Event-Driven Programming

Lecture 4

Jenny Walter

Fall 2008

Simple Graphics Program

```
import acm.graphics.*;
import java.awt.*;
import acm.program.*;

public class Circle extends GraphicsProgram {

   public void run() {
      GOval circle = new GOval(200,200,200,200);
      circle.setFilled(true);
      circle.setFillColor(new Color(200,0,0));
      add(circle);
   }

   public static void main(String[] args) {
      new Circle().start();
   }
}
```

The Java Event Model

Programs that support user control via mouse or keyboard are called **interactive programs**.

User actions such as clicking or moving the mouse are called **events**. Programs that respond to events are said to be **event-driven**.

When you write an event-driven graphics program, you indicate the events to which you wish to respond by designating an object as a **listener** for that event. When the event occurs, a message is sent to the listener, triggering a response.

Event Types

- Some Java event types:
  - **Mouse events**, which occur when the user moves or clicks the mouse
  - **Keyboard events**, which occur when the user types on the keyboard
  - **Action events**, which occur in response to user-interface actions
- Each event type is associated with a set of methods that specify how listeners should respond. These methods are defined in a **listener interface** for each event type.
- As an example, one of the methods in the mouse listener interface is `mouseClicked`. As you would expect, Java calls that method when you click the mouse.
- Listener methods like `mouseClicked` define a parameter that contains information about the event. In the case of `mouseClicked`, the argument is a `MouseEvent` indicating the location at which the click occurred.

Responding to Mouse Events

You can make programs respond to mouse events by following these general steps:

1. Define a run method that calls `addMouseListeners()`
2. Write new definitions of any listener methods you need.

The most common mouse events are shown in the following table, along with the name of the appropriate listener method:

`mouseClicked(`*e*`)`	Called when the user clicks the mouse
`mousePressed(`*e*`)`	Called when the mouse button is pressed
`mouseReleased(`*e*`)`	Called when the mouse button is released
`mouseMoved(`*e*`)`	Called when the user moves the mouse
`mouseDragged(`*e*`)`	Called when the mouse is dragged with the button down

The parameter *e* is a `MouseEvent` object, which provides more data about the event, such as the location of the mouse.

Mouse Listeners in the ACM Libraries

Java's approach to mouse listeners is not as simple as the previous slide implies. To maximize efficiency, Java defines two distinct mouse listener interfaces:

The `MouseListener` interface responds to mouse events that happen in isolation or infrequently, such as clicking the mouse button.

The `MouseMotionListener` interface responds to the much more rapid-fire events that occur when you move or drag the mouse.

The packages in the ACM Java Libraries adopt the following strategies to make mouse listeners easier to use:

The `Program` class includes empty definitions for every method in the `MouseListener` and the `MouseMotionListener` interfaces. Doing so means that you don't need to define all of these methods but can instead simply override the ones you need.

The `GraphicsProgram` class defines the `addMouseListeners` method, which adds the program as a listener for both types of events.

The net effect of these simplifications is that you don't have to think about the difference between these two interfaces.

A Simple Line-Drawing Program

```
public class DrawLines extends GraphicsProgram {
/* Initializes the program by enabling the mouse listeners */
   public void run() {
      addMouseListeners();
   }
/* Called on mouse press to create a new line */
   public void mousePressed(MouseEvent e) {
      line = new GLine(e.getX(), e.getY(), e.getX(), e.getY());
      add(line);
   }
/* Called on mouse drag to extend the endpoint */
   public void mouseDragged(MouseEvent e) {
      line.setEndPoint(e.getX(), e.getY());
   }
/* Private instance variables */
   private GLine line;

   public static void main(String[] args){
      new DrawLines().start();
   }
}
```

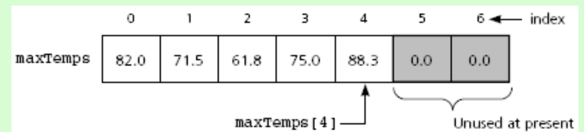
In this program, the Circle moves with the mouse

```
import acm.graphics.*;
import java.awt.*;
import acm.program.*;
import java.awt.event.*;

public class MovingCircle extends GraphicsProgram {

   private GOval circle;

   public void run() {
      circle = new GOval(200,200,200,200);
      circle.setFilled(true);
      circle.setFillColor(new Color(200,0,0));
      add(circle);
      addMouseListeners();
   }
   public void mouseMoved(MouseEvent e) {
      double x = e.getX() - circle.getWidth()/2;
      double y = e.getY() - circle.getHeight()/2;
      if (x < 0) x = 0;
      if (x > getWidth() - circle.getWidth())
         x = getWidth() - circle.getWidth();
      if (y < 0) y = 0;
      if (y > getHeight() - circle.getHeight())
         y = getHeight() - circle.getHeight();
      circle.setLocation(x,y);
   }
   public static void main(String[] args) {
      new MovingCircle().start();
   }
}
```

Arrays

- Collection of elements with the same data type and pre-defined, fixed size
- Array elements have an order
- Support direct and random access
- One-dimensional arrays
  - Declaration example
    - `final int DAYS_PER_WEEK = 7;`
    - `double [] maxTemps = new double[DAYS_PER_WEEK];`
  - Length of an array is accessible using data field `length` (e.g., `maxTemps.length = 7`)
  - Use an index or subscript to access an array element (e.g., `maxTemps[0] = 5.0;`)

Arrays

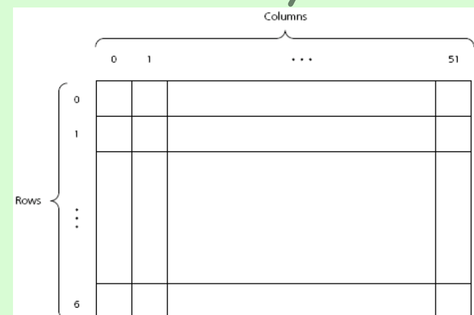


One-dimensional array of at most seven elements

Arrays

- One-dimensional arrays (continued)
  - Initializer list example
    - `double [] weekDayTemps = {82.0, 71.5, 61.8, 75.0, 88.3};`
  - You can also declare array of object references
- Multidimensional arrays
  - Use more than one index
  - Declaration example
    - `final int DAYS_PER_WEEK = 7;`
    - `final int WEEKS_PER_YEAR = 52;`
    - `double[][] minTemps = new double[DAYS_PER_WEEK][WEEKS_PER_YEAR];`

Arrays



A two-dimensional array

Arrays

- Passing an array to a method
  - Declare the method as follows:

```
public double averageTemp(double[] temps,
  int n)
```

  - Invoke the method by writing:

```
double avg = averageTemp(maxTemps, 6);
```

  - Location of array is passed to the method
    - Cannot return a new array through this value
  - Method can modify content of the array

Enhanced For Statement

- The `for` loop and arrays

```
for (ArrayElementType variableName : arrayName)
  statement
```

- The enhanced `for` statement

```
int[] numbers = {1,2,3,4,5,6,7,8,9,10};
for (int item : numbers) {
   System.out.println("Count is: " + item);
}
```

Java Exceptions

- Exception
  - Handles an error during execution
- Throw an exception
  - To indicate an error during a method execution
- Catch an exception
  - To deal with the error condition

Catching Exceptions

- Java provides `try-catch` blocks
  - To handle an exception
- Place statement that might throw an exception within the `try` block
  - Must be followed by one or more `catch` blocks
  - When an exception occurs, control is passed to `catch` block
- `Catch` block indicates type of exception you want to handle

Catching Exceptions

- `try-catch` blocks syntax

```
try {
  statement(s);
}
catch (exceptionClass identifier) {
  statement(s);
}
```

- Some exceptions from the Java API cannot be totally ignored
  - You must provide a handler for that exception

Catching Exceptions

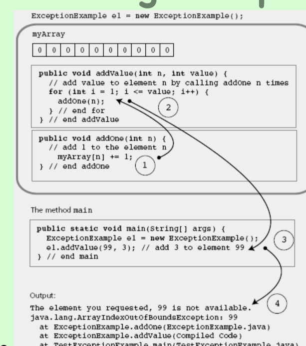


Figure 1-9

Flow of control in a simple Java application

Catching Exceptions

- Types of exception
  - Checked exceptions
    - Instances of classes that are subclasses of `java.lang.Exception`
    - Must be handled locally or thrown by the method
    - Used when method encounters a serious problem
  - Runtime exceptions
    - Occur when the error is not considered serious
    - Instances of classes that are subclasses of `java.lang.RuntimeException`

Throwing Exceptions

- `throws` clause
  - Indicates a method may throw an exception
    - If an error occurs during its execution
  - Syntax

```
public methodName throws ExceptionClassName
```

- `throw` statement
  - Used to throw an exception at any time
  - Syntax

```
throw new exceptionClass(stringArgument);
```

- You can define your own exception class

Text Input and Output

- Input and output consist of streams
- Streams
  - Sequence of characters that either come from or go to an I/O device
  - `InputStream` – Input stream class
  - `PrintStream` – Output stream class
- java.lang.System provides three stream variables
  - `System.in` – standard input stream
  - `System.out` – standard output stream
  - `System.err` – standard error stream

Input

- Prior to Java 1.5

```
BufferedReader stdin = new BufferedReader(new
  InputStreamReader(System.in));

String nextLine = stdin.readLine();

StringTokenizer input = new
  StringTokenizer(nextLine);
x = Integer.parseInt(input.nextToken());
y = Integer.parseInt(input.nextToken());
```

Input

- Java 1.5 – The `Scanner` class

```
int nextValue;
int sum=0;
Scanner kbInput = new Scanner(System.in);
nextValue = kbInput.nextInt();
while (nextValue > 0) {
  sum += nextValue;
  nextValue = kbInput.nextInt();
} // end while
kbInput.close();
```

Input

- Java 1.5 – The `Scanner` class (continued)
  - More useful `next` methods
    - `String next();`
    - `boolean nextBoolean();`
    - `double nextDouble();`
    - `float nextFloat();`
    - `int nextInt();`
    - `String nextLine();`
    - `long nextLong();`
    - `short nextShort();`

Output

- Methods `print` and `println`
  - Write character strings, primitive types, and objects to `System.out`
  - `println` terminates a line of output so next one starts on the next line
  - When an object is used with these methods
    - Return value of object's `toString` method is displayed
    - You usually override this method with your own implementation
  - Problem
    - Lack of formatting abilities

Output

- Method `printf`
  - C-style formatted output method
  - Syntax

```
printf(String format, Object... args)
```

  - Example:

```
String name = "Jamie";
int x = 5, y = 6;
int sum = x + y;
System.out.printf("%s, %d + %d = %d", name,
  x, y, sum);
//produces output Jamie, 5 + 6 = 11
```

Output

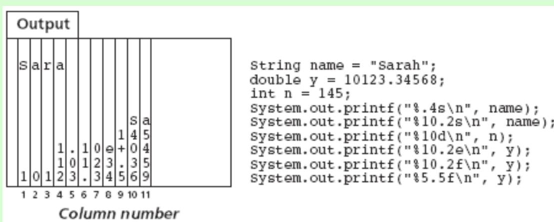


Figure 1-10
Formatting example with `printf`

File Input and Output

- File
  - Sequence of components of the same type that resides in auxiliary storage
  - Can be large and exists after program execution terminates
- Files vs. arrays
  - Files grow in size as needed; arrays have a fixed size
  - Files provides both sequential and random access; arrays provide random access
- File types
  - Text and binary (general or nontext) files

Text Files

- Designed for easy communication with people
  - Flexible and easy to use
  - Not efficient with respect to computer time and storage
- End-of-line symbol
  - Creates the illusion that a text file contains lines
- End-of-file symbol
  - Follows the last component in a file
- Scanner class can be used to process text files

Text Files

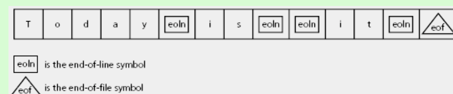


Figure 1-11
A text file with end-of-line and end-of-file symbols

Text Files

- Example

```
String fname, lname;
int age;
Scanner fileInput;
File inFile = new File("Ages.dat");
try {
   fileInput = new Scanner(inFile);
   while (fileInput.hasNext()) {
     fname = fileInput.next();
     lname = fileInput.next();
     age = fileInput.nextInt();
     age = fileInput.nextInt();
     System.out.printf("%s %s is %d years old.\n",
     fname, lname, age);
   } // end while
   fileInput.close();
} // end try
catch (FileNotFoundException e) {
   System.out.println(e);
} // end catch
```

Text Files

- Open a stream to a file
  - Before you can read from or write to a file
  - Use class `FileReader`
    - Constructor throws a `FileNotFoundException`
  - Stream is usually embedded within an instance of class `BufferedReader`
    - That provides text processing capabilities
  - `StringTokenizer`
    - Used to break up the string returned by `readLine` into tokens for easier processing

Text Files

- Example

```
BufferedReader input;
StringTokenizer line;
String inputLine;
try {
   input = new BufferedReader(new FileReader("Ages.dat"));
   while ((inputLine = input.readLine()) != null) {
    line = new StringTokenizer(inputLine);
    // process line of data
    ...
   }
} // end try
catch (IOException e) {
   System.out.println(e);
   System.exit(1); // I/O error, exit the program
} // end catch
```

Text Files

File output

You need to open an output stream to the file

Use class `FileWriter`

Stream is usually embedded within an instance of class `PrintWriter`

That provides methods `print` and `println`

Text Files

Example

```
try {
  PrintWriter output = new PrintWriter(new
  FileWriter("Results.dat"));
  output.println("Results of the survey");
  output.println("Number of males: " +
                 numMales);
  output.println("Number of females: " +
                  numFemales);
  // other code and output appears here...
} // end try
catch (IOException e) {
  System.out.println(e);
  System.exit(1); // I/O error, exit the program
} // end catch
```

Text Files

Closing a file

Syntax

```
myStream.close();
```

Adding to a text file

When opening a file, you can specify if file should be replaced or appended

Syntax

```
PrintWriter ofStream = new
     PrintWriter(new FileOutputStream
              ("Results.dat", true));
```