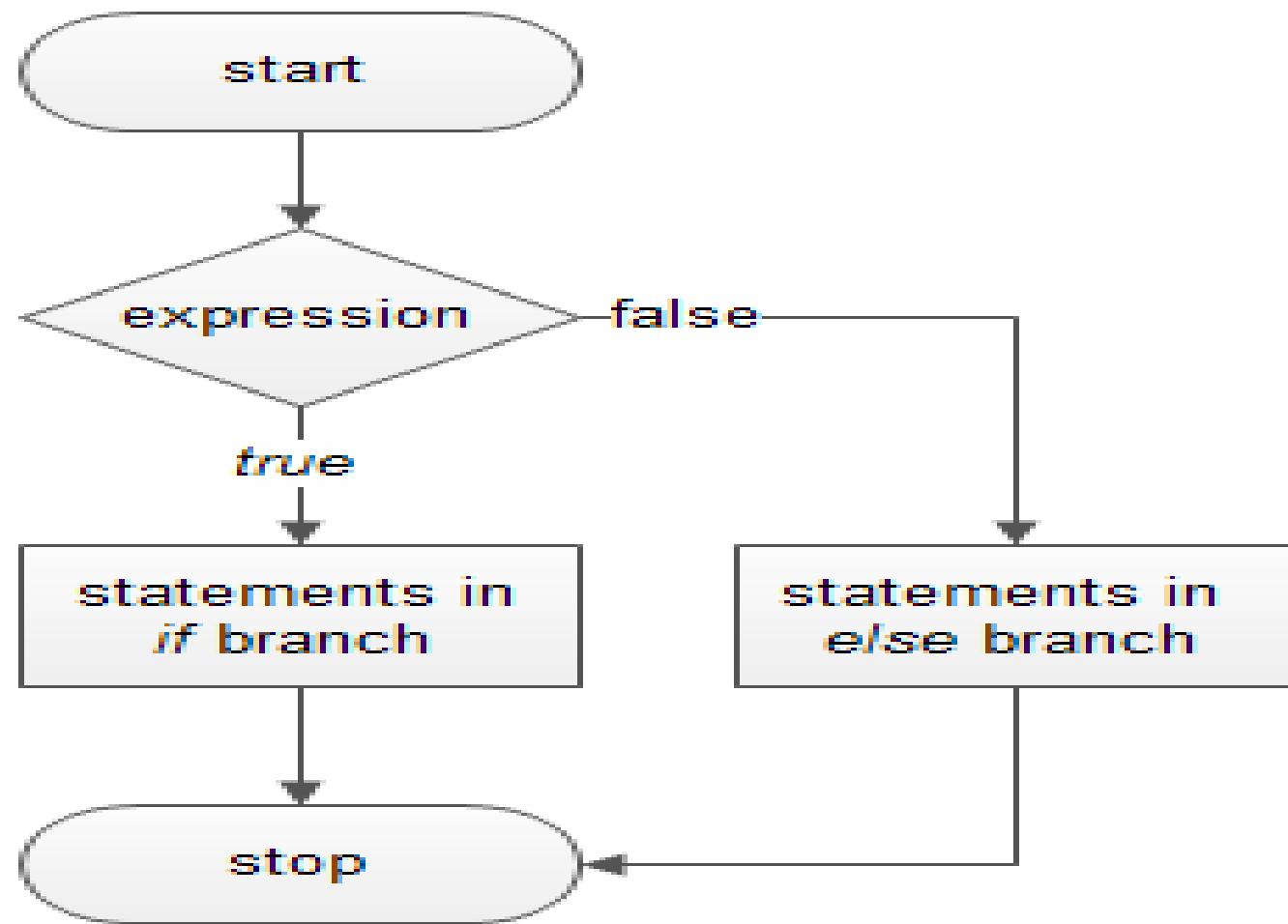
}

Else

{ // executes this block if  
// condition is false

}

# Selection statements:



# Selection statements:

## □ Switch statement:

- The switch statement is a multi-way branch statement.
- It provides an easy way to dispatch execution to different parts of code based on the value of the expression
- It executes single statement from multiple condition
- case must literal, constant.

## Syntax :

Switch(expression)

{

    Case value 1:

        statement;

        break;

    Case value 2:

        statement;

        break;

    -

    -

    Case value N:

        statement N;

        break;

    default:

        statement;

}

# Iteration/looping statements:

- Single statements or block of statements executes “n” no.of times is called iteration or looping.
- This are classified into 3 types:
  - For
  - While
  - Do-while

# Iteration/looping statements:

## □ For loop

- For loop provides a easy way of writing the loop structure.
- For the loop statement consumes the initialization, condition, increment/decrement.
- **Initialization:**
  - ❑ Here we initialize the variable in use. It marks the start of a for loop.
  - ❑ Already declared variable can be used or a variable can be declared, local to loop only.
- **Condition:**
  - It is used for testing the exit condition for a loop control loop as the condition is checked prior to the execution of the loop body are executed.

# Iteration/looping statements:

- Syntax:

```
for(initialization;condition;increment/decrement)
```

```
{  
    Statement;  
}
```

# Iteration/looping statements:

## □ While loop:

- A while loop is a control flow statement that allows code to be executed repeatedly based on a given boolean condition.
- Also called a pre-test loop

Syntax:

While(boolean condition)

{

    loop statements;

}

# Iteration/looping statements:

## □ Do-while:

- While loop starts with the checking of condition.
- If it evaluated true, then the loop body statements are executed.
- Once the condition is evaluated to true, the statements in the loop body are executed.
- Normally the statements contain an update value for the variable being processed for the next iteration
- When the condition becomes false the loop terminates which marks the end of its lifecycles

# Iteration/looping statements:

## ▫ Syntax:

do

{

    statements;

}

while(condition);

# Jump statements:

- Java supports three jump statements
  - Break
  - Continue
  - Return
- These three statements transfer control to other part of the program
- Break:
  - Terminate a sequence in a switch statement
  - To exit a loop

By using break we can force immediate termination of a loop, passing the conditional expression and any remaining code in the body of the loop.

# Jump statements:

## □ Continue:

- It is useful to force an early iteration of a loop.
- That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration.

## □ Return:

- The return statement is used to explicitly return from method
- It causes a program control to transfer back to the caller of the method.

# Array

- An array is a similar data type of collection name.
- Array is a collection of homogenous elements
- An array is a container object that holds fixed number of values of a single type.

datatype[] arrayname;

- In the index value start with zero.
- Here we are using new keyword. It used to create a memory dynamically

# Array

## □ Declaring Array Variables

- An array declaration has two components: the type and the name. *type* declares the element type of the array.
- The element type determines the data type of each element that comprises the array.
- 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).
- Thus, the element type for the array determines what type of data the array will hold.

## □ **Array initialization:**

- When an array is declared, only a reference of array is created.
- To actually create or give memory to array, you create an array like

```
var-name = new type [size];
```

Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *var-name* is the name of array variable that is linked to the array.

That is, to use *new* to allocate an array, **you must specify the type and number of elements to allocate.**

# arrays

## □ Advantages

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

## □ Disadvantages

- **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

# arrays

- Arrays are two types
  - 1. one dimensional array
  - 2. two dimensional array
- One dimensional array:
  - A one dimensional array is an array with a single index. An array has a variable named length in which the size of the array is stored.
- Syntax to Declare an Array in Java

dataType[] arr;

(or)

dataType []arr;

(or)

dataType arr[];

# arrays

## □ Instantiation of an Array in Java

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

## 2. multi-dimensional array:

**Multidimensional Arrays** can be defined in simple words as array of arrays. Data in multidimensional arrays are stored in tabular form (in row major order).

Syntax:

```
data_type[1st dimension][2nd dimension][][]..[Nth dimension]
```

```
array_name = new data_type[size1][size2]....[sizeN];
```

# array

- **data\_type**: Type of data to be stored in the array.  
For example: int, char, etc.
- **dimension**: The dimension of the array created.  
For example: 1D, 2D, etc.
- **array\_name**: Name of the array
- **size1, size2, ..., sizeN**: Sizes of the dimensions respectively.

# Command line arguments

A command-line argument is an information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy. They are stored as strings in the String array passed to main().

Example

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

Output:

compile by > javac CommandLine.java

run by > java CommandLine this is a command line 200

# **UNIT-2**

Introducing Classes -Class Fundamentals -  
Declaring Objects -Introducing Methods  
Introduction to Constructors, Constructor  
Overloading- Garbage Collection -Introducing  
final -Inheritance -Using Super- Method  
Overloading, Method Overriding -abstract  
classes - Packages and Interfaces.

# INTRODUCING CLASSES

- Class is a logical construct upon which the entire java language is built because it defines the shape and nature of an object.
- Any concept you wish to implement in a java program must be encapsulated within a class.

# **CLASS FUNDAMENTALS**

- Class is blue print that object follows.
- Class defines a new data types.
- Once defined this new type can be used to create objects of that type.
- Thus a class is a template for an object and an object is an instance of a class.
- Without class we cannot access the object.

# CLASS FUNDAMENTALS

## general form of class:

```
Class ClassName{  
    Type instance_variable1;  
    Type instance_variable2;  
    {  
        // body of method  
    }  
    ---  
    ---  
    ---  
    Type methodname N(param_list)  
    {  
        //body of method  
    }
```

# Simple class

- A class called Box that defines three instance variable width, height and depth.
- Box does not contain any method.

```
Box
```

```
{
```

```
    double width;
```

```
    double height;
```

```
    double depth;
```

```
}
```

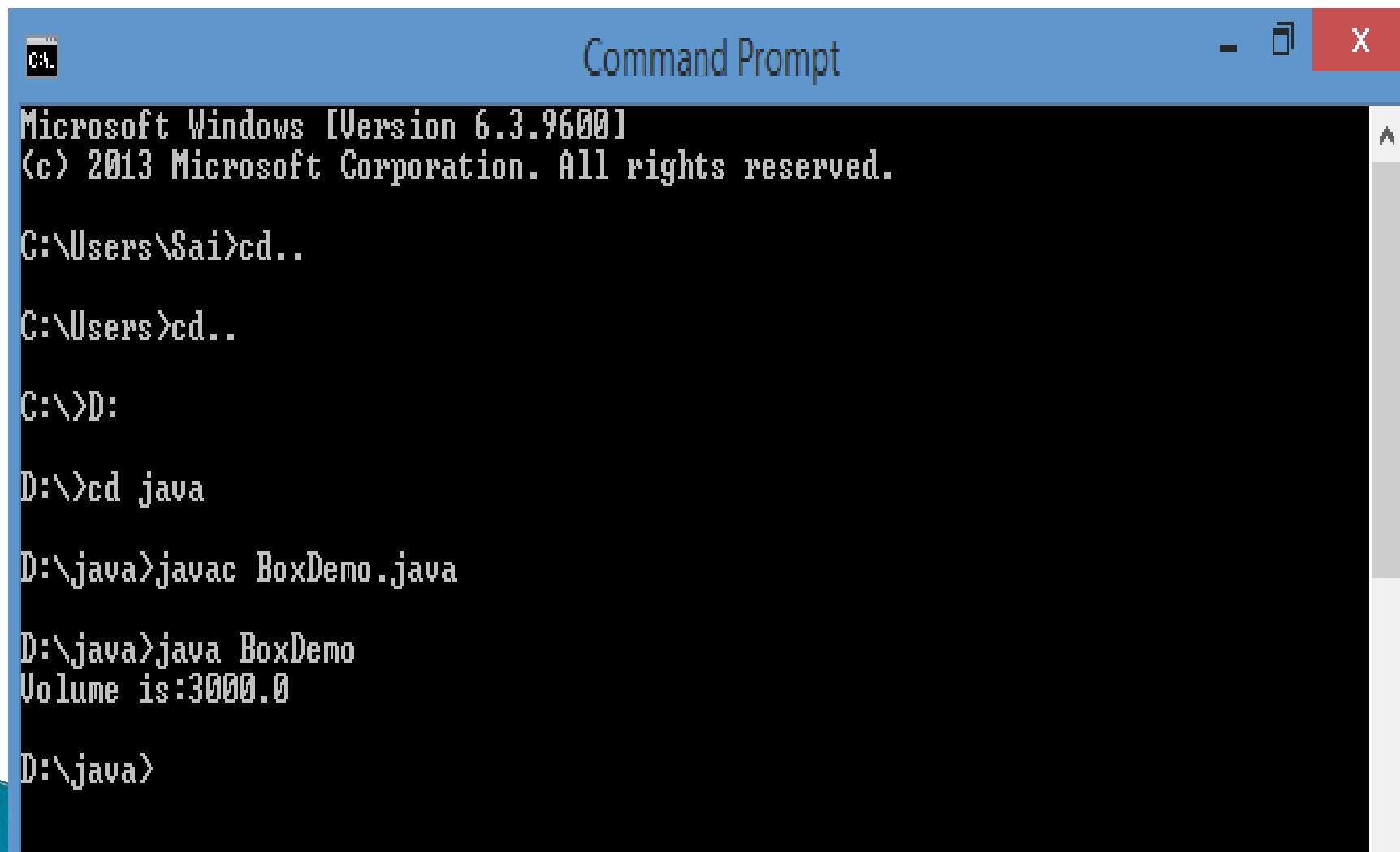
# Introducing classes

Example:

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

class BoxDemo
{
    public static void main(String args[])
    {
        Box mybox=new Box();
        double volume;
        //assign value to mybox instance variable
        mybox.width=10;
        mybox.height=20;
        mybox.depth=15;
        //compute volume of box
        volume=mybox.width*mybox.height*mybox.depth;
        System.out.println("Volume is:"+volume);
    }
}
```

# Introducing classes



The screenshot shows a Microsoft Windows Command Prompt window. The title bar reads "Command Prompt". The window content displays a series of commands and their outputs:

```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\Sai>cd..
C:\Users>cd..
C:\>D:
D:\>cd java
D:\java>javac BoxDemo.java
D:\java>java BoxDemo
Volume is:3000.0
D:\java>
```

# Declaring objects

- Object is instance of class

Syntax of object:

**Classname refference\_variable=new Classname();**

# Introducing methods

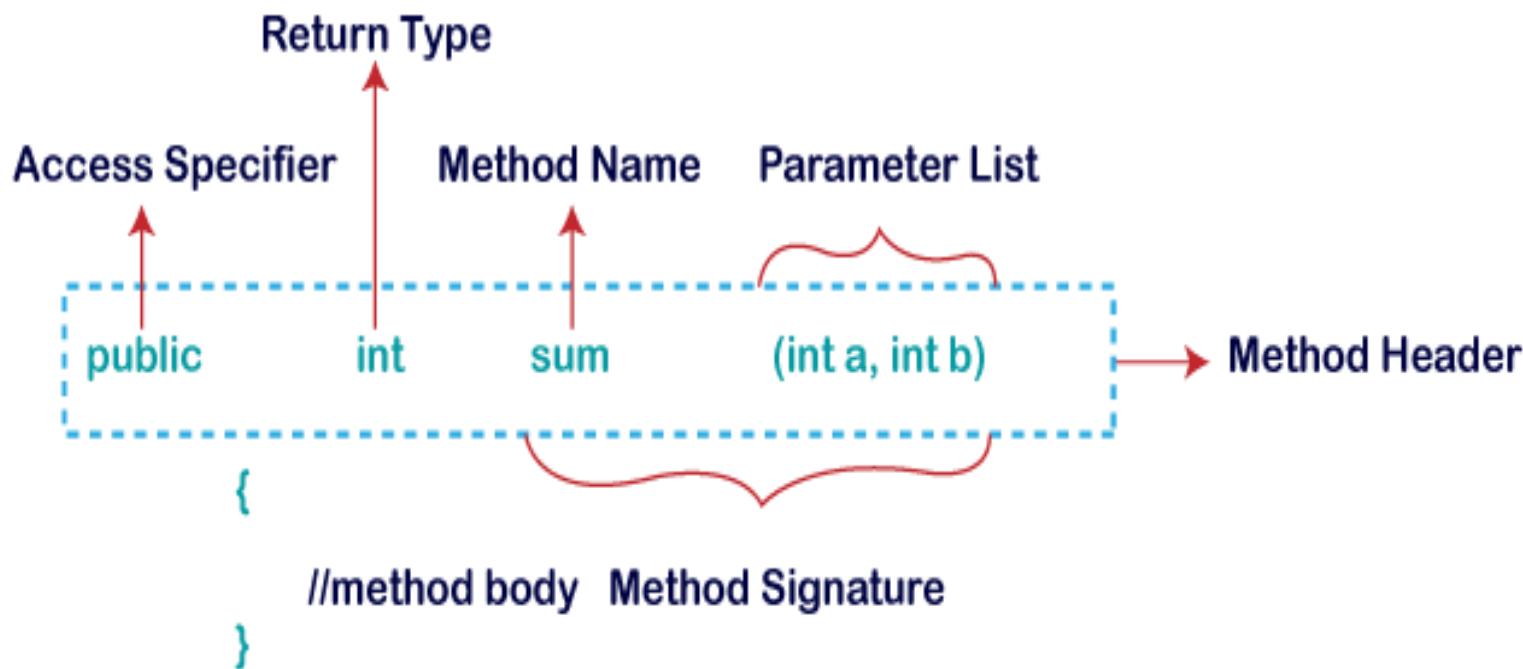
- A Java method is a collection of statements that are grouped together to perform an operation.
- When you call the `System.out.println()` method,
- for example, the system actually executes several statements in order to display a message on the console.
- Now you will learn how to create your own methods with or without return values, invoke a method with or without parameters, and apply method abstraction in the program design.

# Introducing methods

- Syntax:

```
type name (parameter_list)  
{  
    //body of method  
}
```

# Method Declaration



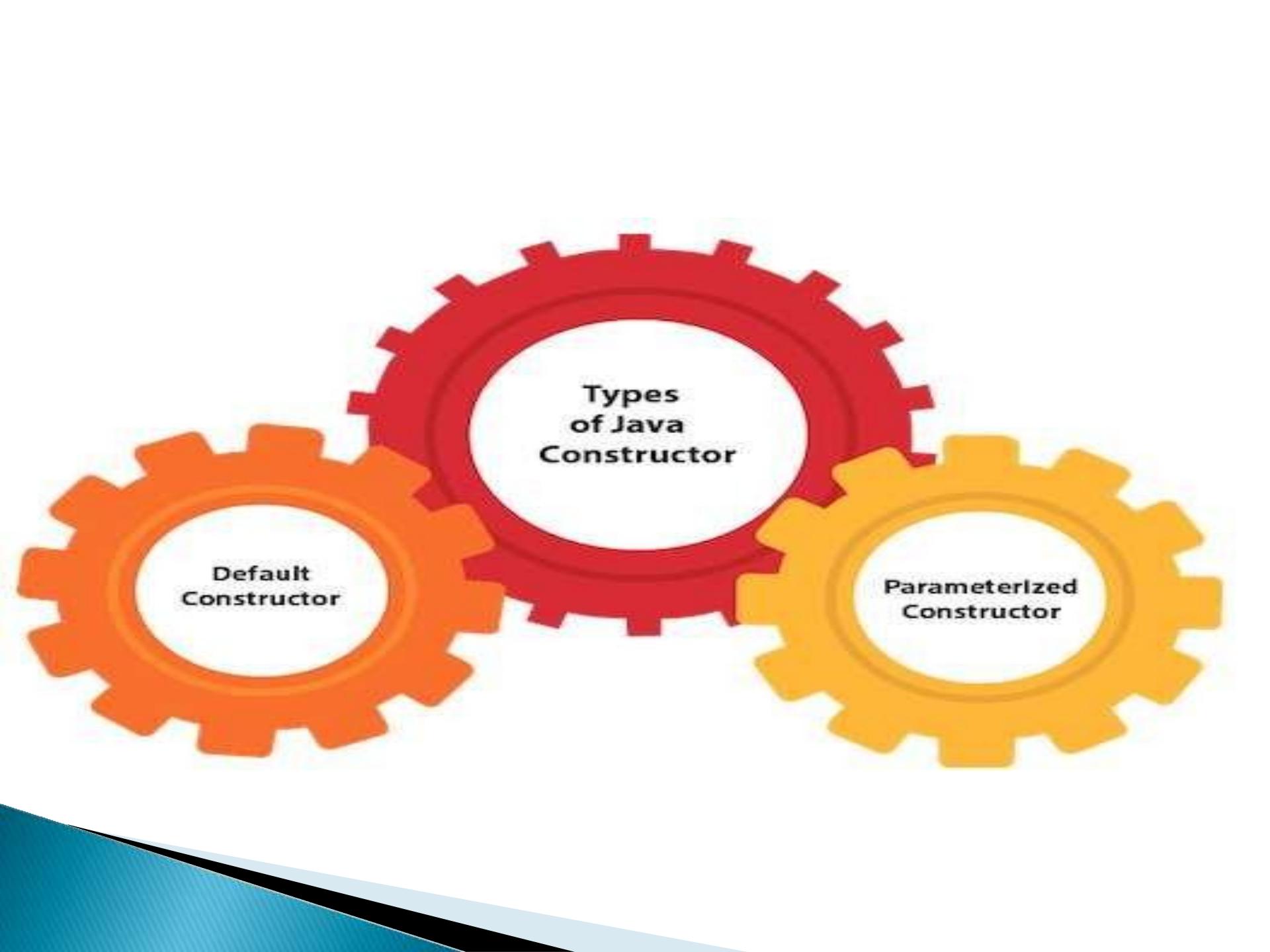
# Access Specifier

**Access Specifier:** Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides **four** types of access specifier:

- **Public:** The method is accessible by all classes when we use public specifier in our application.
- **Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.
- **Protected:** When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
- **Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

# constructors

- In Java, a constructor is a block of codes similar to the method.
- It is called when an instance of the class is created.
- At the time of calling constructor, memory for the object is allocated in the memory.
- It is a special type of method which is used to initialize the object.
- Every time an object is created using the new() keyword, at least one constructor is called.



## Types of Java Constructor

Default Constructor

Parameterized Constructor

# constructors

## Rules for creating Java constructor

- There are two rules defined for the constructor.
- Constructor name must be the same as its class name
- A Constructor must have no explicit return type
- A Java constructor cannot be abstract, static, final, and synchronized

# constructors

## Types of Java constructors

There are two types of constructors in Java:

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

### Java Default Constructor:

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

### Syntax of default constructor:

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

# Default constructor

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

# Default constructor

## Compilation and execution

```
javac Hello.java  
java Hello
```

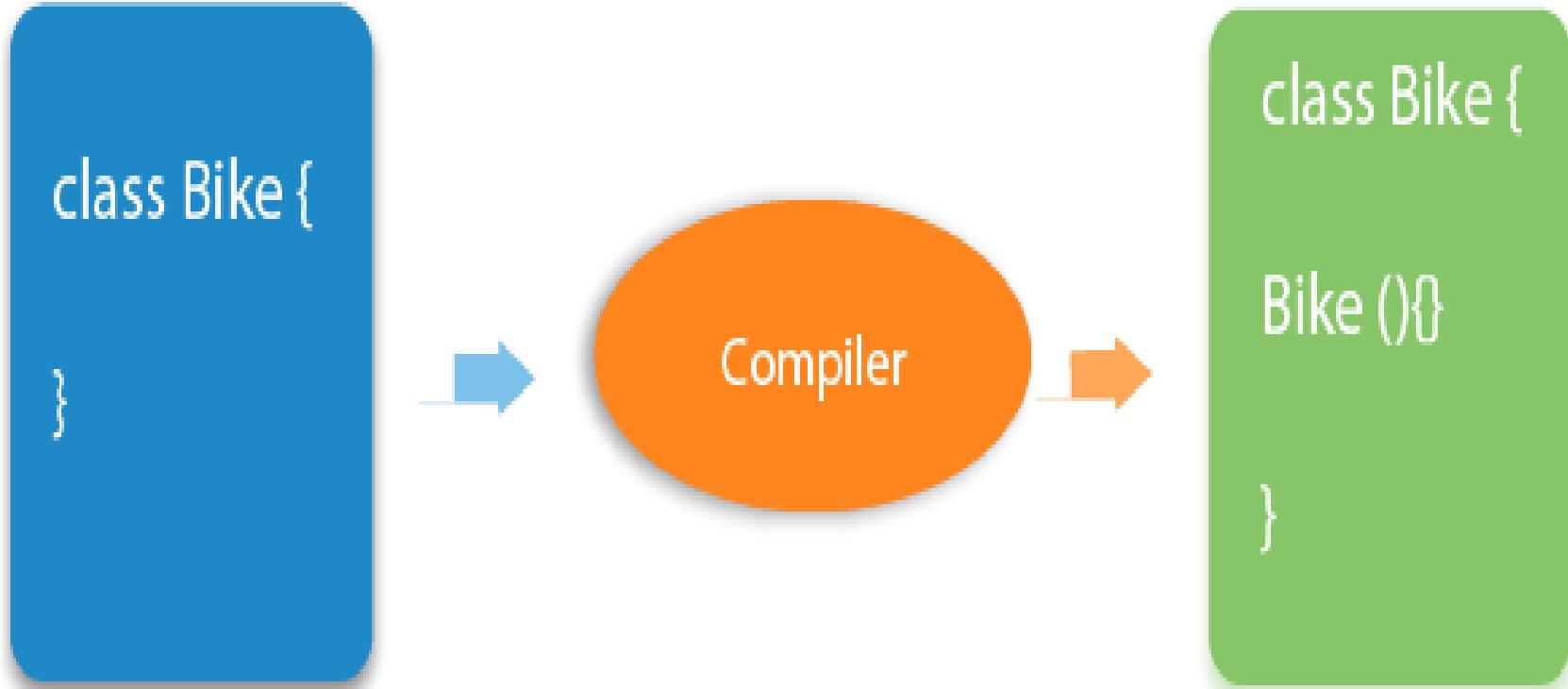
  

```
javac Hello.java  
HelloWorld
```

```
javac Hello.java  
HelloWorld
```

# constructors



# constructors

## Parameterized Constructor:

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

## Why use the parameterized constructor?

- The parameterized constructor is used to provide different values to distinct objects.

# Parameterized constructor

```
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();
    }
}
```

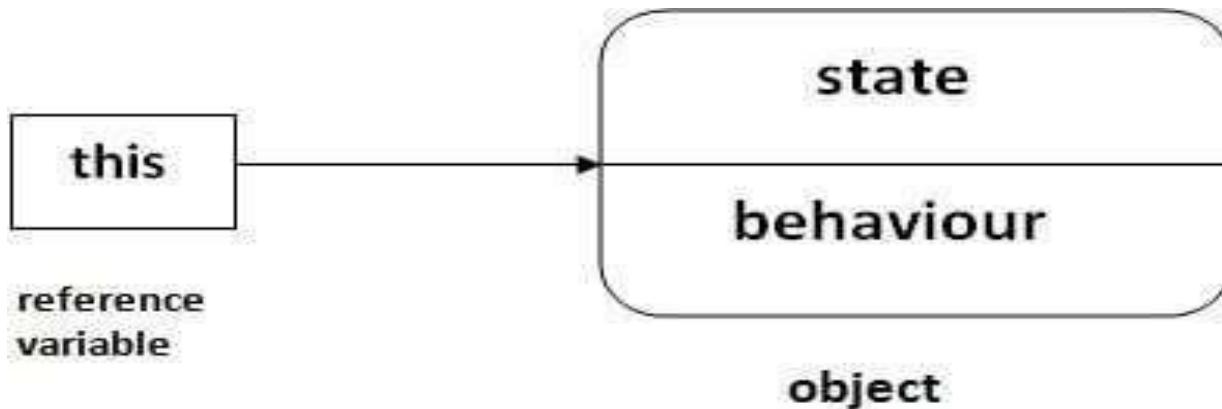
# Parameterized constructor

## □ Compilation and execution:

```
D:\java>java Student4  
111 Karan  
222 Aryan  
D:\java>
```

# Using this keyword

- There can be a lot of usage of java **this keyword**. In java, this is a **reference variable** that refers to the current object.



# This keyword

## Usage of java this keyword

- this can be used to refer current class instance variable.
- this can be used to invoke current class method (implicitly)
- this() can be used to invoke current class constructor.
- this can be passed as an argument in the method call.
- this can be passed as argument in the constructor call.
- this can be used to return the current class instance from the method.

# Garbage collection

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

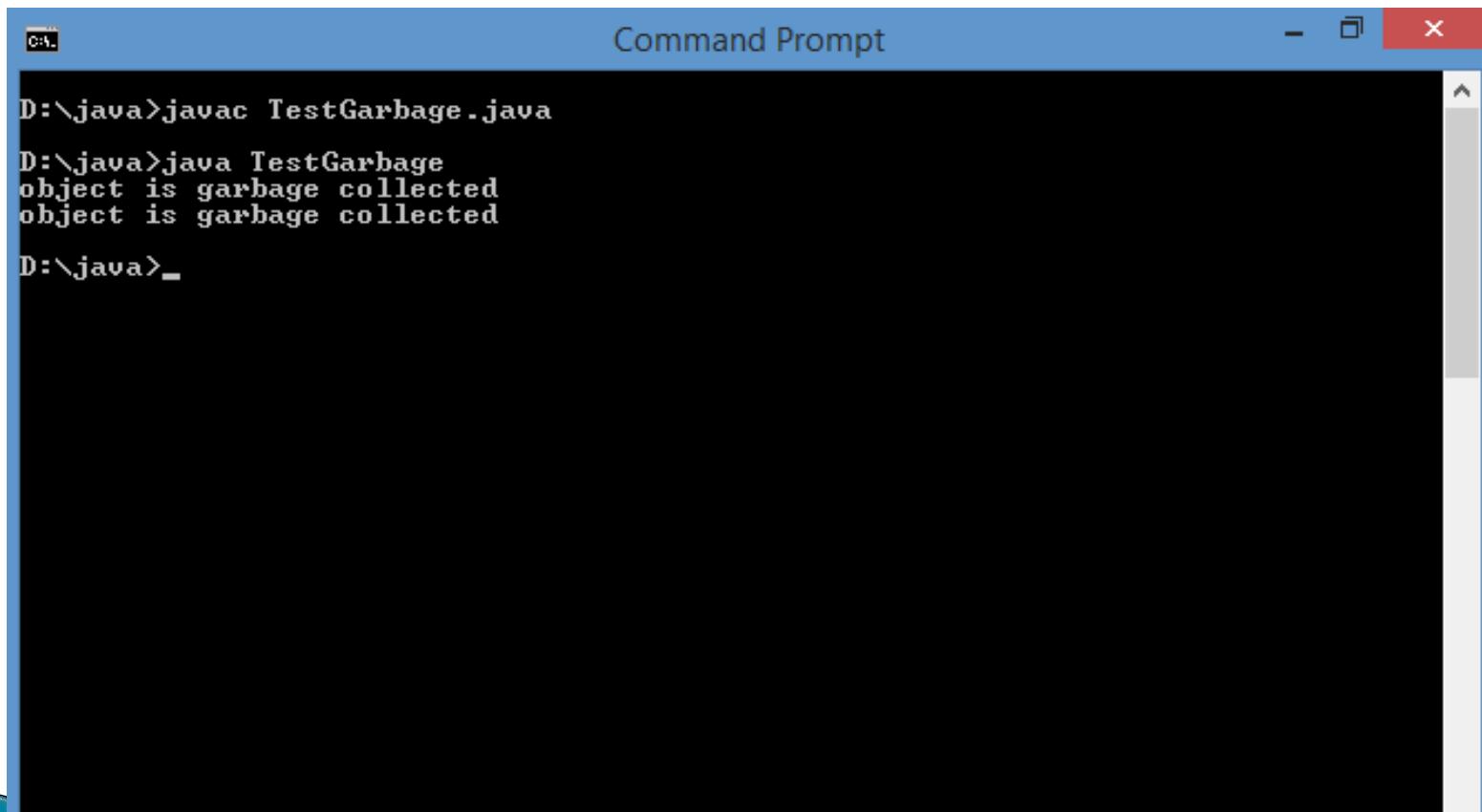
# Garbage collection

## Advantage of Garbage Collection

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

# Garbage collection

## Compilation and execution:



```
D:\>javac TestGarbage.java
D:\>java TestGarbage
object is garbage collected
object is garbage collected
D:\>_
```

# Final keyword

- The final keyword is a non-access modifier used for classes, attributes and methods, which makes them non-changeable (impossible to inherit or override).
- The final keyword is useful when you want a variable to always store the same value, like PI (3.14159...).
- The final keyword is called a "modifier".
- This can be used 3 ways:
  1. Final variable
  2. Final method
  3. Final class

# Final variable

## 1. Java final variable

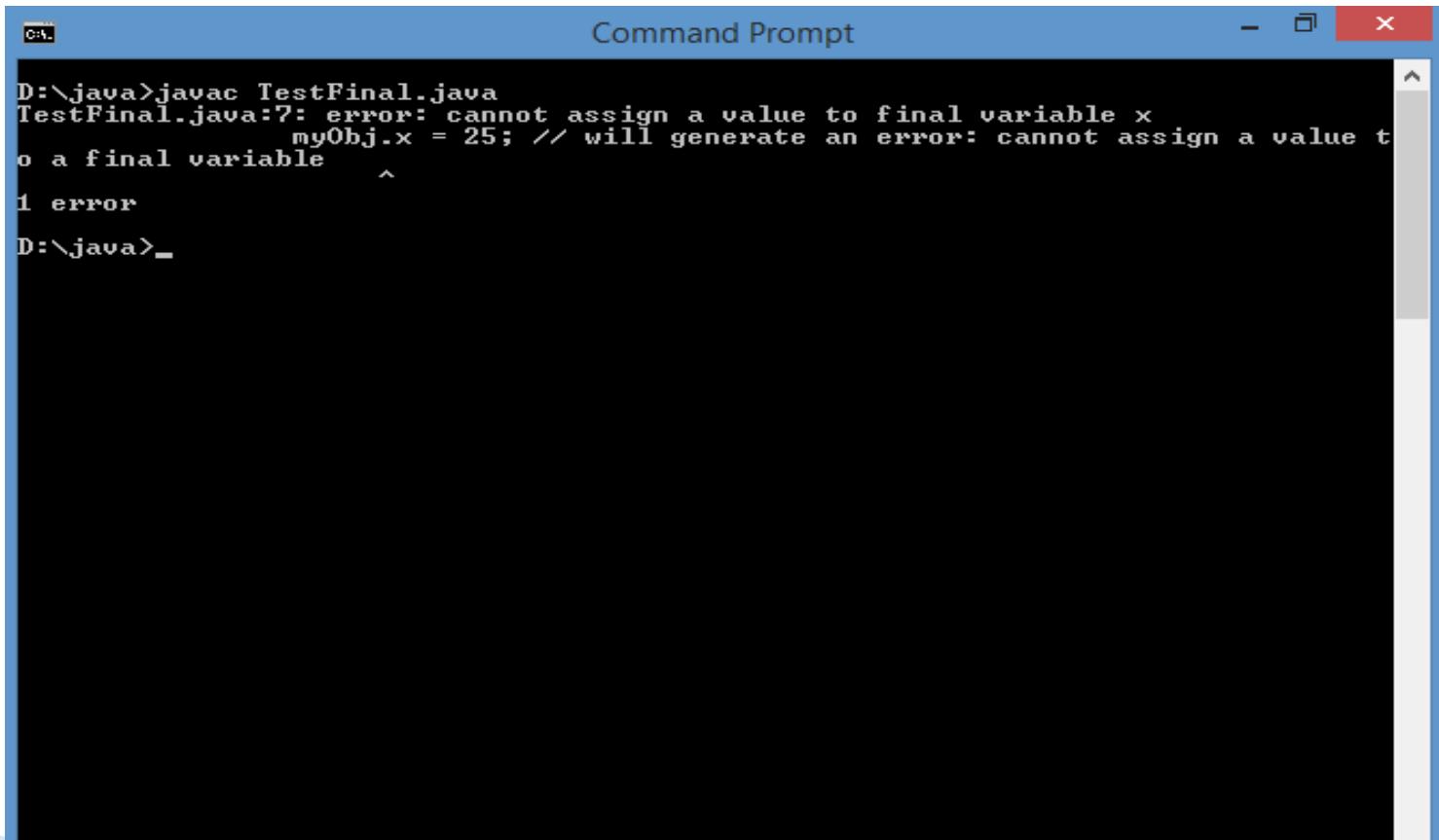
If you make any variable as final, you cannot change the value of final variable(It will be constant).

### Example:

```
public class TestFinal
{
    final int x = 10;
    public static void main(String[] args)
    {
        TestFinal myObj = new TestFinal();
        myObj.x = 25; // will generate an error: cannot assign a value to a final
variable
        System.out.println(myObj.x);
    }
}
```

# Final keyword

## □ Compile and execution



```
D:\>javac TestFinal.java
TestFinal.java:7: error: cannot assign a value to final variable x
        myObj.x = 25; // will generate an error: cannot assign a value t
o a final variable      ^
1 error
D:\>
```

# Final method

## 2. Java final method

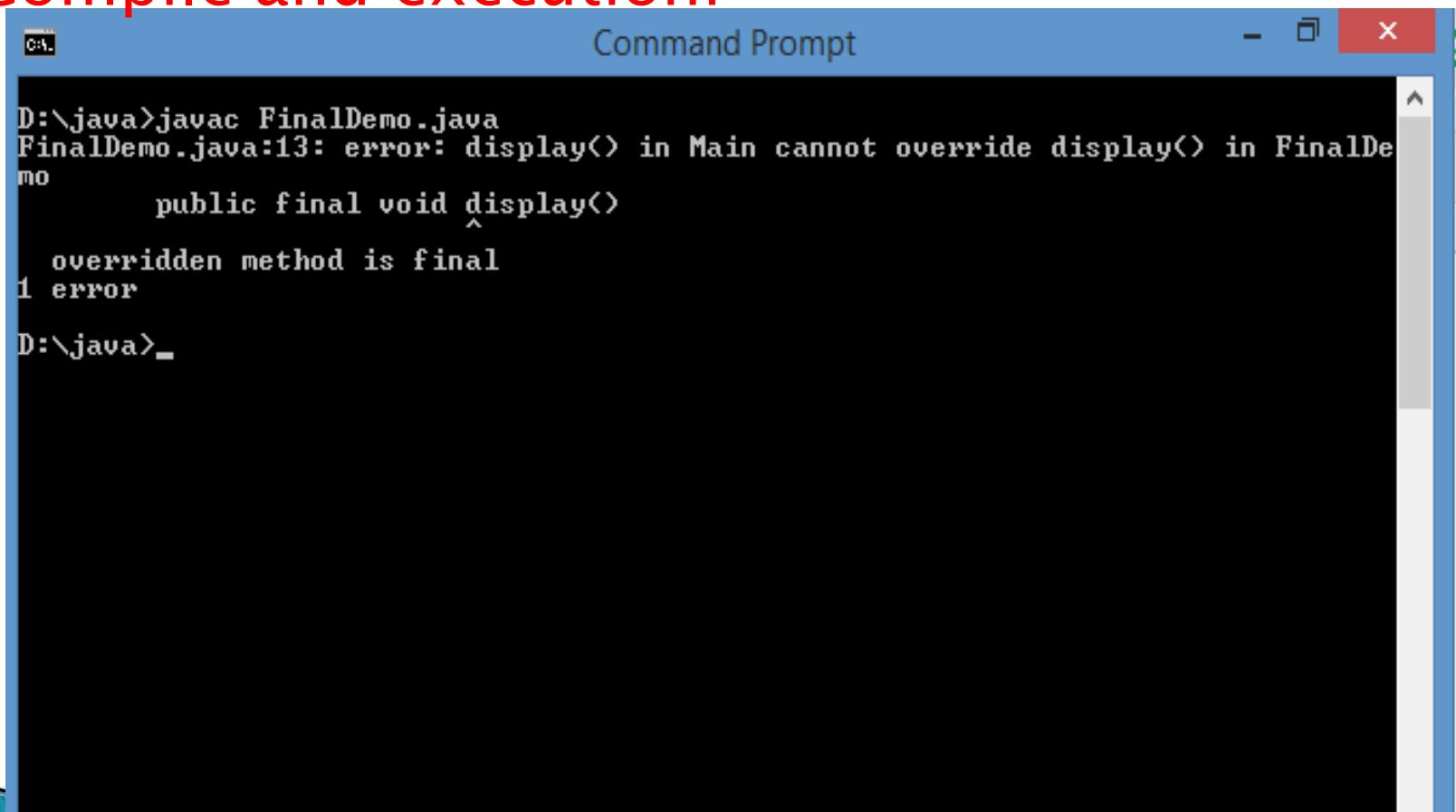
If you make any method as final, you cannot override it.

```
class FinalDemo
{
    // create a final method
    public final void display()
    {
        System.out.println("This is a final method.");
    }
}
```

```
class Main extends FinalDemo
{
    // try to override final method
    public final void display()
    {
        System.out.println("The final method is overridden.");
    }
    public static void main(String[] args)
    {
        Main obj = new Main();
        obj.display();
    }
}
```

# Final method

Compile and execution:



```
D:\java>javac FinalDemo.java
FinalDemo.java:13: error: display() in Main cannot override display() in FinalDe
mo
        public final void display()
                           ^
  overridden method is final
1 error

D:\java>_
```

# Final class

## 3. Java Final class:

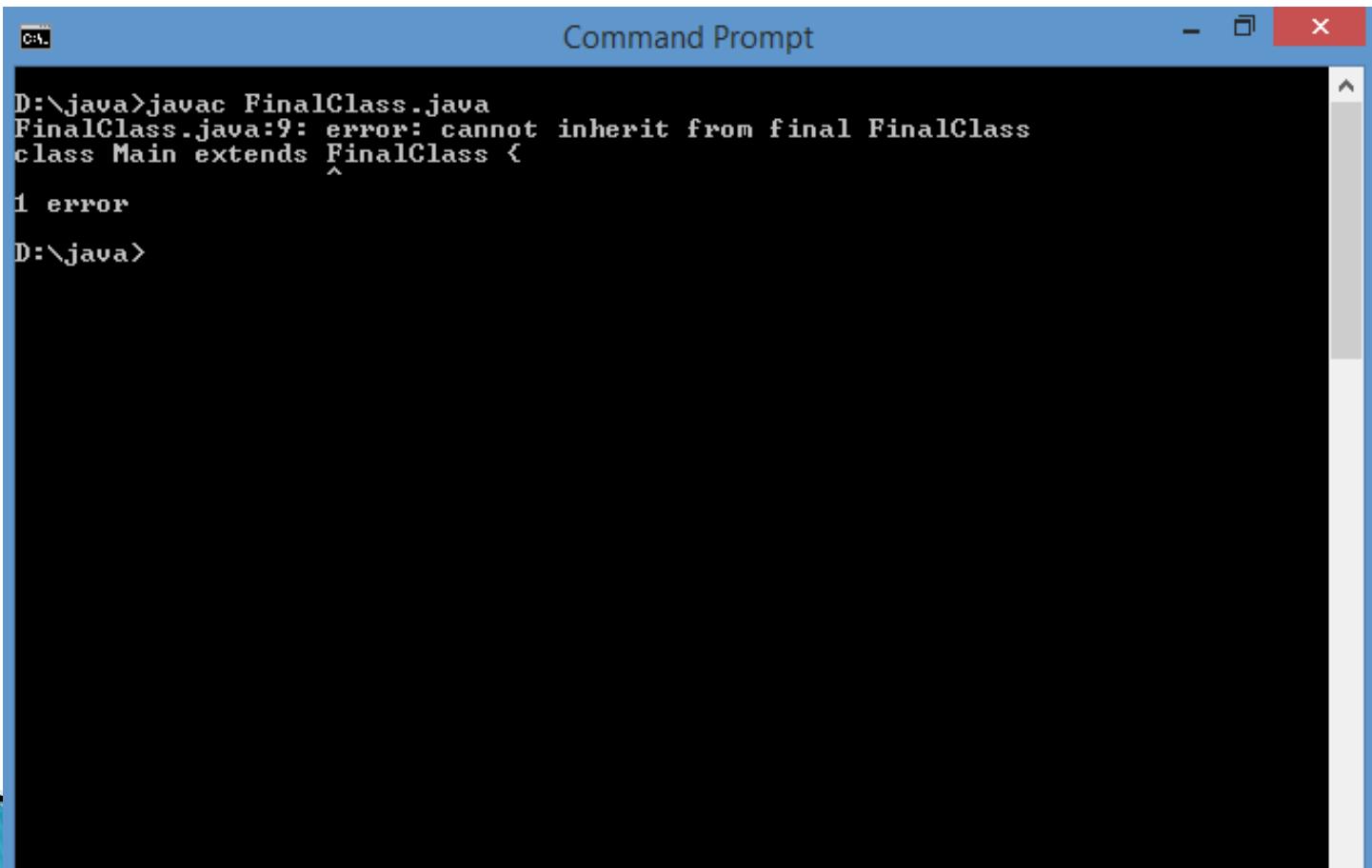
- In Java, the final class cannot be inherited by another class.

```
// create a final class
final class FinalClass
{
    public void display()
    {
        System.out.println("This is a final method.");
    }
}
```

```
// try to extend the final class
class Main extends FinalClass
{
    public void display()
    {
        System.out.println("The final method is overridden.");
    }
    public static void main(String[] args)
    {
        Main obj = new Main();
        obj.display();
    }
}
```

# Final class

Compilation and execution:



```
Command Prompt -> X

D:\java>javac FinalClass.java
FinalClass.java:9: error: cannot inherit from final FinalClass
class Main extends FinalClass {
                           ^
1 error
D:\java>
```

# Inheritance

- Inheritance is one of the object oriented feature in java

In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance

Or

- It is the mechanism in java by which one class is allowed to inherit the features(fields and methods) of another class.

concept" into two categories:

- **subclass** (child) - the class that inherits from another class
- **superclass** (parent) - the class being inherited from

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

## About inheritance

# Inheritance

Syntax:

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

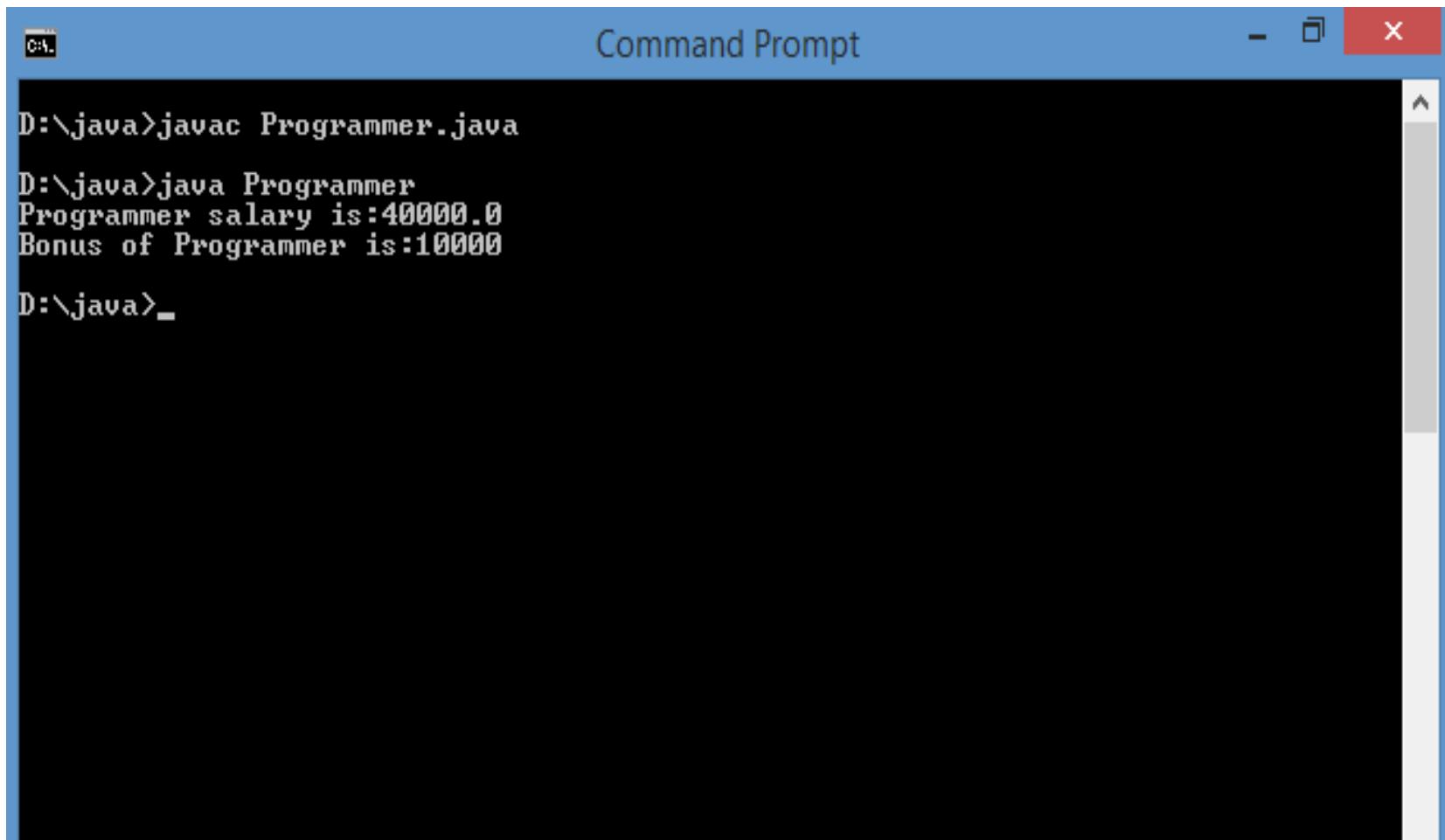
# Inheritance Example

```
class Employee
{
    float salary=40000;
}

class Programmer extends Employee
{
    int bonus=10000;
    public static void main(String args[])
    {
        Programmer p=new Programmer();
        System.out.println("Programmer salary is:"+p.salary);
        System.out.println("Bonus of Programmer is:"+p.bonus);
    }
}
```

# Inheritance

## □ Compilation and execution:



D:\java>javac Programmer.java  
D:\java>java Programmer  
Programmer salary is:40000.0  
Bonus of Programmer is:10000  
D:\java>\_

# inheritance

## Type of inheritance:

1. Single Inheritance
2. Multi level Inheritance
3. Hierarchical Inheritance
4. **Single Inheritance:** In single inheritance, subclasses inherit the features of one superclass. In the image below, class A serves as a base class for the derived class B.

# Single Inheritance

```
class Animal
{
    void eat()
    {
        System.out.println("eating...");
    }
}

class Dog extends Animal
{
    void bark()
    {
        System.out.println("barking...");
    }
}

class TestInheritance
{
    public static void main(String args[])
    {
        Dog d=new Dog();
        d.bark();
        d.eat();
    }
}
```

# Single Inheritance



Command Prompt



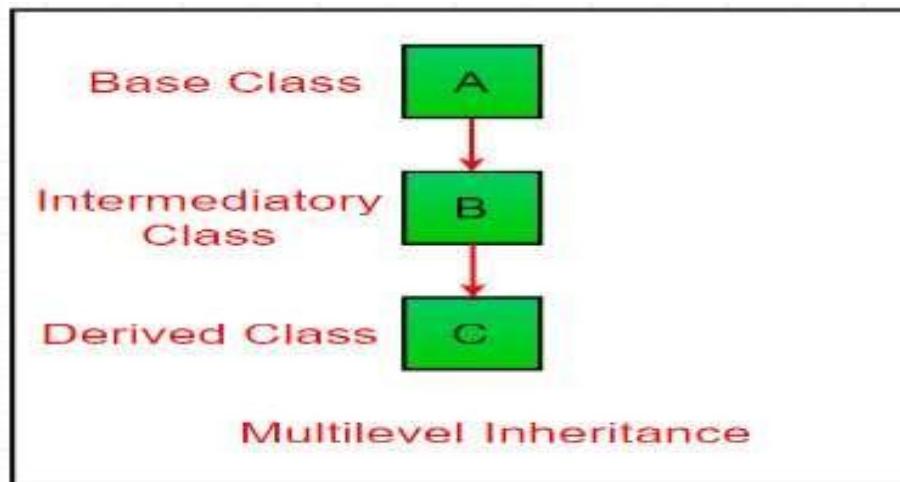
```
D:\java>javac TestInheritance.java
```

```
D:\java>java TestInheritance
barking...
eating...
```

```
D:\java>
```

# Multilevel Inheritance

- When there is a chain of inheritance, it is known as multilevel inheritance.
- As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

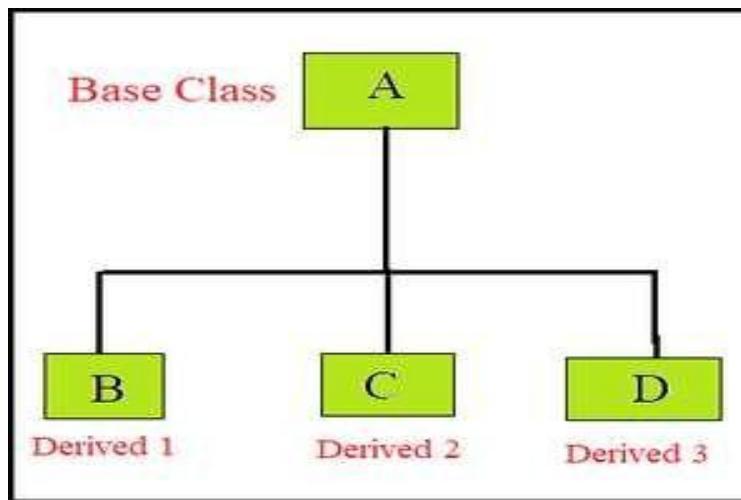


# Multi Level Inheritance

```
C:\> Command Prompt
D:\java>javac TestMulti.java
D:\java>java TestMulti
weeping...
barking...
eating...
D:\java>
```

# Hierarchical Inheritance

- In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived class B, C and D



# Hierarchical Inheritance

D:\>javac TestHierachial.java  
D:\>java TestHierachial  
meowing...  
eating...  
D:\>

# Method Overriding

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

Or

In the class hierarchy when a method in subclass has the same return type and method name as in superclass(parent class), then method in the subclass is said to override the method in the superclass

# Method Overriding

## Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Or

- If two or more methods has same name and parameters list and located in inherited class is called method overriding

## Rules for Java Method Overriding

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

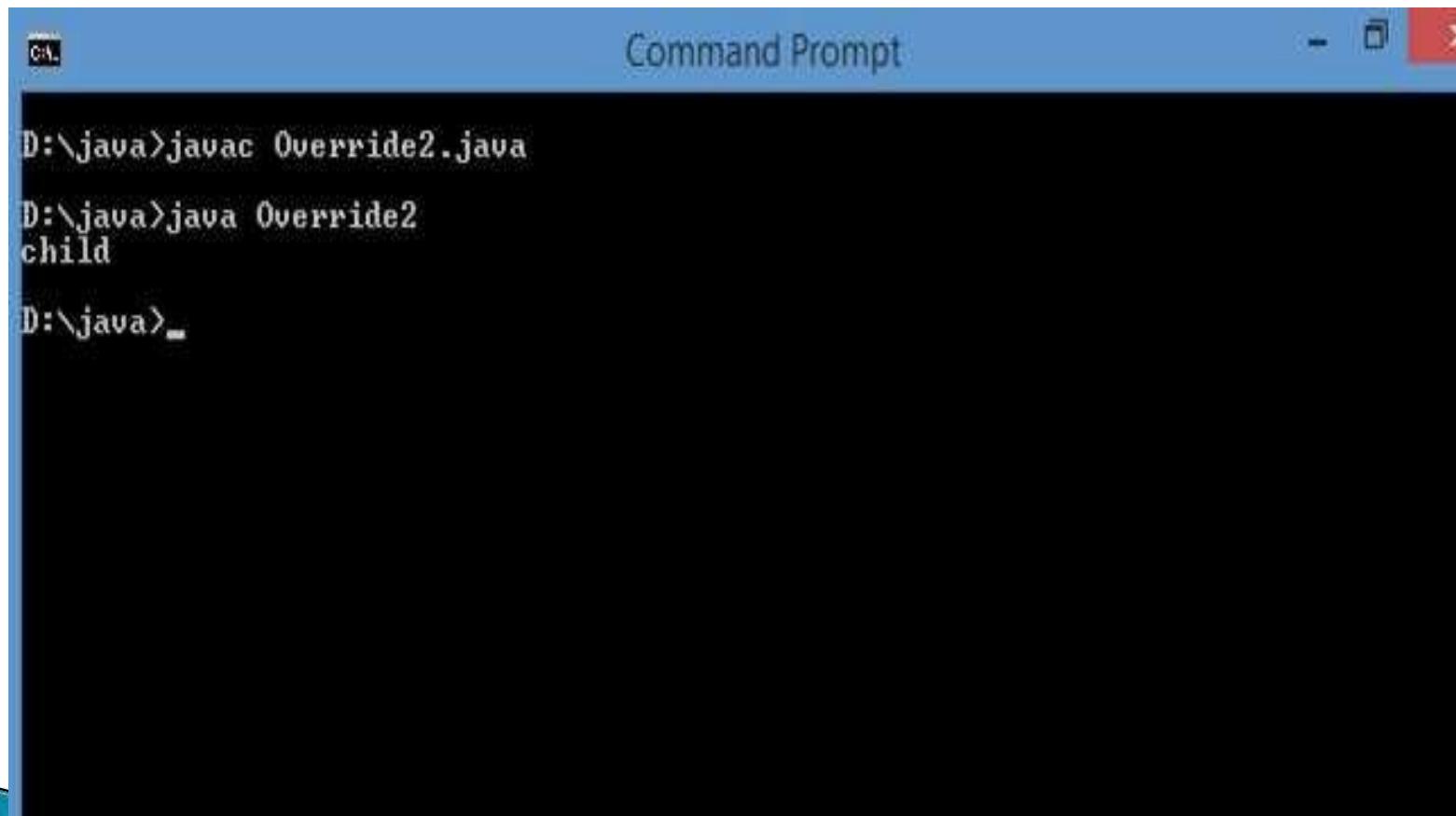
```
class Parent
{
    void display()
    {
        System.out.println("parent");
    }
}

class Child extends Parent
{
    void display()
    {
        System.out.println("child");
    }
}

class Override2
{
    public static void main(String args[])
    {
        Child c=new Child();
        c.display();
    }
}
```

# Method overriding

Compilation and execution process:



```
Command Prompt -> D:\java>javac Override2.java  
D:\java>java Override2  
child  
D:\java>_
```

The image shows a screenshot of a Windows Command Prompt window. The title bar reads "Command Prompt". The window contains the following text:  
D:\java>javac Override2.java  
D:\java>java Override2  
child  
D:\java>\_

# Method overloading

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

Different ways to overload method:

1. By changing member of arguments.
2. By changing the data type.

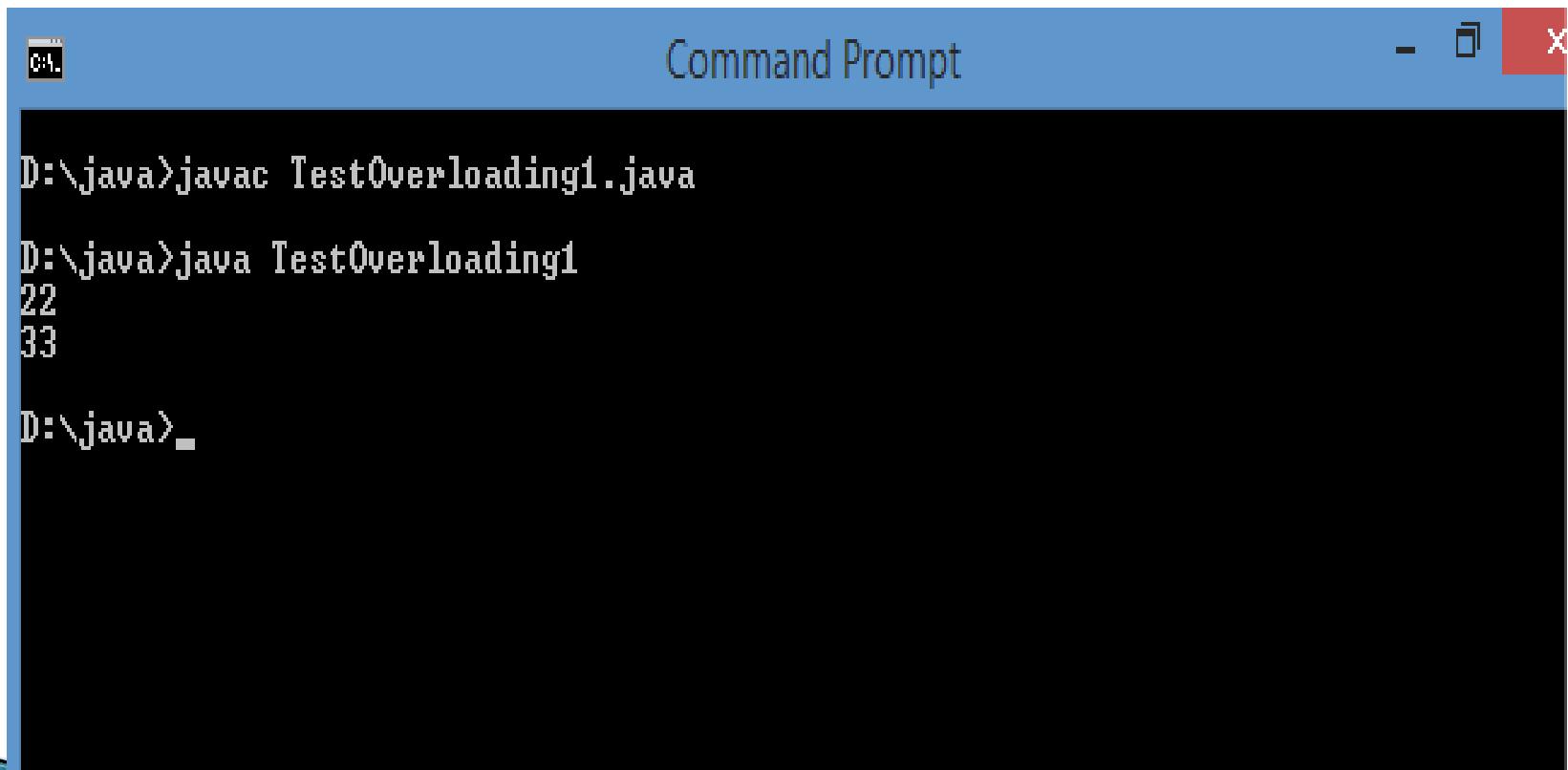
But in java method overloading is not possible by changing the return type of the method only.

# Method overloading

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

# Method overloading

## Compilation and Execution process:



The screenshot shows a Windows Command Prompt window with the title "Command Prompt". The window contains the following text:

```
D:\java>javac TestOverloading1.java
D:\java>java TestOverloading1
22
33
D:\java>_
```

# Abstract classes

- Abstraction is a process of hiding the implementation details from the user. Only the functionality will be provided to the user
- or
- A class which is declared with the abstract keyword is known as an abstract class in Java.
- It needs to be extended and its method implemented. It cannot be instantiated.

# Abstract classes

- There are two ways to achieve abstraction in java
  1. Abstract class ( 0 to 100%)
  2. Interface (100%)
- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

**Syntax::**

**abstract class A**

```
{  
}
```

**Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

**Syntax:**

**abstract void printStatus();**//no method body and abstract

## Rules for Java Abstract class



- 1** An abstract class must be declared with an abstract keyword.
- 2** It can have abstract and non-abstract methods.
- 3** It cannot be instantiated.
- 4** It can have final methods
- 5** It can have constructors and static methods also.

```
abstract class Example1
{
    abstract void print();
}

class Example2 extends Example1
{
    void print()
    {
        System.out.println("Abstract method body in
example");
    }

    public static void main(String args[])
    {
        Example2 e=new Example2();
        e.print();
    }
}
```

# Abstract classes

Compilation and execution process:



The screenshot shows a Windows Command Prompt window with the title "Command Prompt". The window contains the following text:

```
D:\java>javac Example2.java
D:\java>java Example2
Abstract method body in example
D:\java>
```

# Interfaces and packages

## Interfaces:

- An interface is a collection of abstract methods
- An interface can contain both variable and methods
- An interface cannot be instantiated
- In order to access the members of interface we need to inherit the interface into a class using **implements** keyword.
- In the sub class we have to override all the abstract methods
- If the sub class is overriding all the methods of an interface. Then it is called as implementation class
- A class can implement any number of interfaces

# Interfaces and packages

## Why use Java interface?

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

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

# Interfaces and packages

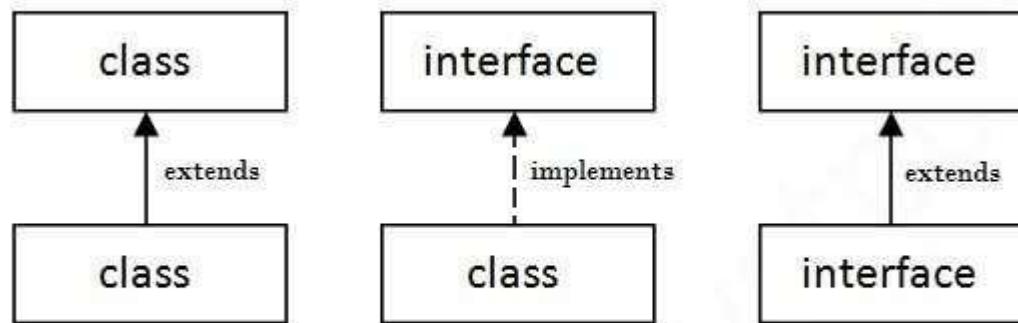
Syntax:

**interface** <interface\_name>{

// declare constant fields  
// declare methods that abstract  
// by default.

}

# Interfaces and packages



# Interfaces and packages

## Example:

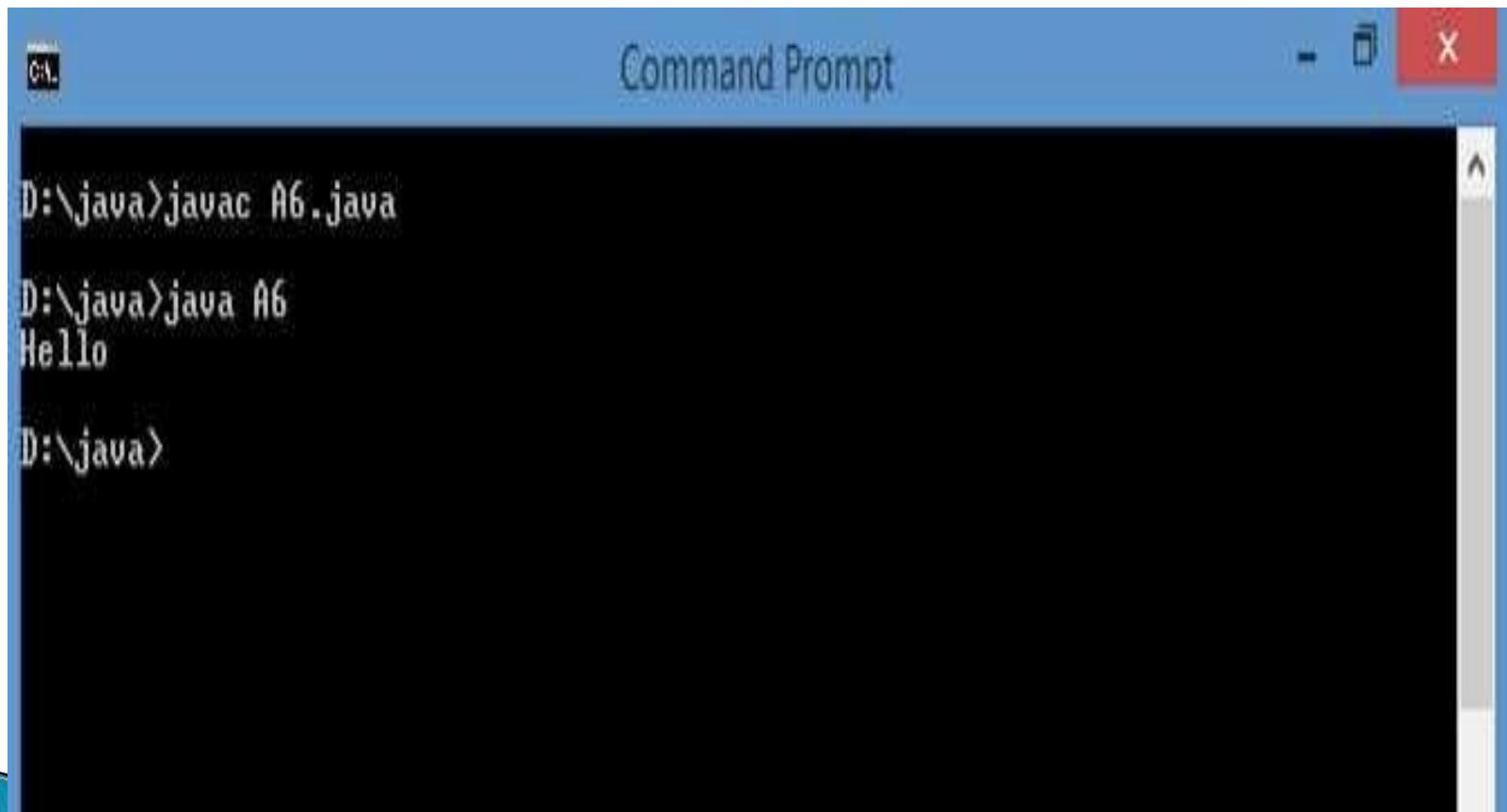
```
interface printable
{
    void print();
}

class A6 implements printable
{
    public void print()
    {
        System.out.println("Hello");
    }

    public static void main(String args[])
    {
        A6 obj = new A6();
        obj.print();
    }
}
```

# Interfaces and packages

Compilation and execution:



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The window has a blue header bar with the title and standard window controls (minimize, maximize, close). The main body of the window is black and contains the following text:

```
D:\java>javac A6.java
D:\java>java A6
Hello
D:\java>
```

# Interfaces and packages

## Packages:

packages is a collection of class that helps you to organize your classes into a folder structure and make it easy to locate and use them.

or

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

Package in java can be categorized in two form,

1. built-in package and
2. user-defined package.

# Interfaces and packages

## 1. Built-in package:

- These packages consists of large number of classes which are a part of java API(Application Program Interface).
- Some of the commonly used built-in packages are:
  - Java.lang
  - java.util
  - Java.io.\*
  - Java.awt
  - Java.net

Ex:

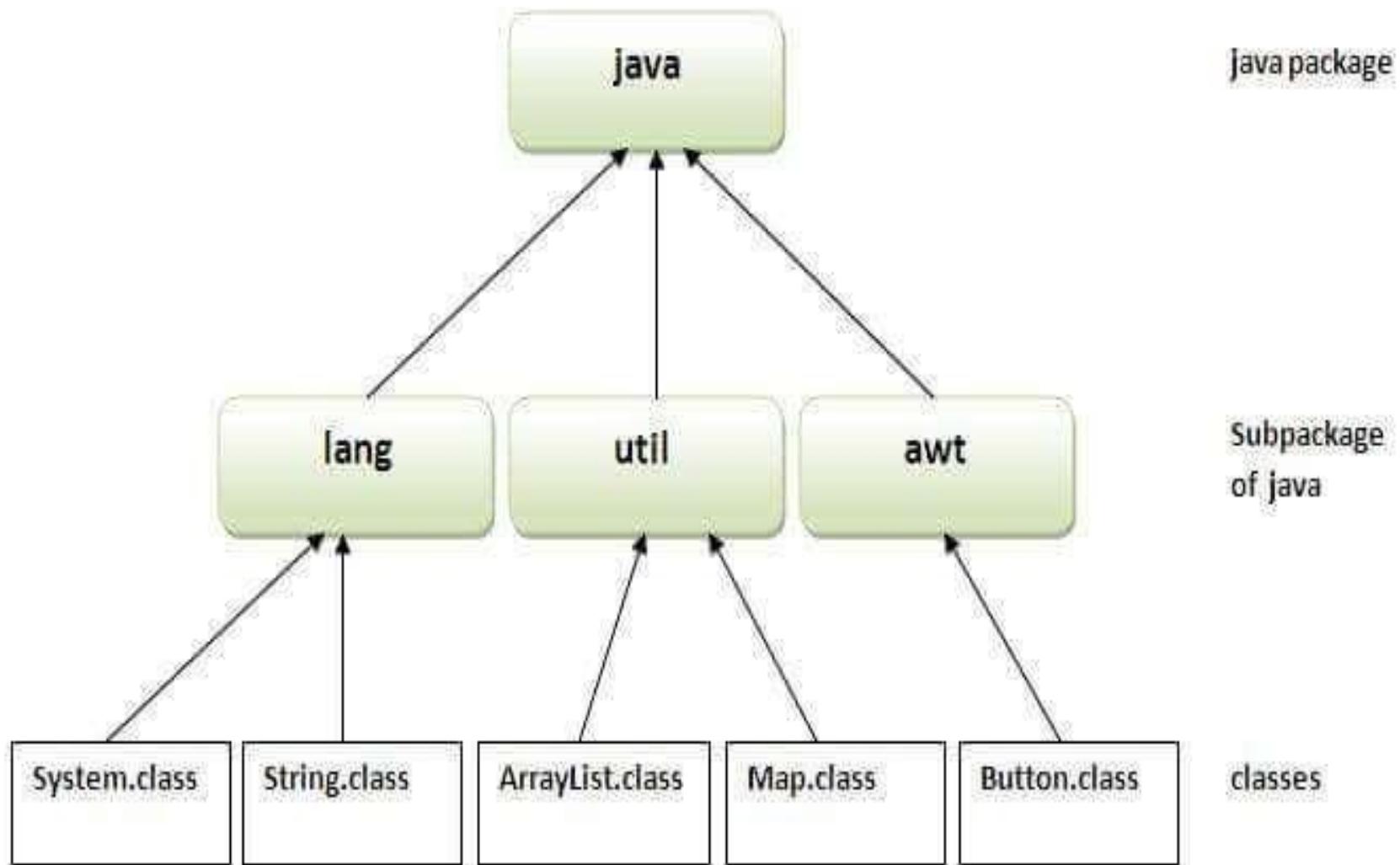
```
import.java.util.vector;
```

# Interfaces and packages

## 2. Package:

- These are the packages that are defined by the user.
- First we will create a directory **mypackage** then create the **myclass** inside the directory with the first statement being the package name

Package mypackage;



# Interfaces and packages

## Advantages of packages:

- Naming conflicts: unique identification of components
- Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- Java package provides access protection.
- Provides Security

# The class member access

	Private	No modifier	Protected	public
<b>Same class</b>	Yes	Yes	Yes	yes
<b>Same package subclass</b>	No	Yes	Yes	yes
<b>Same package-non subclass</b>	No	Yes	Yes	yes
<b>Different package subclass</b>	No	No	Yes	yes
<b>Different package- non subclass</b>	No	No	No	yes

# Interfaces and packages

## Example 1:

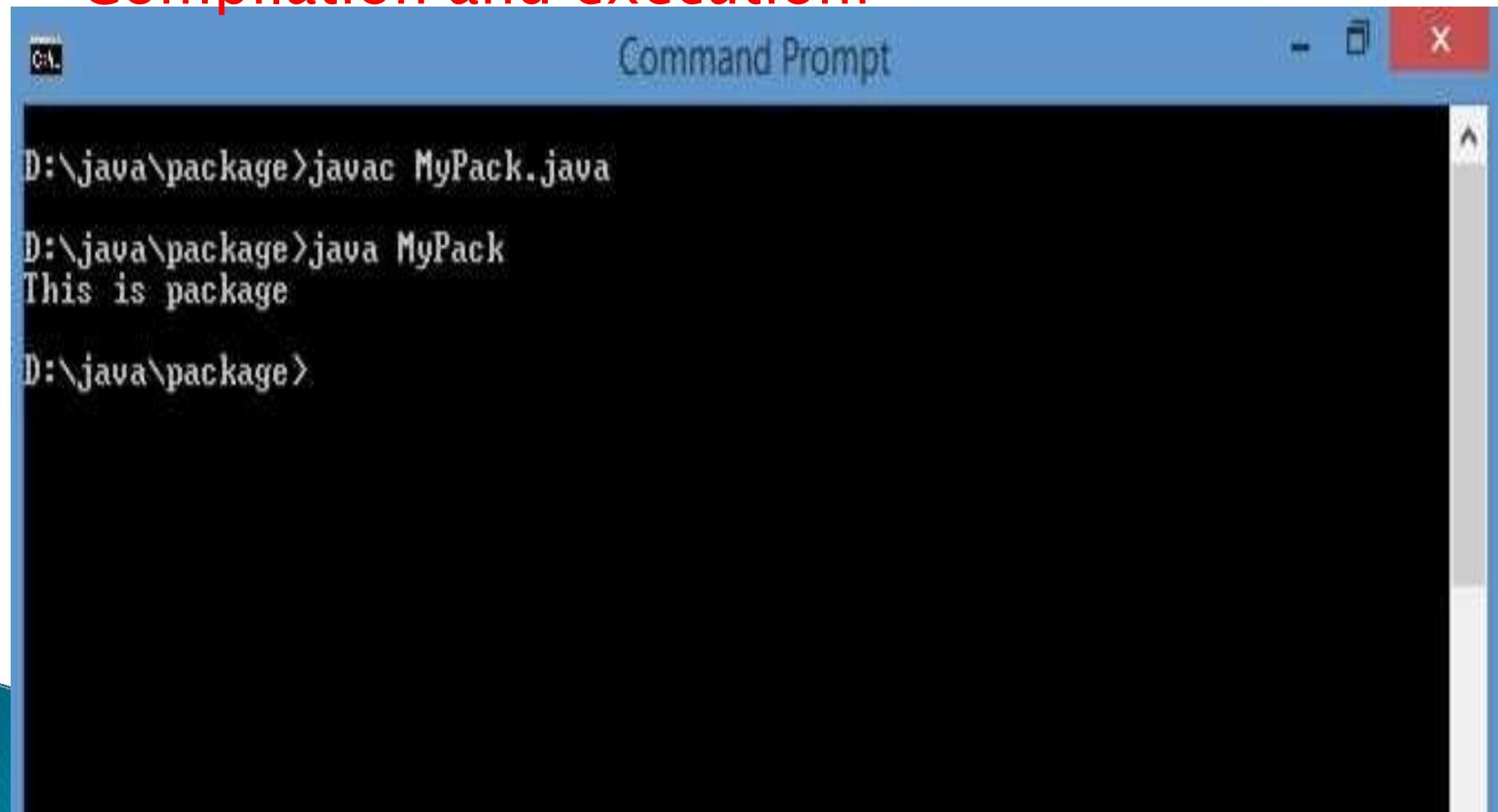
```
// save with P1.java
package pack;
public class P1
{
    public void show()
    {
        System.out.println("This is
package");
    }
}
```

# Interfaces and packages

```
// save with MyPack.java
import pack.*;
class MyPack
{
    public static void main(String args[])
    {
        P1 p=new P1();
        p.show();
    }
}
```

# Interfaces and packages

Compilation and execution:



D:\java\package>javac MyPack.java

D:\java\package>java MyPack

This is package

D:\java\package>

A screenshot of a Windows Command Prompt window titled "Command Prompt". The window has a blue header bar with the title and standard window controls (minimize, maximize, close). The main body of the window is black and contains white text representing a Java application's output. It shows the command "javac MyPack.java" being run, followed by the output of the "java MyPack" command, which prints the string "This is package". Finally, it shows the command "java" again, indicating the application is still running.

# **Unit - 3**

Exception Handling - Exception Fundamentals - Exception Types -Uncaught Exceptions - Using try and catch - Nested try Statements -throw -throws –finally. Multithreaded Programming - The Java Thread Model -Thread Priorities - The Thread Class and the Runnable Interface - Creating Multiple Threads -Using is Alive( ) and join( ) – Thread Priorities -Synchronization-String Handling.

# Exception Fundamentals

The **Exception Handling in Java** is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained

Or

Exception is abnormal condition that arises in a code sequences at run-time

# Exception Fundamentals

## Error vs Exception:

**Error:** An Error indicates serious problem that a reasonable application should not try to catch.

**Exception:** Exception indicates conditions that a reasonable application might try to catch.

# Exception Fundamentals

An exception is an abnormal condition that arises in a code sequence at run-time.

Exceptions are generated by Java Run-time System or they can be manually generated by your code.

Java Exception handling is managed via 5 keywords:

- Try
- Catch
- Throw
- Throws
- finally

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.

throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.
--------	---

The checked exception ACheckedException can be thrown from the body of the method foo().

The code inside try block can throw exceptions.

If the code in try block throws an exception of type Exception or its derived classes, this catch block code will handle it.

The code in finally block will always be executed (doesn't matter if the try block throw an exception or not).

This throw statements throws the custom checked exception (and since there is no catch handler for this exception, it must be declared in throws clause of the method).

```
public static void foo() throws ACheckedException {  
    try {  
        // some code that can throw an exception ...  
    } catch (Exception e){  
        // handle the exception  
    }  
    finally {  
        // release resources acquired in the try block  
    }  
    if(someCondition) {  
        throw new ACheckedException();  
    } else {  
        throw new AnUnCheckedException();  
    }  
}
```

This throw statement throws AnUnCheckedException (since this is an unchecked exception it is not declared in the throws clause).

# Exception Types

There are mainly two types of exceptions:  
checked and  
unchecked.

Here, an error is considered as the unchecked exception.  
According to Oracle, there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

## 1. Checked Exception

The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

# Exception types

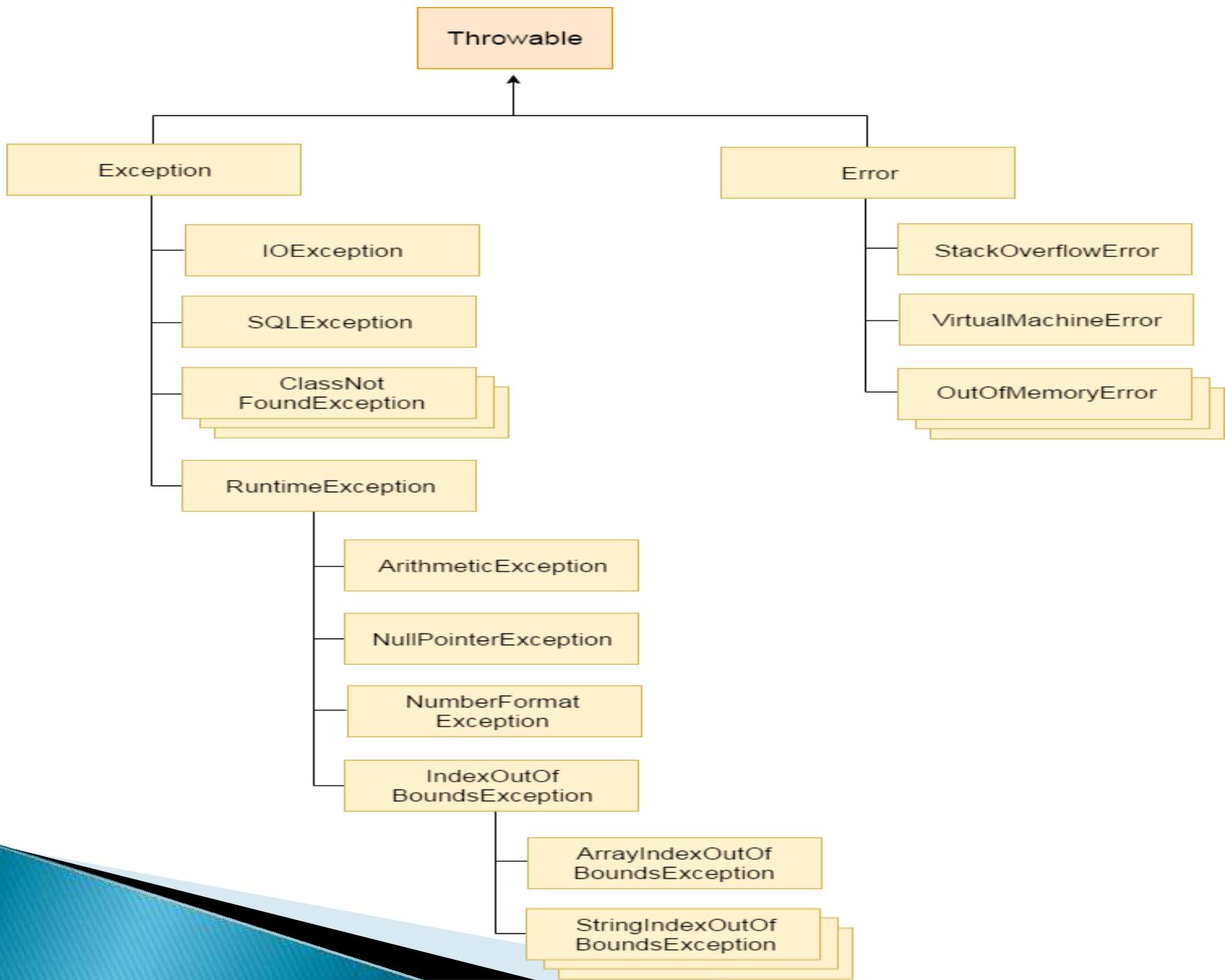
## 2. Unchecked Exception

The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

## 3. Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.





# Example Program

```
public class Exception Test
{
    public static void main(String args[])
    {
        try
        {
            //code that may raise exception
            int data=100/0;
        }
        catch(ArithmaticException e)
        {
            System.out.println(e);
        }
        //rest code of the program
        System.out.println("rest of the code...");
    }
}
```

# Uncaught Exception

Before going to see how we can handle exceptions we need to know- what happened when exception is not handled.

```
class UncaughtExample
```

```
{
```

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

```
{
```

```
        int d=0;
```

```
        int a=42/d;
```

```
}
```

```
}
```

# Using try and catch

## Try block:

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement of try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

# Using try and catch

## Catch block:

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception ( i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

# Using try and catch

Syntax:

**Try**

{

//code that may throw an exception

}

**catch(Exception\_class\_Name ref)**

{

}

```
public class TryCatch2
{
    public static void main(String[] args)
    {
        try
        {
            int data=50/0; //may throw exception
        }
        //handling the exception
        catch(ArithmetricException e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }
}
```

# Nested Try Statements

The try block within a try block is known as nested try block in java.

## Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error.

In such cases, exception handlers have to be nested.

## Syntax:

....

**try**

{

    statement 1;

    statement 2;

**try**

{

    statement 1;

    statement 2;

}

**catch**(Exception e)

{

}

}

**catch**(Exception e)

{

}

....

```
class NestedTry
{
    public static void main(String args[])
    {
        try
        {
            System.out.println("going to divide");
            int b =39/0;
        }
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
        try
        {
            int a[] =new int[5];
            a[5]=4;
        }
        catch(ArrayIndexOutOfBoundsException e)
```

```
{  
    System.out.println(e);  
}  
    System.out.println("other statement");  
}  
catch(Exception e)  
{  
    System.out.println("handled");  
}  
System.out.println("normal flow..");  
}  
}
```

# Throw

The throw keyword is used to create a custom error.

The throw statement is used together with an **exception type**.

There are many exception types available in Java: `ArithmeticException`, `ClassNotFoundExcepcion`, `ArrayIndexOutOfBoundsException`, `SecurityException`, etc.

The exception type is often used together with a custom **method**.

# Throw

## Differences between **throw** and **throws**:

throw	throws
Used to throw an exception for a method	Used to indicate what exception type may be thrown by a method
Cannot throw multiple exceptions	Can declare multiple exceptions
<ul style="list-style-type: none"><li>•<b>Syntax:</b> throw is followed by an object (<i>new type</i>)</li><li>•used inside the method</li></ul>	<ul style="list-style-type: none"><li>•<b>Syntax:</b> throws is followed by a class</li><li>•and used with the method signature</li></ul>

# Throw

## Syntax::

**throw throwableInstance;**

**Example:** **throw new ArithmeticException("/ by zero");**

```
public class Throw1
{
    static void checkAge(int age)
    {
        if (age < 18)
        {
            throw new ArithmeticException("Access denied – You must be
                                         at least 18 years old.");
        }
        else
        {
            System.out.println("Access granted – You are old enough!");
        }
    }

    public static void main(String[] args)
    {
        checkAge(15); // Set age to 15 (which is below 18...)
    }
}
```

# Throws

The Java throws keyword is used to declare an exception.

It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

# Throws

There are many exception types available in Java: `ArithmaticException`, `ClassNotFoundException`, `ArrayIndexOutOfBoundsException`, `SecurityException`, etc.

## Syntax:

```
type method_name(parameter_list) throws  
exception_list  
{  
// body of method  
}
```

# Throws

## **Advantage of Java throws keyword**

Now Checked Exception can be propagated (forwarded in call stack).

It provides information to the caller of the method about the exception.

## Exception Hierarchy:

```
class ThrowsExecp
{
    static void fun() throws IllegalAccessException
    {
        System.out.println("Inside fun(). ");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[])
    {
        try
        {
            fun();
        }
        catch(IllegalAccessException e)
        {
            System.out.println("caught in main.");
        }
    }
}
```

# **Finally**

Java finally block is a block that is used to execute important code such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block.

## **Why use java finally**

Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

```
public class HelloWorld1
{
    public static void main(String []args)
    {
        try
        {
            int data = 25/5;
            System.out.println(data);
        }
        catch(NullPointerException e)
        {
            System.out.println(e);
        }
        finally
        {
            System.out.println("finally block is always
executed");
        }
        System.out.println("rest of the code...");
    }
}
```

}

# Creating Your Own Exception Subclasses

You can create your own exception class by defining a subclass of `Exception`.

The `Exception` class does not define any methods of its own. It inherits methods provided by `Throwable`.

All exceptions have the methods defined by `Throwable` available to them. They are shown in the following list.

**Exception** defines four public constructors. Two support chained exceptions,

## Methods in Throwable class

Method	Description
Throwable fillInStackTrace( )	Returns a Throwable object that contains a completed stack trace.
Throwable getCause( )	Returns the exception that underlies the current exception.
String getLocalizedMessage( )	Returns a localized description.
String getMessage( )	Returns a description of the exception.
StackTraceElement[ ] getStackTrace( )	Returns an array that contains the stack trace.
Throwable initCause(Throwable causeExc)	Associates causeExc with the invoking exception as a cause of the invoking exception.
void printStackTrace()	Displays the stack trace.

## Methods in Throwable class

Method	Description
void printStackTrace(PrintStream stream)	Sends the stack trace to the stream.
void printStackTrace(PrintWriter stream)	Sends the stack trace to the stream.
void setStackTrace(StackTraceElem ent elements[ ])	Sets the stack trace to the elements passed in elements.
String toString()	Returns a String object containing a description of the exception.

# MULTITHREADING PROGRAMMING

## Thread:

small piece of code design to do a particular task in a program is called thread

Or

A thread is a lightweight sub process, the smallest unit of processing. It is a separate path of execution. Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

## ABOUT MULTITHREADING

# Multithreading programming

**Multithreading in Java** is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

# Multithreading programming

## Advantages of Java Multithreading

- 1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.
- 2) You **can perform many operations together**, so it **saves time**.
- 3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

# Multithreading programming

## Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

Process-based Multitasking (Multiprocessing)

Thread-based Multitasking (Multithreading)

# Multithreading programming

## 1) Process-based Multitasking (Multiprocessing)

If more than one program is under execution at the same time (simultaneously). It is called multitasking

Each process has an address in memory. In other words, each process allocates a separate memory area.

A process is heavyweight.

Cost of communication between the process is high.

Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

# Multithreading programming

**Example:** listening to songs while editing a program in text editor

## 2) Thread-based Multitasking (Multithreading)

If a single program performs more than one task simultaneously that is called thread based multitasking

Threads share the same address space.

A thread is lightweight.

Cost of communication between the thread is low.

**Example:** Editing text in a text editor while sending it and printer for printing

# Java Thread Models

During the life time of a thread there are may states it can be enter. They are:

New

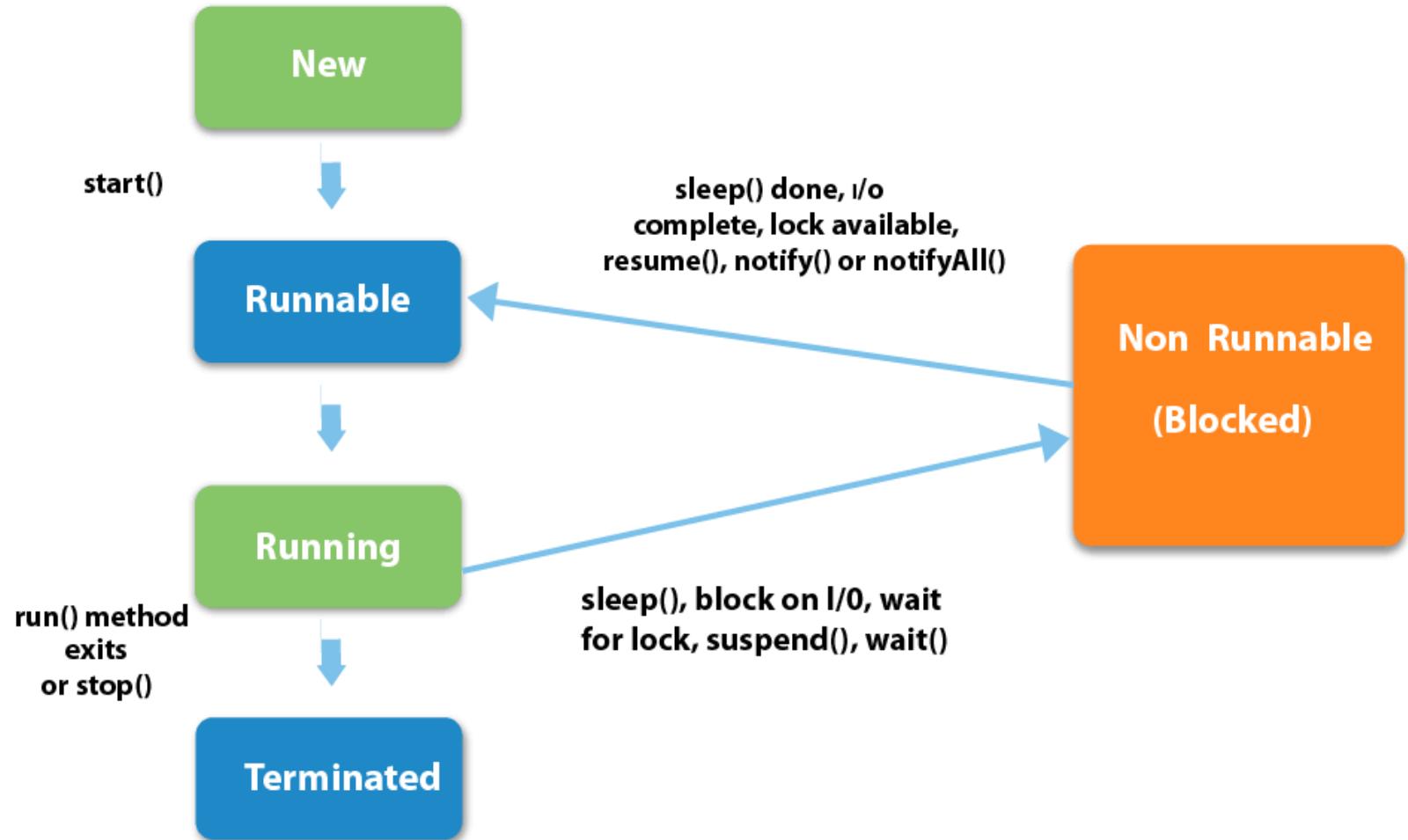
Runnable

Running

Non-Runnable (Blocked)

Terminated

# Java Thread Models



# Java Thread Models

## 1. New:

- When we create a thread object, the thread born and is said to be in new born state.
- At this state, we can do only one of the following with it.
  - Schedule it for running using start() method.
  - Kill it using stop() method.

## 2. Runnable:

- The runnable state means that the thread is ready for execution and is waiting for the availability of the that is the thread has joined the queue of threads that are waiting for execution.

# Java Thread Models

- ❑ If all threads have equal priority then they are given time starts for execution is round robin fashion( First Come First Serve).

## 3. Running:

- ❑ Running means that the processor has given its time to the thread for its execution.
- ❑ The thread run until it control on its own or its preempted by a higher priority thread

# Java Thread Models

## 4. Blocked:

- ❑ This is the state when the thread is still alive, but is currently not eligible to run.

## 5. Dead:

- ❑ A running threads ends with its life when its has completed executing its run() method.
- ❑ a thread can be killed as soon it is born, or while it is running, or even when it is in “not runnable”(Blocked Condition)

# Thread Priorities

Each thread have a priority. Priorities are represented by a number between 1 and 10.

In most cases, thread scheduler schedules the threads according to their priority (known as preemptive scheduling).

But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

# Thread Priorities

There are 3 static variables defined in Thread class for priority:

**public static int MIN\_PRIORITY:** This is minimum priority that a thread can have. Value for this is 1.

**public static int NORM\_PRIORITY:** This is default priority of a thread if do not explicitly define it. Value for this is 5.

**public static int MAX\_PRIORITY:** This is maximum priority of a thread. Value for this is 10.

# **Thread Priorities**

## **Get and Set Thread Priority:**

**public final int getPriority():**

java.lang.Thread.getPriority() method returns priority of given thread.

**public final void setPriority(int newPriority):**

java.lang.Thread.setPriority() method changes the priority of thread to the value newPriority. This method throws IllegalArgumentException if value of parameter newPriority goes beyond minimum(1) and maximum(10) limit.

```
class TestMultiPriority1 extends Thread
{
    public void run()
    {
        System.out.println("running thread name is:"+Thread.currentThread().getName());
    ;
        System.out.println("running thread priority is:"+Thread.currentThread().getPriorit
y());
    }

    public static void main(String args[])
    {
        TestMultiPriority1 m1=new TestMultiPriority1();
        TestMultiPriority1 m2=new TestMultiPriority1();
        m1.setPriority(Thread.MIN_PRIORITY);
        m2.setPriority(Thread.MAX_PRIORITY);
        m1.start();
        m2.start();
    }
}
```

# Thread Priorities

## Output:

running thread name is:Thread-0

running thread priority is:10

running thread name is:Thread-1

running thread priority is:1

# Thread Class and the Runnable Interface

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

## Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.

Thread class extends Object class and implements Runnable interface.

# Thread Class and the Runnable Interface

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r, String name)

# **Thread Class and the Runnable Interface**

Commonly used methods of Thread class:

**public void run():** is used to perform action for a thread.

**public void start():** starts the execution of the thread.JVM calls the run() method on the thread.

**public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

**public void join():** waits for a thread to die.

# **Thread Class and the Runnable Interface**

**public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.

**public int getPriority():** returns the priority of the thread.

**public int setPriority(int priority):** changes the priority of the thread.

**public String getName():** returns the name of the thread.

**public void setName(String name):** changes the name of the thread.

**public Thread currentThread():** returns the reference of currently executing thread.

# Thread Class and the Runnable Interface

**public int getId():** returns the id of the thread.

**public Thread.State getState():** returns the state of the thread.

**public boolean isAlive():** tests if the thread is alive.

**public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.

**public void suspend():** is used to suspend the thread(deprecated).

# **Thread Class and the Runnable Interface**

**public void resume():** is used to resume the suspended thread(deprecated).

**public void stop():** is used to stop the thread(deprecated).

**public boolean isDaemon():** tests if the thread is a daemon thread.

**public void setDaemon(boolean b):** marks the thread as daemon or user thread.

**public void interrupt():** interrupts the thread.

**public boolean isInterrupted():** tests if the thread has been interrupted.

**public static boolean interrupted():** tests if the current thread has been interrupted.

# Thread Class and the Runnable Interface

## Runnable interface:

The runnable interface declares the run() method that is required for implementing threads in our program.

- Declare the class as implementing the Runnable interface
- Implement the run() method.
- Create a thread by defining an object that is instantiated from this “runnable” class as the target of the thread.
- Call the threads start() method to run the thread

```
class X implements Runnable
{
    public void run()
    {
        for(int i=1;i<=10;i++)
        {
            System.out.println("\t threadX:"+i);
        }
        System.out.println("end of thread X");
    }
}
class RunnableTest
{
    public static void main(String args[])
    {
        X runnable=new X();
        Thread threadx=new Thread(runnable);
        threadx.start();
        System.out.println("end of main thread");
    }
}
```

# Thread Class and the Runnable Interface

## Output:

End of main Thread

Threadx:1

Threadx:2

Threadx:3

Threadx:4

Threadx:5

Threadx:6

Threadx:7

Threadx:8

Threadx:9

Threadx:10

End of the Threadx

# Creating Multiple Threads

Multithreading in Java is a process of executing two or more threads simultaneously to maximum utilization of CPU.

Multithreaded applications execute two or more threads run concurrently.

Hence, it is also known as Concurrency in Java.

Each thread runs parallel to each other.

Multitple threads don't allocate separate memory area, hence they save memory.

Also, context switching between threads takes less time.

So far, we have been using only two threads: the **main** thread and one **child** thread. However, our program can affect as many threads as it needs. Let's see how we can create multiple threads.

# Using isAlive() and join()

## isAlive():

- It tests if this thread is alive.
- A thread is alive if it has been started and has not yet died.
- There is a transitional period from when a thread is running to when a thread is not running.
- After the run() method returns, there is a short period of time before the thread stops.
- If we want to know if the start method of the thread has been called or if thread has been terminated, we must use isAlive() method.
- This method is used to find out if a thread has actually been started and has yet not terminated.

```
// Create multiple threads.
class NewThread implements Runnable
{
    String name; // name of thread
    Thread t;
    NewThread(String threadname)
    {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
    }
    // This is the entry point for thread.
    public void run()
    {
        try
        {
            for(int i = 5; i > 0; i--)
            {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e)
    }
}
```

```
{  
    System.out.println(name + "Interrupted");  
}  
System.out.println(name + " exiting.");  
}  
  
class MultiThreadDemo  
{  
    public static void main(String args[])  
    {  
        new NewThread("One"); // start threads  
        new NewThread("Two");  
        new NewThread("Three");  
        try  
        {  
            // wait for other threads to end  
            Thread.sleep(10000);  
        }  
        catch (InterruptedException e)  
        {  
            System.out.println("Main thread Interrupted");  
        }  
        System.out.println("Main thread exiting.");  
    }  
}
```

## Output:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

# Using isAlive( ) and join( )

## Syntax:

```
public final boolean isAlive()
```

## Return

This method will return true if the thread is alive otherwise returns false.

```
class MyThread extends Thread
{
    public void run()
    {
        for(int i=0;i<3;i++)
        {
            System.out.println("Status:"+isAlive());
        }
    }
}

class IsAliveDemo
{
    public static void main(String args[])
    {
        MyThread t=new MyThread();
        t.start();
        System.out.println("new status:"+isAlive());
    }
}
```

# Using isAlive( ) and join()

## Output:

Main thread status: true

Child thread status: true

Child thread status: true

Child thread status: true

# Using isAlive( ) and join()

## Join():

When the join() method is called, the current thread will simply wait until the thread it is joining with is no longer alive.

Or we can say the method that you will more commonly use to wait for a thread to finish is called join( ).

This method waits until the thread on which it is called terminates.

Its name comes from the concept of the calling thread waiting until the specified thread joins it.

Additional forms of join( ) allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

# Using isAlive( ) and join()

- If a thread wants to wait until completing some other thread then we should go for join method.
  - For example if a thread t1 wants to wait until completing t2 then “t1 has to call t2.join()”.
  - If t1 executes t2.join() then immediately t1 will enter into waiting state until t2 completes.
  - Once t2 completes then t1 will continue its execution.
- 
- **Syntax :**
  - **final void join( ) throws InterruptedException**

```
public class MyThread extends Thread
{
    public void run()
    {
        System.out.println("r1 ");
        try {
            Thread.sleep(500);
        }catch(InterruptedException ie){}
        System.out.println("r2 ");
    }
    public static void main(String[] args)
    {
        MyThread t1=new MyThread();
        MyThread t2=new MyThread();
        t1.start();

        try{
            t1.join(); //Waiting for t1 to finish
        }catch(InterruptedException ie){}
        t2.start();
    }
}
```

# Using isAlive( ) and join()

## Output:

r1

r2

r1

r2

```
public class MyThread extends Thread
{
    MyThread(String str)
    {
        super(str);
    }
    public void run()
    {
        System.out.println(Thread.currentThread().getName());
    }
    public static void main(String[] args)
    {
        MyThread t1=new MyThread("first thread");
        MyThread t2=new MyThread("second thread");
        t1.start();
        try
        {
            t1.join(1500);      //Waiting for t1 to finish
        }
        catch(InterruptedException ie)
        {
            System.out.println(ie);
        }
    }
}
```

}

```
t2.start();
try
{
    t2.join(1500);      //Waiting for t2 to finish
}
catch(InterruptedException ie)
{
    System.out.println(ie);
}
}
```

## Output:

r1

r2

r1

r2

# **Synchronization**

Synchronization in java is the capability to control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

## **Why use Synchronization**

The synchronization is mainly used to

- 1)To prevent thread interference.
- 2)To prevent consistency problem.

# Synchronization

## Types of thread synchronization:

There are two types of synchronization

- Process Synchronization
- Thread Synchronization

Here, we will discuss only thread synchronization.

## Thread Synchronization

There are two types of thread synchronization

### **1. Mutual Exclusive**

- | Synchronized method.
- | Synchronized block.
- | static synchronization.

### **2. Cooperation (Inter-thread communication in java)**

# Synchronization

## 1. Mutual Exclusive:

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

### ❖ Synchronized method:

- ② If you declare any method as synchronized, it is known as synchronized method.
- ② Synchronized method is used to lock an object for any shared resource.
- ② When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

```
//example of java synchronized method
class Table
{
    synchronized void printTable(int n)
    { //synchronized method
        for(int i=1;i<=5;i++)
        {
            System.out.println(n*i);
            try
            {
                Thread.sleep(400);
            }
            catch(Exception e)
            {
                System.out.println(e);
            }
        }
    }
}
```

```
class MyThread1 extends Thread
{
    Table t;
    MyThread1(Table t)
    {
        this.t=t;
    }
    public void run()
    {
        t.printTable(5);
    }
}

class MyThread2 extends Thread
{
    Table t;
    MyThread2(Table t)
    {
        this.t=t;
    }
    public void run()
    {
        t.printTable(100);
    }
}
```

```
public class TestSynchronization2
{
    public static void main(String args[])
    {
        Table obj = new Table(); //only one object
        MyThread1 t1=new MyThread1(obj);
        MyThread2 t2=new MyThread2(obj);
        t1.start();
        t2.start();
    }
}
```

### **Output:**

```
Output: 5
10
15
20
25
100
200
300
400
500
```

# Synchronization

## ❖ Synchronized block:

Synchronized block can be used to perform synchronization on any specific resource of the method.

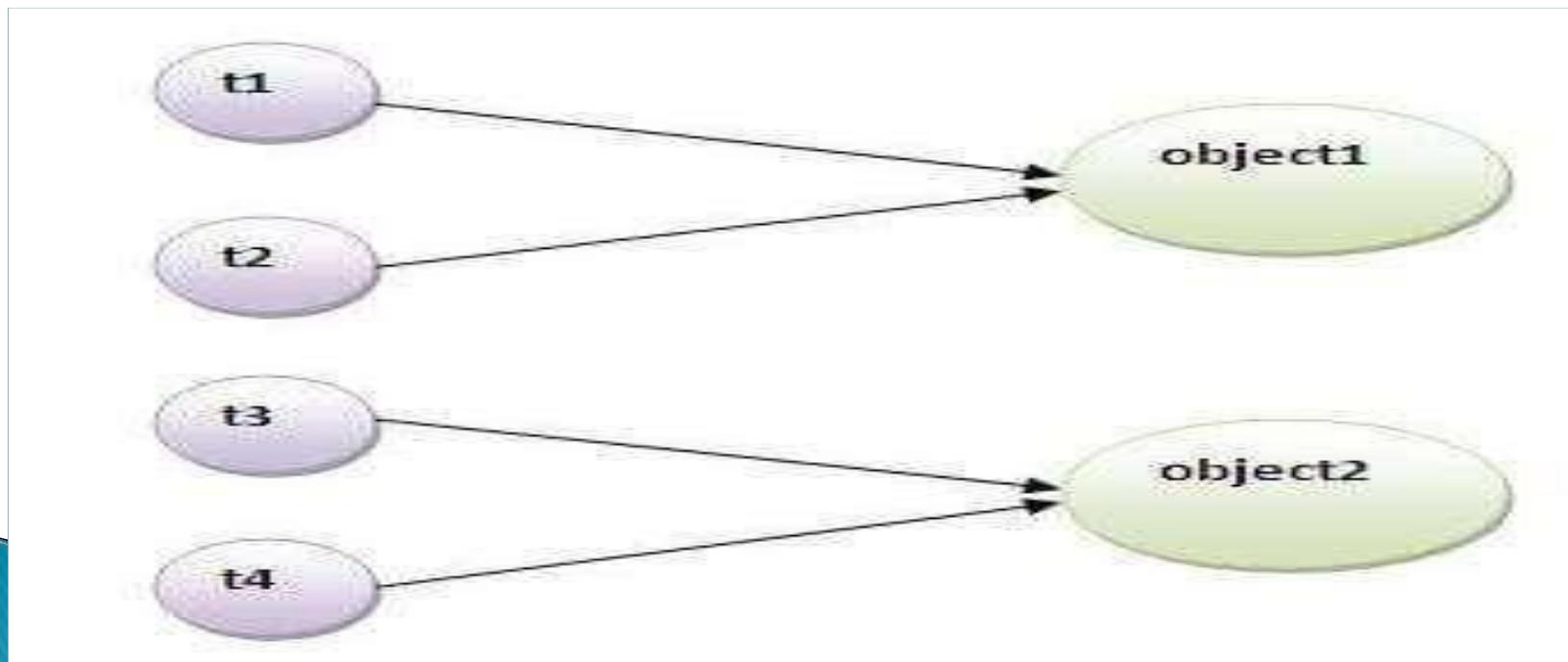
Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.

If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

# Synchronization

## ❖ static synchronization:

If you make any static method as synchronized, the lock will be on the class not on object.



# Synchronization

## Problem without static synchronization

Suppose there are two objects of a shared class(e.g. Table) named object1 and object2.

In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock.

But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock.

I want no interference between t1 and t3 or t2 and t4.Static synchronization solves this problem.

```
class Table
{
    synchronized static void printTable(int n)
    {
        for(int i=1;i<=10;i++)
        {
            System.out.println(n*i);
            try
            {
                Thread.sleep(400);
            }
            catch(Exception e)
            {
            }
        }
    }
}

class MyThread1 extends Thread
{
    public void run()
    {
        Table.printTable(1);
    }
}
```

```
class MyThread2 extends Thread
{
    public void run()
    {
        Table.printTable(10);
    }
}

class MyThread3 extends Thread
{
    public void run()
    {
        Table.printTable(100);
    }
}

class MyThread4 extends Thread
{
    public void run()
    {
        Table.printTable(1000);
    }
}
```

```
public class TestSynchronization4
```

```
{
```

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

```
{
```

```
    MyThread1 t1=new MyThread1();
```

```
    MyThread2 t2=new MyThread2();
```

```
    MyThread3 t3=new MyThread3();
```

```
    MyThread4 t4=new MyThread4();
```

```
    t1.start();
```

```
    t2.start();
```

```
    t3.start();
```

```
    t4.start();
```

```
}
```

```
}
```

# Synchronization

## Output:

1	10	100	1000
2	20	200	2000
3	30	300	3000
5	40	400	4000
6	50	500	5000
7	60	600	6000
8	70	700	7000
9	80	800	8000
10	90	900	9000
10	100	1000	10000

# Synchronization

## 2. Cooperation (Inter-thread communication in java)

**Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.

It is implemented by following methods of **Object class**:

- ❑ wait()
- ❑ notify()
- ❑ notifyAll()

# Synchronization

## 1) wait() method:

- Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

? public final void wait()throws  
InterruptedException

? public final void wait(long timeout) throws  
InterruptedException

# Synchronization

## 2) notify() method:

- Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened.
- The choice is arbitrary and occurs at the discretion of the implementation.

### Syntax:

```
public final void notify()
```

## 3) notifyAll() method:

- Wakes up all threads that are waiting on this object's monitor.

### Syntax:

```
public final void notifyAll()
```

# String Handling

String is a sequence of character. But in java, a string is an object that represents a sequence of character the `java.lang.String` class is used to create string object there are two ways to create string object

1. By string literals
2. By new keywords

## By string literals:

java string literals is created by using double quotes.

## For example :

`String s="Welcome";`

# String Handling

## By new keywords:

java string is created by using a keyword “new”.

It creates two objects(in string pool and in heap) and one reference variable where the variable ‘s’ will refer to the object in the heap

## For example:

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

# String Handling

## Java string methods

1. length()
2. CompareTo()
3. Concat()
4. IsEmpty()
5. Trim()
6. toLowerCase()
7. toUpper()
8. ValueOf()
9. Replace()

# String Handling

- 10. Contains()
- 11. Equals()
- 12. equalsIgnoreCase()
- 13. endsWith()
- 14. toCharArray()
- 15. GetBytes()

# String Handling

## 1. length():

The java string length method tells the length of the string.

It returning count of total number of character present in the string.

```
public class StringLengthMethod
{
    public static void main(String args[])
    {
        String s1="hello";
        String s2="java";
        System.out.println("String length is:"+s1.length());
        System.out.println("String length is:"+s2.length());
    }
}
```

The screenshot shows a Windows Command Prompt window titled "Command Prompt". The window contains the following text:

```
D:\>javac StringLengthMethod.java

D:\>java StringLengthMethod
String length is:5
String length is:4

D:\>_
```

The window has a blue title bar with standard window controls (minimize, maximize, close) and a black body area where the command-line output is displayed.

# String Handling

## 2. CompareTo():

The java string compareTo method compares the given string with current string.

It is a method of comparable interface which is implemented by string class

```
public class StringComparisonToMethod
{
    public static void main(String args[])
    {
        String s1="hello";
        String s2="hello";
        String s3="hemlo";
        String s4="flag";
        System.out.println("Because both are
equal:"+s1.compareTo(s2));
        System.out.println(" Because l is obly one time lower
than m:"+s1.compareTo(s3));
        System.out.println("Beacuse h is 2 times greaterthan f
:"+s1.compareTo(s4));
    }
}
```

# String Handling

## 3. Concat():

The java string concat method combines a specific string at the end of another string and ultimately returns a combined string.

It is like appending another string.

```
public class StringConcatMethod
{
    public static void main(String args[])
    {
        String s1="hello";
        s1=s1.concat("\t how are you");
        System.out.println(s1);
    }
}
```

The screenshot shows a Windows Command Prompt window titled "Command Prompt". The window has a blue header bar with standard window controls (minimize, maximize, close) on the right. The main area of the window is black, displaying the following text:

```
D:\>java>javac StringConcatMethod.java
D:\>java>java StringConcatMethod
hello      how are you
D:\>java>
```

The output of the program is displayed in white text on the black background. The command "javac" is used to compile the Java source code, and the command "java" is used to run the compiled class. The output shows the string "hello" followed by a tab character and the string "how are you".

# String Handling

## 4. IsEmpty():

This method checks whether the string contains anything or not.

If the java string is empty, it returns true else false.

```
public class StringIsEmptyMethod
{
    public static void main(String args[])
    {
        String s1="";
        String s2="hello";
        System.out.println(s1.isEmpty());
        System.out.println(s2.isEmpty());
    }
}
```

The screenshot shows a Windows Command Prompt window titled "Command Prompt". The window has a blue header bar with the title and standard window controls (minimize, maximize, close). The main area of the window is black and contains the following text output from a Java application:

```
D:\java>javac StringIsEmptyMethod.java
D:\java>java StringIsEmptyMethod
true
false
D:\java>
```

# String Handling

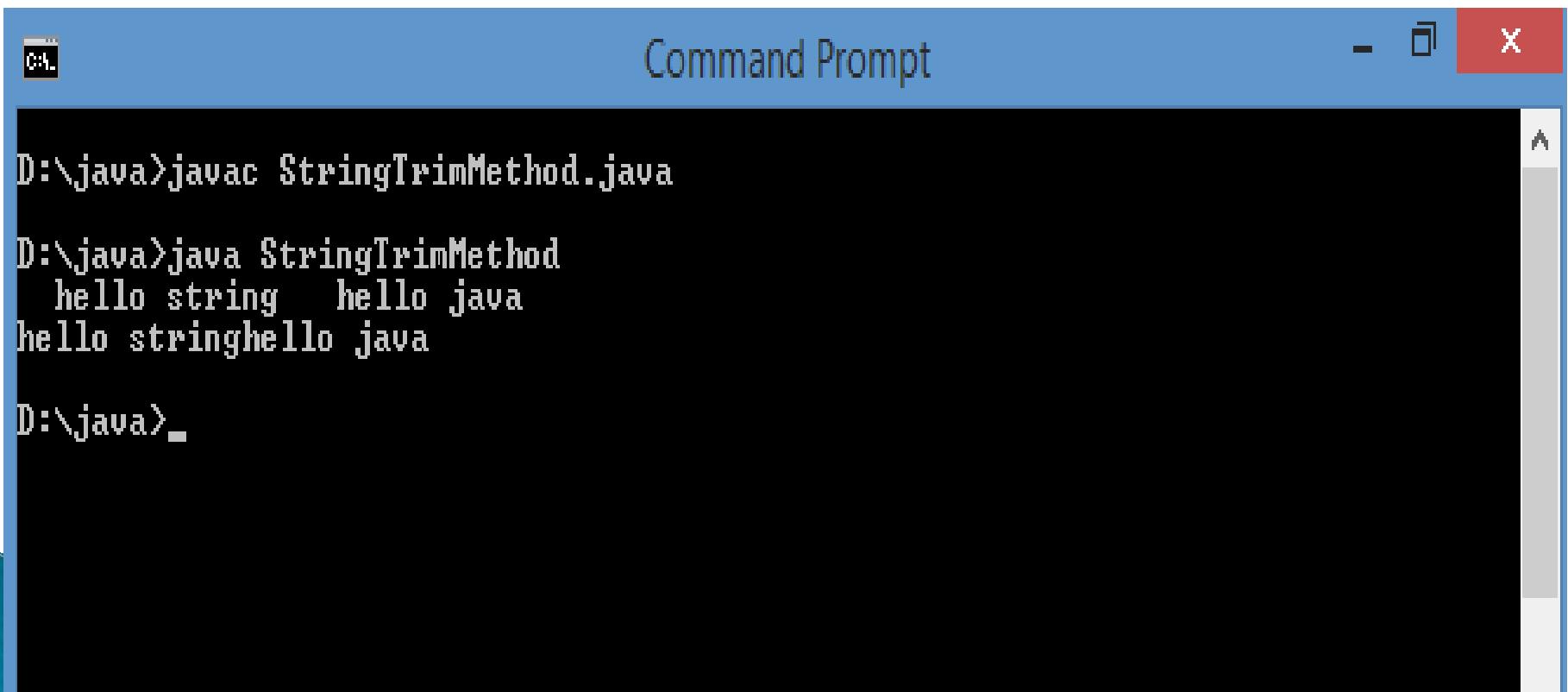
## 5. Java string Trim():

The java string trim() method removes the leading and trailing spaces.

It checks the unicode value of space character (U0020) before and after the string.

If it exists, then removes the spaces and return the omitted string.

```
public class StringTrimMethod
{
    public static void main(String args[])
    {
        String s1=" hello string ";
        System.out.println(s1+"hello java");//without trim()
        System.out.println(s1.trim()+"hello java");//with trim()
    }
}
```



D:\>javac StringTrimMethod.java

D:\>java StringTrimMethod

hello string hello java

hello stringhello java

D:\>\_

# String Handling

## 6. toLowerCase():

the java string toLowerCase() method converts all the characters of the string to lowercase

### Syntax:

```
public String toLowerCase()
```

```
public class StringLowerMethod
{
    public static void main(String args[])
    {
        String s1="JAVA HELLO stRIng";
        String s1lower=s1.toLowerCase();
        System.out.println(s1lower);
    }
}
```

The screenshot shows a Windows Command Prompt window titled "Command Prompt". The window has a blue header bar with the title and standard window controls (minimize, maximize, close). The main area of the window is black, displaying the following text:

```
D:\java>javac StringLowerMethod.java
D:\java>java StringLowerMethod
java hello string
```

The command "javac StringLowerMethod.java" is run to compile the Java source code. The command "java StringLowerMethod" is then run to execute the compiled program, which outputs the string "hello string" to the console.

# String Handling

## 7. toUpper():

the java string toUpperCase() method converts all the characters lower case to uppercase.

### Syntax:

```
public String toUpperCase()
```

```
public class StringUpperMethod
{
    public static void main(String args[])
    {
        String s1="hello java programming";
        String s1upper=s1.toUpperCase();
        System.out.println(s1upper);
    }
}
```

The screenshot shows a Windows Command Prompt window titled "Command Prompt". The window has a blue header bar with the title and standard window controls (minimize, maximize, close). The main area of the window is black and displays the following text:

```
D:\java>javac StringUpperMethod.java
D:\java>java StringUpperMethod
HELLO JAVA PROGRAMMING
D:\java>
```

# String Handling

## 8. **ValueOf():**

The **java string valueOf()** method converts different types of values into string.

By the help of string valueOf() method, you can convert int to string, long to string, boolean to string, character to string, float to string, double to string, object to string and char array to string.

```
public class ValueOfTest
{
    public static void main(String args[])
    {
        Integer x = Integer.valueOf(9);
        Double c = Double.valueOf(5);
        Float a = Float.valueOf("80");
        Integer b = Integer.valueOf("444",16);
        System.out.println(x);
        System.out.println(c);
        System.out.println(a);
        System.out.println(b);
    }
}
```

# String Handling

```
D:\java>javac ValueOfTest.java
D:\java>java ValueOfTest
9
5.0
80.0
1092
D:\java>
```

# String Handling

## 9. Replace ():

The java string replace() method returns a string, replacing all the old character of char sequence to new character.

There are 2 ways to replace methods in a java string

**public String replace(char oldChar, char newChar)**

and

**public String replace(CharSequence target, CharSequence replacement)**

```
public class StringReplaceMethod
{
    public static void main(String args[])
    {
        String s1="java is a very good programming";
        String replaceString=s1.replace('a','e');//replaces all
occurrences of 'a' to 'e'
        System.out.println(replaceString);
    }
}
```



```
D:\>javac StringReplaceMethod.java

D:\>java StringReplaceMethod
jeve is e very good progremming

D:\>java
```

# String Handling

## 10. contains():

The java string contains () method searches the sequence of character in the string.

If the sequences of characters are found then it returns true otherwise returns false.

```
class StringContainsMethod
{
    public static void main(String args[])
    {
        String name="hello how are you";
        System.out.println(name.contains("how are you"));
        System.out.println(name.contains("hello"));
        System.out.println(name.contains("fine"));
    }
}
```

A screenshot of a Windows Command Prompt window titled "Command Prompt". The window has a blue header bar with standard window controls (minimize, maximize, close) on the right. The main area is black with white text. The command history shows:

```
D:\>java>javac StringContainsMethod.java
D:\>java>java StringContainsMethod
true
true
false
D:\>java>
```

# String Handling

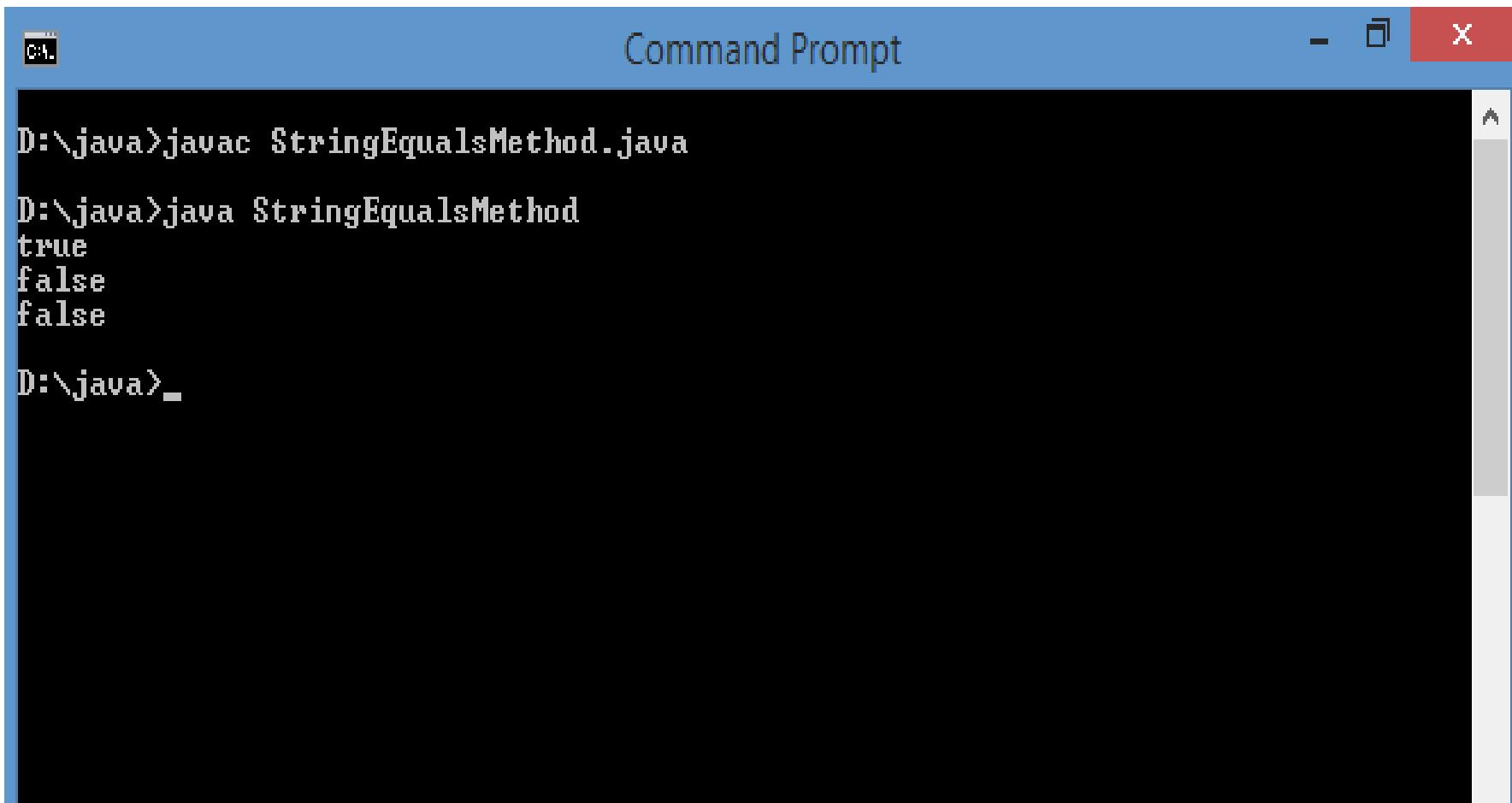
## 11. equals():

The **java string equals()** method compares the two given strings based on the content of the string.

If any character is not matched, it returns false. If all characters are matched, it returns true otherwise false.

```
public class StringEqualsIgnoreMethod
{
    public static void main(String args[])
    {
        String s1="java";
        String s2="java";
        String s3="JAVA";
        String s4="python";
        System.out.println(s1.equals(s2));//true because
content and case is same
        System.out.println(s1.equals(s3));//false because case
is not same
        System.out.println(s1.equals(s4));//false because
content is not same
    }
}
```

# String Handling



D:\java>javac StringEqualsMethod.java

D:\java>java StringEqualsMethod

```
true
false
false
```

D:\java>

# String Handling

## 12. EqualsIgnoreCase():

The **String equalsIgnoreCase()** method compares the two given strings on the basis of content of the string irrespective of case of the string.

It is like equals() method but doesn't check case.

If any character is not matched, it returns false otherwise it returns true.

```
public class StringEqualsIgnoreCaseMethod
{
    public static void main(String args[])
    {
        String s1="java";
        String s2="java";
        String s3="JAVA";
        String s4="python";
        System.out.println(s1.equalsIgnoreCase(s2));
        System.out.println(s1.equalsIgnoreCase(s3));
        System.out.println(s1.equalsIgnoreCase(s4));
    }
}
```

# String Handling



Command Prompt



```
D:\java>javac StringEqualsIgnoreMethod.java
```

```
D:\java>java StringEqualsIgnoreMethod
```

```
true
```

```
true
```

```
false
```

```
D:\java>
```

# String Handling

## 13. toCharArray():

the java method converts the string into a character array first it will calculate the length of the given java string including spaces and then create an array of char type with the same contents.

```
public class StringToCharArrayMethod2
{
    public static void main(String[] args)
    {
        String s1 = "Welcome to Java";
        char[] ch = s1.toCharArray();
        int len = ch.length;
        System.out.println("Char Array length: " + len);
        System.out.println("Char Array elements: ");
        for (int i = 0; i < len; i++)
        {
            System.out.println(ch[i]);
        }
    }
}
```

# String Handling

```
Command Prompt - X
D:\java>java StringToCharArrayMethod2
Char Array length: 15
Char Array elements:
W
e
l
c
o
m
e
t
o
J
a
v
a
D:\java>
```

# String Handling

## 14. getBytes():

The **java string getBytes()** method returns the byte array of the string. In other words, it returns sequence of bytes.

```
public class StringGetBytesmethod
{
    public static void main(String args[])
    {
        String s1="ABCDEFG";
        byte[] barr=s1.getBytes();
        for(int i=0;i<barr.length;i++)
        {
            System.out.println(barr[i]);
        }
    }
}
```

65

66

67

68

69

70

71

# String Handling

## 15.endwith():

The **java string endsWith()** method checks if this string ends with given suffix.

It returns true if this string ends with given suffix else returns false.

```
public class StringEndsWithMethod
{
    public static void main(String args[])
    {
        String s1="hello how are you";
        System.out.println(s1.endsWith("u"));
        System.out.println(s1.endsWith("you"));
        System.out.println(s1.endsWith("hello"));
    }
}
```

Command Prompt

```
D:\java>javac StringEndsWithMethod.java

D:\java>java StringEndsWithMethod
true
true
false

D:\java>
```

# **UNIT - 4**

**Generics– A simple Generic Example–General form of Generic class–Generic Interfaces Collections overview, Collection class, Collection interfaces. Introducing File Handling –File handling in java –Stream –Java File Method –File Operation in Java–Create file –Write to a file –Read from a file.**

# Generics

- The term Generics means PARAETERIZED TYPES.
- Using Generics, it is possible to create a single class, for example that automatically works with different types of data.
- “ A class, interface and method that operates on parameterized types are called generic, as in generic class or generic method”
- Generics introduced in **jdk5** first but due to large change some programmers reluctant to adopt its use. But later the release of **Jdk6**, generics can no longer be ignored. Fortunately generics are not difficult to use.

## ➤ About Generics:

# Generics

For example, suppose you have a method that adds two numbers together. In order to work with the types themselves, you might have to create multiple versions of this method. For instance:

```
public int Add(int a, int b)
```

```
public double Add(double a, double b)
```

```
public float Add(float a, float b)
```

Generics allow you to create a single method that is customized for the type that invokes it.

```
public T Add<T>(T a, T b)
```

T is substituted for whatever type you use.

# Generics

## Why use Generics in java:

Code that uses generics has many benefits over non-generic code:

Stronger type checks at compile time.

A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety.

Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

Elimination of casts.

The following code snippet without generics requires casting:  
`List list = new ArrayList(); list.add("hello"); String s = (String) list.get(0);`

# Generics

When re-written to use generics, the code does not require casting:

```
List<String> list = new ArrayList<String>();  
list.add("hello"); String s = list.get(0); // no cast
```

Enabling programmers to implement generic algorithms.

By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

# Generics

## Advantage of Java Generics:

### 1) Type-safety:

We can hold only a single type of objects in generics. It doesn't allow to store other objects.

Without Generics, we can store any type of objects.

# Generics

```
List list = new ArrayList();  
list.add(10);  
list.add("10");
```

With Generics, it is required to specify the type of object we need to store.

```
List<Integer> list = new ArrayList<Integer>();  
list.add(10);  
list.add("10");// compile-time error
```

# Generics

## 2) Type casting is not required:

There is no need to typecast the object.

Before Generics, we need to type cast.

```
List list = new ArrayList();
```

```
list.add("hello");
```

```
String s = (String) list.get(0); //typecasting
```

After Generics, we don't need to typecast the object.

```
List<String> list = new ArrayList<String>();
```

```
list.add("hello");
```

```
String s = list.get(0);
```

# Generics

## 3) Compile-Time Checking:

It is checked at compile time so problem will not occur at runtime.

The good programming strategy says it is far better to handle the problem at compile time than runtime.

```
List<String> list = new ArrayList<String>();  
list.add("hello");  
list.add(32); //Compile Time Error
```

# Generics

## Simple Generic Example:

The Java Generics allows us to create a single class, interface, and method that can be used with different types of data (objects).

This helps us to reuse our code.

**Note:** Generics does not work with primitive types (int, float, char, etc).

It shows how to create a simple generics class. We have created SimpleGeneric class, which accepts single type parameter.

# Generics

## General form of generic class:

A generic class declaration look like a non generic class declaration, except that the class name is followed by a type parameter section.

As with generic class can have one or more type parameters separated by commas.

These classes are known as paramerized classes or parameterized types because they accept one or more parameter

# Generics

## Declaring syntax for generic class:

Class class-name <type-param-list>

{

//.....

## Syntax for declaring a reference to generic class:

Class-name<type-arg-list> var-name=new class-  
name<type-arg-list>(cons-arg-list)

```
class Test<T, U>
{
    T obj1; // An object of type T
    U obj2; // An object of type U

    // constructor
    Test(T obj1, U obj2)
    {
        this.obj1 = obj1;
        this.obj2 = obj2;
    }

    // To print objects of T and U
    public void print()
    {
        System.out.println(obj1);
        System.out.println(obj2);
    }
}
```

```
// Driver class to test above
class Main
{
    public static void main (String[] args)
    {
        Test <String, Integer> obj = new Test<String, Integer>("SIETK", 15);
        obj.print();
    }
}
```

The screenshot shows a Windows Command Prompt window titled "Command Prompt". The window has a blue header bar with the title and standard window controls (minimize, maximize, close). The main area of the window is black and contains white text representing the command-line session.

```
D:\>java>javac Main.java
D:\>java>java Main
SIETK
15
```

The session starts with the command "javac Main.java" followed by "java Main". The output "SIETK" and "15" is displayed, indicating the values passed to the constructor and printed by the print() method.

```
// A generic interface example.  
// A Min/Max interface.  
interface MinMax<T extends Comparable<T>>  
{  
    T min();  
    T max();  
}  
// Now, implement MinMax  
class MyClass<T extends Comparable<T>> implements MinMax<T>  
{  
    T[] vals;  
    MyClass(T[] o)  
    {  
        vals = o;  
    }  
    // Return the minimum value in vals.  
    public T min()  
    {  
        T v = vals[0];  
        for(int i=1; i < vals.length; i++)  
            if(vals[i].compareTo(v) < 0) v = vals[i];  
        return v;  
    }  
}
```

# Generics

## Collection framework:

Any group of individual objects which are represented as a single unit is known as the collection of the objects

In java a separate framework named “ collection framework”.

The collection interface(`java.util.collection`) and (`java.util.map`) are the two main “root” interfaces java collection classes

## **What is a framework:**

A framework is a set of class and interfaces which provide a ready-made architecture

In order to implement a new feature or a class

# Generics

There is no need to define a framework

An optimal object-oriented design always includes a framework with a collection of class such that all the classes perform the same kind of task.

## Need for a separate collection framework:

Before collection framework (or before JDK 1.2) was introduced, the standard method for grouping java object (or collection) were Array or Vectors or Hashtable.

All of these collections had no common interface.

The main aim of all the collections are same, the implementation of all these collections were defined independently and had no correlation among them.

```
// Java program to demonstrate  
// why collection framework was needed  
import java.io.*;  
import java.util.*;  
  
class CollectionDemo  
{  
    public static void main(String[] args)  
    {  
        // Creating instances of the array,  
        // vector and hashtable  
        int arr[] = new int[] { 1, 2, 3, 4 };  
        Vector<Integer> v = new Vector();  
        Hashtable<Integer, String> h = new Hashtable();  
        // Adding the elements into the  
        // vector  
        v.addElement(1);  
        v.addElement(2);  
        // Adding the element into the  
        // hashtable  
        h.put(1, "java");  
        h.put(2, "programming");
```

```
// Array instance creation requires [],  
    // while Vector and hashtable require ()  
    // Vector element insertion requires addElement(),  
    // but hashtable element insertion requires put()  
  
    // Accessing the first element of the  
    // array, vector and hashtable  
System.out.println(arr[0]);  
System.out.println(v.elementAt(0));  
System.out.println(h.get(1));  
  
    // Array elements are accessed using [],  
    // vector elements using elementAt()  
    // and hashtable elements using get()  
}
```

}

output:

1

1

java

# Generics

**The primary advantages of a collections framework are that it:**

## **Reduces programming effort:**

Reduces programming effort by providing data structures and algorithms so you don't have to write them yourself.

## **Increases performance :**

Increases performance by providing high-performance implementations of data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be tuned by switching implementations.

## **Provides interoperability between unrelated APIs:**

Provides interoperability between unrelated APIs by establishing a common language to pass collections back and forth.

# Generics

## **Reduces the effort required to learn APIs:**

Reduces the effort required to learn APIs by requiring you to learn multiple ad hoc collection APIs.

## **Reduces the effort required to design and implement APIs:**

Reduces the effort required to design and implement APIs by not requiring you to produce ad hoc collections APIs.

## **Fosters software reuse :**

Fosters software reuse by providing a standard interface for collections and algorithms with which to manipulate them.

# Generics

## The collections framework consists of:

**Collection interfaces.** Represent different types of collections, such as sets, lists, and maps. These interfaces form the basis of the framework.

**General-purpose implementations.** Primary implementations of the collection interfaces.

**Legacy implementations.** The collection classes from earlier releases, Vector and Hashtable, were retrofitted to implement the collection interfaces.

**Special-purpose implementations.** Implementations designed for use in special situations. These implementations display nonstandard performance characteristics, usage restrictions, or behavior.

**Concurrent implementations.** Implementations designed for highly concurrent use.

# Generics

**Wrapper implementations.** Add functionality, such as synchronization, to other implementations.

**Convenience implementations.** High-performance "mini-implementations" of the collection interfaces.

**Abstract implementations.** Partial implementations of the collection interfaces to facilitate custom implementations.

**Algorithms.** Static methods that perform useful functions on collections, such as sorting a list.

**Infrastructure.** Interfaces that provide essential support for the collection interfaces.

**Array Utilities.** Utility functions for arrays of primitive types and reference objects. Not, strictly speaking, a part of the collections framework, this feature was added to the Java platform at the same time as the collections framework and relies on some of the same infrastructure.

# Generic Interface

## Generic Interfaces:

In addition to generic classes and methods, you can also have generic interfaces.

Generic interfaces are specified just like generic classes.

Here is an example. It creates an interface called **MinMax** that declares the methods **min( )** and **max( )**, which are expected to return the minimum and maximum value of some set of objects.

# **Generics**

## **Collection Overview:**

The Java Collections Framework standardizes the way in which groups of objects are handled by your programs.

Collections were not part of the original Java release, but were added by J2SE 1.2.

Prior to the Collections Framework, Java provided ad hoc classes such as Dictionary, Vector, Stack, and Properties to store and manipulate groups of objects.

The way that you used Vector was different from the way that you used Properties,

# Generics

## Collection class

providing skeletal implementations that are used as starting points for creating concrete collections.

general rule, the collection classes are not synchronized.

Class	Description
AbstractCollection	Implements most of the <b>Collection</b> interface.
AbstractList	Extends <b>AbstractCollection</b> and implements most of the <b>List</b> interface.
AbstractQueue	Extends <b>AbstractCollection</b> and implements parts of the <b>Queue</b> interface.
AbstractSequentialList	Extends <b>AbstractList</b> for use by a collection that uses sequential rather than random access of its elements
LinkedList	Implements a linked list by extending <b>AbstractSequentialList</b> .
ArrayList	Implements a dynamic array by extending <b>AbstractList</b> .
ArrayDeque	Implements a dynamic double-ended queue by extending <b>AbstractCollection</b> and implementing the <b>Deque</b> interface.

AbstractSet	Extends <b>AbstractCollection</b> and implements most of the <b>Set</b> interface.
EnumSet	Extends <b>AbstractSet</b> for use with <b>enum</b> elements.
HashSet	Extends <b>AbstractSet</b> for use with a hash table.
LinkedHashSet	Extends <b>HashSet</b> to allow insertion-order iterations.
PriorityQueue	Extends <b>AbstractQueue</b> to support a priority-based queue.
TreeSet	Implements a set stored in a tree. Extends <b>AbstractSet</b> .

# Generics

## Collection interfaces

The Collections Framework defines several core interfaces.

This section provides an overview of each interface.

Beginning with the collection interfaces is necessary because they determine the fundamental nature of the collection classes.

Put differently, the concrete classes simply provide different implementations of the standard interfaces.

# **Collection interface**

**Interface Description-** Collection Enables you to work with groups of objects; it is at the top of the collections hierarchy.

**Deque** - Extends **Queue** to handle a double-ended queue.

**List** - Extends **Collection** to handle sequences (lists of objects).

**NavigableSet**- Extends **SortedSet** to handle retrieval of elements based on closest-match searches.

**Queue**- Extends **Collection** to handle special types of lists in which elements are removed only from the head.

**Set**- Extends **Collection** to handle sets, which must contain unique elements.

**SortedSet** Extends **Set** to handle sorted sets

# Collection interface

collection interfaces, collections also use the **Comparator**, **RandomAccess**, **Iterator**, and **ListIterator** interfaces.

**Comparatorm** defines how two objects are compared; **Iterator**, **ListIterator**, and **Spliterator** enumerate the objects within a collection.

the collection interfaces allow some

methods to be optional. The optional methods enable you to modify the contents of a

collection. Collections that support these methods are called modifiable.

Collections that do not allow their contents to be changed are called unmodifiable.

If an attempt is made to use one of these methods on an unmodifiable collection, an **UnsupportedOperationException**

is thrown. All the built-in collections are modifiable.

# Collection interface

List Interface

Set Interface

SortedSet Interface

Navigatable Interface

Queue Interface

Deque Interface

# Introducing File Handling

File handling refers to **the method of storing data in the C program in the form of an output or input that might have been generated while running a C program in a data file**, i.e., a binary file or a text file for future analysis and reference in that very program.

# File handling in java

In Java, with the help of File Class, we can work with files. This File Class is inside the java.io package. The File class can be used by creating an object of the class and then specifying the name of the file.

# File Handling In Java

File Handling is an integral part of any programming language as file handling enables us to store the output of any particular program in a file and allows us to perform certain operations on it. In simple words, file handling means reading and writing data to a file.

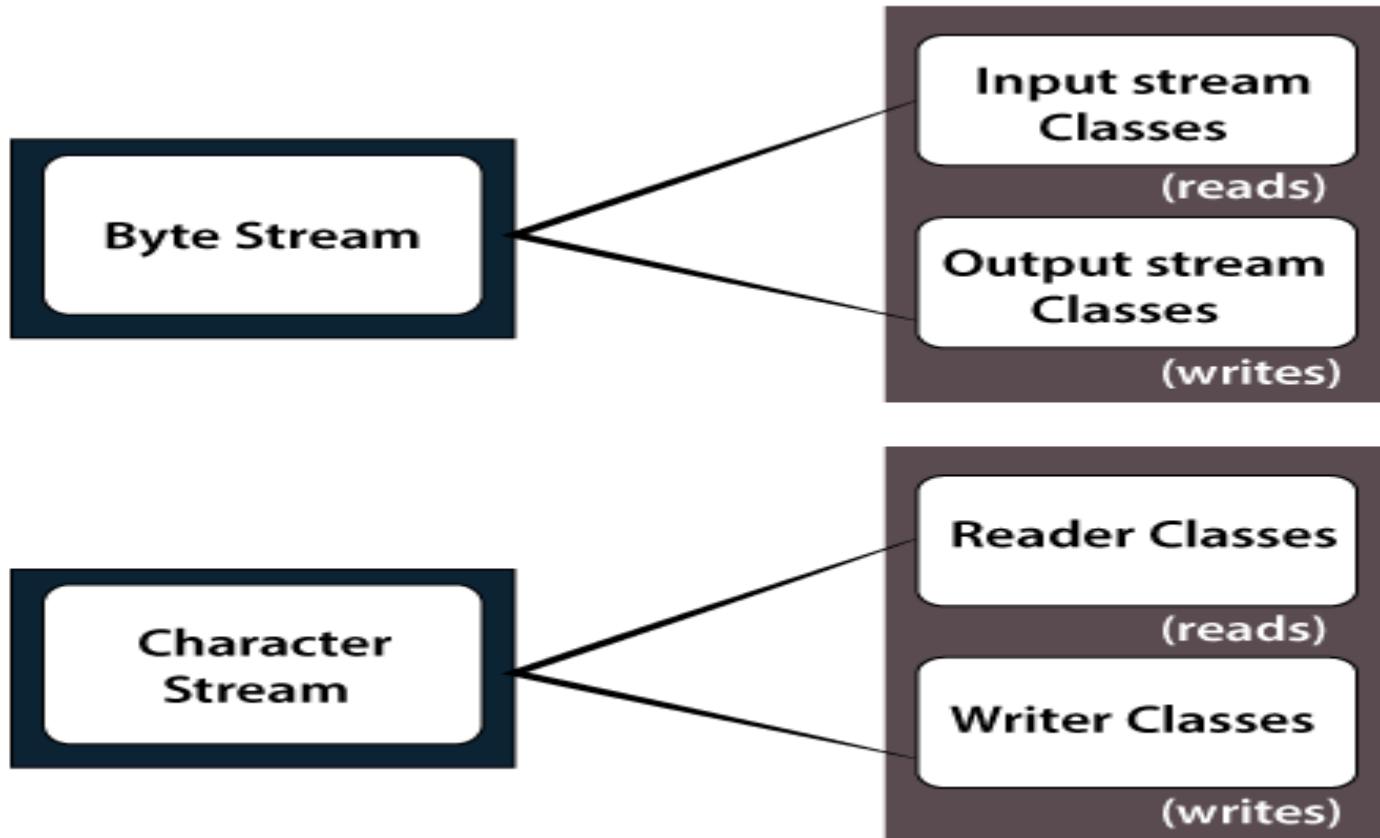
# **Stream**

A stream is **a sequence of objects that supports various methods which can be pipelined to produce the desired result.** The features of Java stream are – A stream is not a data structure instead it takes input from the Collections

# Java File Method

a File is an abstract data type. A named location used to store related information is known as a File. There are several File Operations like **creating a new File, getting information about File, writing into a File, reading from a File and deleting a File.**

# File Operation in Java



**Brief classification of I/O streams**

# Create file

Whenever you want to work with a file, the first step is to create a file. A file is nothing but space in a memory where data is stored. To create a file in a ‘C’ program following syntax is used,

```
FILE *fp; fp = fopen ("file_name", "mode");
```

# Write to a file

In C, when you write to a file, newline characters '\n' must be explicitly added.

The stdio library offers the necessary functions to write to a file:

**fputc(char, file\_pointer)**: It writes a character to the file pointed to by file\_pointer.

**fputs(str, file\_pointer)**: It writes a string to the file pointed to by file\_pointer.

**fprintf(file\_pointer, str, variable\_lists)**: It prints a string to the file pointed to by file\_pointer. The string can optionally include format specifiers and a list of variables variable\_lists.

# Read from a file

There are three different functions dedicated to reading data from a file

**fgetc(file\_pointer):** It returns the next character from the file pointed to by the file pointer. When the end of the file has been reached, the EOF is sent back.

**fgets(buffer, n, file\_pointer):** It reads  $n - 1$  characters from the file and stores the string in a buffer in which the NULL character '\0' is appended as the last character.

**fscanf(file\_pointer, conversion\_specifiers, variable\_adresses):** It is used to parse and analyze data. It reads characters from the file and assigns the input to a list of variable pointers variable\_adresses using conversion specifiers.

# **UNIT- V**

Introducing the AWT -Using AWT Controls-  
Introducing Swing -Exploring Swing. Introducing  
Java8 Features -Lambda Expression -Method  
references -forEach() method – Method and  
Constructor reference by double colon(:) operator  
– Stream API -Date & Time API.

# Introducing the AWT

AWT (Abstract Window Toolkit) is a java *API* that is used to develop GUI applications in Java.

AWT contains number of classes and interfaces that provide the basic infrastructure or core functionalities for developing specific type of application.

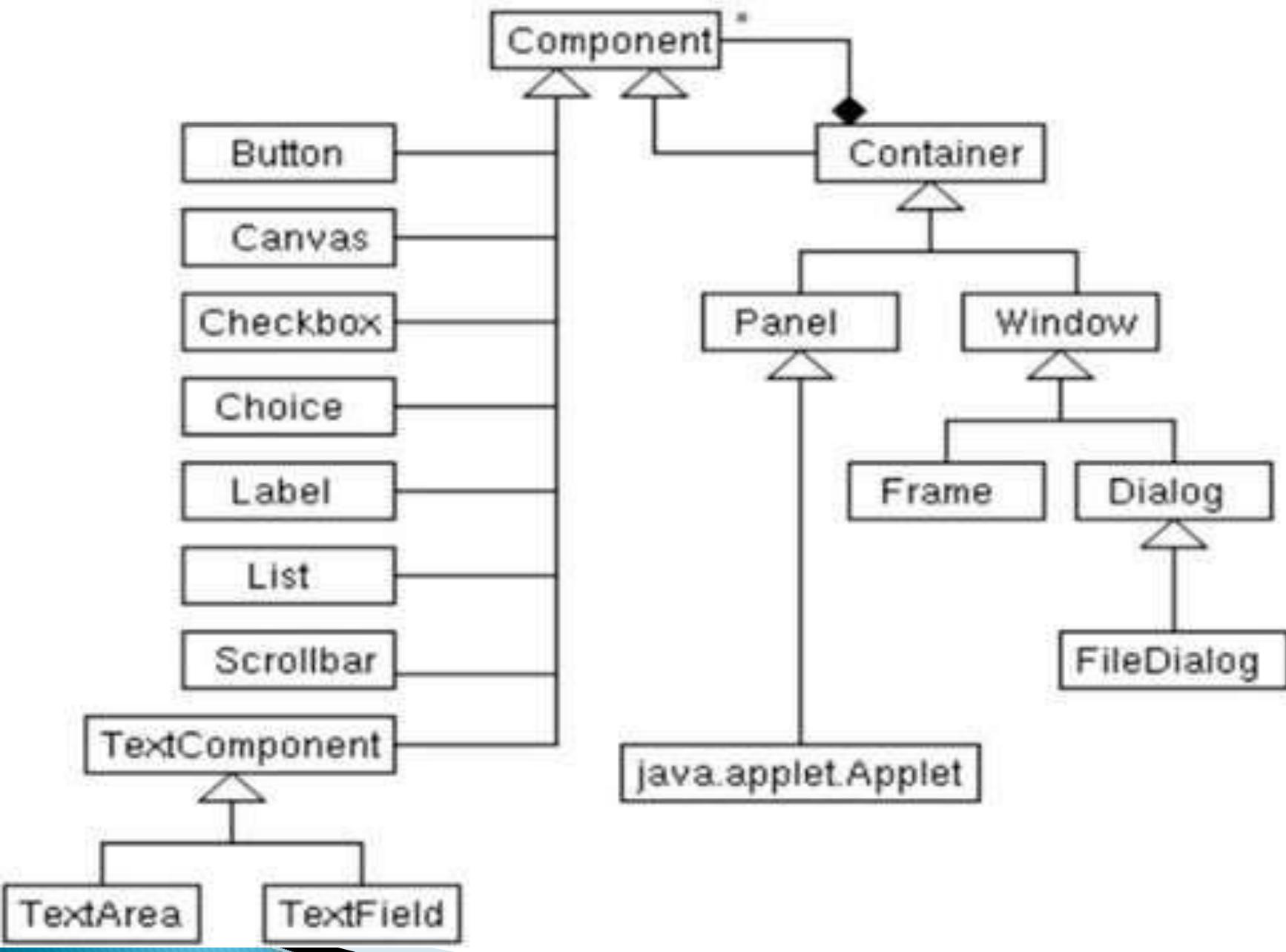
The common use of AWT is in Applets, it is also used to create stand-alone windows that run in a GUI environment, such as windows.

Swings is built-on top of the AWT. furthermore, many AWT classes are used directly or indirectly by Swing.

# Introducing the AWT

## AWT Classes:

The AWT classes are contained in the `java.awt` package. It is one of Java's largest packages. Fortunately, because it is logically organized in a top-down, hierarchical fashion, it is easier to understand and use than you might at first believe. Table lists some of the many AWT classes.



# Introducing the AWT

Component is an abstract class that describes the basic functionality supported by all AWT components.

Container is a sub-class of Component that adds the functionality of containership to a component.

A Container component can contain another container or non-container components.

Window and Panel are two non-abstract sub-classes of Container.

Panel represents a rectangular region that does not have a border and title bar.

Window is a Panel with border and title bar.

Window can independently exist whereas panel can't.

# Introducing the AWT

Frame is a sub-class of Window .

Button, Label, TextField, CheckBox etc. are non-container components.

## **Component:**

Component is an abstract class that encapsulates all of the attributes of a visual component.

All user interface elements that are displayed on the screen and that interact with the user are subclasses of Component.

It defines over a hundred public methods that are responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting.

# Introducing the AWT

**Component object** is responsible for remembering the current foreground and background colors and the currently selected text font.

Commonly used methods of Component class:

**setBackground()**: used to change the background color of a component.

*public void setBackground ( Color c )*

**setForeground()**: used to change the foreground color of a component.

*public void setForeground ( Font obj )*

Font is represented by java.awt.Font.

*public Font (String FontName, int Style, int Size)*

e.g. : Font f = (“Times New Roman”, Font.BOLD, 20);

**setBounds()**: used to specify size and position of component in a container.

*public void setBounds (int left, int top, int width, int height)*

# Introducing the AWT

## **Container:**

Container class is a subclass of Component.

It has additional methods that allow other Component objects to be nested within it.

Other Container objects can be stored inside of a Container (since they are themselves instances of Component).

This makes for a multileveled containment system.

A container is responsible for laying out (that is, positioning) any components that it contains. It does this through the use of various layout managers.

# Introducing the AWT

## Commonly used methods of Container class:

**add():** used to add components to a container.

public void add (Component c)

**setSize():** used to specify the size of a container.

public void setSize(int width, int height)

**setLayout():** used to specify the layout manager for a container.

public void setLayout (LayoutManager mgr)

**setVisible(boolean visibility):** used to set the visibility of container.

public void setVisible (boolean visibility)

# Introducing the AWT

## Panel:

Panel is a window that does not contain a title bar, menu bar, or border.

The screen output of an applet is drawn on the surface of a Panel object. But it is not visible when applet is run inside a browser.

When we run an applet using an applet viewer, the applet viewer provides the title and border.

Other components can be added to a Panel object by its add( ) method (inherited from Container).

We can position and resize the components of a panel manually using the setLocation(), setSize( ), setPreferredSize( ), or setBounds( ) methods defined by Component.

# Introducing the AWT

## **Window:**

Window class creates a top-level window.

A top-level window is not contained within any other object; it sits directly on the desktop.

Generally, we won't create Window objects directly. Instead, we use a subclass of Window called Frame.

## **Frame:**

It is a subclass of Window and has a title bar, menu bar, borders, and resizing corners.

Frame encapsulates what is commonly thought of as a “window.”

# Introducing the AWT

## Constructor:

public Frame ()

public Frame (String Title)

## Methods:

public String getTitle ()

public void setTitle (String Title)

public void setVisible(boolean Visibility)

## Canvas:

Canvas class is a part of Java AWT. Canvas is a blank rectangular area where the user can draw or trap input from the user. Canvas class inherits the Component class.

## Constructor of the Canvas class are :

**Canvas():** Creates a new blank canvas.

**Canvas(GraphicsConfiguration c):** Creates a new canvas with a specified graphics configuration.

# Introducing the AWT

## T:1 Usinng AWT Controls:

**AWT Supports the following types of controls.**

Label,

Button,

CheckBox, CheckBoxGroup,

Choice,

Lists,

ScrollBars,

TextField,

TextArea.

# Using AWT Controls

## 1. Label:

A label displays a string of read only text (does not support interaction) ,the label cannot changed by the end user anyway.

Creating the Label

The following constructor used to create the Label

**Label()** – empty label constructor

**Label(String text)** –text is the string you are going to pass to the label

**Label(String text,int alignment)** –you can align the label by align the value of alignment must be one of these three constants :  
Label.LEFT , Label.RIGHT , Label.CENTER.

Set the text of the label or change the text in a label void setText(String str);

Obtain the current label text by calling String getText()

Set the Alignment of the label by calling Void setAlignment(int align)

Obtain the current label Alignment by calling int getAlignment()

```
// Demonstrate Labels
import java.awt.*;
import java.applet.*;
/*
<applet code="LabelDemo" width=300 height=200>
</applet>
*/
public class LabelDemo extends Applet
{
    public void init()
    {
        Label one = new Label("One");
        Label two = new Label("Two");
        Label three = new Label("Three");
        // add labels to applet window
        add(one);
        add(two);
        add(three);
    }
}
```

# Using AWT Controls

Output:



# Using AWT Controls

## 2. Button:

Buttons are used to initiate action

A commonly used control is **push button**, it is a component that contains a label and that generates an event when it is pressed.

Push buttons are objects of type **Button**.

Button defines these two constructors: Constructor:

public Button ()//creates empty button

label public Button (String Name)//it contains string as

### Methods:

public String getLabel()

public void setLabel(String Name)

After button is created you can set its label by calling setLabel().you can retrieve its label by calling getLabel()

```
// Demonstrate Buttons
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ButtonDemo" width=250 height=150>
</applet>
*/
public class ButtonDemo extends Applet implements ActionListener
{
    String msg = "";
    Button yes, no, maybe;
    public void init()
    {
        yes = new Button("Yes");
        no = new Button("No");
        maybe = new Button("Undecided");
        add(yes);
        add(no);
        add(maybe);
        yes.addActionListener(this);
        no.addActionListener(this);
        maybe.addActionListener(this);
    }
}
```

```
public void actionPerformed(ActionEvent ae)
{
    String str = ae.getActionCommand();
    if(str.equals("Yes"))
    {
        msg = "You pressed Yes.";
    }
    else if(str.equals("No"))
    {
        msg = "You pressed No.";
    }
    else
    {
        msg = "You pressed Undecided.";
    }
    repaint();
}
public void paint(Graphics g)
{
    g.drawString(msg, 6, 100);
}
```

# Using AWT Controls



# **INTRODUCING SWING**

**Swing in java** is part of Java foundation class which is lightweight and platform independent. It is used for creating window based applications. It includes components like button, scroll bar, text field etc. Putting together all these components makes a graphical user interface.

## **What is Swing In Java?**

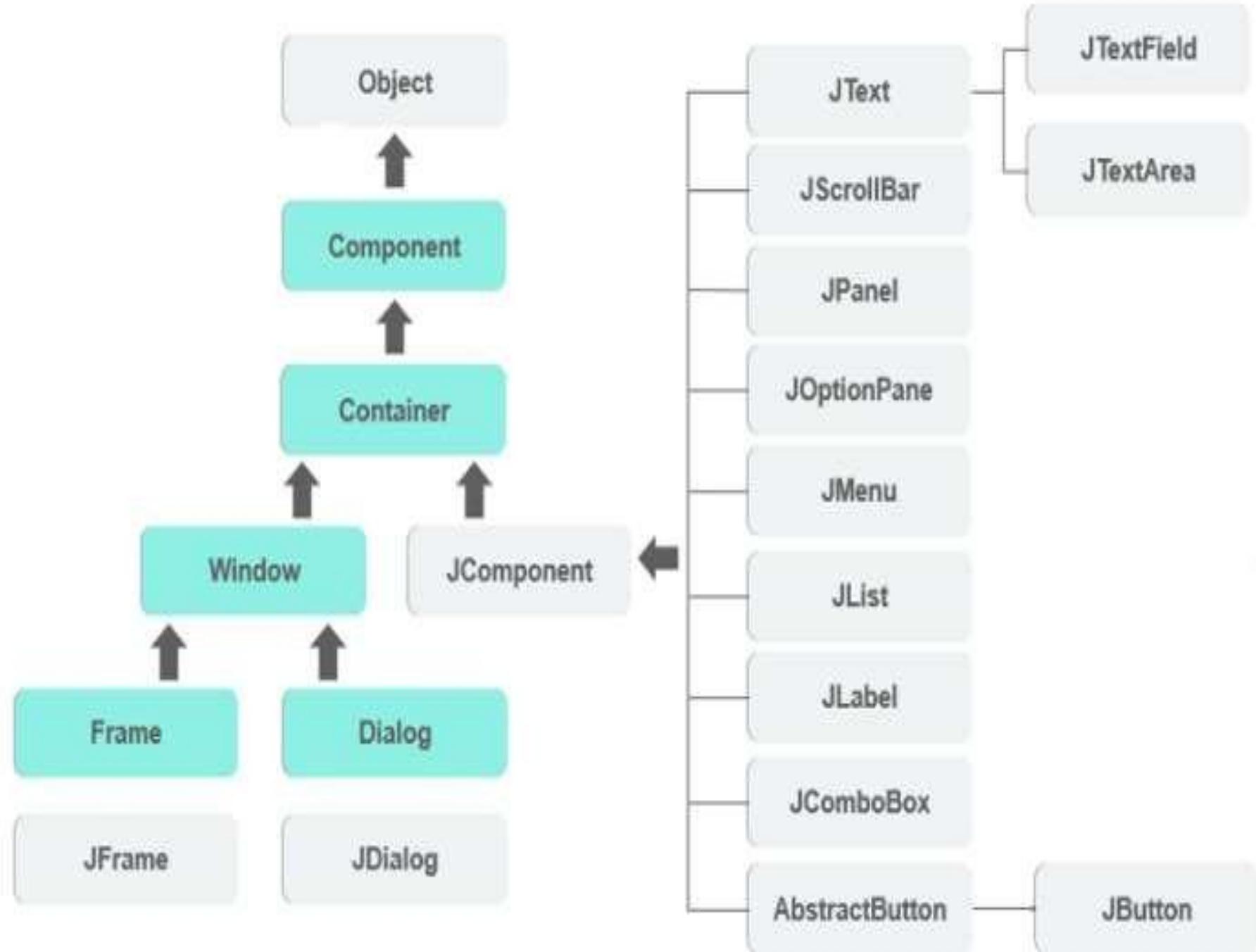
Swing in Java is a lightweight GUI toolkit which has a wide variety of widgets for building optimized **window based applications**. It is a part of the JFC (Java Foundation Classes). It is build on top of the **AWT API** and entirely written in java. It is platform independent unlike AWT and has lightweight components.

## **Container Class**

Any class which has other components in it is called as a container class. Following are the three types of container classes:

- **Panel** – It is used to organize components on to a window
- **Frame** – A fully functioning window with icons and titles
- **Dialog** – It is like a pop up window but not fully functional like the frame

## **About swing:**



# **INTRODUCING JAVA8 FEATURES**

## **LAMBDA EXPRESSION**

Lambda expressions are an anonymous function, meaning that they have no name or identifier. They can be passed as a parameter to another function. They are paired with a functional interface and feature a parameter with an expression that references that parameter.

**The syntax of a basic lambda expression is:**

*parameter -> expression*

# Java Lambda Expressions

Lambda expression is a new and important feature of Java which was included in Java SE 8.

It provides a clear and concise way to represent one method interface using an expression.

It is very useful in collection library.

It helps to iterate, filter and extract data from collection.

# Lambda expressions are added in Java 8 and provide below functionalities.

- Enable to treat functionality as a method argument, or code as data.
- A function that can be created without belonging to any class.
- A lambda expression can be passed around as if it was an object and executed on demand.

(int arg1, String arg2) -> {System.out.println("Two arguments "+arg1+" and "+arg2);}

The diagram illustrates the structure of a lambda expression with three horizontal lines. The first line contains the argument list '(int arg1, String arg2)'. The second line contains the arrow token '->'. The third line contains the body of the lambda expression '{System.out.println("Two arguments "+arg1+" and "+arg2);}'. Vertical lines connect each component to its corresponding label below: 'Argument List' under the first line, 'Arrow token' under the second line, and 'Body of lambda expression' under the third line.

Argument List      Arrow token      Body of lambda expression

# Why use Lambda Expression

To provide the implementation of Functional interface

Less coding

# Java Lambda Expression Syntax

(argument-list) -> {body}

Java lambda expression is consisted of three components.

- 1) **Argument-list:** It can be empty or non-empty as well.
- 2) **Arrow-token:** It is used to link arguments-list and body of expression.
- 3) **Body:** It contains expressions and statements for lambda expression.

# **METHOD REFERENCES**

Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference

## **Types of Method References**

There are following types of method references in java:

- Reference to a static method.
- Reference to an instance method.
- Reference to a constructor.

# Java Lambda Expression Example

```
@FunctionalInterface //It is optional
interface Drawable{
    public void draw();
}

public class LambdaExpressionExample2 {
    public static void main(String[] args) {
        int width=10;

        //with lambda
        Drawable d2=()->{
            System.out.println("Drawing "+width);
        };
        d2.draw();
    }
}
```

# Java Lambda Expression Example: No Parameter

A lambda expression can have zero or any number of arguments.

```
interface Sayable{
    public String say();
}

public class LambdaExpressionExample3{
    public static void main(String[] args) {
        Sayable s=()->{
            return "I have nothing to say.";
        };
        System.out.println(s.say());
    }
}
```

## No Parameter Syntax

```
( ) -> {  
    //Body of no parameter lambda  
}
```

## One Parameter Syntax

```
(p1) -> {  
    //Body of single parameter lambda  
}
```

## Two Parameter Syntax

```
(p1,p2) -> {  
    //Body of multiple parameter lambda  
}
```

# Method references

Method references help to point to methods by their names.

A method reference is described using ":" symbol.

Types of Method References in java:

1. Reference to a static method.
2. Reference to an instance method.
3. Reference to a constructor.

## **FOREACH() METHOD**

Java provides a new method `forEach()` to iterate the elements. It is defined in `Iterable` and `Stream` interface. It is a default method defined in the `Iterable` interface. Collection classes which extends `Iterable` interface can use `forEach` loop to iterate elements.

This method takes a single parameter which is a functional interface. So, you can pass lambda expression as an argument.

### **forEach() Signature in Iterable Interface**

```
default void forEach(Consumer<super T>action)
```

# For Each loop

Java provides a new method `forEach()` to iterate the elements.

It is defined in `Iterable` and `Stream` interface.

It is a default method defined in the `Iterable` interface.

Collection classes which extends `Iterable` interface can use `forEach` loop to iterate elements.

This method takes a single parameter which is a functional interface.

So, you can pass lambda expression as an argument.

syntax

**default void forEach(Consumer<super T>action)**

forEach() Signature in Iterable Interface  
syntax

**default void** forEach(Consumer<super T>action)

Java 8 forEach() example 1

```
import java.util.ArrayList;
import java.util.List;
public class ForEachExample {
    public static void main(String[] args) {
        List<String> gamesList = new ArrayList<String>();
        gamesList.add("Football");
        gamesList.add("Cricket");
        gamesList.add("Chess");
        gamesList.add("Hockey");
        System.out.println("-----");
        Iterating by passing lambda expression-----");
        gamesList.forEach(games -> System.out.println(games));
    }
}
```

## **METHOD AND CONSTRUCTOR REFERENCE BY DOUBLE COLON(::) OPERATOR**

The double colon operator (:) introduced in java 8, also known as method reference operator in java. It is used to call a method by referring to it with the help of its class/instance.

They behave exactly as the lambda expressions. The only difference it has from lambda expressions is that this uses direct reference to the method by name instead of providing a delegate to the method.

# **Double Colon Operator and Lambda**

Double colon operator (::) also a short hand for lambdas. Let's see a very simple example that print list items.

Using Lambda:

```
List<String> list = List.of("Peter", "Thomas",
"Edvard", "Gerhard");
// print using lambda
list.forEach(item -> System.out.println(item));
```

## **Using :: operator :**

To concise code and readable, we used Lambda in above example. Java method reference makes it even more shorter and readable.

```
// print using :: (method reference operator)
list.forEach(System.out::println);
```

To reference a method or to reference a constructor, we must use double colon operator. Following are the various scenarios where it can be used.

- Reference to static method.
- Reference to an instance method of a particular object.
- Reference to an instance method of an arbitrary object of a particular type
- Reference to a Constructor

## **STREAM API**

Java provides a new additional package in Java 8 called `java.util.stream`. This package consists of classes, interfaces and enum to allows functional-style operations on the elements. You can use stream by importing `java.util.stream` package.

## **Stream provides following features:**

- Stream does not store elements. It simply conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations.
- Stream is functional in nature. Operations performed on a stream does not modify it's source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
- Stream is lazy and evaluates code only when required.
- The elements of a stream are only visited once during the life of a stream.

# **DATE & TIME API**

Java 8 introduces a new date-time API under the package `java.time`. Following are some of the important classes introduced in `java.time` package.

- Local – Simplified date-time API with no complexity of timezone handling.
- Zoned – Specialized date-time API to deal with various timezones.

## **Local Date-Time API**

`LocalDate`/`LocalTime` and `LocalDateTime` classes simplify the development where timezones are not required. Let's see them in action.

# DATE & TIME API

Java 8 introduces a new date-time API under the package `java.time`. Following are some of the important classes introduced in `java.time` package.

**Local** – Simplified date-time API with no complexity of timezone handling.

**Zoned** – Specialized date-time API to deal with various timezones.

**Local Date -Time API** `LocalDate`/`LocalTime` and `LocalDateTime` classes simplify the development where timezones are not required.

# New Date Time API in Java 8

The new date API helps to overcome the drawbacks mentioned above with the legacy classes. It includes the following classes:

**java.time.LocalDate:** It represents a year-month-day in the ISO calendar and is useful for representing a date without a time. It can be used to represent a date only information such as a birth date or wedding date.

**java.time.LocalTime:** It deals in time only. It is useful for representing human-based time of day, such as movie times, or the opening and closing times of the local library.

**java.time.LocalDateTime:** It handles both date and time, without a time zone. It is a combination of LocalDate with LocalTime.

**java.time.ZonedDateTime:** It combines the LocalDateTime class with the zone information given in ZonId class. It represent a complete date time stamp along with timezone information.

**java.time.OffsetTime:** It handles time with a corresponding time zone offset from Greenwich/UTC, without a time zone ID.

**java.time.OffsetDateTime**: It handles a date and time with a corresponding time zone offset from Greenwich/UTC, without a time zone ID.

**java.time.Clock** : It provides access to the current instant, date and time in any given time-zone. Although the use of the Clock class is optional, this feature allows us to test your code for other time zones, or by using a fixed clock, where time does not change.

**java.time.Instant** : It represents the start of a nanosecond on the timeline (since EPOCH) and useful for generating a timestamp to represent machine time. An instant that occurs before the epoch has a negative value, and an instant that occurs after the epoch has a positive value.

**java.time.Duration** : Difference between two instants and measured in seconds or nanoseconds and does not use date-based constructs such as years, months, and days, though the class provides methods that convert to days, hours, and minutes.

**java.time.Period** : It is used to define the difference between dates in date-based values (years, months, days).

**java.time.ZoneId** : It states a time zone identifier and provides rules for converting between an Instant and a LocalDateTime.

**java.time.ZoneOffset** : It describes a time zone offset from Greenwich/UTC time.

**java.time.format.DateTimeFormatter** : It comes up with various predefined formatter, or we can define our own. It has parse() or format() method for parsing and formatting the date time values.

# WEB LINKS

OOP GEEKS AND GEEKS TUTORIAL

<https://www.geeksforgeeks.org/oops-object-oriented-design/>

OOP MYGREATLEARNING ACADEMY

<https://www.mygreatlearning.com/academy>

OOP ONLINE COURSE

<https://www.java67.com>