Exception handling

Like people, processes cannot always foresee the consequences of their actions.

Java divides the realm of the unpredictable into exceptions and events.

- An exception happens *synchronously* with a process; it is the immediate consequence of some action, such as division by 0.
- An event happens *asynchronously*, often as a result of something the user does. The Java GUI is event driven; user actions initiate events.

Normally, a programmer writes code to deal with an exceptional condition where that condition is detected.

For example, in Smalltalk’s lazy initialization, the programmer codes a check for a null instance variable, and if it is found to be null, then initializes it.

```plaintext
sum isNil ifTrue: [sum := 0].
```

This is a good way to handle exceptions in code that doesn’t have to deal with a lot of them.

Why isn’t it a good idea in production code?

Therefore, code is more readable if the language provides a mechanism to deal with exceptions elsewhere.

These exceptions include events like—

- _______ out of bounds,
- arithmetic overflow and _________,
- memory exhaustion,
- nonexistent files or insufficient permissions.
It is possible for a programmer to define additional exceptions, but it is best not to overuse exceptions.

They are expected to be unusual, so the compiler may not concentrate on optimizing code for dealing with them.

When an error is detected by a method, the method will throw an exception.

The programmer can choose whether to write code that deals with the exception.

- If (s)he chooses not to deal with the exception, (s)he writes in the procedure heading, e.g.,
  
  ```
  throws IOException
  ```

  to indicate that the method will not deal with any exceptions involving I/O.

  If this is done in the `main` method, its header might read—

  ```
  public static void main(string argv[])
  throws IOException {
  ...
  }
  ```

  In this case, handling the exception is the responsibility of

- If (s)he chooses to deal with the exception, (s)he encloses the code that may generate the exception in a try block.

  The try block is immediately followed by one or more catch blocks.

Java stops executing statements in the try block as soon as an exception is thrown.

Suppose we are trying to read from a file. If an exception occurs, exception-handling code prints out the name of the exception.
try {
    ...
    FileInputStream inputFile =
        new FileInputStream("input.data");
    ...
} catch (IOException e) {
    System.out.println(e);
}

Note that the I/O exception is actually an instance of an exception class. The exception may be an instance of FileNotFoundException, which is a subclass of IOException.

Multiple catch blocks permit different code to be used to handle different exceptions. For example,

    • one piece of code can be used to handle a FileNotFoundException (perhaps a retry, in case the file is being created “as we speak”), and
    • another piece of code to handle any other kind of I/O exception (perhaps just reporting the exception that occurred).

    public static String readInputFile(String fileName) {
        boolean tryAgain = true;
        while (tryAgain) {
            try {
                tryAgain = false;
                FileInputStream inputfile =
                    new FileInputStream(fileName);
                ...
            }
            catch (FileNotFoundException)
                tryAgain = true;
        }
        catch (IOException e {
            System.out.println(e);
        }
    }
Occasionally, you may want to have a piece of code executed after a `try` statement, regardless of whether an exception occurred.

In this case, Java provides a `finally` block:

```java
try {
 ...
}
catch (exception-class name e){
 ...
}
finally {
  //clean-up statements
}
```

**Event handling in Java**

An event happens *asynchronously*, often as a result of something the user does.

We have already seen one kind of event—closing a window—in Lecture 12. Other events are initiated by the user, for example,

- moving the mouse,
- clicking a mouse button,
- pressing a key, or
- resizing a component.

Events are sent to a Java program by the underlying windowing system.

Information about event is stored in an instance of a subclass of `EventObject`. Most of the events we will see are `AWTEvents`.

Here is the hierarchy of `AWTEvents`.

```java
java.lang.Object
java.util.EventObject
java.awt.AWTEvent
  java.awt.event.ActionEvent
  java.awt.event.AdjustmentEvent
```
Java.awt.ItemEvent
java.awt.ComponentEvent
  java.awt.ContainerEvent
java.awt.FocusEvent
java.awt.InputEvent
  java.awt.KeyEvent
  java.awt.MouseEvent
java.awt.PaintEvent
java.awt.WindowEvent

To get an event processed, the programmer must write code to do two things:

- register an event listener, and
- implement an event handler.

Listener interfaces
In our example from Lecture 12, we used
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE).

This could also be done by having the HelloWorldSwing class implement a windowClosing method to exit the system:

public void windowClosing(WindowEvent e) {
    System.exit(0);
}

To do this, HelloWorldSwing would have to implement the WindowListener interface.

Exercise: What other changes would we have to make in the code for the example?

WindowListener is an example of an event-listener interface. In general, for each leaf Event class there is a corresponding listener. Two exceptions are

- MouseEvent, which has two listeners, MouseListener and MouseMotionListener, and
• **PaintEvent**, which is a special event type used to ensure that paint/update method calls are serialized properly with other events.

Now, let’s list the **Listener** interfaces.

### Adapters

Some interfaces, e.g., **WindowListener**, have many methods:

```java
public void windowOpened(WindowEvent e)
public void windowIconified(WindowEvent e)
public void windowDeiconified(WindowEvent e)
public void windowClosed(WindowEvent e)
public void windowActivated(WindowEvent e)
public void windowDeactivated(WindowEvent e)
```

We could just implement all of these methods in a class like **HelloWorldSwing**.

But, because this is tedious, the `java.awt.event` package defines **adapters** for most of the **Listeners**.

Adapters are classes that implement all of the methods in their corresponding interfaces, with null bodies.

If the programmer only wants to define a few methods of the interface, (s)he can extend an adapter class, and override a few of its methods.

However, as we shall see, this usually means that an object can no longer serve as a handler for its own events.
Types of events
Let us take a look at a few components and the events that they generate.

Button events
A button generates a button event when a user clicks it with a mouse.

Buttons are usually created with a one-parameter constructor. This parameter specifies the button’s label:

```java
JButton myButton = new JButton("OK");
```

An ActionListener receives events from the button. Here is a simple code segment that defines a button and processes its event:

```java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class OKButton extends JFrame {
    private JButton myButton;
    public void init() {
        myButton = new JButton("OK");
        myButton.addActionListener(new ButtonListener(this));
        add(myButton);
    }
}

class ButtonListener implements ActionListener {
    JFrame myFrame;
    public ButtonListener(JFrame f) {
        myFrame = f;
    }
    public void actionPerformed(ActionEvent e) {
        myFrame.setTitle("Button was pressed!");
    }
}
```

Note that the ActionListener interface is implemented. This is because clicking a button with a mouse generates an ActionEvent.
When a button is pressed and released,

- the AWT sends an instance of `ActionEvent` to the button, by calling `processEvent` on the button.
- the button’s `processEvent` method calls `processActionEvent`.
- `processActionEvent` passes the action event on to any `ActionListener` s that have registered an interest in events from this button. It does this by calling the `ActionListener`’s `actionPerformed` method.

In the code above, it is an instance of `ButtonHandler` that is interested in events from `myButton`.

- `myButton` adds an instance of `ButtonListener` as its `ActionListener`.
- Conversely, `ButtonListener` keeps track of which frame it is listening for, by taking the frame as a parameter to its constructor.

This makes it possible for the `ButtonListener` to perform an action involving the frame (specifically, uses `setTitle` to display a string in the title bar at the top of the frame.

*Keyboard events*

Pressing or releasing a key generates a keyboard event.

The `KeyListener` interface recognizes three kinds of events:

- `keyPressed(KeyEvent)` is called when an *action key* is depressed. Action keys include arrow keys, function keys, Home, End, Page Up, Page Down, Num Lock, Print Screen, Scroll Lock, Caps Lock, and Pause.
- `keyTyped(KeyEvent)` is called when any other key is depressed.
- `keyReleased(KeyEvent)` is called after any key is released that has generated either of the above two events.
Four methods are useful in figuring out which key has been typed:

- `int getKeyCode()` returns the integer key code associated with the key in this event (only for `keyPressed` events).
- `char getKeyChar()` returns the character associated with the key (if any).
- `String getKeyText(int keyCode)` returns a string describing the `keyCode`, such as “PAUSE” or “F1”.
- `String getKeyModifiersText(int modifiers)` returns a string describing the modifier key(s), such as “Shift” or “Ctrl+Shift”.

A sample program that captures and processes key events is found at:

http://java.sun.com/docs/books/tutorial/uiswing/events/keylistener.html

```java
public class KeyEventDemo ... implements KeyListener ... {
    ...//where initialization occurs:
    typingArea = new JTextField(20);
    typingArea.addKeyListener(this);

    /** Handle the key-typed event from the text field. */
    public void keyTyped(KeyEvent e) {
        displayInfo(e, "KEY TYPED: ");
    }

    /** Handle the key-pressed event from the text field. */
    public void keyPressed(KeyEvent e) {
        displayInfo(e, "KEY PRESSED: ");
    }

    /** Handle the key-released event from the text field. */
    public void keyReleased(KeyEvent e) {
        displayInfo(e, "KEY RELEASED: ");
    }

    protected void displayInfo(KeyEvent e, String s){
        ...}
        char c = e.getKeyChar();
        int keyCode = e.getKeyCode();
        int modifiers = e.getModifiers();
```
tmpString = KeyEvent.getKeyModifiersText(modifiers);

...//display information about the KeyEvent ...

Mouse events

There are two interfaces for dealing with mouse events, MouseListener and MouseMotionListener.

MouseListener defines these methods:

- `mousePressed(MouseEvent)` Invoked when a mouse button is pressed with the mouse cursor over a component.
- `mouseClicked(MouseEvent)` Invoked when a mouse button is pressed and released over a component without the mouse being moved.
- `mouseReleased(MouseEvent)` Invoked when a mouse's button is released after the mouse has been dragged. This event is always preceded by a `MouseDragged` event.
- `mouseEntered(MouseEvent)` Invoked when the mouse cursor enters the bounds of a component.
- `mouseExited(MouseEvent)` Invoked when the mouse cursor moves outside the bounds of a component.

MouseMotionListener defines these methods:

- `mouseDragged(MouseEvent)` Invoked when a mouse button is pressed while the mouse is over a component, and then dragged. Always preceded by a `MousePressed` event.
- `mouseMoved(MouseEvent)` Invoked when the mouse is moved with the cursor is over a component, without any buttons being pressed.

What is the sequence of events when a mouse is dragged?
What is the difference between `MouseClicked` and `MouseReleased`?

Among other things, each `MouseEvent` object contains the $x$ and $y$ coordinates where the event occurred. These values can be retrieved via the `int getX()` and `int getY()` methods of the mouse event.

On some systems, mice have more than one button. `MouseEvent` inherits methods from `InputEvent` to allow it to distinguish among the buttons.

- `isMetaDown()` returns true when the user clicks the right button on a mouse with two or three buttons. How do you think this might be simulated on a one-button mouse?

- `isAltDown()` returns true when the user clicks the middle button on a three-button mouse. How do you think this might be simulated on a one- or two-button mouse?

For examples of mouse listeners, see http://java.sun.com/docs/books/tutorial/uiswing/events/mouselistener.html.

In general, documentation on events can be found at http://java.sun.com/docs/books/tutorial/uiswing/overview/event.html

*Item events*

Several of the Swing classes implement the interface `ItemSelectable`, which indicates that they contain a set of items from which zero or more items may be selected.

Among these classes are `JButton`, `JCheckBox`, `JRadioButton`, and `JComboBox`. 
• A checkbox is a box that may be selected or deselected. Any number of check boxes in a group may be selected.

• “Radio buttons” are a set of buttons of which only one can be selected at the same time. Such a set of buttons are members of the same ButtonGroup.

• A combo box, sometimes called a drop-down list or dropbox, allows a user to select an item from the list.

Choosing an item generates an ItemEvent. The ItemEvent class contains a method getItemSelectable() to get the ItemSelectable object that generated the item event. Whenever an item’s state changes, the itemStateChanged() of the ItemListener class is called.

Here is a way that such an event might be processed, assuming that selected is a Boolean array that tells for each item whether it has been selected:

```java
class ItemEventHandler implements ItemListener {
    ...
    public void itemStateChanged(ItemEvent e) {
        Choice chosen = (Choice) e.getItemSelectable();
        selected[chosen.getSelectedIndex()] = true;
    }
}
```

**Inner classes**

Languages like Algol, PL/1 and Pascal allow a programmer to nest procedures within procedures.

The idea behind this was that certain names in the program need to be known only in the vicinity where they are defined.

From the beginning, Java adopted this idea of limited scope for variables and methods, but not for packages.

In Java 1.0, all classes were top level; all classes were defined at the same level. If a class was known anywhere in a source file, it was known throughout the file.
Java 1.1 allows classes to be nested within other classes. A class defined inside another class is known as an *inner class*.

Inner classes are useful for at least two reasons:

- The name of the inner class is known only within its scope. Thus, it does not “pollute” the namespace of the package.
- The code of an inner class can refer directly to names from enclosing scopes, including both class and instance variables and methods, and local variables of enclosing blocks.

**A Fixed Stack**

Here is an example adapted from the Java tutorial documentation, a class *FixedStack*, which implements a stack, and defines an enumerator of elements from the stack, from top to base:

```java
public class FixedStack {
    Object array[];
    int top = 0;
    FixedStack(int fixedSizeLimit) {
        array = new Object[fixedSizeLimit];
    }

    public void push(Object item) {
        array[top++] = item;
    }

    public boolean isEmpty() {
        return top == 0;
    }

    // other stack methods go here...

    /** This adapter class is defined as part of its
    *  target class, It is placed alongside the
    *  variables it needs to access.
    */
    class Iterator implements java.util.Iterator {
        int count = top;
        public boolean hasNext() {
            return count > 0;
        }

        public Object next() {
            if (count == 0)
                throw new java.util.NoSuchElementException("FixedStack");
        }
    }
}
```
return array[--count];
}
public void remove() { // We won’t specify this.
    ...
}
public java.util.Iterator elements() {
    return new Iterator();
}

The interface `java.util.Iterator` is used to communicate a series of values to a client.

Note that `FixedStack` does not directly implement the `Iterator` interface. Why not?

Thus, a separate adapter class is used to return the elements one by one.

Note that the adapter class needs to access the array containing the stack elements.

It can directly refer to instance variables of the stack, since the adapter is defined inside `FixedStack`.

**Inner classes for adapters**

Since an AWT component and its event listener are closely related, it is helpful to use inner classes for event adapters too. A useful example is found at http://java.sun.com:81/docs/books/faq/src/awt/EventListener11Example.java.

It contains code for implementing a `WindowListener` and a `MouseMotionListener`. Here are some excerpts:

```java
import java.awt.*;
import java.awt.event.*;
import java.swing.*;
```
public class EventListener11Example {
/** An inner class that closes the window and  
* quits the app. */  
class WindowQuitter extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        e.getWindow().dispose();
        System.exit(0);
    }
}

Frame mainFrame = new JFrame("EventListener11Example");
/** Performs initialization, builds the interface. */
public void init() {
    /* Initialize event handlers. */
    mainFrame.addWindowListener(new WindowQuitter());

    /* Show the main window. */
    mainFrame.setBounds(100, 100, 500, 200);
    mainFrame.show();
}
}

Sometimes it is not necessary to define a named class (like WindowQuitter) at all. If the code for a class is very simple, it can be declared in line with the add...Listener method.

Here is an example of how this could be done in the HelloWorldSwing example from the start of Lecture 12:

frame.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
});

Summary

- Events and exceptions happen asynchronously with the rest of the program.
- Exceptions are handled with try ... catch blocks.
• Events are handled by registering an event listener and implementing an event handler.

• Inner classes are often a useful way to implement an event handler.

• The `java.util.Iterator` provides a way to return each element of a data structure.