**ROSE NEEMA MWANYIKA**

**SCM211/0292-2021**

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.

This section covers the following topics:

* [Declaring Anonymous Classes](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html" \l "declaring-anonymous-classes)
* [Syntax of Anonymous Classes](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html" \l "syntax-of-anonymous-classes)
* [Accessing Local Variables of the Enclosing Scope, and Declaring and Accessing Members of the Anonymous Class](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html" \l "accessing)
* [Examples of Anonymous Classes](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html" \l "examples-of-anonymous-classes)

**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" \t "_blank), 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" \l "accessing-members-of-an-enclosing-class" \t "_top); 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" \l "shadowing" \t "_top) 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" \t "_top) 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);

}

}

Preview

**What can you include in a class definition?**

There are several different kinds of items that can be included in a  
class definition.  As you learned in the earlier lessons in this  
series, the list includes:

* Static variables
* Instance variables
* Static methods
* Instance methods
* Constructors
* Static initializer blocks
* Instance initializers

**Can also contain other class definitions**

As you also learned in previous lessons, a class definition can also  
contain the following four kinds of *inner classes:*

* Member classes
* Local classes
* Anonymous classes
* Nested top-level classes and interfaces

The previous two lessons explained member classes and local  
classes.  This lesson will explain anonymous classes.  The  
next lesson will explain nested top-level classes and interfaces.

*(Note That It Is Questionable Whether A Nested Top-Level Class  
Or Interface Should Be Referred To As An Inner Class, Because An Object  
Of A Nested Top-Level Class Can Exist In The Absence Of An Object Of  
The Enclosing Class.  Regardless Of Whether The Term Inner Class  
Applies, A Nested Top-Level Class Is Defined Within The Definition Of  
Another Class, So Its Definition Is Internal To The Definition Of  
Another Class.)*

**What is an anonymous class?**

I’m going to begin my discussion with a quotation from one of my  
favorite authors, David Flanagan, author of Java in a Nutshell.

*“An Anonymous Class Is Essentially A Local Class Without  
A Name.”*

If you have read the previous lesson, you should know quite a lot  
about local classes at this point in time.  Continuing with  
Flanagan’s words,

*“Instead Of Defining A Local Class And Then  
Instantiating It, You Can Often Use An Anonymous Class To Combine These  
Two Steps…  An Anonymous Class Is Defined By A Java Expression,  
Not A Java Statement.  This Means That An Anonymous Class  
Definition Can Be Included Within A  
Larger Java Expression…”*

As you will see from the sample program in this lesson, anonymous  
class definitions are often included as arguments to method calls.

As is the case for an object of a member class or a local class *(discussed  
in previous lessons),* an object of an anonymous class must be  
internally linked to an object of the enclosing class.

Thus, an anonymous class is truly an inner class, because an object  
of the anonymous class cannot exist in the absence of an object of the  
enclosing class.

**What about an anonymous interface?**

Interfaces defined within classes are implicitly static.  This  
means that they are always *top-level.*  There is no such  
thing as a member interface, a local interface, or an anonymous  
interface.

**Why use anonymous classes?**

As with local classes, objects instantiated from anonymous classes  
share many of the characteristics of objects instantiated from member  
classes.  However, in some cases, an anonymous class can be  
defined closer to its point of use than would be possible with a member  
class or a local class.  Once  
you become accustomed to the somewhat cryptic syntax used with  
anonymous  
classes, this can often lead to improved code readability.

Probably the most important benefit of anonymous classes has to do  
with accessing the members of enclosing classes.  Just like with  
member classes and local classes, methods of an anonymous class have  
direct access  
to all the members of the enclosing classes, including private  
members.   
Thus the use of anonymous classes can often eliminate the requirement  
to connect objects together via constructor parameters.

In addition, although not demonstrated in this lesson, as with local  
classes, objects of anonymous classes have access to **final**  
local variables that are declared within the scope of the anonymous  
class.

**Can be particularly useful when …**

An anonymous class can be particularly useful in those cases where

* There is no reason for an object of the anonymous class  
  to exist in the absence of an object of the enclosing class.
* There is no reason for an object of the anonymous class to exist  
  outside a method of the enclosing class.
* Methods of the object of the anonymous class need access to  
  members of the object of the enclosing class.
* Methods of the object of the anonymous class need access to **final**local variables and method parameters belonging to the method  
  in which the anonymous class is defined.
* Only one instance of the anonymous class is needed.
* There is no need for the class to have a name that is accessible  
  elsewhere in the program.

**Purpose of this lesson**

This lesson explains anonymous classes from a practical viewpoint,  
including a comparison between anonymous classes and local classes.

**Anonymous classes versus local classes**

Once again, according to David Flanagan,

*“…An Anonymous Class Behaves Just Like A Local Class,  
And Is Distinguished From A Local Class Merely In The Syntax Used To  
Define And Instantiate It.”*

Unlike a local class, however, an anonymous class cannot define a  
constructor.  An anonymous class can define an instance  
initializer, which can provide some of the benefits of a constructor.

*(I Discussed Instance Initializers In Detail In An  
Earlier Lesson Entitled [The  
Essence Of OOP Using Java, Instance Initializers](http://www.developer.com/java/other/article.php/3065621).  As  
You May Recall, A Primary Shortcoming  
Of An Instance Initializer As Compared To A Constructor Is That An  
Instance Initializer Cannot Accept Incoming Parameters.)*

**Restrictions on the use of anonymous classes**

Because an anonymous class has no name, and the definition and  
instantiation of the class appear in a single expression, only one  
instance of each anonymous class can be created.  If you need more  
than one instance of the class, you should probably use a local class,  
a member class, or a top-level class instead.

As mentioned above, it is not possible to define constructors for  
anonymous classes.  If you need to use a constructor when you  
instantiate the class, you should probably use a local class, a member  
class, or a top-level class instead.

As with member classes and local classes, anonymous classes cannot  
contain **static** members.

As with local variables and local classes, anonymous classes cannot  
be declared **public**,**protected**,**private**,or  
**static**.  In fact, no modifiers can be specified in the  
definition of an anonymous class.

**Smoke and mirrors**

As I told you in my earlier lessons on local classes, the methods in  
an anonymous class don’t really have access to local variables and  
method parameters.  Rather, when an object of the anonymous class  
is instantiated, copies of the **final** local variables and method  
parameters referred to by the object’s methods are stored as instance  
variables in the object.  The methods in the object of the  
anonymous class really access those hidden instance variables.

Thus, the local variables and method parameters accessed by the  
methods of the local class must be declared **final** to prevent  
their values from changing after the object is instantiated.

There are some additional activities involving *smoke and mirrors*  
taking place behind the scenes when you define and instantiate an  
anonymous class.  Generally speaking, this involves the automatic  
generation of code to cause things to behave as they do.  The good  
news is  
that you don’t have to write that extra code, and you don’t have to  
maintain it.  The extra code is written for you, and if you modify  
your class structure, the extra code is automatically modified  
accordingly.

You can read about the code that is automatically generated in my  
earlier lessons on local classes and member classes.

**Syntax for anonymous classes**

Before getting into actual code in the sample program, I want to  
explain the syntax used to define and instantiate an anonymous class.

The definition and instantiation of an anonymous class uses one or  
the other of the two expressions shown in Figure 1.

|  |
| --- |
| **new** className(optional argument list){classBody}  new interfaceName(){classBody}  **Figure 1** |

Usually, this expression is included inside a larger overall  
expression, such as an argument to a method call.

**What does the first expression mean?**

Here is how I usually explain this syntax to my students.  The  
first expression in Figure 1 starts out fairly normal, but becomes  
cryptic very quickly.  This expression instantiates a new object  
from an unnamed and previously undefined class, which automatically *extends*the class named **className**, and which cannot explicitly  
implement any interfaces.  The body of the new class is given by **classBody**.

The result of executing this expression is that a new class that  
extends **className** is defined, a new object of the new class is  
instantiated, and the expression is replaced by a reference to the new  
object.

**Example usage**

If this expression appears as the right operand of an assignment  
operator, the object’s reference is saved in the left operand of the  
assignment operator.   If the expression appears as an argument in  
a method call, the object’s reference is passed to the method.  If  
the expression appears in some other form of larger overall expression,  
the object’s reference is handed over to the surrounding expression to  
be used appropriately.

**What about instantiating an interface?**

The second expression in Figure 1 starts out weird.  To my  
knowledge, there is no other situation in Java where you apply the **new**  
operator to the name of an interface.  From the beginning, you  
have been told that you cannot instantiate an object of an  
interface.  *(An interface is implicitly abstract and it  
doesn’t have a constructor, not even a default constructor.)*   
However, you can instantiate an object of a class that implements none,  
one, or more interfaces.

The correct interpretation of the second expression in Figure 1 is as  
follows.   This expression instantiates a new object from an  
unnamed  
and previously undefined class, which automatically *implements*the  
interface named **interfaceName**, and automatically *extends*  
the  
class named **Object**.   The class can explicitly implement  
one,  
and only one interface, and cannot extend any class other than **Object**.   
Once again, the body of the new class is given by **classBody**.

As in the case of the first expression in Figure 1, the result of  
executing this expression is that a new class that implements **interfaceName**  
is defined, a new object of the new class is instantiated, and the  
expression is replaced by a reference to the new object.  That  
reference is handed over to the surrounding expression to be used  
appropriately.

**What about constructor parameters?**

As mentioned earlier in this lesson, since the new class doesn’t have  
a name, it isn’t possible to define a constructor for the new  
class.  According to Flanagan,

*“Any Arguments You Specify Between The Parentheses  
Following The Superclass Name In An Anonymous Class Definition Are  
Implicitly Passed To The Superclass Constructor.”*

Thus, it is possible to define an anonymous class that extends a class  
whose constructor requires parameters, and to pass those parameters to  
the superclass constructor when the anonymous class is instantiated.

The parentheses following **interfaceName** in the second  
expression  
in Figure 1 must always be empty.  In this case, the superclass is  
always **Object**, which never expects constructor parameters.

**Enough talk, let’s see some code**

The paragraphs that follow will explain a program named **InnerClasses08**,  
which is designed specifically to illustrate anonymous classes, and to  
compare anonymous classes with local classes.  I will discuss the  
program in fragments.  A complete listing of the program is  
provided in Listing 10 near the end of the lesson.

Discussion and Sample Code

When the program is executed, it produces the GUI shown in Figure  
2.  I will refer back to this figure during the discussion of the  
program.

Figure 2  Program GUI

**Class file names**

This program consists of a total of six classes:

* Two top-level classes
* One local class
* Three anonymous classes

When compiled, the program produces the class files shown  
in Figure 3.

|  |
| --- |
| GUI$1$BaldButton.class GUI$1.class GUI$2.class GUI$3.class GUI.class InnerClasses08.class  **Figure 3** |

As you will see later, the file named **GUI$1$BaldButton.class**  
represents the local class.  The three files named **GUI$1.class**,  
**GUI$2.class**, and **GUI$3.class** represent anonymous  
classes.  The two remaining files represent top-level classes.

*(As You Can See, The Anonymous Classes Are Not Truly  
Anonymous, Since The Files That Represent Them Must Have Names.   
Generally, However, The Establishment Of The Individual Names Is Beyond  
The Control Of The Programmer, And The Names Are Not Known To The  
Program In A Way That Makes It Possible To Refer To Them By Name.)*

**Program structure and behavior**

This program is designed to illustrate the use of local classes and  
anonymous classes in a very practical way. It illustrates one  
implementation of a local class and three different implementations of  
anonymous classes. The program compares the local class with an  
anonymous class designed to accomplish the same purpose.  The  
program also illustrates the use of instance initializers as an  
alternative to constructors.

**A local class**

The program defines and uses a local class to instantiate an object to  
handle mouse clicked events on a button with low-level event handling.  
This  
class uses a constructor to enable mouse events on a new extended **Button**class. It also uses a constructor to display the name of the class  
file.

**An anonymous class to compare with the local  
class**

The program also defines and uses an anonymous class to instantiate an  
object to handle mouse clicked events on a button with low-level event  
handling. This class uses an instance initializer to enable mouse  
events on a new extended **Button**class. It also uses an  
instance initializer to display the name of the class file.  This  
class and the local class described above provide a direct comparison  
between the use of local classes and anonymous classes to serve the  
same purpose.

**An anonymous class that implements an interface**

The program illustrates the use of an anonymous class, which implements  
the **MouseListener** interface, to instantiate an object to handle  
mouse clicked events using the source-listener event model *(sometimes  
referred to as the delegation event model or the JavaBeans event model).*  
The anonymous class uses an instance initializer to display the name of  
the class file.

**An anonymous class that extends an existing  
class**

The program illustrates the use of an anonymous class, which extends  
the **WindowAdapter** class, to instantiate an object to handle  
window events fired by the close button in the upper-right corner of  
the **Frame** object shown in Figure 2.  This class also uses  
the source-listener event model, and uses an instance initializer to  
display the name of the class  
file.

**The screen output**

The program produces the screen output shown in Figure 4 when

* The program is started
* Each button shown in Figure 2 is clicked once in succession,  
  going from left to right
* The close button in the upper-right corner of the **Frame**object  
  in Figure 2 is clicked

|  |
| --- |
| Local class name: GUI$1$BaldButton Anonymous class B name: GUI$1 Anonymous class C name: GUI$2 Anonymous window listener class name: GUI$3 buttonA clicked buttonB clicked buttonC clicked Close button clicked  **Figure 4** |

When the close button is clicked, the program produces the last line of  
text in Figure 4 and terminates.

I will identify the code that produces each line of output text in the  
discussion of the program that follows.

**The controlling class**

The controlling class for the program is shown in Listing 1.

|  |
| --- |
| public class InnerClasses08 {  public static void **main**(String[] args){  **new GUI()**;  }//end main }//end class InnerClasses08  **Listing 1** |

As you can see, the controlling class is very simple, with the **main**  
method instantiating an object of the **GUI**class.  This  
results in the GUI that is pictured in Figure 2.

**Local and anonymous classes inside GUI  
constructor**

The local class and the three anonymous classes are defined inside the  
constructor for the **GUI**class.

*(Recall That Local Classes And Anonymous Classes Are  
Defined Inside Code Blocks, Which Often Place Them Inside Methods And  
Constructors, But You Can Also Place Them Inside Static Initializer  
Blocks And Instance Initializers.)*

The first four lines of the output text in Figure 4 are produced by  
constructors and instance initializers in the local and anonymous  
classes.  Therefore, those four lines of text are produced when  
the new object of the **GUI** class is instantiated.

**The GUI class**

As is often the case, the **GUI** classused to create the  
visual GUI shown in Figure 2 consists solely of a constructor.   
Basically,  
this constructor places three buttons in the frame and registers event  
handlers on the buttons and on the frame.  Once the **GUI**object  
is constructed and appears on the screen, all further activity in the  
program occurs under control of the event handlers associated with the  
buttons and the frame.

*(You Can Learn More About Event Handling By Reviewing My  
Online Tutorial Lessons At [Http://Www.Dickbaldwin.Com/Tocmed.Htm](http://www.dickbaldwin.com/tocmed.htm).)*

**The GUI constructor**

The **GUI** class, and the constructor for that class begin in  
Listing 2.

|  |
| --- |
| class GUI extends Frame{  public GUI(){//constructor  setLayout(new FlowLayout());  setSize(250,75);  setTitle("Copyright 2003 R.G.Baldwin");  **Listing 2** |

As you can see, the **GUI** class extends **Frame**, so that an  
object of the class is a frame.

The constructor code shown in Listing 2 simply sets values for the *layout,  
size,* and *title*properties of the frame.

**The BaldButton class**

The **BaldButton** class, whose definition begins in Listing 3, is  
a  
local class that extends **Button**.  This class extends the **Button**  
class to make it possible to override the **processMouseEvent**  
method in order to handle mouse events that are fired by the  
button.  This  
is a form of low-level event handling, which will be contrasted with  
source-listener event handling later in the program.

Listing 3 shows the constructor for the **BaldButton** class.

|  |
| --- |
| class **BaldButton** extends Button{  BaldButton(String text){//constructor  enableEvents(AWTEvent.MOUSE\_EVENT\_MASK);  setLabel(text);  System.out.println("Local class name: " +  getClass().getName());  }//end constructor  **Listing 3** |

**Enable mouse events**

The most important code in the constructor is the statement that  
enables mouse events on the button.  If you are unfamiliar with  
the **enableEvents** method, you should look it up in the Sun  
documentation.

Briefly, this method must be invoked on the button to cause the  
overridden **processMouseEvent** method to be invoked later when  
the button fires a mouse event.

**The remaining constructor code**

The remaining code in the constructor

* Sets the text value on the face of the button
* Gets and displays the name of the class file that represents this  
  local class

**The screen output**

Construction of the button by the code in Listing 3 causes the text  
shown in Figure 5 to appear on the screen.  This is how I was able  
to identify the name of the class file that represents the local class  
in my earlier discussion  
of class file names.

|  |
| --- |
| Local class name: GUI$1$BaldButton  **Figure 5** |

We will see later that this button will be added as the leftmost button  
in the GUI shown in Figure 2.

**The processMouseEvent method**

Continuing with the constructor for the **BaldButton** class,  
Listing 4 shows the overridden **processMouseEvent** method for an  
object of  
the **BaldButton** class.

|  |
| --- |
| public void processMouseEvent(  MouseEvent e){  if (e.getID() ==  MouseEvent.MOUSE\_CLICKED){  System.out.println("buttonA clicked");  }//end if   //The following is required of overridden  // processMouseEvent method.  super.processMouseEvent(e);  }//end processMouseEvent  }//end class BaldButton   //Add button to Frame  add(new BaldButton("A")); **Listing 4** |

This method is invoked each time an object instantiated from this class  
fires a mouse event.  That is why I refer to the method as an  
event handler for the button.

**Different kinds of mouse events**

A button can fire a variety of different kinds or subcategories of  
mouse events:

* MOUSE\_CLICKED
* MOUSE\_DRAGGED
* MOUSE\_ENTERED
* MOUSE\_EXITED
* MOUSE\_MOVED
* MOUSE\_PRESSED
* MOUSE\_RELEASED

In this case, I elected to ignore all but MOUSE\_CLICKED.  This  
subcategory of mouse event occurs when a mouse button is pressed and  
then released.

Thus the code in the event handler of Listing 4 first confirms that the  
event was of the MOUSE\_CLICKED variety, and if so, it displays a  
message that  
matches the fifth line of text in the output shown in Figure 4.

**Invoke processMouseEvent on the superclass**

Without getting into the details of why this is required, I’m simply  
going to tell you that when you use this low-level event model to  
handle events, your overridden **processMouseEvent** method must  
call the same method in the superclass, passing the incoming parameter  
of type **MouseEvent** as a parameter to the superclass version of  
the method.

**Add a button to the frame**

The last statement in Listing 4 instantiates a new **BaldButton**  
object, setting the text on the face of the button to **A**, and  
adds that new object to the frame.  Because the layout property of  
the frame has  
been set to **FlowLayout**, and because this is the first component  
added  
to the frame, this button appears as the leftmost button in the GUI  
shown  
in Figure 2.

**Could instantiate multiple buttons of this type**

Although I instantiated the button object as an anonymous object in  
this case, that wasn’t necessary.  Using this local class, I could  
instantiate more than one object of this type, saving the object’s  
references in reference variables of the appropriate type.  Later  
we will see that this is  
not possible for anonymous classes.

It is interesting to note, however, that with this event handle model,  
if I were to instantiate multiple buttons of this type, the same **processMouseEvent**  
method would be invoked no matter which of the buttons fired a mouse  
event.  If I wanted different behavior as a result of the  
different buttons firing mouse events, I would have to write code  
inside the **processMouseEvent** method to deal with that  
issue.  The source-listener event model that I will illustrate  
later doesn’t suffer from that restriction.

**An anonymous inner class for low-level event  
handling**

Listing 5 shows the beginning of an anonymous class to perform  
low-level event handling similar to that shown in Listing 4.  This  
code defines an anonymous inner class, which extends **Button**,  
and which has mouse events enabled.  I provided this class  
primarily for comparison with the local class named **BaldButton**.   
This class is an anonymous alternative to the local **BaldButton**  
class.

|  |
| --- |
| **add**(new Button("B")  {//Begin class definition  {//Instance initializer  enableEvents(  AWTEvent.MOUSE\_EVENT\_MASK);  System.out.println(  "Anonymous class B name: " +  getClass().getName());  }//end instance initializer  **Listing 5** |

**An argument to the add method**

Note that the definition of this anonymous class appears as an argument  
to the **add** method for the frame.  Thus, the anonymous  
object instantiated from the anonymous class is added as the second *(middle)*  
button in Figure 2.

**Extends the Button class**

Note also that this form of anonymous class implicitly extends the **Button**  
class.  Once again, this makes it possible to override the **processMouseEvent**  
method belonging to the **Button** class.

**An instance initializer**

As I mentioned earlier in this lesson, it is not possible to define a  
constructor for an anonymous class.  However, it is possible to  
define an instance initializer.  This class defines an instance  
initializer, which

* Enables mouse events on an anonymous object instantiated from the  
  anonymous class
* Gets and displays the name of the class file that represents the  
  anonymous class

**The screen output**

Therefore, the instantiation of this anonymous object causes the text  
shown in Figure 6 to appear on the screen.  About all you can tell  
by looking at this class name is that it is the name of a file that  
represents an anonymous class.

|  |
| --- |
| Anonymous class B name: GUI$1  **Figure 6** |

**Overridden processMouseEvent method**

The remaining code in the anonymous class definition is shown in  
Listing 6.

|  |
| --- |
| public void processMouseEvent(  MouseEvent e){  if (e.getID() ==  MouseEvent.MOUSE\_CLICKED){  System.out.println(  "buttonB clicked");  }//end if   //Required of overridden  // processMouseEvent method.  super.processMouseEvent(e);  }//end processMouseEvent  }//end class definition  );//end add method call  **Listing 6** |

Basically, the remaining code consists of an overridden **processMouseEvent**  
method, and the curly braces, parentheses, and semicolon necessary to  
complete the expression and the statement.

**Same code as before**

The code in this overridden **processMouseEvent** method is  
essentially the same as that shown for the local class in Listing 4,  
except that it  
produces a different message on the screen when the user clicks the  
button.

Clicking the middle button in Figure 2 produces the screen output shown  
by the sixth line in Figure 4.

**Implementing a listener interface**

Now I’m going to switch from low-level event handling to  
source-listener event handling.  With this event handling model

* A listener object is instantiated from a class that implements a  
  specific listener interface.  In this case, that interface will be  
  the **MouseListener** interface.
* The listener object is *registered*on an object that  
  knows  
  how to fire events of a type that is associated with the listener  
  interface.  In this case, that will be events of type **MouseEvent**.
* When the source object fires an event of the specified type, one  
  of the methods belonging to the registered listener object will be  
  invoked  
  to handle the event.  The different methods belonging to the  
  listener  
  object are declared in the implemented listener interface.

**Instantiating and registering a MouseListener  
object**

The code to accomplish this begins in Listing 7.  Listing 7 begins  
by instantiating a new **Button** object.

*(Note That With This Event Model, It Is Not Necessary To  
Extend The****Button****Class, Because It Is Not Necessary To  
Override Methods Belonging  
To The****Button****Object.)*

After instantiating a new **Button** object, the code in Listing 7  
invokes the **addMouseListener** method to register a **MouseListener**  
object on that button.  The argument to the **addMouseListener**  
method must be a reference to an object instantiated from a class that  
implements the **MouseListener** interface.

|  |
| --- |
| Button buttonC = new Button("C");   buttonC.addMouseListener(new MouseListener()  {//begin class definition  //Instance initializer  {System.out.println(  "Anonymous class C name: " +  getClass().getName());}  **Listing 7** |

**Instantiate the listener object**

In this case, that listener object is created by writing an expression  
to instantiate an anonymous object from an anonymous class and placing  
that  
expression as an argument to the **addMouseListener** method.

**Implement the MouseListener interface**

The definition of the anonymous class in this example uses the syntax  
that implements an interface.

**An instance initializer**

As before, an instance initializer is used to get and display the name  
of the class file that represents the anonymous class.  Thus, when  
the  
new anonymous object of the anonymous class is instantiated, the text  
shown  
in Figure 7 appears on the screen.  Note the similarity of this  
class  
file name to that shown earlier in Figure 6.  The names of the two  
class files differ only with respect to a number that is provided by  
the  
compiler to guarantee that each class file name is unique.

|  |
| --- |
| Anonymous class C name: GUI$2  **Figure 7** |

**Implementing the interface**

Whenever a class implements an interface, it must provide a concrete  
definition for each of the methods declared in the interface, even if  
some of those methods are empty.

Continuing with the definition of the anonymous class, Listing 8  
provides definitions for all five of the methods declared in the **MouseListener**  
interface.  Four of those methods are defined as empty methods.

|  |
| --- |
| public void **mouseClicked**(MouseEvent e){  System.out.println("buttonC clicked");  }//end mouseClicked   //All interface methods must be defined  public void **mousePressed**(MouseEvent e){}  public void **mouseReleased**(MouseEvent e){}  public void **mouseEntered**(MouseEvent e){}  public void **mouseExited**(MouseEvent e){}   }//end class definition  );//end addMouseListener call   add(buttonC);//add button to frame  **Listing 8** |

**Separation of event subcategories**

One of the major differences between the low-level event model  
discussed  
earlier and the source-listener model being discussed here has to do  
with  
where the separation between the different subcategories *(mouseClicked,  
mousePressed, mouseReleased, etc.)* of a given event type is  
accomplished

In the low-level model, the separation must be accomplished by code in  
the overridden event handler method, such as with the **if**  
statement  
in the **processMouseEvent** method defined in Listing 6.

In the source-listener model, the separation is accomplished before the  
event handler method is invoked, and a specific event handler method,  
such as the **mouseClicked** method is invoked on the listener  
object.

**When the button fires a mouse event …**

In this case, whenever the button fires a **MouseEvent** of the **mouseClicked**  
subcategory, the **mouseClicked** method defined in Listing 8 will  
be  
invoked, causing the seventh line of text in Figure 4 to appear on the  
screen.

Whenever the button fires a **MouseEvent** of one of the other  
subcategories, one of the empty methods defined in Listing 8 will be  
invoked.  This method will return immediately, doing nothing but  
wasting a little computer time.

*(In Case You Are Wondering What Happened To The****MouseMoved*** *And****MouseDragged****Methods, They Are Defined In The****MouseMotionListener****Interface Instead Of The****MouseListener****Interface.)*

**Add the button to the frame**

Finally, the last statement in Listing 8 adds the new button to the  
frame as the rightmost button in Figure 2.

**A disclaimer**

I wrote this code the way that I did in Listing 8 to illustrate an  
anonymous class that implements an interface.  In real life, I  
would probably cause the anonymous class to extend the **MouseAdapter**  
class and override the **mouseClicked** method instead of  
implementing the **MouseListener** interface.  That would  
eliminate the requirement for me to define the four empty methods in  
Listing 8.

**Extending the WindowAdapter class**

The above disclaimer provides a perfect lead-in to the definition of  
the anonymous class shown in Listing 9.

|  |
| --- |
| **addWindowListener**(new WindowAdapter()  {//begin class definition  //Instance initializer  {System.out.println(  "Anonymous window listener class " +  "name: " + getClass().getName());}   public void **windowClosing**(WindowEvent e){  System.out.println(  "Close button clicked");  System.exit(0);  }//end windowClosing  }//end class definition  );//end addWindowListener   setVisible(true);   }//end constructor }//end GUI class  **Listing 9** |

**Registering a WindowListener on the frame**

The code in Listing 9 instantiates an anonymous object of an anonymous  
class, which extends the **WindowAdapter** class.  That  
anonymous object is registered as a **WindowListener** on the frame  
by passing the object’s reference to the **addWindowListener**  
method belonging to the frame.

*(The****AddWindowListener****Method Requires An  
Incoming  
Parameter Of Type****WindowListener****.  This Is Satisfied By  
The  
Fact That The****WindowAdapter****Class Implements The****WindowListener*** *Interface.  Thus, An Object Instantiated From A Class That Extends****WindowAdapter****Can Also Be Treated As Type****WindowListener****.)*

**The screen output**

This anonymous class definition uses an instance initializer to get and  
display the name of the class that represents the anonymous  
class.  Thus, when the anonymous object of the anonymous class is  
instantiated,  
the text shown in Figure 8 appears on the screen.

|  |
| --- |
| Anonymous window listener class name: GUI$3  **Figure 8** |

**Class file names**

In an earlier lesson explaining member classes, I told you that it is  
possible to examine the names of the class files that represent the  
member classes and to determine quite a lot about the structure of the  
program in terms of which classes are members of which other  
classes.  However, in the case of local classes and anonymous  
classes, about all that you can determine from the name of the class  
file is that the file either represents a local class or represents an  
anonymous class *(see the summary of class named in Figure 3).*

**The windowClosing method**

The code in Listing 9 overrides the **windowClosing** method  
inherited from the **WindowAdapter** class.

Clicking the *close* button with the X in the upper right hand  
corner of Figure 2 causes the **windowClosing** method to be  
invoked on any **WindowListener** objects that have been registered  
on the frame.  In this case, the  
overridden **windowClosing** method in Listing 9 cases the last  
line  
of text in Figure 4 to be displayed on the screen.

Following that, the overridden **windowClosing** method invokes the  
**System.exit** method to terminate the program.

**The remaining code**

The remaining code in Listing 9

* Causes the frame to become visible
* Signals the end of the constructor
* Signals the end of the **GUI** class

**The GUI remains on the screen until terminated**

Once the constructor is executed, the GUI simply remains on the screen  
waiting for someone to click one of the buttons or to click the close  
button in  
the upper right corner of the frame.  When these buttons are  
clicked,  
the event handlers are invoked, causing text such as that shown in  
Figure  
9 to appear on the screen.

|  |
| --- |
| buttonA clicked buttonB clicked buttonC clicked Close button clicked  **Figure 9** |

**Simple event handlers**

In this demo program, the event handlers simply display messages on the  
screen, and in the case of the close button, terminate the  
program.  In a real world program, the behavior of the event  
handlers would likely  
be much more substantive, but the overall skeleton of the program need  
not  
be any different from that illustrated here.

Run the Program

At this point, you may find it useful to compile and run the  
program shown in Listing 10 near the end of the lesson.

Summary

In addition to a number of other items, a class definition can  
contain:

* Member classes
* Local classes
* Anonymous classes
* Nested top-level classes and interfaces

Member classes and local classes were explained in previous  
lessons.  This lesson explains anonymous classes.  The next  
lesson will explain nested top-level classes and interfaces.

Although there are some differences, an anonymous class is very similar  
to a local class without a name.

Instead of defining a local class and then instantiating it, you can  
often use an anonymous class to combine these two steps.

An anonymous class is defined by a Java expression, not a  
statement.  Therefore, an anonymous class definition can be  
included within a larger  
overall Java expression.

Anonymous class definitions are often included as arguments to method  
calls, or as the right operand to assignment operators.

An object of an anonymous class must be internally linked to an object  
of the enclosing class.

There is no such thing as an anonymous interface, a local interface, or  
a member interface.

An anonymous class can often be defined very close to its point of use.  
 Once you become accustomed to the somewhat cryptic syntax used  
with anonymous classes, this can lead to improved code readability.

Probably the most important benefit of anonymous classes has to do with  
accessing the members of enclosing classes.  As with member  
classes  
and local classes, methods of an anonymous class have direct access to  
all the members of the enclosing classes, including private  
members.  Thus the use of anonymous classes can sometimes  
eliminate the requirement to connect objects together via constructor  
parameters.

In addition, objects of anonymous classes have access to **final**  
local variables that are declared within the scope of the anonymous  
class.

An anonymous class can be particularly useful in those cases where

* There is no reason for an object of the anonymous class to exist  
  in the absence of an object of the enclosing class.
* There is no reason for an object of the anonymous class to exist  
  outside a method of the enclosing class.
* Methods of the object of the anonymous class need access to  
  members of the object of the enclosing class.
* Methods of the object of the anonymous class need access to **final**local variables and method parameters belonging to the method  
  in which the anonymous class is defined.
* Only one instance of the anonymous class is needed.
* There is no need for the class to have a name that is accessible  
  elsewhere in the program.

An anonymous class cannot define a constructor.  However, it can  
define an instance initializer.  Any arguments that you specify  
between  
the parentheses following the superclass name in an anonymous class  
definition are implicitly passed to the superclass constructor.

Only one instance of an anonymous class can be created.

As with member classes and local classes, anonymous classes cannot  
contain **static** members.

As with local variables and local classes, anonymous classes cannot be  
declared **public**,**protected**,**private**,or **static**.

The next lesson in this series will explain top-level nested classes.

Complete  
Program Listing

A complete listing of the program discussed in this lesson is show in  
Listing 10 below.

|  |
| --- |
| /\*File InnerClasses08.java Copyright 2003 R.G.Baldwin  This program is designed to illustrate the use of local classes, and anonymous classes. It illustrates three different implementations of anonymous classes. It also illustrates the use of instance initializers as an alternative to constructors.  Illustrates use of local class to instantiate object to handle mouse clicked event with low-level event handling. This class uses constructor to enable mouse events on a new extended Button class. Also uses constructor to display the class file name.  Illustrates use of anonymous class to instantiate object to handle mouse clicked event with low-level event handling. This class uses an instance initializer to enable mouse events on a new extended Button class. Also uses instance initializer to display name of class file.  Illustrates use of anonymous class, which implements MouseListener interface, to instantiate object to handle mouse clicked event using source-listener event model. Uses instance initializer to display name of class file.  Illustrates use of anonymous class, which extends WindowAdapter class, to instantiate object to handle window events fired by the close button in the upper-right corner of a Frame object, using source-listener event model. Uses instance initializer to display name of class file.  This program produces the following class files when compiled:  GUI$1$BaldButton.class GUI$1.class GUI$2.class GUI$3.class GUI.class InnerClasses08.class  The program produces the following output when the program is started, each button is clicked once in succession, and then the close button in the upper-right corner of the Frame is clicked:  Local class name: GUI$1$BaldButton Anonymous class B name: GUI$1 Anonymous class C name: GUI$2 Anonymous window listener class name: GUI$3 buttonA clicked buttonB clicked buttonC clicked Close button clicked  Tested using JDK 1.4.1 under Win \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  import java.awt.\*; import java.awt.event.\*;  public class InnerClasses08 {  public static void main(String[] args){  new GUI();  }//end main }//end class InnerClasses08 //=============================================//  class GUI extends Frame{   public GUI(){//constructor  setLayout(new FlowLayout());  setSize(250,75);  setTitle("Copyright 2003 R.G.Baldwin");   //Local class w/mouse events enabled. The new  // class extends Button, and uses low-level  // event handling to handle mouse clicked  // events on the button.  class BaldButton extends Button{  BaldButton(String text){//constructor  enableEvents(AWTEvent.MOUSE\_EVENT\_MASK);  setLabel(text);  //Display the name of the class file  System.out.println("Local class name: " +  getClass().getName());  }//end constructor   //This is the event handling method.  public void processMouseEvent(  MouseEvent e){  if (e.getID() ==  MouseEvent.MOUSE\_CLICKED){  System.out.println("buttonA clicked");  }//end if  //The following is required of overridden  // processMouseEvent method.  super.processMouseEvent(e);  }//end processMouseEvent  }//end class BaldButton   //Add button to Frame  add(new BaldButton("A"));    //This code defines an anonymous Inner Class  // w/mouse events enabled. The new class  // extends Button. This class uses low-level  // event handling to handle mouse clicked  // events on the button. This is an  // anonymous alternative to the local class  // defined above.  add(new Button("B")  {//Begin class definition  {//Instance initializer  enableEvents(  AWTEvent.MOUSE\_EVENT\_MASK);  System.out.println(  "Anonymous class B name: " +  getClass().getName());  }//end instance initializer   //Override the inherited  // processMouseEvent method.  public void processMouseEvent(  MouseEvent e){  if (e.getID() ==  MouseEvent.MOUSE\_CLICKED){  System.out.println(  "buttonB clicked");  }//end if  //Required of overridden  // processMouseEvent method.  super.processMouseEvent(e);  }//end processMouseEvent  }//end class definition  );//end add method call    Button buttonC = new Button("C");  //Anonymous inner class that implements  // MouseListener interface  buttonC.addMouseListener(new MouseListener()  {//begin class definition  //Instance initializer  {System.out.println(  "Anonymous class C name: " +  getClass().getName());}   public void mouseClicked(MouseEvent e){  System.out.println("buttonC clicked");  }//end mouseClicked   //All interface methods must be defined  public void mousePressed(MouseEvent e){}  public void mouseReleased(MouseEvent e){}  public void mouseEntered(MouseEvent e){}  public void mouseExited(MouseEvent e){}   }//end class definition  );//end addMouseListener call   add(buttonC);//add button to frame    //Use an anonymous class to register a window  // listener on the Frame. This class extends  // WindowAdapter  addWindowListener(new WindowAdapter()  {//begin class definition  //Instance initializer  {System.out.println(  "Anonymous window listener class " +  "name: " + getClass().getName());}   public void windowClosing(WindowEvent e){  System.out.println(  "Close button clicked");  System.exit(0);  }//end windowClosing  }//end class definition  );//end addWindowListener   setVisible(true);   }//end constructor  }//end GUI class //=============================================//  **Listing 10** |