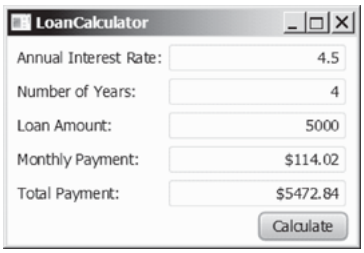
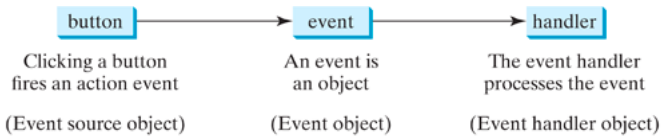
- **Event Handlers**

A GUI interacts with the users. You can write code to process **events** such as a button click, mouse movement, and keystrokes. Suppose you wish to write a GUI program that lets the user enter a loan amount, annual interest rate, and number of years then click the Calculate button to obtain the monthly payment and total payment. You have to use event-driven programming to write the code to respond to the button-clicking event.



To respond to a button click, you need to write the code to process the button-clicking action. The button is an event source object—where the action originates. You need to create an object capable of handling the action event on a button. This object is called an **event handler**.

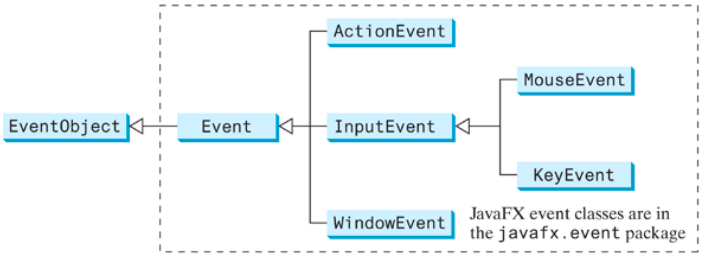


Not all objects can be the handlers for an action event. To be a handler of an action event, two requirements must be met:

1. The object must be an instance of the EventHandler interface. This interface defines the common behavior for all handlers.
2. The EventHandler object handler must be registered with the event source object using the method **source.setOnAction(handler).**

The EventHandler interface contains the **handle(ActionEvent)** method for processing the action event. Your handler class must override this method to respond to the event.

An **event** is an object created from an event source. Firing an event means to create an event and delegate the handler to handle the event. When you run a Java GUI program, the program interacts with the user and the events drive its execution. This is called event-driven programming. An event can be defined as a signal to the program that something has happened. **Events are triggered by external user actions**, such as mouse movements, mouse clicks, and keystrokes. The program can choose to respond to or ignore an event. The component that creates an event and fires it is called the **event source object**, or simply source object or source component. For example, a button is the source object for a button-clicking action event. An event is an instance of an event class. The root class of the Java event classes is **java.util.EventObject**. The root class of the JavaFX event classes is **javafx.event.Event**. The hierarchical relationships of some event classes are shown below.



You can identify the source object of an event using the **getSource()** instance method in the **EventObject** class. The subclasses of EventObject deal with specific types of events, such as action events, window events, mouse events, and key events. For example, **when clicking a button, the button creates and fires an ActionEvent**. Here, the button is an event source object, and an ActionEvent is the event object fired by the source object.

A handler is an object that must be registered with an event source object and it must be an instance of an appropriate event-handling interface.

- java.util.**EventObject** (implements java.io.Serializable)
  - javafx.event.**Event** (implements java.lang.Cloneable)
    - javafx.event.**ActionEvent**
      - javafx.scene.media.**MediaMarkerEvent**
    - javafx.scene.control.**CheckBoxTreeItem.TreeModificationEvent**<T>
    - javafx.scene.control.**DialogEvent**
    - javafx.scene.input.**InputEvent**
      - javafx.scene.input.**ContextMenuEvent**
      - javafx.scene.input.**DragEvent**
      - javafx.scene.input.**GestureEvent**
        - javafx.scene.input.**RotateEvent**
        - javafx.scene.input.**ScrollEvent**
        - javafx.scene.input.**SwipeEvent**
        - javafx.scene.input.**ZoomEvent**
      - javafx.scene.input.**InputMethodEvent**
      - javafx.scene.input.**KeyEvent**
      - javafx.scene.input.**MouseEvent**
        - javafx.scene.input.**MouseDragEvent**
      - javafx.scene.input.**TouchEvent**
    - javafx.scene.control.**ListView.EditEvent**<T>
    - javafx.scene.media.**MediaErrorEvent**
    - javafx.scene.control.**ScrollToEvent**<T>
    - javafx.scene.control.**SortEvent**<C>
    - javafx.scene.control.**TableColumn.CellEditEvent**<S,T>
    - javafx.scene.transform.**TransformChangedEvent**
    - javafx.scene.control.**TreeItem.TreeModificationEvent**<T>
    - javafx.scene.control.**TreeTableColumn.CellEditEvent**<S,T>
    - javafx.scene.control.**TreeTableView.EditEvent**<S>
    - javafx.scene.control.**TreeView.EditEvent**<T>
    - javafx.scene.web.**WebErrorEvent**
    - javafx.scene.web.**WebEvent**<T>
    - javafx.stage.**WindowEvent**
    - javafx.concurrent.**WorkerStateEvent**
- javafx.event.**EventType**<T> (implements java.io.Serializable)

- **Registering an Event**

Java uses a delegation-based model for event handling: A source object fires an event, and an object interested in the event handles it. The latter object is called an **event handler** or an **event listener**. For an object to be a handler for an event on a source object, **two things are needed.**

1. The **handler object** must be an instance of the corresponding event–handler interface to ensure the handler has the correct method for processing the event. JavaFX defines a unified handler interface EventHandler for an event T. The handler interface contains the **handle(T e)** method for processing

the event. For example, the handler interface for ActionEvent is EventHandler; each handler for ActionEvent should implement the handle(ActionEvent e) method for processing an ActionEvent.

2. The **handler object must be registered** by the source object. Registration methods depend on the event type. For ActionEvent, the method is **setOnAction**. For a mouse-pressed event, the method is **setOnMousePressed**. For a key-pressed event, the method is **setOnKeyPressed**.

- **Mouse Event**

A MouseEvent is fired whenever a mouse button is pressed, released, clicked, moved, or dragged on a node or a scene. The MouseEvent object captures the event, such as the number of clicks associated with it, the location (the x- and y-coordinates) of the mouse, or which mouse button was pressed. https://openjfx.io/javadoc/17/javafx.graphics/javafx/scene/input/MouseEvent.html

- **Key Event**

A KeyEvent is fired whenever a key is pressed, released, or typed on a node or a scene. Key events enable the use of the keys to control and perform actions, or get input from the keyboard. The KeyEvent object describes the nature of the event (namely, that a key has been pressed, released, or typed) and the value of the key. https://openjfx.io/javadoc/17/javafx.graphics/javafx/scene/input/KeyEvent.html, key code: https://openjfx.io/javadoc/17/javafx.graphics/javafx/scene/input/KeyCode.html

- **Listeners**

You can add a listener to process a value change in an observable object. An instance of Observable is known as an **observable object**, which contains the **addListener(InvalidationListener listener)** method for adding a listener. The listener class must implement the functional interface InvalidationListener to override the invalidated(Observable o) method for handling the value change. Once the value is changed in the Observable object, the listener is notified by invoking its invalidated(Observable o) method. Every binding property is an instance of Observable.

- **Anonymous Inner classes for Event Handlers**

An inner class, or nested class, is a class defined within the scope of another class. Normally, you define a class as an inner class **if it is used only by its outer class**. An inner class has the following features:

1. An inner class is compiled into a class named OuterClass$InnerClass.class.
2. An inner class can reference the data and the methods defined in the outer class in which it nests, so you need not pass the reference of an object of the outer class to the constructor of the inner class. For this reason, inner classes can make programs simple and concise. For example, the "data" in the sample code below is directly reference in the inner class.
3. An inner class can be defined with a visibility modifier subject to the same visibility rules applied to a member of the class. An inner class can be defined as static. A static inner class can be accessed using the outer class name. A static inner class cannot access non-static members of the outer class.
4. Objects of an inner class are often created in the outer class. However, you can also create an object of an inner class from another class. If the inner class is non-static, you must first create an instance of the outer class, then use the following syntax to create an object for the inner class:

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

5. If the inner class is static, use the following syntax to create an object for it:

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

For example,

```
public class Outerclass {
    private int data = 0;
    /** A method in the outer class */
    public void m() { //do something
    }
    // An inner class
    class InnerClass {
    /**
    A method in the inner class referencing data and method in outer class
    */
        public void mi() {
            data++;
            m();
        }
    }
}
```

A simple use of inner classes is to combine dependent classes into a primary class. This reduces the number of source files. It also makes class files easy to organize since they are all named with the primary class as the prefix. For example, rather than creating the two source files OuterClass.java and InnerClasss.java, you can merge class InnerClass into class OuterClass and create just one source file, OuterClass.java. The resulting class files are Outerclass.class and OuterClass$InnerClass.class.

Another practical use of inner classes is to **avoid class-naming conflicts**. A handler class is designed specifically to create a handler object for a GUI component (e.g., a button). The handler class will not be shared by other applications and therefore is appropriate to be defined as an inner class. For example,

```
public void start(Stage primaryStage) {
    //...
    btEnlarge.setOnAction(new EnlargeHandler()); //new instance of inner class
}
//inner class
class EnlargeHandler implements EventHandler<ActionEvent> {
    public void handle(ActionEvent e) {
        circlePane.enlarge();
    }
}
```

An **anonymous inner class is an inner class without a name**. It combines defining an inner class and creating an instance of the class into one step. For example, the code above could be rewritten and simplified as an anonymous inner as follows.

```
public void start(Stage primaryStage) {
           //the grey color part can be omitted.
    btEnlarge.setOnAction(new class EnlargeHandler implements
                                EventHandler<ActionEvent>() {
        public void handle(ActionEvent e) {
            circlePane.enlarge();
        } });
}
```

Anonymous classes are expressions, which means that you define the class in another expression. The anonymous class expression consists of the following:

1. The **new** operator.
2. The name of an interface to implement or a class to extend.
3. 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 the example in the next page.
4. A body, which is a class declaration body. More specifically, in the body, method declarations are allowed but statements are not.

Another example of anonymous classes is shown at the end of this note.

- **Lambda Expressions for Event Handlers**

One issue with anonymous classes is that if the implementation of your anonymous class is very simple, such as an interface that contains only one method, then the syntax of anonymous classes may seem unwieldy and unclear. In these cases, you're usually trying to pass functionality as an argument to another method, such as what action should be taken when someone clicks a button. Lambda expressions enable you to do this, **to treat functionality as method argument, or code as data.** Lambda expressions let you express instances of single-method classes more compactly.

The **EventHandler<ActionEvent> interface** contains only one method, **handle**. Instead of implementing this method with a new class, the example in the previous page uses an anonymous class expression. Notice that this expression is **the argument passed to the btEnlarge.setOnAction method**. Because the EventHandler<ActionEvent> interface contains only one method, you can use **a lambda expression** instead of an anonymous class expression.

Lambda expressions can be viewed as an anonymous class with a concise syntax. The two code segments below are equivalent.

```
//anonymous inner class
  bt.setOnAction(new EventHandler<ActionEvent>() {
      @Override // Override the handle method
      public void handle(ActionEvent e) {
        bt.setText("Welcome World!!");
      }
  } );
//Lambda expression
bt.setOnAction(e-> {
  bt.setText("Welcome World!!");
} );
```

As another example of the lambda expressions.

```
//anonymous innner class

printPersons(
    roster,
    new CheckPerson() {
        public boolean test(Person p) {
            return p.getGender() == Person.Sex.MALE
                && p.getAge() >= 18
                && p.getAge() <= 25;
        }
    }
);
```

```
//lambda expression

printPersons(
    roster,
    (Person p) -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);
```

In the above example, the CheckPerson is a functional interface. A functional interface is any interface that contains only one abstract method. (A functional interface may contain one or more default methods or static methods.) Because a functional interface contains only one abstract method, **you can omit the name of that method when you implement it.**

```
printPersonsWithPredicate(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);
```

The syntax of a lambda expression consists of the following:

1. A comma-separated list of formal parameters enclosed in parentheses. The CheckPerson.test method below contains one parameter, p, which represents an instance of the Person class. Note: You can omit the data type of the parameters in a lambda expression. In addition, you can omit the parentheses if there is only one parameter like the above example.
2. The arrow token, **->**
3. A body, which consists of a single expression or a statement block.

If you specify a single expression, then the Java runtime evaluates the expression and then returns its value. Alternatively, you can use a return statement:

```
p -> {
    return p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25;
}
```

A return statement is not an expression; in a lambda expression, you **must enclose statements in braces** ({}). However, you **do not have to enclose a void method invocation in braces**. For example, the following is a valid lambda expression:

```
email -> System.out.println(email);
```

Note that a lambda expression looks a lot like a method declaration; **you can consider lambda expressions as anonymous methods—methods without a name.** The following example, Calculator, is an example of lambda expressions that take more than one formal parameter:

```
public class Calculator {
    interface IntegerMath {
        int operation(int a, int b);
    }
    public int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    }
    public static void main(String... args) {
        Calculator myApp = new Calculator();
        IntegerMath addition = (a, b) -> a + b;
        IntegerMath subtraction = (a, b) -> a - b;
        System.out.println("40 + 2 = " +
            myApp.operateBinary(40, 2, addition));
        System.out.println("20 - 10 = " +
            myApp.operateBinary(20, 10, subtraction));
    }
}
```

- **Example of anonymous inner classes.**

```
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();

        // Consider the code below.
        // 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.

        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();
    }
}
```