# NAME: DAVID OWINO

# REG NO: SCM211-1359/2021

# UNIT: OOPII

# ASSIGNMENT

## **Abstract class in java**

A class which is declared with the abstract keyword is known as an abstract class in [java](https://www.javatpoint.com/java-tutorial)

. It can have abstract and non-abstract methods (method with the body).

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

**Abstraction in java**

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

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

Abstraction lets you focus on what the [object does instead of how it does it.](https://www.javatpoint.com/object-and-class-in-java)

Ways to achieve abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

Abstract class in java

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

**Rules for java Abstract class**

* An abstract class must be declared with an abstract keyword.
* It can have abstract and non-abstract methods.
* It cannot be instantiated.
* It can have  [constructors](https://www.javatpoint.com/java-constructor)

and static methods also.

* It can have final methods which will force the subclass not to change the body of the method.

**Example of abstract class**

1. **Abstract class** A{}

**Abstract Method in java**

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

**Example of abstract method**

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

Example of Abstract class that has an abstract method

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

1. **abstract** **class** Bike{
2. **abstract** **void** run();  //abstract method run
3. }
4. **class** Honda4 **extends** Bike{
5. **void** run(){System.out.println("running safely");}
6. **public** **static** **void** main(String args[]){
7. Bike obj = **new** Honda4();
8. obj.run();
9. }
10. }

Understanding the real scenario of Abstract class

Another example of abstract class in java.

1. **abstract** **class** Bank{
2. **abstract** **int** getRateOfInterest();
3. }
4. **class** SBI **extends** Bank{
5. **int** getRateOfInterest(){**return** 7;}
6. }
7. **class** PNB **extends** Bank{
8. **int** getRateOfInterest(){**return** 8;}
9. }
10. **class** TestBank{
11. **public** **static** **void** main(String args[]){
12. Bank b;
13. b=**new** SBI();
14. System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
15. b=**new** PNB();
16. System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
17. }}

Rate of Interest is: 7 %

Rate of Interest is: 8 %

Abstract class having constructor

An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

1. //Example of an abstract class that has abstract and non-abstract methods
2. **abstract** **class** Bike{
3. Bike(){System.out.println("bike is created");}
4. **abstract** **void** run();
5. **void** changeGear(){System.out.println("gear changed");}
6. }
7. //Creating a Child class which inherits Abstract class
8. **class** Honda **extends** Bike{
9. **void** run(){System.out.println("running safely..");}
10. }
11. //Creating a Test class which calls abstract and non-abstract methods
12. **class** TestAbstraction2{
13. **public** **static** **void** main(String args[]){
14. Bike obj = **new** Honda();
15. obj.run();
16. obj.changeGear();
17. }
18. }

bike is created

running safely..

gear changed

Rule: If there is an abstract method in a class ,that class must be abstract.

Rule:if you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.

**Another real scenario of abstract class**

The abstract class can also be used to provide some implementation of the [interface](https://www.javatpoint.com/interface-in-java)

. In such case, the end user may not be forced to override all the methods of the interface. *Note: If you are beginner to java, learn interface first and skip this example.*

1. **interface** A{
2. **void** a();
3. **void** b();
4. **void** c();
5. **void** d();
6. }
7. **abstract** **class** B **implements** A{
8. **public** **void** c(){System.out.println("I am c");}
9. }
10. **class** M **extends** B{
11. **public** **void** a(){System.out.println("I am a");}
12. **public** **void** b(){System.out.println("I am b");}
13. **public** **void** d(){System.out.println("I am d");}
14. }
15. **class** Test5{
16. **public** **static** **void** main(String args[]){
17. A a=**new** M();
18. a.a();
19. a.b();
20. a.c();
21. a.d();
22. }}

Output:I am a

I am b

I am c

I am d

Abstract Class in Java does the process of hiding the intricate code implementation details from the user and just provides the user with the necessary information. This phenomenon is called **Data Abstraction in Object-Oriented Programming (java)**

**What is an Abstract Class?**

Generally, an abstract class in Java is a template that stores the data members and methods that we use in a program. Abstraction in Java keeps the user from viewing complex code implementations and provides the user with necessary information.

We cannot instantiate the abstract class in Java directly. Instead, we can subclass the abstract class. When we use an abstract class as a subclass, the abstract class method implementation becomes available to all of its parent classes.

**Features of Abstract Class**

Template

The abstract class in Java enables the best way to execute the process of data abstraction by providing the developers with the option of hiding the code implementation. It also presents the end-user with a template that explains the methods involved.

Loose Coupling

Data abstraction in Java enables loose coupling, by reducing the dependencies at an exponential level.

Code Reusability

Using an abstract class in the code saves time. We can call the abstract method wherever the method is necessary. Abstract class avoids the process of writing the same code again.

Abstraction

Data abstraction in Java helps the developers hide the code complications from the end-user by reducing the project's complete characteristics to only the necessary components.

Dynamic Resolution

Using the support of dynamic method resolution, developers can solve multiple problems with the help of one abstract method.

**The Syntax for Abstract Class**

To declare an abstract class, we use the access modifier first, then the "abstract" keyword, and the class name shown below.

//Syntax:

<Access\_Modifier> abstract class <Class\_Name> {

//Data\_Members;

//Statements;

//Methods;

}

**Examples for Abstract Classes**

The following programs are a few examples of the abstract class in Java.

//Example 1:

package abstraction;

public abstract class Person {

private String Name;

private String Gender;

public Person(String nm, String Gen) {

this.Name = nm;

this.Gender = Gen;

}

public abstract void work();

@Override

public String toString() {

return "Name=" + this.Name + "::Gender=" + this.Gender;

}

public void changeName(String newName) {

this.Name = newName;

}

public void Exam() {

// TODO Auto-generated method stub

}

public void Office() {

// TODO Auto-generated method stub

}

}

package abstraction;

public class Employee extends Person {

private int EmpId;

public Employee(String EmployeeName, String Gen, int EmployeeID) {

super(EmployeeName, Gen);

this.EmpId = EmployeeID;

}

public void Office() {

if (EmpId == 0) {

System.out.println("Employee Logged Out");

} else {

System.out.println("Employee Logged In");

}

}

public static void main(String args[]) {

Person employee = new Employee("Pavithra", "Female", 1094826);

employee.Office();

employee.changeName("Pavithra Tripathy");

System.out.println(employee.toString());

}

@Override

public void work() {

// TODO Auto-generated method stub

}

}

//Output:

Employee Logged In

Name=Pavithra Tripathy::Gender=Female

There are certain rules that one should know while declaring an abstract class. Let us explore them in detail.

**Rules to Declare Abstract Class**

* The keyword "abstract" is mandatory while declaring an abstract class in Java.
* Abstract classes cannot be instantiated directly.
* An abstract class must have at least one abstract method.
* An abstract class includes final methods.
* An abstract class may also include non-abstract methods.
* An abstract class can consist of constructors and static methods.

**Procedure to Achieve Abstraction in Java**

The process of Data Abstraction in Java is possible in two different ways. The first method is obviously by using the abstract class in Java, and the other one is by using an interface.

**Interface**

An interface in Java is a boundary between the method and the class implementing it. An interface in Java holds the method signature in it, but never the implementation of the method. In Java, we use the interface to achieve abstraction.

//Syntax:

interface <Class\_Name>{

//Method\_Signatures;

}

//Example:

//Interface

package simplilearn;

public interface Area {

public void Square();

public void Circle();

public void Rectangle();

public void Triangle();

}

//Class

package simplilearn;

import java.util.Scanner;

public class shapeArea implements Area {

public void Circle() {

Scanner kb = new Scanner(System.in);

System.out.println("Enter the radius of the circle");

double r = kb.nextInt();

double areaOfCircle = 3.142 \* r \* r;

System.out.println("Area of the circle is" + areaOfCircle);

}

@Override

public void Square() {

// TODO Auto-generated method stub

Scanner kb2 = new Scanner(System.in);

System.out.println("Value of the side the square");

double s = kb2.nextInt();

double areaOfSquare = s \* s;

System.out.println("Area of the square is" + areaOfSquare);

}

@Override

public void Rectangle() {

// TODO Auto-generated method stub

Scanner kb3 = new Scanner(System.in);

System.out.println("Enter the length of the Rectangle");

double l = kb3.nextInt();

System.out.println("Enter the breadth of the Rectangle");

double b = kb3.nextInt();

double areaOfRectangle = l \* b;

System.out.println("Area of the Rectangle is" + areaOfRectangle);

}

@Override

public void Triangle() {

// TODO Auto-generated method stub

Scanner kb4 = new Scanner(System.in);

System.out.println("Enter the base of the Triangle");

double base = kb4.nextInt();

System.out.println("Enter the height of the Triangle");

double h = kb4.nextInt();

double areaOfTriangle = 0.5 \* base \* h;

System.out.println("Area of the Triangle is" + areaOfTriangle);

}

public static void main(String[] args) {

shapeArea geometry = new shapeArea();

geometry.Circle();

geometry.Square();

geometry.Rectangle();

geometry.Triangle();

}

}

//Output:

Enter the radius of the circle

5

Area of the circle is78.55

Enter the length of the side of the square

10

Area of the square is100.0

Enter the length of the Rectangle

25

Enter the breadth of the Rectangle

45

Area of the Rectangle is1125.0

Enter the base of the Triangle

20

Enter the height of the Triangle

25

Area of the Triangle is250.0

**Interface v/s Abstract class**

Though an interface and abstract class perform a similar operation of data abstraction in Java, some differences separate them. The differences between them are as follows:

|  |  |
| --- | --- |
| **Interface** | **Abstract Class** |
| Keyword used: interface | Keyword used: abstract |
| Subclasses can implement an interface | Subclasses have to extend abstract class |
| Multiple interfaces can be implemented | One abstract class can be extended |
| Supports Multiple Inheritance | Cannot support Multiple Inheritance |

**Advantages of Abstract classes**

* Abstract class in Java is highly beneficial in writing shorter codes
* Abstraction in Java avoids code duplication
* Abstract classes enable code reusability
* Changes to internal code implementation are done without affecting classes

**Disadvantages of Abstract Classes**

* Abstraction in Java is expensive, as sometimes you need to handle cases and situations which are not always necessary
* Object-relational impedance mismatch, in case of RDBMS
* Object-relational mapping occurs in case of frameworks like hibernate

## 2.Anonymous classes

Anonymous classes enable you to make your code more concise. They enable you to declare and instantiate a class at the same time. They are like local classes except that they do not have a name. Use them if you need to use a local class only once.

**Declaring Anonymous Classes**

While local classes are class declarations, anonymous classes are expressions, which means that you define the class in another expression. The following example, [HelloWorldAnonymousClasses](https://docs.oracle.com/javase/tutorial/java/javaOO/examples/HelloWorldAnonymousClasses.java), uses anonymous classes in the initialization statements of the local variables frenchGreeting and spanishGreeting, but uses a local class for the initialization of the variable englishGreeting:

public class HelloWorldAnonymousClasses {

interface HelloWorld {

public void greet();

public void greetSomeone(String someone);

}

public void sayHello() {

class EnglishGreeting implements HelloWorld {

String name = "world";

public void greet() {

greetSomeone("world");

}

public void greetSomeone(String someone) {

name = someone;

System.out.println("Hello " + name);

}

}

HelloWorld englishGreeting = new EnglishGreeting();

HelloWorld frenchGreeting = new HelloWorld() {

String name = "tout le monde";

public void greet() {

greetSomeone("tout le monde");

}

public void greetSomeone(String someone) {

name = someone;

System.out.println("Salut " + name);

}

};

HelloWorld spanishGreeting = new HelloWorld() {

String name = "mundo";

public void greet() {

greetSomeone("mundo");

}

public void greetSomeone(String someone) {

name = someone;

System.out.println("Hola, " + name);

}

};

englishGreeting.greet();

frenchGreeting.greetSomeone("Fred");

spanishGreeting.greet();

}

public static void main(String... args) {

HelloWorldAnonymousClasses myApp =

new HelloWorldAnonymousClasses();

myApp.sayHello();

}

}

**Syntax of Anonymous Classes**

As mentioned previously, an anonymous class is an expression. The syntax of an anonymous class expression is like the invocation of a constructor, except that there is a class definition contained in a block of code.

Consider the instantiation of the frenchGreeting object:

HelloWorld frenchGreeting = new HelloWorld() {

String name = "tout le monde";

public void greet() {

greetSomeone("tout le monde");

}

public void greetSomeone(String someone) {

name = someone;

System.out.println("Salut " + name);

}

};

The anonymous class expression consists of the following:

* The new operator
* The name of an interface to implement or a class to extend. In this example, the anonymous class is implementing the interface HelloWorld.
* Parentheses that contain the arguments to a constructor, just like a normal class instance creation expression. **Note**: When you implement an interface, there is no constructor, so you use an empty pair of parentheses, as in this example.
* A body, which is a class declaration body. More specifically, in the body, method declarations are allowed but statements are not.

Because an anonymous class definition is an expression, it must be part of a statement. In this example, the anonymous class expression is part of the statement that instantiates the frenchGreeting object. (This explains why there is a semicolon after the closing brace.)

**Accessing Local Variables of the Enclosing Scope, and Declaring and Accessing Members of the Anonymous Class**

Like local classes, anonymous classes can [capture variables](https://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html#accessing-members-of-an-enclosing-class); they have the same access to local variables of the enclosing scope:

* An anonymous class has access to the members of its enclosing class.
* An anonymous class cannot access local variables in its enclosing scope that are not declared as final or effectively final.
* Like a nested class, a declaration of a type (such as a variable) in an anonymous class shadows any other declarations in the enclosing scope that have the same name. See [Shadowing](https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html#shadowing) for more information.

Anonymous classes also have the same restrictions as local classes with respect to their members:

* You cannot declare static initializers or member interfaces in an anonymous class.
* An anonymous class can have static members provided that they are constant variables.

Note that you can declare the following in anonymous classes:

* Fields
* Extra methods (even if they do not implement any methods of the supertype)
* Instance initializers
* Local classes

However, you cannot declare constructors in an anonymous class.

**Examples of Anonymous Classes**

Anonymous classes are often used in graphical user interface (GUI) applications.

Consider the JavaFX example [HelloWorld.java](https://docs.oracle.com/javase/8/javafx/get-started-tutorial/hello_world.htm) (from the section [Hello World, JavaFX Style](https://docs.oracle.com/javase/8/javafx/get-started-tutorial/hello_world.htm) from [Getting Started with JavaFX](https://docs.oracle.com/javase/8/javafx/get-started-tutorial/javafx_get_started.htm)). This sample creates a frame that contains a **Say 'Hello World'** button. The anonymous class expression is highlighted:

import javafx.event.ActionEvent;

import javafx.event.EventHandler;

import javafx.scene.Scene;

import javafx.scene.control.Button;

import javafx.scene.layout.StackPane;

import javafx.stage.Stage;

public class HelloWorld extends Application {

public static void main(String[] args) {

launch(args);

}

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("Hello World!");

Button btn = new Button();

btn.setText("Say 'Hello World'");

btn.setOnAction(**new EventHandler<ActionEvent>() {**

**@Override**

**public void handle(ActionEvent event) {**

**System.out.println("Hello World!");**

**}**

**}**);

StackPane root = new StackPane();

root.getChildren().add(btn);

primaryStage.setScene(new Scene(root, 300, 250));

primaryStage.show();

}

}

In this example, the method invocation btn.setOnAction specifies what happens when you select the **Say 'Hello World'** button. This method requires an object of type EventHandler<ActionEvent>. The EventHandler<ActionEvent> interface contains only one method, handle. Instead of implementing this method with a new class, the example uses an anonymous class expression. Notice that this expression is the argument passed to the btn.setOnAction method.

Because the EventHandler<ActionEvent> interface contains only one method, you can use a lambda expression instead of an anonymous class expression. See the section [Lambda Expressions](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html) for more information.

Anonymous classes are ideal for implementing an interface that contains two or more methods. The following JavaFX example is from the section [Customization of UI Controls](https://docs.oracle.com/javase/8/javafx/user-interface-tutorial/custom.htm). The highlighted code creates a text field that only accepts numeric values. It redefines the default implementation of the TextField class with an anonymous class by overriding the replaceText and replaceSelection methods inherited from the TextInputControl class.

import javafx.application.Application;

import javafx.event.ActionEvent;

import javafx.event.EventHandler;

import javafx.geometry.Insets;

import javafx.scene.Group;

import javafx.scene.Scene;

import javafx.scene.control.\*;

import javafx.scene.layout.GridPane;

import javafx.scene.layout.HBox;

import javafx.stage.Stage;

public class CustomTextFieldSample extends Application {

final static Label label = new Label();

@Override

public void start(Stage stage) {

Group root = new Group();

Scene scene = new Scene(root, 300, 150);

stage.setScene(scene);

stage.setTitle("Text Field Sample");

GridPane grid = new GridPane();

grid.setPadding(new Insets(10, 10, 10, 10));

grid.setVgap(5);

grid.setHgap(5);

scene.setRoot(grid);

final Label dollar = new Label("$");

GridPane.setConstraints(dollar, 0, 0);

grid.getChildren().add(dollar);

final TextField sum = **new TextField() {**

**@Override**

**public void replaceText(int start, int end, String text) {**

**if (!text.matches("[a-z, A-Z]")) {**

**super.replaceText(start, end, text);**

**}**

**label.setText("Enter a numeric value");**

**}**

**@Override**

**public void replaceSelection(String text) {**

**if (!text.matches("[a-z, A-Z]")) {**

**super.replaceSelection(text);**

**}**

**}**

**};**

sum.setPromptText("Enter the total");

sum.setPrefColumnCount(10);

GridPane.setConstraints(sum, 1, 0);

grid.getChildren().add(sum);

Button submit = new Button("Submit");

GridPane.setConstraints(submit, 2, 0);

grid.getChildren().add(submit);

submit.setOnAction(new EventHandler<ActionEvent>() {

@Override

public void handle(ActionEvent e) {

label.setText(null);

}

});

GridPane.setConstraints(label, 0, 1);

GridPane.setColumnSpan(label, 3);

grid.getChildren().add(label);

scene.setRoot(grid);

stage.show();

}

public static void main(String[] args) {

launch(args);

}

}

## 3. INNER CLASSES

**Java inner class** or nested class is a class that is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place to be more readable and maintainable.

Additionally, it can access all the members of the outer class, including private data members and methods.

Syntax of Inner class

1. **class** Java\_Outer\_class{
2. //code
3. **class** Java\_Inner\_class{
4. //code
5. }
6. }

Advantage of Java inner classes

There are three advantages of inner classes in Java. They are as follows:

1. Nested classes represent a particular type of relationship that is **it can access all the members (data members and methods) of the outer class,** including private.
2. Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.
3. **Code Optimization**: It requires less code to write.

Need of Java Inner class

Sometimes users need to program a class in such a way so that no other class can access it. Therefore, it would be better if you include it within other classes.

If all the class objects are a part of the outer object then it is easier to nest that class inside the outer class. That way all the outer class can access all the objects of the inner class.

Difference between nested class and inner class in Java

An inner class is a part of a nested class. Non-static nested classes are known as inner classes.

Types of Nested classes

There are two types of nested classes non-static and static nested classes. **The non-static nested classes are also known as inner classes.**

* Non-static nested class (inner class)
  1. Member inner class
  2. Anonymous inner class
  3. Local inner class
* Static nested class

|  |  |
| --- | --- |
| **Type** | **Description** |
| [Member Inner Class](https://www.javatpoint.com/member-inner-class) | A class created within class and outside method. |
| [Anonymous Inner Class](https://www.javatpoint.com/anonymous-inner-class) | A class created for implementing an interface or extending class. The java compiler decides its name. |
| [Local Inner Class](https://www.javatpoint.com/local-inner-class) | A class was created within the method. |
| [Static Nested Class](https://www.javatpoint.com/static-nested-class) | A static class was created within the class. |
| [Nested Interface](https://www.javatpoint.com/nested-interface) | An interface created within class or interface. |

**Java Member Inner Class**

A non-static class that is created inside a class but outside a method is called **member inner class**. It is also known as a **regular inner class**. It can be declared with access modifiers like public, default, private, and protected.

**Syntax:**

1. Inner{
2. //code
3. }
4. }

**Java Member Inner Class Example**

In this example, we are creating a msg() method in the member inner class that is accessing the private data member of the outer class.

**TestMemberOuter1.java**

1. **class** TestMemberOuter1{
2. **private** **int** data=30;
3. **class** Inner{
4. **void** msg(){System.out.println("data is "+data);}
5. }
6. **public** **static** **void** main(String args[]){
7. TestMemberOuter1 obj=**new** TestMemberOuter1();
8. TestMemberOuter1.Inner in=obj.**new** Inner();
9. in.msg();
10. }
11. }

**Output:**

data is 30

How to instantiate Member Inner class in java?

An object or instance of a member's inner class always exists within an object of its outer class. The new operator is used to create the object of member inner class with slightly different syntax.

The general form of syntax to create an object of the member inner class is as follows:

**Syntax:**

1. OuterClassReference.**new** MemberInnerClassConstructor();

**Example:**

1. obj.**new** Inner();

Here, OuterClassReference is the reference of the outer class followed by a dot which is followed by the new operator.

Internal working of Java member inner class

The java compiler creates two class files in the case of the inner class. The class file name of the inner class is "Outer$Inner". If you want to instantiate the inner class, you must have to create the instance of the outer class. In such a case, an instance of inner class is created inside the instance of the outer class.

Internal code generated by the compiler

The Java compiler creates a class file named Outer$Inner in this case. The Member inner class has the reference of Outer class that is why it can access all the data members of Outer class including private.

1. **import** java.io.PrintStream;
2. **class** Outer$Inner
3. {
4. **final** Outer **this**$0;
5. Outer$Inner()
6. {   **super**();
7. **this**$0 = Outer.**this**;
8. }
9. **void** msg()
10. {
11. System.out.println((**new** StringBuilder()).append("data is ")
12. .append(Outer.access$000(Outer.**this**)).toString());
13. }
14. }

**Java Anonymous inner class**

Java anonymous inner class is an inner class without a name and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain "extras" such as overloading methods of a class or interface, without having to actually subclass a class.

In simple words, a class that has no name is known as an anonymous inner class in Java. It should be used if you have to override a method of class or interface. Java Anonymous inner class can be created in two ways:

1. Class (may be abstract or concrete).
2. Interface

Java anonymous inner class example using class

**TestAnonymousInner.java**

1. **abstract** **class** Person{
2. **abstract** **void** eat();
3. }
4. **class** TestAnonymousInner{
5. **public** **static** **void** main(String args[]){
6. Person p=**new** Person(){
7. **void** eat(){System.out.println("nice fruits");}
8. };
9. p.eat();
10. }
11. }

**Output:**

nice fruits

Internal working of given code.

1. Person p=**new** Person(){
2. **void** eat(){System.out.println("nice fruits");}
3. };
4. A class is created, but its name is decided by the compiler, which extends the Person class and provides the implementation of the eat() method.
5. An object of the Anonymous class is created that is referred to by 'p,' a reference variable of Person type.

Internal class generated by the compiler

1. **import** java.io.PrintStream;
2. **static** **class** TestAnonymousInner$1 **extends** Person
3. {
4. TestAnonymousInner$1(){}
5. **void** eat()
6. {
7. System.out.println("nice fruits");
8. }
9. }

Java anonymous inner class example using interface

1. **interface** Eatable{
2. **void** eat();
3. }
4. **class** TestAnnonymousInner1{
5. **public** **static** **void** main(String args[]){
6. Eatable e=**new** Eatable(){
7. **public** **void** eat(){System.out.println("nice fruits");}
8. };
9. e.eat();
10. }
11. }

**Output:**

nice fruits

**Internal Working of given code**

It performs two main tasks behind this code:

1. Eatable p=**new** Eatable(){
2. **void** eat(){System.out.println("nice fruits");}
3. };
4. A class is created, but its name is decided by the compiler, which implements the Eatable interface and provides the implementation of the eat() method.
5. An object of the Anonymous class is created that is referred to by 'p', a reference variable of the Eatable type.

Internal class generated by the compiler

1. **import** java.io.PrintStream;
2. **static** **class** TestAnonymousInner1$1 **implements** Eatable
3. {
4. TestAnonymousInner1$1(){}
5. **void** eat(){System.out.println("nice fruits");}
6. }

**Java Local inner class**

A class i.e., created inside a method, is called local inner class in java. Local Inner Classes are the inner classes that are defined inside a block. Generally, this block is a method body. Sometimes this block can be a for loop, or an if clause. Local Inner classes are not a member of any enclosing classes. They belong to the block they are defined within, due to which local inner classes cannot have any access modifiers associated with them. However, they can be marked as final or abstract. These classes have access to the fields of the class enclosing it.

If you want to invoke the methods of the local inner class, you must instantiate this class inside the method.

Java local inner class example

**LocalInner1.java**

1. **public** **class** localInner1{
2. **private** **int** data=30;//instance variable
3. **void** display(){
4. **class** Local{
5. **void** msg(){System.out.println(data);}
6. }
7. Local l=**new** Local();
8. l.msg();
9. }
10. **public** **static** **void** main(String args[]){
11. localInner1 obj=**new** localInner1();
12. obj.display();
13. }

**Output:**

30

Internal class generated by the compiler.

In such a case, the compiler creates a class named Simple$1Local that has the reference of the outer class.

1. **import** java.io.PrintStream;
2. **class** localInner1$Local
3. {
4. **final** localInner1 **this**$0;
5. localInner1$Local()
6. {
7. **super**();
8. **this**$0 = Simple.**this**;
9. }
10. **void** msg()
11. {
12. System.out.println(localInner1.access$000(localInner1.**this**));
13. }
14. }

Rule: Local variables can't be private, public, or protected.

Rules for Java Local Inner class

1) Local inner class cannot be invoked from outside the method.

2) Local inner class cannot access non-final local variable till JDK 1.7. Since JDK 1.8, it is possible to access the non-final local variable in the local inner class.

Example of local inner class with local variable**LocalInner2.java**

1. **class** localInner2{
2. **private** **int** data=30;//instance variable
3. **void** display(){
4. **int** value=50;//local variable must be final till jdk 1.7 only
5. **class** Local{
6. **void** msg(){System.out.println(value);}
7. }
8. Local l=**new** Local();
9. l.msg();
10. }
11. **public** **static** **void** main(String args[]){
12. localInner2 obj=**new** localInner2();
13. obj.display();
14. }
15. }

**Output:**

50

Java static nested class

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

* It can access static data members of the outer class, including private.
* The static nested class cannot access non-static (instance) data members or

Java static nested class example with instance method

**TestOuter1.java**

1. **class** TestOuter1{
2. **static** **int** data=30;
3. **static** **class** Inner{
4. **void** msg(){System.out.println("data is "+data);}
5. }
6. **public** **static** **void** main(String args[]){
7. TestOuter1.Inner obj=**new** TestOuter1.Inner();
8. obj.msg();
9. }
10. }

**Output:**

data is 30

In this example, you need to create the instance of static nested class because it has instance method msg(). But you don't need to create the object of the Outer class because the nested class is static and static properties, methods, or classes can be accessed without an object.

Internal class generated by the compiler

1. **import** java.io.PrintStream;
2. **static** **class** TestOuter1$Inner
3. {
4. TestOuter1$Inner(){}
5. **void** msg(){
6. System.out.println((**new** StringBuilder()).append("data is ")
7. .append(TestOuter1.data).toString());
8. }
9. }

Java static nested class example with a static method

If you have the static member inside the static nested class, you don't need to create an instance of the static nested class.

**TestOuter2.java**

1. **public** **class** TestOuter2{
2. **static** **int** data=30;
3. **static** **class** Inner{
4. **static** **void** msg(){System.out.println("data is "+data);}
5. }
6. **public** **static** **void** main(String args[]){
7. TestOuter2.Inner.msg();//no need to create the instance of static nested class
8. }
9. }

[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=TestOuter2)

**Output:**

data is 30

**Java Nested Interface**

An interface, i.e., declared within another interface or class, is known as a nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred to by the outer interface or class. It can't be accessed directly.

Points to remember for nested interfaces

There are given some points that should be remembered by the java programmer.

* The nested interface must be public if it is declared inside the interface, but it can have any access modifier if declared within the class.
* Nested interfaces are declared static

Syntax of nested interface which is declared within the interface

1. **interface** interface\_name{
2. ...
3. **interface** nested\_interface\_name{
4. ...
5. }
6. }

Syntax of nested interface which is declared within the class

1. **class** class\_name{
2. ...
3. **interface** nested\_interface\_name{
4. ...
5. }
6. }

Example of nested interface which is declared within the interface

In this example, we will learn how to declare the nested interface and how we can access it.

**TestNestedInterface1.java**

1. **interface** Showable{
2. **void** show();
3. **interface** Message{
4. **void** msg();
5. }
6. }
7. **class** TestNestedInterface1 **implements** Showable.Message{
8. **public** **void** msg(){System.out.println("Hello nested interface");}
10. **public** **static** **void** main(String args[]){
11. Showable.Message message=**new** TestNestedInterface1();//upcasting here
12. message.msg();
13. }
14. }

**Output:**

Hello nested interface