UNIT -1

Overview of Java

Java is one of the world’s most important and widely used computer languages, and it has held this distinction for many years. Unlike some other computer languages whose influence has weared with passage of time, while java's has grown.

Till today it is the first and best choice for developing console/web-based applications.

Creation of Java

Java was developed by James Ghosling, Patrick Naughton, Mike Sheridan at Sun Microsystems Inc. in 1991. It took 18 months to develop the first working version.

The initial name was Oak but was renamed to Java in 1995.

Evolution of Java

Java was initially launched as Java 1.0 but soon after its initial release, Java 1.1 was launched. Java 1.1 redefined event handling, new library elements were added.

In Java 1.2 Swing and Collection framework was added and suspend(), resume() and stop() methods were deprecated from Thread class.

No major changes were made into Java 1.3 but the next release that was Java 1.4 contained several important changes. Keyword assert, chained exceptions and channel based I/O System was introduced.

Java 1.5 was called J2SE 5, it added following major new features :

- Generics
- Annotations
- Autoboxing and autounboxing
- Enumerations
- For-each Loop
- Varargs
- Static Import
- Formatted I/O
- Concurrency utilities

Next major release was **Java SE 7** which included many new changes like:

- Now **String** can be used to control `Switch` statement.
- Multi Catch Exception
- `try-with-resource` statement
- Binary Integer Literals
- **Underscore** in numeric literals.

---

**Application of Java**

Java is widely used in every corner of world and of human life. Java is not only used in softwares but is also widely used in designing hardware controlling software components. There are more than 930 million JRE downloads each year and 3 billion mobile phones run java.

Following are some other usage of Java:

1. Developing Desktop Application
2. Web Applications like Linkedin
3. Mobile OS like Android
4. Embedded System
5. Robotics and games etc.

Features of Java

The prime reason behind creation of Java was to bring portability and security feature into a computer language. Beside these two major features, there were many other features that played an important role in moulding out the final form of this outstanding language. Those features are:

1) Simple

Java is easy to learn and its syntax is quite simple, clean and easy to understand. The confusing and ambiguous concepts of C++ are either left out in Java or they have been re-implemented in a cleaner way.

Eg : Pointers and Operator Overloading are not there in java but were an important part of C++.

2) Object Oriented

In java everything is Object which has some data and behaviour. Java can be easily extended as it is based on Object Model.

3) Robust

Java makes an effort to eliminate error prone codes by emphasizing mainly on compile time error checking and runtime checking. But the main areas which Java improved were
Memory Management and mishandled Exceptions by introducing automatic Garbage Collector and Exception Handling.

4) Platform Independent

Unlike other programming languages such as C, C++ etc which are compiled into platform specific machines. Java is guaranteed to be write-once, run-anywhere language.

On compilation Java program is compiled into bytecode. This bytecode is platform independent and can be run on any machine, plus this bytecode format also provide security. Any machine with Java Runtime Environment can run Java Programs.

5) Secure

When it comes to security, Java is always the first choice. With java secure features it enable us to develop virus free, temper free system. Java program always runs in Java runtime environment with almost null interaction with system OS, hence it is more secure.
6) Multi Threading

Java multithreading feature makes it possible to write program that can do many tasks simultaneously. Benefit of multithreading is that it utilizes same memory and other resources to execute multiple threads at the same time, like While typing, grammatical errors are checked along.

7) Portable

Java Byte code can be carried to any platform.

8) Architectural Neutral

No implementation dependent features. Everything related to storage is predefined, example: size of primitive data types

8) High Performance

Java enables high performance with the use of just-in-time compiler.

Setting Classpath for Java

Java is freely available on Oracle's Website. Download the latest version of JDK (Java Development Kit) on your operating system. Install JDK on your machine. Once you have installed Java on your machine you would need to set environment variable to point to correct installation directory.

Setting up path for windows (2000/XP/vista/Window 7,8)

Assuming that you have installed Java in C:\ Program files/ Java / JDK directory

Prof. Dhaval Thaker
Step 1: Right click on my computer and select properties.

Step 2: Go to the Advance System Settings tab.
**Step 3:** Click on Environment Variables button.

**Step 4:** Now alter the path variable so that it also contains the path to JDK installed directory.
For e.g:- Change `C:\windows\system 32.` to `C:\windows\system 32; C:\program files / Java/ JDK`.

### Setting up path for window 95/98/ME

Assuming that you have installed Java in `C:\program files\java\ JDK` directory, do the following:

**Step 1**: Edit the `C:\autoexec.bat` file and add the following line at the end.

```
SET PATH =% PATH% C:\ PROGRAM FILE/JAVA/JDK/bin
```

### What is JVM?

Java virtual Machine(JVM) is a virtual Machine that provides runtime environment to execute java byte code. JVM are most often implemented to run on existing operating system.

JVM control execution of every Java program. It enables features such as automated exception handling, Garbage-collected heap.

JVM is distributed with Java class library, a set of standard class library in byte code. That implements the Java Application Programming Interface (API).
JVM Architecture

Class Loader: Class loader loads the Class for execution.

Method area: Stores pre-class structure as constant pool.

Heap: Heap is in which objects are allocated.

Stack: Local variables and partial results are store here. Each thread has a private JVM stack created when the thread is created.

Program register: Program register holds the address of JVM instruction currently being executed.
Native method stack: It contains all native used in application.

Executive Engine: Execution engine controls the execute of instructions contained in the methods of the classes.

Native Method Interface: Native method interface gives an interface between java code and native code during execution.

Native Method Libraries: Native Libraries consist of files required for the execution of native code.

First Java Program

Let us look at a simple java program.

class Hello
{
    public static void main (String[] args)
    {
        System.out.println ("Hello World program");
    }
}

Step to compile and run your first java program

Step 1: Open a text editor and write the code as above.

Step 2: Save the file as Hello.java

Step 3: Open command prompt and go to the directory where you saved your first java program assuming it is same in C:\

Step 4: Type `javac Hello.java` and press Return to compile your code. If there is no error in the code the command prompt will take you to the next line.
Step 5: Now type `java Hello` on command prompt to run your program.

Step 6: You will be able to see **Hello world program** printed on your command prompt.

---

**Now let us see what happen at runtime**

After writing your Java program, when you will try to compile it. Compiler will perform some compilation operation on your program.

Once it is compiled successfully byte code(.class file) is generated by the compiler.
After compiling when you will try to run the byte code (.class file), the following steps are performed at runtime:-

1. Class loader loads the java class. It is subsystem of JVM Java Virtual machine.
2. Byte Code verifier checks the code fragments for illegal code that can violate access right to object.
3. Interpreter read the byte code stream and then executes the instructions.

**Data-type**

Java language has a rich implementation of data type. Data type specify size and the type of values that can be stored in an identifier.

In java, data types are classified into two category.

1. Primitive Data type
2. Non-Primitive Data type

---

**1) Primitive Data type**

A primitive data type can be of eight types:

<table>
<thead>
<tr>
<th>Primitive Data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
</tr>
</tbody>
</table>

Once a primitive data type has been declared its type can never change, although in most cases its value can change. These eight primitive type can be put into four groups.
**Integer**

This group includes `byte`, `short`, `int`, `long`.

**byte**: It is 8 bit integer data type. Value range from -128 to 127. Default value zero.  
Example: `byte b=10;`

**short**: It is 16 bit integer data type. Value range from -32768 to 32767. Default value zero.  
Example: `short s=11;`

**int**: It is 32 bit integer data type. Value range from -2147483648 to 2147483647. Default value zero.  
Example: `int i=10;`

Example: `long l=100012;`

**Floating-Point Number**

This group includes `float`, `double`.

**float**: It is 32 bit float data type. Default value 0.0f.  
Example: `float ff=10.3f;`

**double**: It is 64 bit float data type. Default value 0.0d.  
Example: `double db=11.123;`

**Characters**

This group represents `char`, which represent symbols in a character set, like letters and numbers.

**char**: It is 16 bit unsigned unicode character. Range 0 to 65,535.  
Example: `char c='a';`

**Boolean**

This group represents `boolean`, which is a special type for representing true/false values.  
They are defined constant of the language.  
Example: `boolean b=true;`
2) Non-Primitive(Reference) Data type

A reference data type is used to refer to an object. A reference variable is declared to be of specific and that type can never be change. We will talk a lot more about reference data type later in Classes and Object lesson.

Identifier

All Java components require name. Name used for classes, methods, interfaces and variables are called Identifier. Identifier must follow some rule. Here are the rules:

- All identifier must start with either a letter (a to z or A to Z) or currency character ($) or an underscore.
- After the first character, an identifier can have any combination of character.
- A keyword cannot be used as an identifier.
- Identifiers in Java are case sensitive, foo and Foo are two different identifier.

Variable

Java Programming language defines mainly three kind of variables.

1. Instance variables
2. Static Variables
3. Local Variables
1) *Instance variables*

Instance variables are variables that are declare inside a class but outside any method, constructor or block. Instance variable are also variable of object commonly known as field or property.

```java
class Student {
    String name;
    int age;
}
```

Here `name` and `age` are instance variable of Student class.

2) *Static variables*

Static are class variables declared with static keyword. Static variables are initialized only once. Static variables are also used in declaring constant along with final keyword.

```java
class Student {
    String name;
    int age;
    static int instituteCode=1101;
}
```

Here `instituteCode` is a static variable. Each object of Student class will share instituteCode property.
3) Local variables

Local variables are declared in method constructor or blocks. Local variables are initialized when method or constructor block start and will be destroyed once its end. Local variable reside in stack. Access modifiers are not used for local variable.

```java
float getDiscount(int price)
{
    float discount;
    discount=price*(20/100);
    return discount;
}
```

Array

An array is a collection of similar data types. Array is a container object that hold values of homogenous type. It is also known as static data structure because size of an array must be specified at the time of its declaration.

An array can be either primitive or reference type. It gets memory in heap area. Index of array starts from zero to size-1.

---

Array Declaration

Syntax :

datatype[] identifier;

or

datatype identifier[];

Both are valid syntax for array declaration. But the former is more readable.

Example :
```java
int[] arr;
char[] arr;
short[] arr;
long[] arr;
int[][] arr; //two dimensional array.
```

**Initialization**

`new` operator is used to initialized an array.

**Example:**

```java
int[] arr=new int[10]; //10 is the size of array.
```

or

```java
int[] arr={10,20,30,40,50};
```

---

**Accessing array element**

As mention earlier array index starts from 0. To access nth element of an array. Syntax

```
arrayname[n-1];
```

**Example** : To access 4th element of a given array

```java
int[] arr={10,20,30,40};
System.out.println("Element at index 3"+arr[2]);
```

The above code will print the 4th element of array arr on console.
foreach or enhanced for loop

J2SE 5 introduces special type of for loop called foreach loop to access elements of array. Using foreach loop you can access complete array sequentially without using index of array. Let us see an example of foreach loop.

class Test
{
    public static void main(String[] args)
    {
        int[] arr={10,20,30,40};
        for(int x:arr)
        {
            System.out.println(x);
        }
    }
}

output: 10
20
30
40

Java Operators

Java provides a rich set of operators environment. Java operators can be divided into following categories

- Arithmetic operators
- Relation operators
- Logical operators
• Bitwise operators
• Assignment operators
• Conditional operators
• Misc operators

---

**Arithmetic operators**

Arithmetic operators are used in mathematical expression in the same way that are used in algebra.

<table>
<thead>
<tr>
<th>operator</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>adds two operands</td>
</tr>
<tr>
<td>-</td>
<td>subtract second operands from first</td>
</tr>
<tr>
<td>*</td>
<td>multiply two operand</td>
</tr>
<tr>
<td>/</td>
<td>divide numerator by denominator</td>
</tr>
<tr>
<td>%</td>
<td>remainder of division</td>
</tr>
<tr>
<td>++</td>
<td>Increment operator increases integer value by one</td>
</tr>
<tr>
<td>--</td>
<td>Decrement operator decreases integer value by one</td>
</tr>
</tbody>
</table>

Prof. Dhaval Thaker
Relation operators

<table>
<thead>
<tr>
<th>operator</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Check if two operand are equal</td>
</tr>
<tr>
<td>!=</td>
<td>Check if two operand are not equal.</td>
</tr>
<tr>
<td>&gt;</td>
<td>Check if operand on the left is greater than operand on the right</td>
</tr>
<tr>
<td>&lt;</td>
<td>Check operand on the left is smaller than right operand</td>
</tr>
<tr>
<td>&gt;=</td>
<td>check left operand is greater than or equal to right operand</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Check if operand on left is smaller than or equal to right operand</td>
</tr>
</tbody>
</table>

The following table shows all relation operators supported by Java.

Logical operators

Java supports following 3 logical operator. Suppose a=1 and b=0;

<table>
<thead>
<tr>
<th>operator</th>
<th>description</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
<td>(a &amp;&amp; b) is false</td>
</tr>
</tbody>
</table>
Logical OR

(a || b) is true

Logical NOT

(!a) is false

---

**Bitwise operators**

Java defines several bitwise operators that can be applied to the integer types long, int, short, char and byte.

<table>
<thead>
<tr>
<th>operator</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Bitwise exclusive OR</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>left shift</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>right shift</td>
</tr>
</tbody>
</table>

Now let's see truth table for bitwise `&`, `|` and `^`:

| a | b | a & b | a | b | a ^ b |
|---|---|-------|---|---|--
| 0 | 0 |   0   | 0 |   0 |
The bitwise shift operators shifts the bit value. The left operand specifies the value to be shifted and the right operand specifies the number of positions that the bits in the value are to be shifted. Both operands have the same precedence. **Example**

\[
\begin{align*}
a &= 0001000 \\
b &= 2 \\
a &\ll b = 0100000 \\
a &\gg b = 000010
\end{align*}
\]

**Assignment Operators**

Assignment operator supported by Java are as follows

<table>
<thead>
<tr>
<th>operator</th>
<th>description</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>assigns values from right side operands to left side operand</td>
<td>a = b</td>
</tr>
<tr>
<td>+=</td>
<td>adds right operand to the left operand and assign the result to left</td>
<td>a += b is same as a = a + b</td>
</tr>
<tr>
<td>-=</td>
<td>subtracts right operand from the left operand and assign the result to left</td>
<td>a -= b is same as a = a - b</td>
</tr>
<tr>
<td>Operator</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>*=</td>
<td>Multiply left operand with the right operand and assign the result to left operand</td>
<td>a*=b is same as a=a*b</td>
</tr>
<tr>
<td>/=</td>
<td>Divide left operand with the right operand and assign the result to left operand</td>
<td>a/=b is same as a=a/b</td>
</tr>
<tr>
<td>%=</td>
<td>Calculate modulus using two operands and assign the result to left operand</td>
<td>a%=b is same as a=a%b</td>
</tr>
</tbody>
</table>

**Misc operator**

There are few other operators supported by the Java language.

**Conditional operator**

It is also known as ternary operator and used to evaluate Boolean expression.

\[ epr1 \ ? \ expr2 \ : \ expr3 \]

If \( epr1 \) Condition is true? Then value \( expr2 \) : Otherwise value \( expr3 \)

**instanceOf operator**

This operator is used for object reference variables. The operator checks whether the object is of particular type (class type or interface type).
UNIT -2

Object and Classes

Since Java is an object oriented language, complete java language is build on classes and object. Java is also known as a strong **Object oriented programming language** (oops).

OOPS is a programming approach which provides solution to problems with the help of algorithms based on real world. It uses real world approach to solve a problem. So object oriented technique offers better and easy way to write program then procedural programming model such as C, ALGOL, PASCAL.

---

**Main Features of OOPS**

- Inheritance
- Polymorphism
- Encapsulation
- Abstraction

As an object oriented language Java supports all the features given above. We will discuss all these features in detail later.

---

Class

In Java everything is encapsulated under classes. Class is the core of Java language. Class can be defined as a template/ blueprint that describe the behaviors /states of a particular entity. A class defines new data type. Once defined this new type can be used to create object of that type. Object is an instance of class. You may also call it as physical existence of a logical template class.
A class is declared using **class** keyword. A class contain both data and code that operate on that data. The data or variables defined within a **class** are called **instance variables** and the code that operates on this data is known as **methods**.

---

**Rules for Java Class**

- A class can have only public or default(no modifier) access specifier.
- It can be either abstract, final or concrete (normal class).
- It must have the class keyword, and class must be followed by a legal identifier.
- It may optionally extend one parent class. By default, it will extend java.lang.Object.
- It may optionally implement any number of comma-separated interfaces.
- The class's variables and methods are declared within a set of curly braces `{}`.
- Each .java source file may contain only one public class. A source file may contain any number of default visible classes.
- Finally, the source file name must match the public class name and it must have a .java suffix.

---

**A simple class example**

Suppose, Student is a **class** and student’s name, roll number, age will be its property. Let's see this in Java syntax

class Student {
  String name;
  int rollno;
  int age;
}
When a reference is made to a particular student with its property then it becomes an **object**, physical existence of Student class.

```java
Student std=new Student();
```

After the above statement `std` is instance/object of Student class. Here the `new` keyword creates an actual physical copy of the object and assigns it to the `std` variable. It will have physical existence and get memory in heap area. The `new` operator dynamically allocates memory for an object

**Methods in Java**

Method describe behavior of an object. A method is a collection of statements that are group together to perform an operation.

**Syntax :**

```
return-type methodName(parameter-list)
```

Prof. Dhaval Thaker
Example of a Method

public String getName(String st)
{
    String name="StudyTonight";
    name=name+st;
    return name;
}

**Modifier** : Modifier are access type of method. We will discuss it in detail later.

**Return Type** : A method may return value. Data type of value return by a method is declare in method heading.

**Method name** : Actual name of the method.

**Parameter** : Value passed to a method.
Method body: collection of statement that defines what method does.

Parameter Vs. Argument

While talking about method, it is important to know the difference between two terms parameter and argument.

Parameter is variable defined by a method that receives value when the method is called. Parameter are always local to the method they don't have scope outside the method. While argument is a value that is passed to a method when it is called.

```
public void sum( int x, int y )
{
    System.out.println(x+y);
}
public static void main( String[ ] args )
{
    Test b=new Test();
    b.sum( 10, 20 );
}
```

Call-by-value and call-by-reference

There are two ways to pass an argument to a method
1. **call-by-value** : In this approach, a copy of an argument value is passed to a method. Changes made to the argument value inside the method will have no effect on the arguments.

2. **call-by-reference** : In this approach, a reference of an argument is passed to a method. Any changes made inside the method will affect the argument value.

**NOTE** : In Java, when you pass a primitive type to a method it is passed by value whereas when you pass an object of any type to a method it is passed as reference.

---

**Example of call-by-value**

```java
class Test {
    public void callByValue(int x) {
        x = 100;
    }
    public static void main(String[] args) {
        int x = 50;
        Test t = new Test();
        t.callByValue(x); //function call
        System.out.println(x);
    }
}
```

Output : 50
Example of call-by-reference

```java
public class Test {
    int x=10;
    int y=20;
    public void callByReference(Test t) {
        t.x=100;
        t.y=50;
    }
    public static void main(String[] args) {
        Test ts = new Test();
        System.out.println("Before "+ts.x+" "+ts.y);
        b.callByReference(ts);
        System.out.println("After "+ts.x+" "+ts.y);
    }
}
```

Output:

Before 10 20
After 100 50
**Method overloading**

If two or more methods in a class have the same name but different parameters, it is known as method overloading.

Method overloading is one of the ways through which Java supports polymorphism. Method overloading can be done by changing the number of arguments or by changing the data type of arguments. If two or more methods have the same name and same parameter list but differ in return type are not said to be overloaded methods.

---

**Different ways of Method overloading**

There are two different ways of method overloading.

**Method overloading by changing data type of Arguments**

*Example:*

```java
class Calculate {
    void sum (int a, int b) {
        System.out.println("sum is"+(a+b));
    }
    void sum (float a, float b) {
        System.out.println("sum is"+(a+b));
    }
    public static void main (String[] args) {
        Calculate cal = new Calculate();
    }
}
```
cal.sum (8, 5);  //sum(int a, int b) is method is called.
cal.sum (4.6, 3.8); //sum(float a, float b) is called.
}
}

Output:

Sum is 13
Sum is 8.4

You can see that sum() method is overloaded two times. The first takes two integer arguments, the second takes two float arguments.

---

**Method overloading by changing no. of argument.**

**Example:**

class Area
{
    void find(int l, int b)
    {
        System.out.println("Area is"+(l*b));
    }
    void find(int l, int b, int h)
    {
        System.out.println("Area is"+(l*b*h));
    }
    public static void main (String[] args)
    {
        Area ar = new Area();
In this example the find() method is overloaded twice. The first takes two arguments to calculate area, and the second takes three arguments to calculate area.

When an overloaded method is called java look for match between the arguments to call the method and the method's parameters. This match need not always be exact, sometime when exact match is not found, Java automatic type conversion plays a vital role.

Example of Method overloading with type promotion.

class Area
{
    void find(long l,long b)
    {
        System.out.println("Area is"+(l*b)) ;
    }
    void find(int l, int b,int h)
    {
        System.out.println("Area is"+(l*b*h));
    }
    public static void main (String[] args)
Constructors in Java

A constructor is a special method that is used to initialize an object. Every class has a constructor, if we don't explicitly declare a constructor for any java class the compiler builds a default constructor for that class. A constructor does not have any return type.

A constructor has same name as the class in which it resides. Constructor in Java can not be abstract, static, final or synchronized. These modifiers are not allowed for constructor.

class Car
{
  String name ;
  String model;
  Car( ) //Constructor
  {
    name ="";
    model="";
}
There are two types of Constructor

- Default Constructor
- Parameterized constructor

Each time a new object is created at least one constructor will be invoked.

```java
Car c = new Car(); //Default constructor invoked
Car c = new Car(name); //Parameterized constructor invoked
```

Constructor Overloading

Like methods, a constructor can also be overloaded. Overloaded constructors are differentiated on the basis of their type of parameters or number of parameters. Constructor overloading is not much different than method overloading. In case of method overloading you have multiple methods with same name but different signature, whereas in Constructor overloading you have multiple constructor with different signature but only difference is that Constructor doesn't have return type in Java.

Q. Why do we Overload constructors?

Constructor overloading is done to construct object in different ways.

Example of constructor overloading

class Cricketer
```
{ 
  String name;
  String team;
  int age;
  Cricketer ()    //default constructor.
  {
    name ="";
    team ="";
    age = 0;
  }
  Cricketer(String n, String t, int a)   //constructor overloaded
  {
    name = n;
    team = t;
    age = a;
  }
  Cricketer (Cricketer ckt)    //constructor similar to copy constructor of c++
  {
    name = ckt.name;
    team = ckt.team;
    age = ckt.age;
  }
  public String toString()
  {
    return "this is " + name + " of "+team;
  }
}
```
Class test:
{
  public static void main (String[] args)
  {
    Cricketer c1 = new Cricketer();
    Cricketer c2 = new Cricketer("sachin", "India", 32);
    Cricketer c3 = new Cricketer(c2);
    System.out.println(c2);
    System.out.println(c3);
    c1.name = "Virat";
    c1.team = "India";
    c1.age = 32;
    System.out.print in (c1);
  }
}

output:
this is sachin of india
this is sachin of india
this is virat of india
Q What's the difference between constructors and normal methods?

Constructors must have the same name as the class and can not return a value. They are only called once while regular methods could be called many times and it can return a value or can be void.

Q. What is constructor chaining in Java?

Constructor chaining is a phenomena of calling one constructor from another constructor of same class. Since constructor can only be called from another constructor in Java, constructor chaining is used for this purpose.

class Test
{
    Test()
    {
        this(10);
    }
    Test(int x)
    {
        System.out.println("x="+x);
    }
}

Q. Does constructors return any value?

Yes, constructors return current instant of a class. But yet constructor signature cannot have any return type.

Prof. Dhaval Thaker
**this keyword**

- **this** keyword is used to refer to current object.
- **this** is always a reference to the object on which method was invoked.
- **this** can be used to invoke current class constructor.
- **this** can be passed as an argument to another method.

*Example*:

```java
class Box
{
    Double width, weight, dept;
    Box (double w, double h, double d)
    {
        this.width = w;
        this.height = h;
        this.depth = d;
    }
}
```

Here the **this** is used to initialize member of current object.

---

**The this is used to call overloaded constructor in java**

```java
class Car
{
    private String name;
    public Car()
    {
```

Prof. Dhaval Thaker
this("BMW");    //overloaded constructor is called.
}

public Car(String n)
{
    this.name=n;   //member is initialized using this.
}

The this is also used to call Method of that class.

public void getName()
{
    System.out.println("Studytonight");
}

public void display()
{
    this.getName();
    System.out.println();
}

this is used to return current Object

public Car getCar()
{
    return this;
}
Garbage Collection

In Java destruction of object from memory is done automatically by the JVM. When there is no reference to an object, then that object is assumed to be no longer needed and the memory occupied by the object are released. This technique is called Garbage Collection. This is accomplished by the JVM. Unlike C++ there is no explicit need to destroy object.

Can the Garbage Collection be forced explicitly ?

No, the Garbage Collection can not be forced explicitly. We may request JVM for garbage collection by calling System.gc() method. But This does not guarantee that JVM will perform the garbage collection.

Advantages of Garbage Collection

1. Programmer doesn't need to worry about dereferencing an object.
2. It is done automatically by JVM.
3. Increases memory efficiency and decreases the chances for memory leak.

---

**finalize() method**

Sometime an object will need to perform some specific task before it is destroyed such as closing an open connection or releasing any resources held. To handle such situation `finalize()` method is used. `finalize()` method is called by garbage collection thread before collecting object. Its the last chance for any object to perform cleanup utility.

Signature of `finalize()` method

```java
protected void finalize()
{
    //finalize-code
}
```

---

**Some Important Points to Remember**

1. `finalize()` method is defined in `java.lang.Object` class, therefore it is available to all the classes.
2. `finalize()` method is declare as `protected` inside Object class.
3. `finalize()` method gets called only once by GC threads.

---

**gc() Method**

`gc()` method is used to call garbage collector explicitly. However `gc()` method does not guarantee that JVM will perform the garbage collection. It only request the JVM for garbage collection. This method is present in `System` and `Runtime` class.
Example for gc() method

```java
public class Test {

    public static void main(String[] args) {
        Test t = new Test();
        t = null;
        System.gc();
    }

    public void finalize() {
        System.out.println("Garbage Collected");
    }
}
```

Output:
Garbage Collected

Modifiers in Java

Modifiers are keywords that are added to change meaning of a definition. In Java, modifiers are categorized into two types,

1. Access control modifier
2. Non Access Modifier
1) **Access control modifier**

Java language has four access modifier to control access levels for classes, variable methods and constructor.

- **Default** : Default has scope only inside the same package
- **Public** : Public scope is visible everywhere
- **Protected** : Protected has scope within the package and all sub classes
- **Private** : Private has scope only within the classes
2) Non-access Modifier

Non-access modifiers do not change the accessibility of variables and methods, but they do provide them special properties. Non-access modifiers are of 5 types,

1. Final
2. Static
3. Transient
4. Synchronized
5. Volatile

**Final**

Final modifier is used to declare a field as final i.e. it prevents its content from being modified. Final field must be initialized when it is declared.

*Example:*

class Cloth
{
    final int MAX_PRICE = 999;  //final variable
    final int MIN_PRICE = 699;
    final void display()     //final method
    {
        System.out.println("Maxprice is" + MAX_PRICE);
        System.out.println("Minprice is" + MIN_PRICE);
    }
}
A class can also be declared as final. A class declared as final cannot be inherited. **String** class in java.lang package is an example of a final class. Method declared as final can be inherited but you cannot override (redefine) it.

---

**Static Modifier**

Static Modifiers are used to create class variables and class methods which can be accessed without instance of a class. Let's study how it works with variables and member functions.

**Static with Variables**

Static variables are defined as a class member that can be accessed without any object of that class. Static variable has only one single storage. All the object of the class having static variable will have the same instance of static variable. Static variables are initialized only once.

Static variable are used to represent common property of a class. It saves memory.

Suppose there are 100 employee in a company. All employee have its unique name and employee id but company name will be same all 100 employee. Here company name is the common property. So if you create a class to store employee detail, company_name field will be mark as static.

**Example**

```java
class Employee {
    int e_id;
    String name;
    static String company_name = "StudyTonight";
}
```
Example of static variable

class ST_Employee
{
    int eid;
    String name;
    static String company_name = "StudyTonight";
    public void show()
    {
        System.out.println(eid + " " + name + " " + company_name);
    }
    public static void main(String[] args)
    {
        ST_Employee se1 = new ST_Employee();
        se1.eid = 104;
        se1.name = "Abhijit";
        se1.show();
        ST_Employee se2 = new ST_Employee();
        se2.eid = 108;
        se2.name = "ankit";
        se2.show();
    }
}

Output

104 Abhijit StudyTonight
108 ankit StudyTonight

Prof. Dhaval Thaker
### Static variable vs Instance Variable

<table>
<thead>
<tr>
<th>Static variable</th>
<th>Instance Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Represent common property</td>
<td>Represent unique property</td>
</tr>
<tr>
<td>Accessed using class name</td>
<td>Accessed using object</td>
</tr>
<tr>
<td>get memory only once</td>
<td>get new memory each time a new object is created</td>
</tr>
</tbody>
</table>

**Example**

```java
class Company {
    static String company_name = "StudyTonight";
}

class Employee {
    int eid;
    String name;

    Employee(int e, String n) {
        this.eid = e;
        this.name = n;
    }
}

public class Main {
    public static void main(String[] args) {
        Employee se1 = new Employee(104, "Abhijit");
        Employee se2 = new Employee(108, "Ankit");

        System.out.println(se1.company_name);  // Output: StudyTonight
        System.out.println(se2.company_name);  // Output: StudyTonight
    }
}
```
public class Test
{
    static int x = 100;
    int y = 100;
    public void increment()
    {
        x++; y++;
    }
    public static void main( String[] args )
    {
        Test t1 = new Test();
        Test t2 = new Test();
        t1.increment();
        t2.increment();
        System.out.println(t2.y);
        System.out.println(Test.x); //accessed without any instance of class.
    }
}

Output

101
102

See the difference in value of two variable. Static variable y shows the changes made to it by increment() method on the different object. While instance variable x show only the change made to it by increment() method on that particular instance.
**Static Method**

A method can also be declared as static. Static methods do not need instance of its class for being accessed. `main()` method is the most common example of static method. `main()` method is declared as static because it is called before any object of the class is created.

*Example:*

class Test
{

    public static void square(int x)
    {
        System.out.println(x*x);
    }

    public static void main (String[] arg)
    {

        square(8)  //static method square () is called without any instance of class.
        
    }

}  

**Output: 64**

---

**Static block**

Static block is used to initialize static data member. Static block executes before `main()` method.
Example

class ST_Employee
{
    int eid;
    String name;
    static String company_name;

    static {
        company_name = "StudyTonight";  //static block invoked before main()
    }

    public void show()
    {
        System.out.println(eid+" "+name+" "+company_name);
    }

    public static void main( String[] args )
    {
        ST_Employee se1 = new ST_Employee();
        se1.eid = 104;
        se1.name = "Abhijit";
        se1.show();
    }
}
Q. Why a non-static variable cannot be referenced from a static context?

When you try to access a non-static variable from a static context like main method, java compiler throws a message like "a non-static variable cannot be referenced from a static context". This is because non-static variables are related with instance of class(object) and they get created when instance of a class is created by using `new` operator. So if you try to access a non-static variable without any instance compiler will complain because those variables are not yet created and they don't have any existence until an instance is created and associated with it.

Example of accessing non-static variable from a static context

class Test
{
    int x;
    public static void main(String[] args)
    {
        x=10;
    }
}

Output

compiler error: non-static variable count cannot be referenced from a static context

Same example using instance of class

class Test
{  
  int x;
  
  public static void main(String[] args)  
  {
    Test tt=new Test();
    tt.x=10;  //works fine with instance of class
  
  }
}

Q. Why main() method is static in java?

Because static methods can be called without any instance of a class and main() is called before any instance of a class is created.

Transient modifier

When an instance variable is declared as transient, then its value doesn't persist when an object is serialized.

Synchronized modifier

When a method is synchronized it can be accessed by only one thread at a time. We will discuss it in detail in Thread.

Volatile modifier

Volatile modifier tells the compiler that the volatile variable can be changed unexpectedly by other parts of your program. Volatile variables are used in case of multithreading program.
Inheritance (IS-A)

Inheritance is one of the key features of Object Oriented Programming. Inheritance provided mechanism that allowed a class to inherit property of another class. When a Class extends another class it inherits all non-private members including fields and methods. Inheritance in Java can be best understood in terms of Parent and Child relationship, also known as Super class (Parent) and Sub class (child) in Java language.

Inheritance defines is-a relationship between a Super class and its Sub class. extends and implements keywords are used to describe inheritance in Java.

Let us see how extend keyword is used to achieve Inheritance.

```java
class Vehicle {
    {
        ......
    }

class Car extends Vehicle {
    {
        .......    //extends the property of vehicle class.
    }
```
Now based on above example. In OOPs term we can say that,

- **Vehicle** is super class of **Car**.
- **Car** is sub class of **Vehicle**.
- Car IS-A Vehicle.

---

**Purpose of Inheritance**

1. To promote code reuse.
2. To use Polymorphism.

---

**Simple example of Inheritance**

class Parent
{
    public void p1()
    {
        System.out.println("Parent method");
    }
}

public class Child extends Parent {
    public void c1()
    {
        System.out.println("Child method");
    }

    public static void main(String[] args)
    {

Child cobj = new Child();
cobj.c1();  //method of Child class
cobj.p1();  //method of Parent class
}
}

Output
Child method
Parent method

Another example of Inheritance

class Vehicle
{
    String vehicleType;
}

public class Car extends Vehicle {

    String modelType;
    public void showDetail()
    {
        vehicleType = "Car";  //accessing Vehicle class member
        modelType = "sports";
        System.out.println(modelType+" "+vehicleType);
    }

    public static void main(String[] args)
    {

Car car = new Car();
car.showDetail();

Output

sports Car

**Types of Inheritance**

1. Single Inheritance
2. Multilevel Inheritance
3. Heirarchical Inheritance

**Note:** Multiple inheritance is not supported in Java
Why multiple inheritance is not supported in Java

- To remove ambiguity.
- To provide more maintainable and clear design.

Super keyword

In Java, `super` keyword is used to refer to immediate parent class of a class. In other words `super` keyword is used by a subclass whenever it need to refer to its immediate super class.
Example of Child class referring Parent class property using `super` keyword

class Parent {
    String name;
}

class Child extends Parent {
    String name;

    public void details() {
        super.name = "Parent";
        name = "Child";
    }
}

super.name = "Parent"; //refers to parent class member
name = "Child";
System.out.println(super.name + " and " + name);
public static void main(String[] args) {
    Child cobj = new Child();
    cobj.details();
}

Output
Parent and Child

Example of Child class refering Parent class methods using super keyword

class Parent {
    String name;
    public void details() {
        name = "Parent";
        System.out.println(name);
    }
}

public class Child extends Parent {
    String name;
    public void details() {
    }
super.details();  //calling Parent class details() method
name = "Child";
System.out.println(name);
}
public static void main(String[] args)
{
    Child cobj = new Child();
cobj.details();
}

Output

Parent
Child

Example of Child class calling Parent class constructor using super keyword

class Parent
{
    String name;

    public Parent(String n)
    {
        name = n;
    }
}
public class Child extends Parent {
    String name;

    public Child(String n1, String n2) {
        super(n1); //passing argument to parent class constructor
        this.name = n2;
    }

    public void details() {
        System.out.println(super.name + " and "+name);
    }

    public static void main(String[] args) {
        Child cobj = new Child("Parent","Child");
        cobj.details();
    }
}

Output

Parent and Child

Super class reference pointing to Sub class object.

In context to above example where Class B extends class A.
A a=new B();

is legal syntax because of IS-A relationship is there between class A and Class B.

---

**Q. Can you use both this() and super() in a Constructor?**

NO, because both super() and this() must be first statement inside a constructor. Hence we cannot use them together.

---

**Method Overriding**

When a method in a sub class has same name and type signature as a method in its super class, then the method is known as overridden method. Method overriding is also referred to as runtime polymorphism. The key benefit of overriding is the ability to **define method that's specific to a particular subclass type.**

---

**Example of Method Overriding**

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

class Dog extends Animal
{
    public void eat()    //eat() method overridden by Dog class.
    {

System.out.println("Dog eat meat");
}

As you can see here Dog class gives its own implementation of eat() method. Method must have same name and same type signature.

**NOTE**: Static methods cannot be overridden because, a static method is bounded with class where as instance method is bounded with object.

**Difference between Overloading and Overriding**

<table>
<thead>
<tr>
<th>Method Overloading</th>
<th>Method Overriding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter must be different and name must be same.</td>
<td>Both name and parameter must be same.</td>
</tr>
<tr>
<td>Compile time polymorphism.</td>
<td>Runtime polymorphism.</td>
</tr>
<tr>
<td>Increase readability of code.</td>
<td>Increase reusability of code.</td>
</tr>
<tr>
<td>Access specifier can be changed.</td>
<td>Access specifier most not be more restrictive than original method(can be less restrictive).</td>
</tr>
</tbody>
</table>
Q. Can we Override static method? Explain with reasons?

No, we cannot override static method. Because static method is bound to class whereas method overriding is associated with object i.e at runtime.

Command line argument in Java

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

Example

class cmd
{
    public static void main(String[] args)
Java Package

Package are used in Java, in-order to avoid name conflicts and to control access of class, interface and enumeration etc. A package can be defined as a group of similar types of
classes, interface, enumeration and sub-package. Using package it becomes easier to locate the related classes.

---

**Package are categorized into two forms**

- Built-in Package:-Existing Java package for example `java.lang`, `java.util` etc.
- User-defined-package:- Java package created by user to categorized classes and interface

---

**Creating a package**

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

```java
cpyackage mypack;
public class employee
{
```
The above statement create a package called `mypack`.

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

---

**Example of package creation**

```java
package mypack

class Book {
    String bookname;
    String author;
    Book(String b, String c) {
        this.bookname = b;
        this.author = c;
    }
    public void show() {
        System.out.println(bookname + " " + author);
    }
}

class test
```
To run this program:

- create a directory under your current working development directory (i.e. JDK directory), name it as mypack.
- compile the source file
- Put the class file into the directory you have created.
- Execute the program from development directory.

**NOTE**: Development directory is the directory where your JDK is install.

---

**Uses of Java package**

Package is a way to organize files in Java, it is used when a project consists of multiple modules. It also helps resolve naming conflicts. Package's access level also allows you to protect data from being used by the non-authorized classes.

---

**import keyword**

`import` keyword is used to import built-in and user-defined packages into your Java source file. So that your class can refer to a class that is in another package by directly using its name.
There are 3 different ways to refer to class that is present in different package

1. **Using fully qualified name** (But this is not a good practice.)

   Example:
   ```java
class MyDate extends java.util.Date{
    //statement;
}
```

2. import the only class you want to use.

   Example:
   ```java
   import java.util.Date;
class MyDate extends Date{
    //statement.
}
```

3. import all the classes from the particular package

   Example:
   ```java
   import java.util.*;
class MyDate extends Date{
    //statement;
}
```
import statement is kept after the package statement.

Example:

```java
package mypack;
import java.util.*;
```

But if you are not creating any package then import statement will be the first statement of your java source file.

---

**Static import**

`static import` is a feature that expands the capabilities of `import` keyword. It is used to import `static` member of a class. We all know that static member are referred in association with its class name outside the class. Using `static import`, it is possible to refer to the static member directly without its class name. There are two general form of static import statement.

- The first form of `static import` statement, import only a single static member of a class

  **Syntax**

  ```java
  import static package.class-name.static-member-name;
  ```

  **Example**

  ```java
  import static java.lang.Math.sqrt; //importing static method `sqrt` of `Math` class
  ```

- The second form of `static import` statement, imports all the static member of a class

  **Syntax**

  ```java
  import static package.class-type-name.*;
  ```

  **Example**
Example without using static import

```java
import static java.lang.Math.*;  //importing all static member of Math class

class Test
{
    public static void main(String[] args)
    {
        System.out.println(Math.sqrt(144));
    }
}

Output
12
```

Example using static import

```java
import static java.lang.Math.*;

class Test
{
    public static void main(String[] args)
    {
        System.out.println(sqrt(144));
    }
}

Output
12
```
Abstract class

If a class contain any abstract method then the class is declared as abstract class. An abstract class is never instantiated. It is used to provide abstraction. Although it does not provide 100% abstraction because it can also have concrete method.

Syntax :
abstract class class_name { }

Abstract method

Method that are declared without any body within an abstract class is known as abstract method. The method body will be defined by its subclass. Abstract method can never be final and static. Any class that extends an abstract class must implement all the abstract methods declared by the super class.

Syntax :
abstract return_type function_name ();   // No definition

Example of Abstract class

abstract class A
{
    abstract void callme();
}
class B extends A
{
    void callme()
    {
        System.out.println("this is callme.");
    
}
Abstract class with concrete(normal) method.

Abstract classes can also have normal methods with definitions, along with abstract methods.

abstract class A
{
    abstract void callme();
    public void normal()
    {
        System.out.println("this is concrete method");
    }
}
class B extends A
{
    void callme()
    {
        System.out.println("this is callme.");
    

output: this is callme.
Points to Remember

1. Abstract classes are not Interfaces. They are different, we will study this when we will study Interfaces.

2. An abstract class must have an abstract method.

3. Abstract classes can have Constructors, Member variables and Normal methods.

4. Abstract classes are never instantiated.

5. When you extend Abstract class with abstract method, you must define the abstract method in the child class, or make the child class abstract.
Abstraction using abstract class

Abstraction is an important feature of OOPS. It means hiding complexity. Abstract class is used to provide abstraction. Although it does not provide 100% abstraction because it can also have concrete method. Let's see how abstract class is used to provide abstraction.

```java
abstract class Vehicle
{
    public abstract void engine();
}

public class Car extends Vehicle {

    public void engine()
    {
        System.out.println("Car engine");
        //car engine implementation
    }

    public static void main(String[] args)
    {
        Vehicle v = new Car();
        v.engine();
    }
}
```

**Output**

Car engine
Here by casting instance of **Car** type to **Vehicle** reference, we are hiding the complexity of **Car** type under **Vehicle**. Now the **Vehicle** reference can be used to provide the implementation but it will hide the actual implementation process.

---

**When to use Abstract Methods & Abstract Class?**

Abstract methods are usually declared where two or more subclasses are expected to do a similar thing in different ways through different implementations. These subclasses extend the same Abstract class and provide different implementations for the abstract methods.

Abstract classes are used to define generic types of behaviors at the top of an object-oriented programming class hierarchy, and use its subclasses to provide implementation details of the abstract class.

**Interface**

Interface is a pure abstract class. They are syntactically similar to classes, but you cannot create instance of an **Interface** and their methods are declared without any body. Interface is used to achieve complete **abstraction** in Java. When you create an interface it defines what a class can do without saying anything about how the class will do it.

**Syntax:**

```java
interface interface_name { }
```

---

**Example of Interface**

```java
interface Moveable
{
    int AVERAGE-SPEED=40;
    void move();
}
```
NOTE: Compiler automatically converts methods of Interface as public and abstract, and the data members as public, static and final by default.

**Rules for using Interface**

- Methods inside Interface must not be static, final, native or strictfp.
- All variables declared inside interface are implicitly public static final variables (constants).
- All methods declared inside Java Interfaces are implicitly public and abstract, even if you don’t use public or abstract keyword.
- Interface can extend one or more other interface.
- Interface cannot implement a class.
- Interface can be nested inside another interface.
Example of Interface implementation

```java
interface Moveable {
    int AVG-SPEED = 40;
    void move();
}

class Vehicle implements Moveable {
    public void move() {
        System.out.printin("Average speed is" + AVG-SPEED);
    }
    public static void main(String[] arg) {
        Vehicle vc = new Vehicle();
        vc.move();
    }
}
```

Output:
Average speed is 40.

Interfaces supports Multiple Inheritance

Though classes in java doesn't support multiple inheritance, but a class can implement more than one interface.
interface Moveable
{
    boolean isMoveable();
}

interface Rollable
{
    boolean isRollable
}

class Tyre implements Moveable, Rollable
{
    int width;

    boolean isMoveable()
    {
        return true;
    }

    boolean isRollable()
    {
        return true;
    }
    public static void main(String args[])
    {
        Tyre tr = new Tyre();
System.out.println(tr.isMoveable());
System.out.println(tr.isRollable());
}
}

**Output:**
true
true

---

**Interface extends other Interface**

Classes implements interfaces, but an interface extends other interface.

```java
interface NewsPaper
{
    news();
}

interface Magazine extends NewsPaper
{
    colorful();
}
```

---

**Difference between an interface and an abstract class?**

<table>
<thead>
<tr>
<th>Abstract class</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Abstract class

Abstract class is a class which contain one or more abstract methods, which has to be implemented by its sub classes.

Interface

Interface is a Java Object containing method declaration but no implementation. The classes which implement the Interfaces must provide the method definition for all the methods.

Abstract class is a Class prefix with an abstract keyword followed by Class definition.

Interface is a pure abstract class which starts with interface keyword.

Abstract class can also contain concrete methods.

Whereas, Interface contains all abstract methods and final variable declarations.

Abstract classes are useful in a situation that Some general methods should be implemented and specialization behavior should be implemented by child classes.

Interfaces are useful in a situation that all properties should be implemented.

### Exception Handling

Exception Handling is the mechanism to handle runtime malfunctions. We need to handle such exceptions to prevent abrupt termination of program. The term exception means exceptional condition, it is a problem that may arise during the execution of program. A bunch of things can lead to exceptions, including programmer error, hardware failures, files that need to be opened cannot be found, resource exhaustion etc.
**Exception**

A Java Exception is an object that describes the exception that occurs in a program. When an exceptional events occurs in java, an exception is said to be thrown. The code that's responsible for doing something about the exception is called an **exception handler**.

---

**Exception class Hierarchy**

All exception types are subclasses of class **Throwable**, which is at the top of exception class hierarchy.

- **Exception** class is for exceptional conditions that program should catch. This class is extended to create user specific exception classes.
- **RuntimeException** is a subclass of Exception. Exceptions under this class are automatically defined for programs.
**Exception are categorized into 3 category.**

- **Checked Exception**
  
  The exception that can be predicted by the programmer. *Example*: File that need to be opened is not found. These type of exceptions must be checked at compile time.

- **Unchecked Exception**
  
  Unchecked exceptions are the class that extends RuntimeException. Unchecked exception are ignored at compile time. *Example*: ArithmeticException, NullPointerException, Array Index out of Bound exception. Unchecked exceptions are checked at runtime.

- **Error**
  
  Errors are typically ignored in code because you can rarely do anything about an error. *Example*: if stack overflow occurs, an error will arise. This type of error is not possible handle in code.

---

**Uncaught Exceptions**

When we don't handle the exceptions, they lead to unexpected program termination. Let's take an example for better understanding.

```java
class UncaughtException {
    public static void main(String args[]) {
        int a = 0;
        int b = 7/a;    // Divide by zero, will lead to exception
    }
}
```
This will lead to an exception at runtime, hence the Java run-time system will construct an exception and then throw it. As we don’t have any mechanism for handling exception in the above program, hence the default handler will handle the exception and will print the details of the exception on the terminal.

```java
java.lang.ArithmeticException: / by zero
    at UncaughtException.main(UncaughtException.java:4)
```

### Exception Handling Mechanism

In java, exception handling is done using five keywords,

1. `try`
2. `catch`
3. `throw`
4. `throws`
5. `finally`

Exception handling is done by transferring the execution of a program to an appropriate exception handler when exception occurs.

---

**Using try and catch**

Try is used to guard a block of code in which exception may occur. This block of code is called guarded region. A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in guarded code, the catch block that follows the try is
checked, if the type of exception that occurred is listed in the catch block then the exception is handed over to the catch block which then handles it.

---

**Example using Try and catch**

class Excp
{
    public static void main(String args[])
    {
        int a,b,c;
        try
        {
            a=0;
            b=10;
            c=b/a;
            System.out.println("This line will not be executed");
        }
        catch(ArithmeticException e)
        {
            System.out.println("Divided by zero");
        }
        System.out.println("After exception is handled");
    }
}

output:
Divided by zero
After exception is handled

An exception will be thrown by this program as we are trying to divide a number by zero inside try block. The program control is transferred outside try block. Thus the line "This line will not be executed" is never parsed by the compiler. The exception thrown is handled in catch block. Once the exception is handled the program controls continue with the next line in the program. Thus the line "After exception is handled" is printed.

Multiple catch blocks:

A try block can be followed by multiple catch blocks. You can have any number of catch blocks after a single try block. If an exception occurs in the guarded code the exception is passed to the first catch block in the list. If the exception type of exception, matches with the first catch block it gets caught, if not the exception is passed down to the next catch block. This continue until the exception is caught or falls through all catches.

Example for Multiple Catch blocks

class Excep
{
    public static void main(String[] args)
    {
        try
        {
            int arr[]={1,2};
            arr[2]=3/0;
        }
        catch(ArithmeticException ae)
        {
            System.out.println("divide by zero");
        }
catch(ArrayIndexOutOfBoundsException e)
{
    System.out.println("array index out of bound exception");
}
}

Output:
divide by zero

---

**Example for Unreachable Catch block**

While using multiple `catch` statements, it is important to remember that exception sub classes inside `catch` must come before any of their super classes otherwise it will lead to compile time error.

class Excep
{
    public static void main(String[] args)
    {
        try
        {
            int arr[]={1,2};
            arr[2]=3/0;
        }
        catch(Exception e)    //This block handles all Exception
        {
            System.out.println("Generic exception");
        }
    }
Nested try statement

try statement can be nested inside another block of try. Nested try block is used when a part of a block may cause one error while entire block may cause another error. In case if inner try block does not have a catch handler for a particular exception then the outer try is checked for match.

```java
class Excep {
    public static void main(String[] args) {
        try {
            int arr[] = {5, 0, 1, 2};
            try {
                int x = arr[3] / arr[1];
            } catch (ArithmeticException ae) {
                //This block is unreachable
                System.out.println("array index out of bound exception");
            }
        }
    }
}
```
System.out.println("divide by zero");

arr[4]=3;

catch(ArrayIndexOutOfBoundsException e)
{
    System.out.println("array index out of bound exception");
}

---

**Important points to Remember**

1. If you do not explicitly use the try catch blocks in your program, java will provide a default exception handler, which will print the exception details on the terminal, whenever exception occurs.

2. Super class `Throwable` overrides `toString()` function, to display error message in form of string.

3. While using multiple catch block, always make sure that exception subclasses comes before any of their super classes. Else you will get compile time error.

4. In nested try catch, the inner try block, uses its own catch block as well as catch block of the outer try, if required.

5. Only the object of Throwable class or its subclasses can be thrown.

**throw Keyword**
throw keyword is used to throw an exception explicitly. Only object of Throwable class or its sub classes can be thrown. Program execution stops on encountering throw statement, and the closest catch statement is checked for matching type of exception.

**Syntax:**

```java
throw ThrowableInstance
```

---

**Creating Instance of Throwable class**

There are two possible ways to get an instance of class Throwable,

1. Using a parameter in catch block.
2. Creating instance with `new` operator.
3. `new NullPointerException("test");`

   This constructs an instance of NullPointerException with name test.

---

**Example demonstrating throw Keyword**

```java
class Test {
    static void avg() {
        try {
            throw new ArithmeticException("demo");
        }
        catch(ArithmeticException e) {
```

---
In the above example the avg() method throw an instance of ArithmeticException, which is successfully handled using the catch statement.

**throws Keyword**

Any method capable of causing exceptions must list all the exceptions possible during its execution, so that anyone calling that method gets a prior knowledge about which exceptions to handle. A method can do so by using the `throws` keyword.

**Syntax :**

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

**NOTE :** It is necessary for all exceptions, except the exceptions of type `Error` and `RuntimeException`, or any of their subclass.
Example demonstrating throws Keyword

class Test
{
    static void check() throws ArithmeticException
    {
        System.out.println("Inside check function");
        throw new ArithmeticException("demo");
    }

    public static void main(String args[])
    {
        try
        {
            check();
        }
        catch(ArithmeticException e)
        {
            System.out.println("caught" + e);
        }
    }
}

finally clause

A finally keyword is used to create a block of code that follows a try block. A finally block of code always executes whether or not exception has occurred. Using a finally block, lets you
run any cleanup type statements that you want to execute, no matter what happens in the protected code. A finally block appears at the end of catch block.

Example demonstrating finally Clause

```java
Class ExceptionTest
{
    public static void main(String[] args)
    {
        int a[] = new int[2];
        System.out.println("out of try");
        try
        {
            System.out.println("Access invalid element"+ a[3]);
        }
    }
}
```
/* the above statement will throw ArrayIndexOutOfBoundsException */

}
finally
{
    System.out.println("finally is always executed.");
}

Output:
Out of try
finally is always executed.
Exception in thread main java. Lang. exception array Index out of bound exception.

You can see in above example even if exception is thrown by the program, which is not handled by catch block, still finally block will get executed.

**Introduction to Multithreading**

A program can be divided into a number of small processes. Each small process can be addressed as a single thread (a lightweight process). Multithreaded programs contain two or more threads that can run concurrently. This means that a single program can perform two or more tasks simultaneously. For example, one thread is writing content on a file at the same time another thread is performing spelling check.

In Java, the word thread means two different things.

- An instance of Thread class.
- or, A thread of execution.
An instance of `Thread` class is just an object, like any other object in java. But a thread of execution means an individual "lightweight" process that has its own call stack. In java each thread has its own call stack.

![Call Stack Diagram]

**The main thread**

Even if you don't create any thread in your program, a thread called `main` thread is still created. Although the `main` thread is automatically created, you can control it by obtaining a reference to it by calling `currentThread()` method.

Two important things to know about `main` thread are,

- It is the thread from which other threads will be produced.
- `main` thread must be always the last thread to finish execution.

```java
class MainThread {
    public static void main(String[] args) {
        Thread t=Thread.currentThread();
        t.setName("MainThread");
    }
}
```
System.out.println("Name of thread is "+t);
}
}

Output : Name of thread is Thread[MainThread,5,main]

---

**Life cycle of a Thread**

1. **New**: A thread begins its life cycle in the new state. It remains in this state until the start() method is called on it.

2. **Runnable**: After invocation of start() method on new thread, the thread becomes runnable.

3. **Running**: A method is in running thread if the thread scheduler has selected it.

4. **Waiting**: A thread is waiting for another thread to perform a task. In this stage the thread is still alive.

5. **Terminated**: A thread enter the terminated state when it complete its task.


Thread Priorities

Every thread has a priority that helps the operating system determine the order in which threads are scheduled for execution. In java thread priority ranges between,

- MIN-PRIORITY (a constant of 1)
- MAX-PRIORITY (a constant of 10)

By default every thread is given a NORM-PRIORITY(5). The main thread always have NORM-PRIORITY.

Thread Class

Thread class is the main class on which Java's Multithreading system is based. Thread class, along with its companion interface Runnable will be used to create and run threads for utilizing Multithreading feature of Java.

Constructors of Thread class

1. Thread ( )
2. Thread ( String str )
3. Thread ( Runnable r )
4. Thread ( Runnable r, String str)

You can create new thread, either by extending Thread class or by implementing Runnable interface. Thread class also defines many methods for managing threads. Some of them are,
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setName()</td>
<td>to give thread a name</td>
</tr>
<tr>
<td>getName()</td>
<td>return thread’s name</td>
</tr>
<tr>
<td>getPriority()</td>
<td>return thread’s priority</td>
</tr>
<tr>
<td>isAlive()</td>
<td>checks if thread is still running or not</td>
</tr>
<tr>
<td>join()</td>
<td>Wait for a thread to end</td>
</tr>
<tr>
<td>run()</td>
<td>Entry point for a thread</td>
</tr>
<tr>
<td>sleep()</td>
<td>suspend thread for a specified time</td>
</tr>
<tr>
<td>start()</td>
<td>start a thread by calling run() method</td>
</tr>
</tbody>
</table>

**Some Important points to Remember**

1. When we extend Thread class, we cannot override `setName()` and `getName()` functions, because they are declared final in Thread class.
2. While using `sleep()`, always handle the exception it throws.
Creating a thread

Java defines two ways by which a thread can be created.

- By implementing the `Runnable` interface.
- By extending the `Thread` class.

### Implementing the Runnable Interface

The easiest way to create a thread is to create a class that implements the runnable interface. After implementing runnable interface, the class needs to implement the `run()` method, which is of form,

```java
public void run()
```

- `run()` method introduces a concurrent thread into your program. This thread will end when `run()` returns.
- You must specify the code for your thread inside `run()` method.
- `run()` method can call other methods, can use other classes and declare variables just like any other normal method.

```java
class MyThread implements Runnable {

    public void run() {
        System.out.println("concurrent thread started running..");
    }
}
```
class MyThreadDemo
{
    public static void main( String args[] )
    {
        MyThread mt = new MyThread();
        Thread t = new Thread(mt);
        t.start();
    }
}

Output : concurrent thread started running..

To call the run() method, start() method is used. On calling start(), a new stack is provided to the thread and run() method is called to introduce the new thread into the program.

---

**Extending Thread class**

This is another way to create a thread by a new class that extends Thread class and create an instance of that class. The extending class must override run() method which is the entry point of new thread.

class MyThread extends Thread
{
    public void run()
    {
        System.out.println("Concurrent thread started running..");
    }
}
class MyThreadDemo
{

    public static void main( String args[] )
    {

        MyThread mt = new MyThread();
        mt.start();

    }
}

Output: concurrent thread started running..

In this case also, as we must override the `run()` and then use the `start()` method to start and run the thread. Also, when you create MyThread class object, Thread class constructor will also be invoked, as it is the super class, hence MyThread class object acts as Thread class object.

---

**What if we call run() method directly without using start() method?**

In above program if we directly call `run()` method, without using `start()` method,

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

    MyThread mt = new MyThread();
    mt.run();

}
```

Doing so, the thread won’t be allocated a new call stack, and it will start running in the current call stack, that is the call stack of the `main` thread. Hence Multithreading won’t be there.
Can we Start a thread twice?

No, a thread cannot be started twice. If you try to do so, `IllegalThreadStateException` will be thrown.

```java
public static void main( String args[] ) {
    MyThread mt = new MyThread();
    mt.start();
    mt.start(); //Exception thrown
}
```

When a thread is in running state, and you try to start it again, or any method try to invoke that thread again using `start()` method, exception is thrown.

Joining threads

Sometimes one thread needs to know when another thread is ending. In java, `isAlive()` and `join()` are two different methods to check whether a thread has finished its execution.
The `isAlive()` methods return `true` if the thread upon which it is called is still running otherwise it return `false`.

```java
final boolean isAlive()
```

But, `join()` method is used more commonly than `isAlive()`. This method waits until the thread on which it is called terminates.

```java
final void join() throws InterruptedException
```

Using `join()` method, we tell our thread to wait until the specified thread completes its execution. There are overloaded versions of `join()` method, which allows us to specify time for which you want to wait for the specified thread to terminate.

```java
final void join(long milliseconds) throws InterruptedException
```

---

**Example of `isAlive` method**

```java
public class MyThread extends Thread {

    public void run() {
        try{
            Thread.sleep(500);
        }catch(InterruptedException ie){}

        System.out.println("r1 ");
    }

    public void run() {
        System.out.println("r2 ");
    }
}
```
public static void main(String[] args) {

    MyThread t1=new MyThread();
    MyThread t2=new MyThread();
    t1.start();
    t2.start();

    System.out.println(t1.isAlive());
    System.out.println(t2.isAlive());
}

Output
r1
true
true
r1
r2
r2

Example of thread without join() method

public class MyThread extends Thread
In this above program two thread t1 and t2 are created. t1 starts first and after printing "r1" on console thread t1 goes to sleep for 500 mls. At the same time Thread t2 will start its process and print "r1" on console and then goes into sleep for 500 mls. Thread t1 will wake
up from sleep and print "r2" on console similarly thread t2 will wake up from sleep and print "r2" on console. So you will get output like \texttt{r1 r1 r2 r2}

\textbf{Example of thread with} \texttt{join()} \textbf{method}

```java
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();
}
}

Output
r1
r2
r1
r2
In this above program join() method on thread t1 ensure that t1 finishes it process before thread t2 starts.

---

**Specifying time with join()**

If in the above program, we specify time while using `join()` with `m1`, then `m1` will execute for that time, and then `m2` and `m3` will join it.

```
m1.join(1500);
```

Doing so, initially `m1` will execute for 1.5 seconds, after which `m2` and `m3` will join it.

**Synchronization**

At times when more than one thread try to access a shared resource, we need to ensure that resource will be used by only one thread at a time. The process by which this is achieved is called **synchronization**. The synchronization keyword in java creates a block of code referred to as critical section.

Every Java object with a critical section of code gets a lock associated with the object. To enter critical section a thread need to obtain the corresponding object’s lock.

**General Syntax :**

```
synchronized (object)
```
Why we use Synchronization?

If we do not use synchronization, and let two or more threads access a shared resource at the same time, it will lead to distorted results.

Consider an example, Suppose we have two different threads $T_1$ and $T_2$, $T_1$ starts execution and save certain values in a file `$temporary.txt` which will be used to calculate some result when $T_1$ returns. Meanwhile, $T_2$ starts and before $T_1$ returns, $T_2$ change the values saved by $T_1$ in the file `$temporary.txt` ($temporary.txt$ is the shared resource). Now obviously $T_1$ will return wrong result.

To prevent such problems, synchronization was introduced. With synchronization in above case, once $T_1$ starts using `$temporary.txt` file, this file will be locked (LOCK mode), and no other thread will be able to access or modify it until $T_1$ returns.

Using Synchronized Methods

Using Synchronized methods is a way to accomplish synchronization. But lets first see what happens when we do not use synchronization in our program.

Example with no Synchronization

class First
{
    public void display(String msg)
    {

System.out.print ("["+msg);
try
{
    Thread.sleep(1000);
}
catch(InterruptedException e)
{
    e.printStackTrace();
}
System.out.println ("]");
}

class Second extends Thread
{
    String msg;
    First fobj;
    Second (First fp,String str)
    {
        fobj = fp;
        msg = str;
        start();
    }
    public void run()
    {
        fobj.display(msg);
    }
public class Syncro {
    public static void main (String[] args) {
        First fnew = new First();
        Second ss = new second(fnew, "welcome");
        Second ss1 = new second(fnew, "new");
        Second ss2 = new second(fnew, "programmer");
    }
}

Output :
[welcome [ new [ programmer]
]

In the above program, object fnew of class First is shared by all the three running threads(ss, ss1 and ss2) to call the shared method(void display). Hence the result is unsynchronized and such situation is called Race condition.

**Synchronized Keyword**

To synchronize above program, we must serialize access to the shared display() method, making it available to only one thread at a time. This is done by using keyword synchronized with display() method.

```java
synchronized void display (String msg)
```
Using Synchronised block

If you have to synchronize access to object of a class that has no synchronized methods, and you cannot modify the code. You can use synchronized block to use it.

class First
{
    public void display(String msg)
    {
        System.out.print ("[")+msg);
        try
        {
            Thread.sleep(1000);
        }
        catch(InterruptedException e)
        {
            e.printStackTrace();
        }
        System.out.println ("]");
    }
}

class Second extends Thread
{
    String msg;
    First fobj;
    Second (First fp,String str)
public class Syncro
{
    public static void main (String[] args)
    {
        First fnew = new First();
        Second ss = new second(fnew, "welcome");
        Second ss1= new second (fnew,"new");
        Second ss2 = new second(fnew, "programmer");
    }
}

Output :
[welcome]
Because of synchronized block this program gives the expected output.

**Applet in Java**

- Applets are small Java applications that can be accessed on an Internet server, transported over Internet, and can be automatically installed and run as apart of a web document. Any applet in Java is a class that extends the `java.applet.Applet` class.
- An Applet class does not have any `main()` method.
- It is viewed using JVM. The JVM can use either a plug-in of the Web browser or a separate runtime environment to run an applet application.
- JVM creates an instance of the applet class and invokes `init()` method to initialize an Applet.

```java
A Simple Applet

import java.awt.*;
import java.applet.*;
public class Simple extends Applet
{
    public void paint(Graphics g)
    {
        g.drawString("A simple Applet", 20, 20);
    }
}
```
Every **Applet** application must declare a `paint()` method. This method is defined by **AWT** class and must be overridden by the applet. `paint()` method is called each time an applet needs to redisplay its output. Another important thing to notice about applet application is that, execution of an applet does not begin at `main()` method. In fact an applet application does not have any `main()` method.

---

**Advantages of Applets**

1. Very less response time as it works on the client side.
2. Can be run using any browser, which has JVM running in it.

---

**Applet class**

Applet class provides all necessary support for applet execution, such as initializing and destroying of applet. It also provide methods that load and display images and methods that load and play audio clips.

**An Applet Skeleton**

Most applets override these four methods. These four methods form Applet lifecycle.
• **init()**: init() is the first method to be called. This is where variables are initialized. This method is called only once during the runtime of applet.

• **start()**: start() method is called after init(). This method is called to restart an applet after it has been stopped.

• **stop()**: stop() method is called to suspend thread that does not need to run when applet is not visible.

• **destroy()**: destroy() method is called when your applet needs to be removed completely from memory.

---

**Example of an Applet Skeleton**

```java
import java.awt.*;
import java.applet.*;

public class AppletTest extends Applet
{
    public void init()
    {
        //initialization
    }

    public void start ()
    {
        //start or resume execution
    }

    public void stop()
    {
        //suspend execution
    }
}
```
Example of an Applet

```java
import java.applet.*;
import java.awt.*;

public class MyApplet extends Applet {

    int height, width;
    public void init() {
        height = getSize().height;
        width = getSize().width;
        setName("MyApplet");
    }

    public void paint(Graphics g) {
```

```java
```
How to run an Applet Program

An Applet program is compiled in the same way as you have been compiling your console programs. However there are two ways to run an applet.

- Executing the Applet within Java-compatible web browser.
- Using an Applet viewer, such as the standard tool, applet viewer. An applet viewer executes your applet in a window

For executing an Applet in an web browser, create short HTML file in the same directory. Inside body tag of the file, include the following code. (applet tag loads the Applet class)

```html
<applet code="MyApplet" width=400 height=400 />
</applet>
```
Run the HTML file

Running Applet using Applet Viewer

To execute an Applet with an applet viewer, write short HTML file as discussed above. If name it as run.htm, then the following command will run your applet program.

f:/appletviewer run.htm
Event Handling

Any program that uses GUI (graphical user interface) such as Java application written for windows, is event driven. Event describes the change of state of any object. **Example** : Pressing a button, Entering a character in Textbox.

---

**Components of Event Handling**

Event handling has three main components,

- **Events** : An event is a change of state of an object.
- **Events Source** : Event source is an object that generates an event.
- **Listeners** : A listener is an object that listens to the event. A listener gets notified when an event occurs.
How Events are handled?

A source generates an Event and sends it to one or more listeners registered with the source. Once the event is received by the listener, they process the event and then return. Events are supported by a number of Java packages, like `java.util`, `java.awt`, and `java.awt.event`.

### Important Event Class and Interface

<table>
<thead>
<tr>
<th>Event Class</th>
<th>Description</th>
<th>Listener Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionEvent</td>
<td>generated when button is pressed, menu-item is selected, list-item is double clicked</td>
<td>ActionListener</td>
</tr>
<tr>
<td>MouseEvent</td>
<td>generated when mouse is dragged, moved, clicked, pressed or released also when the enters or exit a component</td>
<td>MouseListener</td>
</tr>
<tr>
<td>KeyEvent</td>
<td>generated when input is received from keyboard</td>
<td>KeyListener</td>
</tr>
<tr>
<td>ItemEvent</td>
<td>generated when check-box or list item is clicked</td>
<td>ItemListener</td>
</tr>
<tr>
<td>TextEvent</td>
<td>generated when value of textarea or textfield is changed</td>
<td>TextListener</td>
</tr>
<tr>
<td>MouseWheelEvent</td>
<td>generated when mouse wheel is moved</td>
<td>MouseWheelListener</td>
</tr>
<tr>
<td>WindowEvent</td>
<td>generated when window is activated, deactivated, deiconified, iconified, opened or closed</td>
<td>WindowListener</td>
</tr>
</tbody>
</table>
### ComponentEvent
Generated when component is hidden, moved, resized or set visible

### ContainerEvent
Generated when component is added or removed from container

### AdjustmentEvent
Generated when scroll bar is manipulated

### FocusEvent
Generated when component gains or loses keyboard focus

#### Example of Event Handling

```java
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.applet.*;
import java.awt.event.*;
import java.awt.*;

public class Test extends Applet implements KeyListener
{
    String msg="";
    public void init()
    {
        addKeyListener(this);
    }
```
public void keyPressed(KeyEvent k)
{
    showStatus("KeyPressed");
}

public void keyReleased(KeyEvent k)
{
    showStatus("KeyRealesed");
}

public void keyTyped(KeyEvent k)
{
    msg = msg+k.getKeyChar();
    repaint();
}

public void paint(Graphics g)
{
    g.drawString(msg, 20, 40);
}

HTML code :
<applet code="Test" width=300, height=100>
AWT

AWT contains large number of classes and methods that allows you to create and manage windows GUI application. AWT is the foundation upon which Swing is made. It is used for GUI programming in Java. But now a days it is merely used because most GUI java programs are implemented using Swing because of its rich implementation of GUI controls and light-weighted nature.
**Component class**

Component class is at the top of AWT hierarchy. Component is an abstract class that encapsulates all attributes of visual component. A component object is responsible for remembering the current foreground and background colours and the currently selected text font.

**Container**

Container is a component in AWT that contains another component like button, text field, tables etc. Container is a subclass of component class. Container class keeps track of components that are added to another component.

**Panel**

Panel class is a concrete subclass of Container. Panel does not contain title bar, menu bar or border.
**Window class**

Window class creates a top level window. Window does not have borders and menubar.

---

**Frame**

Frame is a sub class of Window and have resizing canvas. It is a container that contain several different components like button, title bar, textfield, label etc. In Java, most of the AWT applications are created using Frame window. Frame class has two different constructors,

```java
Frame() throws HeadlessException

Frame(String title) throws HeadlessException
```

---

**Creating a Frame**

There are two ways to create a Frame. They are,

1. By Instantiating Frame class
2. By extending Frame class

---

**Creating Frame Window by Instantiating Frame class**

```java
import java.awt.*;

public class Testawt
{
    Testawt()
```
{  
    Frame fm=new Frame(); //Creating a frame.  
    Label lb = new Label("welcome to java graphics"); //Creating a label  
    fm.add(lb); //adding label to the frame.  
    fm.setSize(300, 300); //setting frame size.  
    fm.setVisible(true); //set frame visibility true.  
}

public static void main(String args[])  
{
    Testawt ta = new Testawt();  
}
}
Creating Frame window by extending Frame class

```java
package testawt;

import java.awt.*;
import java.awt.event.);

public class Testawt extends Frame
{
    public Testawt()
    {
        Button btn=new Button("Hello World");
        add(btn); //adding a new Button.
        setSize(400, 500); //setting size.
        setTitle("StudyTonight"); //setting title.
        setLayout(new FlowLayout()); //set default layout for frame.
        setVisible(true); //set frame visibility true.
    }

    public static void main (String[] args)
    {
        Testawt ta = new Testawt(); //creating a frame.
    }
}
```