Nested Classes

A outer class can not be declared static. Only inner classes can be declared static. Static inner classes imply that the object of the inner class is not dependent upon the outer class object and can exist indepenently.

Nested classes are divided into two categories: static and non-static. Nested classes that are declared static are called *static nested classes*. Non-static nested classes are called *inner classes*.

A nested class is a member of its enclosing class. Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private. Static nested classes do not have access to other members of the enclosing class. As a member of the OuterClass, a nested class can be declaredprivate, public, protected, or *package private*. (Recall that outer classes can only be declared public or*package private*.)

Compelling reasons for using nested classes include the following:

* **It is a way of logically grouping classes that are only used in one place**: If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
* **It increases encapsulation**: Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.
* **It can lead to more readable and maintainable code**: Nesting small classes within top-level classes places the code closer to where it is used.

**Static Nested Classes**

As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class: it can use them only through an object reference.

**Note:** A static nested class interacts with the instance members of its outer class (and other classes) just like any other top-level class. In effect, a static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience.

Static nested classes are accessed using the enclosing class name:

OuterClass.StaticNestedClass

For example, to create an object for the static nested class, use this syntax:

OuterClass.StaticNestedClass nestedObject =

new OuterClass.StaticNestedClass();

**Inner Classes**

As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields. Also, because an inner class is associated with an instance, it cannot define any static members itself.

Objects that are instances of an inner class exist *within* an instance of the outer class. Consider the following classes:

class OuterClass {

...

class InnerClass {

...

}

}

An instance of InnerClass can exist only within an instance of OuterClass and has direct access to the methods and fields of its enclosing instance.

To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:

OuterClass.InnerClass innerObject = outerObject.new InnerClass();

There are two special kinds of inner classes: [local classes](https://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html) and [anonymous classes](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html).

**Shadowing**

If a declaration of a type (such as a member variable or a parameter name) in a particular scope (such as an inner class or a method definition) has the same name as another declaration in the enclosing scope, then the declaration *shadows* the declaration of the enclosing scope. You cannot refer to a shadowed declaration by its name alone. The following example, [ShadowTest](https://docs.oracle.com/javase/tutorial/java/javaOO/examples/ShadowTest.java" \t "_blank), demonstrates this:

public class ShadowTest {

public int x = 0;

class FirstLevel {

public int x = 1;

void methodInFirstLevel(int x) {

System.out.println("x = " + x);

System.out.println("this.x = " + this.x);

System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);

}

}

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

ShadowTest st = new ShadowTest();

ShadowTest.FirstLevel fl = st.new FirstLevel();

fl.methodInFirstLevel(23);

}

}

The following is the output of this example:

x = 23

this.x = 1

ShadowTest.this.x = 0

This example defines three variables named x: the member variable of the class ShadowTest, the member variable of the inner class FirstLevel, and the parameter in the method methodInFirstLevel. The variable xdefined as a parameter of the method methodInFirstLevel shadows the variable of the inner classFirstLevel. Consequently, when you use the variable x in the method methodInFirstLevel, it refers to the method parameter. To refer to the member variable of the inner class FirstLevel, use the keyword this to represent the enclosing scope:

System.out.println("this.x = " + this.x);

Refer to member variables that enclose larger scopes by the class name to which they belong. For example, the following statement accesses the member variable of the class ShadowTest from the methodmethodInFirstLevel:

System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);

**Serialization**

[Serialization](https://docs.oracle.com/javase/tutorial/jndi/objects/serial.html) of inner classes, including [local](https://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html) and [anonymous](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html) classes, is strongly discouraged. When the Java compiler compiles certain constructs, such as inner classes, it creates *synthetic constructs*; these are classes, methods, fields, and other constructs that do not have a corresponding construct in the source code. Synthetic constructs enable Java compilers to implement new Java language features without changes to the JVM. However, synthetic constructs can vary among different Java compiler implementations, which means that .class files can vary among different implementations as well. Consequently, you may have compatibility issues if you serialize an inner class and then deserialize it with a different JRE implementation.

**Local and Anonymous Classes**

There are two additional types of inner classes. You can declare an inner class within the body of a method. These classes are known as [local classes](https://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html). You can also declare an inner class within the body of a method without naming the class. These classes are known as [anonymous classes](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html).

**Local Classes Are Similar To Inner Classes**

Local classes are similar to inner classes because they cannot define or declare any static members. Local classes in static methods, such as the class PhoneNumber, which is defined in the static methodvalidatePhoneNumber, can only refer to static members of the enclosing class. For example, if you do not define the member variable regularExpression as static, then the Java compiler generates an error similar to "non-static variable regularExpression cannot be referenced from a static context."

Local classes are non-static because they have access to instance members of the enclosing block. Consequently, they cannot contain most kinds of static declarations.

# 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.

**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);

}

};

**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();

}

}