ON TO JAVA

A Short Course in the Java Programming Language

Credits

• Lectures correspond to and are based almost entirely on material in the text:
  – Online version at: http://www.ai.mit.edu/people/phw/OnToJava/
• Lecture notes compiled by:
  – R. Scott Cost, UMBC

How this Book Teaches You the Java Programming Language

-1-

Highlights

• Its features make Java ideally suited for writing network-oriented programs.
• Java is an object-oriented programming language. When you use an object-oriented programming language, your programs consist of class definitions.
• Java class definitions and the programs associated with classes are compiled into byte code, which facilitates program portability.
• Java class definitions and the programs associated with them can be loaded dynamically via a network.
• Java's compiler detects errors at compile time; the Java virtual machine detects errors at run time.
• Java programs can be multithreaded, thereby enabling them to perform many tasks simultaneously.
• Java programs collect garbage automatically, relieving you of tedious programming and frustrating debugging, thereby increasing your productivity.
• Java has syntactical similarities with the C and C++ languages.
• This book introduces and emphasizes powerful ideas, key mechanisms, and important principles.
How to Compile and Execute a Simple Program

Java Programs

• Java programs are collections of class definitions
• Edit Java source file (*.java) with any editor, or use one of many IDEs
• Compile source to Java Byte Code

Basic Java Program

• Consider this basic example program:
  
  ```java
  public class Demonstrate {
    public static void main (String argv[]) {
      6 + 9 + 8;
    }
  }
  ```

  Main is called when class is invoked from the command line
  
  * Note: Applets do not have 'main' methods

Basic Java Program...

```java
public class Demonstrate {
    public static void main (String argv[]) {
      6 + 9 + 8;
    }
}
```

• ‘public’ determines the methods accessibility
• ‘static’ declares this to be a class method
• ‘void’ is the return type (in this case, none)
Basic Java Program...

```java
public class Demonstrate {
    public static void main (String argv[]) {
        6 + 9 + 8;
    }
}
```

- Methods have a (possible empty) parameter specification
- For main, this standard parameter is analogous to C's `argv`

Basic Java Program...

```java
public class Demonstrate {
    public static void main (String argv[]) {
        6 + 9 + 8;
    }
}
```

- Method body; in this case, a simple arithmetic expression
- Java's statement separator is the `;`
- Note: no return statement for a void method

Basic Java Program...

```java
public class Demonstrate {
    public static void main (String argv[]) {
        System.out.println("The movie rating is ");
        System.out.println(6 + 9 + 8);
    }
}
```

- Addition of these statements sends output to stdout
- `System` is a class in the `java.lang` package; `out` is an output stream associated with it
Demonstration

Put the source into a file Demonstrate.java with your favorite editor
C:\java>javac Demonstrate.java
C:\java>java Demonstrate
The movie rating is
23

How to Declare Variables

-3-

Variables

- Variables are named by identifiers, and have data types.
- By convention, java variable names begin in lower case, and are punctuated with upper case letters. Examples:
  - fileHasBeenRead
  - previousResponse

Identifiers

- Identifiers: name consisting of letters, digits, underscore, ‘$’
  - cannot begin with a digit
- int is a 32 bit signed integer
- double is a 64 bit signed floating point number
Declaration

public class Demonstrate {
    public static void main (String argv[]) {
        int script;
        int acting;
        int direction;
        ...
    }
}

Assignment

public class Demonstrate {
    public static void main (String argv[]) {
        int script = 6;
        int acting = 9;
        int direction = 8;
        ...
    }
}

• Assignment can occur in declaration
• All Java variables have a default value (0 for int)
• Java compiler will complain if you do not initialize

Assignment Operator

public class Demonstrate {
    public static void main (String argv[]) {
        int result, script = 6, acting = 9, direction = 8;
        result = script;
        result = result + acting;
        result = result + direction;
        System.out.println("The rating of the movie is ");
        System.out.println(result);
    }
}

Declarations

public class Demonstrate {
    public static void main (String argv[]) {
        int result, script = 6;
        result = script;
        int acting = 9;
        result = result + acting;
        ...
        System.out.println("The rating of the movie is ");
        System.out.println(result);
    }
}

• Declarations can occur anywhere in code
Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
<th>Stores</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>1</td>
<td>integer</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>integer</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>integer</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
<td>integer</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>floating-point number</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>floating-point number</td>
</tr>
</tbody>
</table>

Inline Comments

• Short comments
  // comments text to the end of the line

• Multi-line comments
  /*
  Comment continues until an end sequence is encountered
  */

Comments…

/**
 * Many comments in Java are written
 * in this form, for use in auto-
 * documentation
 */

How to Write Arithmetic Expressions

-4-
### Arithmetic Operators

- 6 + 3 // Add, evaluating to 9
- 6 - 3 // Subtract, evaluating to 3
- 6 * 3 // Multiply, evaluating to 18
- 6 / 3 // Divide, evaluating to 2
- 6 + y // Add, evaluating to 6 plus y's value
- x - 3 // Subtract, evaluating to x's value minus 3
- x * y // Multiply, evaluating to x's value times y's value
- x / y // Divide, evaluating to x's value divided by y's value

### Precedence

- Expressions have zero or more operators.
- Java follows standard rules for operator precedence.
- Precedence can be overridden through the use of parentheses.

### Mixed Expressions

- In expressions with different types, Java first unifies the types
  - e.g. int x float -> float x float -> float
- Expressions can be type cast
  - (double) i, where i is an int
  - (int) d, where d is a double

### Nested Assignment

- Assignment and other expressions can be nested as subexpressions
  - e.g. x = (y = 5)
How to Define Simple Methods

-5-

Methods

- The method is the basic unit of code in Java.
- Methods are associated with Classes.
- An example:
  ```java
  public class Demonstrate {
    public static int movieRating (int s, int a, int d) {
      return s + a + d;
    }
    // Definition of main goes here
  }
  ```

Example

Here is what each part of the method definition does:

- Indicates that the method can be called from any other method
- Indicates that the method is a class method
- Tells Java the data type of the returned value
- Tells Java the name of the method
- Tells Java the names and data types of the parameters

```java
public static int movieRating (int s, int a, int d) {
  return s + a + d;
}
```

Example...

```java
public class Demonstrate {
  public static void main (String argv[]) {
    int script = 6, acting = 9, direction = 8;
    System.out.print("The rating of the movie is ");
    System.out.println(movieRating(script, acting, direction));
  }
  public static int movieRating (int s, int a, int d) {
    return s + a + d;
  }
}
```

--- Result ---
The rating of the movie is 23
Naming Convention

- By convention, method names begin with lower case, and are punctuated with upper case
  - myVeryFirstMethod()

Return Value

- Return type must be specified
- All non-void methods require an explicit return statement
  - For void methods, return statement (with no arguments) is optional

Overloading

- A class may have multiple methods with the same name; they must have different signatures
  - Parameter list must differ
  - Overloading

Multiple classes can be defined in the same file
- Only the first class defined will be publicly accessible
Example

```java
public class Demonstrate {
    public static void main (String argv[]) {
        int intScript = 6, intActing = 9, intDirection = 8;
        double doubleScript = 6.0, doubleActing = 9.0, doubleDirection = 8.0;
        displayMovieRating(intScript, intActing, intDirection);
        displayMovieRating(doubleScript, doubleActing, doubleDirection);
    }

    // First, define displayMovieRating with integers:
    public static void displayMovieRating (int s, int a, int d) {
        System.out.print("The integer rating of the movie is ");
        System.out.println(s + a + d);
        return;
    }

    // Next, define displayMovieRating with floating-point numbers:
    public static void displayMovieRating (double s, double a, double d) {
        System.out.print("The floating-point rating of the movie is ");
        System.out.println(s + a + d);
        return;
    }
}
```

--- Result ---
The integer rating of the movie is 23
The floating-point rating of the movie is 23.0

Example...

- Another example of overloading:

```java
int i = 8, j = 7;
System.out.println("Print " + (i + j));
--- result ---
Print 15
```

Math

```java
public class Demonstrate {
    public static void main (String argv[]) {
        System.out.println("Natural logarithm of 10: " + Math.log(10));
        System.out.println("Absolute value of -10: " + Math.abs(-10));
        System.out.println("Maximum of 2 and 3: " + Math.max(2, 3));
        System.out.println("Power of 6: " + Math.pow(6, 5));
        System.out.println("Square root of 7: " + Math.sqrt(7));
        System.out.println("Sin of 8 radians: " + Math.sin(8));
        System.out.println("Random number (0.0 to 1.0): " + Math.random());
    }
}
```

--- Result ---
Natural logarithm of 10: 2.302585092994046
Absolute value of -10: 10
Maximum of 2 and 3: 3
Power of 6: 7776
Square root of 7: 2.6457513110645907
Sin of 8 radians: 0.9893582466233818
Random number (0.0 to 1.0): 0.8520107471627543

How to Understand Variable Scope and Extent
Scoping

• Method parameters are available everywhere within, but only within, the applicable method
• Parameters are implemented *call-by-value* in Java

Example

```java
public class Demonstrate {
    // First, define adder:
    public static int adder () { // BUG!
        return s + a + d;
    }
    // Next, define movieRating:
    public static int movieRating (int s, int a, int d) { // BUG!
        return adder();
    }
    // Then, define main:
    public static void main (String argv[]) {
        int script = 6, acting = 9, direction = 8, result; result = movieRating(script, acting, direction); System.out.print("The rating of the movie is "); System.out.println(s + a + d); // BUG!
    }
}
```

Blocks

• Blocks are defined with curly braces
• Variables within blocks are *local variables*
• Parameters and local variables have *local scope*; allocated memory is lost once block is exited

How to Benefit from Procedural Abstraction

-7-
Procedure Abstraction

• Procedure Abstraction
  – Move some aspect of computation into a unit, or method

Virtues of Procedure Abstraction

• Facilitates reuse
• Push details out of sight/mind
• Facilitate debugging
• Augments repetitive computation
• Facilitates localized improvement/adaptation

Class Variables

• Associated with a particular class, not individual instances of the class
• Persist throughout programs execution, irrespective of scope
• Use the static keyword

How to Declare Class Variables

-8-
Example

public class Movie {
    public static int wScript;

    // Rest of class definition
}

• Syntactically, class variables act just like local variables:
  – Combine multiple declarations
  – Declare and initialize

Example

public class Movie {
    // Define class variables:
    public static int wScript = 6, wActing = 13, wDirection = 11;

    // Define movieRating:
    public static int movieRating (int s, int a, int d) {
        return wScript * s + wActing * a + wDirection * d;
    }
}

Access as:
Movie.wScript

Shadowing

• Local variables and parameters shadow, or override, class variables of the same name
• In these cases only, it is necessary to use the field-selection operator '.'
Constants

- Class variables whose values will not change can be made constant:

```java
public static final int wScript = 6, wActing = 13, wDirection = 11;
```

How to Create Class Instances

Class Instance

- Creating a class instance allocates memory for a unique object, defined by the class
  - Instance variables
  - Instance methods

New

- Class instances are created with the new keyword, and a class constructor
  - `Movie m = new Movie();`

- Note: Because Java uses garbage collection, there is no corresponding operator for object deletion
Constructor

- Constructor is a special method that initializes a class instance
  - Can have many constructors
  - All have a default return type – the given class
  - All classes have a default, zero argument constructor

Instance Methods

- Instance methods ‘look and feel’ like class methods
- Difference:
  - An instance method must be invoked on a specific instance (e.g. `m.rating()`)  
  - Instance methods have access to the instances state – that is, the instances variables are in its scope.

How to Define Instance Methods

Example

```java
public class Symphony {
    public int music, playing, conducting; public int rating () { return music + playing + conducting; }
}
public class Demonstrate {
    public static void main (String argv[]) {
        Movie m = new Movie(); m.script = 8; m.acting = 9; m.direction = 6;
        Symphony s = new Symphony(); s.music = 7; s.playing = 8; s.conducting = 5;
        System.out.println("The rating of the movie is "+ m.rating());
        System.out.println("The rating of the symphony is "+ s.rating());
    }
}
```

--- Result ---

The rating of the movie is 23
The rating of the symphony is 20
‘This’

- ‘this’ refers to the current instance
- Consider:

```java
public class Movie {
    public int script, acting, direction;
    public int rating () {
        return script + acting + direction;
    }
}
```

vs.

```java
public class Movie {
    public int script, acting, direction;
    public int rating () {
        return this.script + this.acting + this.direction;
    }
}
```

Parameters

- Where class methods need arguments on which to operate (e.g. Math.log(n) )
- Instance methods need not always have arguments, as they can act on the instance state (e.g. stack.Pop() )

- * Class methods can operate on class variables, but they have no access to instance variables

Constructors

- Special method that defines the initialization procedure for an instance of a class
- Automatically invoked when an instance is created
  - Movie m = new Movie();
Constructors…

• Constructors:
  – Have the same name as the class to which they are bound
  – Return an instance of that class; no return type is specified

Example

```java
public class Movie {
    public int script, acting, direction;
    public Movie() {
        script = 5; acting = 5; direction = 5;
    }
    public int rating () {
        return script + acting + direction;
    }
}
```

Cascading Example

```java
public class Movie {
    public int script, acting, direction;
    public Movie() {
        this(5); // optionally, more code here
    }
    public Movie(int _script) {
        script = _script; acting = 5; direction = 5;
    }
    public int rating () {
        return script + acting + direction;
    }
}
```

How to Define Getter and Setter Methods

-12-
Getter/Setter

• Methods which provide access to the state of a class instance
  – Getter – return the value of a variable, or some other information
  – Setter (or mutator) – change the internal state of an instance

Rather than…

```java
public class Demonstrate {
    public int x = 0;
}

Demonstrate d = new Demonstrate();
if (d.x == 0) d.x = 5;
```

Prefer this…

```java
public class Demonstrate {
    private int x = 0;
    publicGetX() { return x; }
    public SetX(int _x) { x = _x; }
}

Demonstrate d = new Demonstrate();
if (d.GetX() == 0) d.SetX(4);
```

How to Benefit from Data Abstraction

-13-
Access Methods

• Constructors, getters and setters are called access methods
• Access methods facilitate data abstraction

Virtues of Data Abstraction

• Your programs become easier to reuse
• Your programs become easier to understand
• You can easily augment what a class provides
• You can easily improve the way that data are stored

Example

• Provide data beyond what is present
• Instance has variable dos (data object size), and variable n (number of objects)

```java
public int GetTotalSize() {
    return dos * n;
}
```

Example

• Constrain the values of a variable
• Instance has variable counter

```java
public void SetCounter(int _counter) {
    if (counter < max) counter++;
}
```
How to Define Classes that Inherit Instance Variables and Methods

-14-

Example

```java
public class Attraction {
    // Define instance variable:
    public int minutes;
    // Define zero-parameter constructor:
    public Attraction (){
        System.out.println("Calling zero-parameter Attraction constructor");
        minutes = 75;
    }
    public Attraction (int m){
        minutes = m;
    }
}
```

Example (Subclasses)

```java
public class Movie extends Attraction {
    // Define instance variables:
    public int script, acting, direction;
    // Define zero parameter constructor:
    public Movie (){
        System.out.println("Calling zero-parameter Movie constructor");
        script = 5; acting = 5; direction = 5;
    }
    // Define three parameter constructor:
    public Movie (int s, int a, int d){
        script = s; acting = a; direction = d;
    }
    // Define rating:
    public int rating (){ return script + acting + direction; }
}
```
Example (Subclasses)...

```java
public class Symphony extends Attraction {
    // Define instance variables:
    public int music, playing, conducting;
    // Define zero parameter constructor:
    public Symphony () {
        System.out.println("Calling zero parameter Symphony constructor");
        music = 5; playing = 5; conducting = 5;
    }
    // Define three parameter constructor:
    public Symphony (int m, int p, int c) {
        music = m; playing = p; conducting = c;
    }
    // Define rating:
    public int rating () {
        return music + playing + conducting;
    }
}
```

Extends

- If class A extends class B, it is a direct subclass
- All classes, directly or indirectly, extend Object
  - “class A” (no extension) can also be written as “class A extends Object”

Access and Overriding

- Subclass has access to non-private methods of superclass
- All can be overridden
- All constructors first call the zero argument constructor of the superclass

Example

```java
public class Demonstrate {
    public static void main (String argv[]) {
        Movie m = new Movie();
        Symphony s = new Symphony();
    }
}
```

--- Result ---
Calling zero-parameter Attraction constructor
Calling zero-parameter Movie constructor
Calling zero-parameter Symphony constructor
Overriding vs. Overloading

• Note the distinction between **overloading** and **shadowing** or **overriding**:
  – **Overloading** occurs when Java can distinguish two procedures with the same name by examining the number or types of their parameters
  – **Shadowing** or **overriding** occurs when two procedures with the same name, the same number of parameters, and the same parameter types are defined in different classes, one of which is a superclass of the other

How to Enforce Abstraction
Using Protected and Private Variables and Methods

Data Abstraction

• Support data abstraction with the use of getter and setter methods
  – Constrain values
  – Enhanced functionality
  – Flexibility
  – Information hiding
  – …

Enforce Abstraction

• Use the private keyword to protect access to variables and methods
• Allow access through access methods
Example

```java
public class Attraction {
    // First, define instance variable:
    private int minutes;
    // Define zero-parameter constructor:
    public Attraction () {
        minutes = 75;
    }
    // Define one-parameter constructor:
    public Attraction (int m) {
        minutes = m;
    }
    // Define getter:
    public int getMinutes () {
        return minutes;
    }
    // Define setter:
    public void setMinutes (int m) {
        minutes = m;
    }
}
```

Example...

• With the Attraction class so redefined, attempts to access an attraction's instance-variable values from outside the Attraction class fail to compile:
  
  ```java
  x.minutes <-- Access fails to compile;
  the minutes instance variable is private
  x.minutes = 6 <-- Assignment fails to compile;
  the minutes instance variable is private
  ```

• Thus, attempts to access an attraction's instance-variable values from outside the Attraction class, via public instance methods, are successful:
  
  ```java
  x.getMinutes() <-- Access compiles;
  getMinutes is a public method
  x.setMinutes(6) <-- Assignment compiles;
  setMinutes is a public method
  ```

Private Methods

• Methods can also be marked private
  
  (Note: private methods and variables will by default not appear on auto-generated documentation)

• Some prefer to put all private methods after public methods

Protected Access

• In addition to private and public, variables and methods can be marked `protected`
  
  –Can be accessed by members of that class, and any subclass
  
  –(also, by other classes in the same compilation unit or package)
How to Write Constructors that Call Other Constructors

Motivation

- Want to avoid duplication of code as much as possible, for the usual reasons
- Create a hierarchy of constructors
- Call to alternate constructor must be first line in method

Example

```java
public class Movie extends Attraction {
    public int script, acting, direction;
    public Movie () {
        script = 5; acting = 5; direction = 5;
    }
    public Movie (int s, int a, int d) {
        script = s; acting = a; direction = d; <--------*
    }
    public Movie (int s, int a, int d, int m) {
        script = s; acting = a; direction = d; <--------*
        minutes = m;
    }
    public int rating () {
        return script + acting + direction;
    }
}
```
Code Merging

- Use this to eliminate repetition of code, abstract functionality:
  - To certain constructors, or
  - To constructors in the superclass

Example

```java
public class Movie extends Attraction {
  public Movie (int m) { minutes = m; }     <-------* ...
}

public class Symphony extends Attraction {
  public Symphony (int m) { minutes = m; }  <----* ...
}
```

Example...

- (1) Add a one argument constructor to the superclass

```java
class Attraction {
  private int minutes;
  public int getMinutes() {
    return minutes;
  }
  public void setMinutes(int m) {
    minutes = m;
  }
  public Attraction () {
    minutes = 75;
  }
  public Attraction (int m) {
    minutes = m;
  }
}
```

Example...

- (2) Use super keyword to invoke constructor of superclass with arguments

```java
public class Movie extends Attraction {
  ... public Movie (int m) { minutes = m; }  <--------* ...
  }

public class Symphony extends Attraction {
  ... public Symphony (int m) { minutes = m; } <-----* ...
  }
```
How to Write Methods that Call Other Methods

How to Design Classes and Class Hierarchies

How to Enforce Requirements Using Abstract Classes and Abstract Methods

How to Enforce Requirements and to Document Programs Using Interfaces
How to Perform Tests Using Predicates
-21-

How to Write Conditional Statements
-22-

How to Combine Boolean Expressions
-23-

How to Write Iteration Statements
-24-
How to Write Recursive Methods

How to Write Multiway Conditional Statements

How to Work with File Input Streams

How to Create and Access Arrays
How to Move Arrays Into and Out of Methods
-29-

How to Store Data in Expandable Vectors
-30-

How to Work with Characters and Strings
-31-

How to Catch Exceptions
-32-
How to Work with Output File Streams

-33-

Example

```java
import java.io.*;
import java.util.*;
public class Demonstrate {
    public static void main(String argv[]) throws IOException {
        FileOutputStream stream = new FileOutputStream("output.data");
        PrintWriter writer = new PrintWriter(stream);
        Vector mainVector;
        mainVector = Auxiliaries.readMovieFile("input.data");
        for (Iterator i = mainVector.iterator(); i.hasNext();) {
            writer.println(((Movie) i.next()).rating());
        }
        writer.flush();
        stream.close();
        System.out.println("File written");
    }
}
```

Output File Streams

- Similar to use of input file streams
  - import java.io.*;
  - create an output stream
  - attach a printwriter
  - write
  - ...
  - flush the stream
  - close

How to Write and Read Values Using the Serializable Interface

-34-
Serialization

- Write objects directly to/from files
- Use:
  - ObjectOutputStream
  - ObjectInputStream

- Avoid details of writing and reconstructing data structures

Serializable

- Classes which support serialization must implement the ‘Serializable’ interface
- Use writeObject/readObject
  - must deal with:
    - IOException
    - ClassNotFoundException

Contained Classes

- Objects stored in an instance to be serialized (e.g. elements in a Vector) must also be serializable
- Multiple instances may be saved to a single file

How to Modularize Programs Using Compilation Units and Packages

-35-
Modules

- Good practice to group functionally related classes
  - compilation units
  - packages

Compilation Units

- Compilation Unit = File
- Only first class (with same name as file) can be public
- Example use:
  - Your class uses ‘helper’ classes to store and manipulate local-only data – keep those classes in the same compilation unit

Packages

- Package = Directory
- Packages are arranged hierarchically
  - Does NOT necessarily correspond to class hierarchy

Classpath

- Classpath tells Java where to find source root directories
  - specify as an environment variable
  - on the command line
- Packages are subdirectories from some root on the classpath
Example

- `classpath = c:\java;c:\myfiles\java`

- package-less Java classes can be located in either of the two roots
- `c:\java\test` contains files in the ‘test’ package
- `c:\myfiles\java\blue\v1` – files in the ‘blue.v1’ package

Package Declaration

- Identify a class with a package:
  - `package blue.v1;`

- A file beginning with the above statement **must** be in a `blue/v1` subdirectory of some root on the classpath

Import

- As you have seen, use the ‘import’ statement to tell the compiler you will be using classes from other packages
  - `e.g. import java.io.*;`

How to Combine Private Variables and Methods with Packages
Access

- Four different states:
  - private
  - public
  - protected
  - unspecified

Who Can Access

- Access defined by who can access. Includes:
  - Other members of the compilation unit
  - Other compilation unit, same package
  - Subclass, different CU and package
  - All else

- Members of the same class always have complete access to a variable or method

<table>
<thead>
<tr>
<th>Access</th>
<th>Public</th>
<th>Protected</th>
<th>Unspecified</th>
<th>Private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same Compilation Unit</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Other CU, Same Package</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Subclass in Different Package</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>All Others</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

How to Create Windows and to Activate Listeners

-37-
GUI

- GUI – Graphical User Interface
- Components – Classes whose instances have a graphical representation
- Containers – Components which can (visually) contain other components
- Window => Container

Hierarchy

- Object => Component => Container
- Container => Window => Frame => JFrame
- Container => Panel => Applet => JApplet
- Container => JComponent => JPanel

Packages

- java.awt
  - Component, Container
- java.swing
  - JFrame, JApplet, JComponent, JPanel

  Component/Container machinery is platform independent

Example – Create a Window

```java
import javax.swing.*;
public class Demonstrate {
    public static void main (String argv []) {
        JFrame frame = new JFrame("Movie Application");
        frame.setSize(350, 150);
        frame.show();
    }
}
```
Events and Listeners

- Event
  - something which happens (key press, click…)
  - an extension of the EventObject class

- Listener
  - class which extends a listener class
  - class which implements a listener interface

- Listeners respond to events

Create a Responsive Window

1. Define a listener class
2. Define methods to handle specific events
3. Connect the listener to your application frame
4. Listener will now respond to events on the window as specified

Example

```java
import java.awt.event.*;
public class ApplicationClosingWindowListener implements WindowListener {
    public void windowClosing(WindowEvent e) {System.exit(0);}
    public void windowActivated(WindowEvent e) {} 
    public void windowClosed(WindowEvent e) {} 
    public void windowDeactivated(WindowEvent e) {} 
    public void windowDeiconified(WindowEvent e) {} 
    public void windowIconified(WindowEvent e) {} 
    public void windowOpened(WindowEvent e) {} 
}
```
Example…

```java
import javax.swing.*;
public class Demonstrate {
    public static void main (String argv []) {
        JFrame frame = new JFrame("Movie Application");
        frame.setSize(350, 150);
        frame.addWindowListener(new ApplicationClosingWindowListener());
        frame.show();
    }
}
```

Adapters

- Adapter classes provide trivial implementations of event-handling methods
- Extend an adapter class, and implement only those methods you require

MovieApplication Example

```java
import javax.swing.*;
public class MovieApplication extends JFrame {
    public static void main (String argv []) {
        new MovieApplication("Movie Application");
    }
    public MovieApplication(String title) {
        super(title); setSize(350, 150);
        addWindowListener(new ApplicationClosingWindowListener());
        show();
    }
}
```

How to Define Inner Classes and to Structure Applications
Inner Classes

• Again, group for simplicity and access control
• Inner classes are available only to parent (enclosing) class
• Have access to private members of enclosing class

Example

```java
import javax.swing.*;
import java.awt.event.*;
public class application name extends JFrame {
    public static void main(String argv[]) {
        new application name(argv[0]);
    }
    public application name(String title) {
        super(title); setSize(width, height);
        addWindowListener(new LocalWindowListener());
        show();
    }
}
private class LocalWindowListener extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
}
```

Drawing 101

• Draw on instances of the JComponent class
• JComponent has a definition for ‘paint’
  – Called whenever you call ‘repaint’
  – Called when you iconify, deiconify or expose a frame’s window
Interfaces

• Create an interface for the visual object you wish to create. Example:

```java
public interface MeterInterface {
    // Setters:
    public abstract void setValue (int valueToBeShownByDial);
    public abstract void setTitle (String meterLabel);
    // Getters:
    public int getValueAtCoordinates (int x, int y);
}
```

Example: Meter

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

public class Meter extends JComponent implements MeterInterface {
    String title = "Title to be Supplied";
    int minValue, maxValue, value;

    public Meter (int x, int y) {
        minValue = x;
        maxValue = y;
        value = (y + x) / 2;
    }

    public void setValue(int v) {
        value = v; repaint();
    }

    public void setTitle(String t) {
        title = t;
        repaint();
    }

    public void paint(Graphics g) {
        g.drawLine(0, 50, 100, 50); ...
    }
}
```

Graphics Context

• Graphics Context acts as a controller that determines how graphical commands affect display

• Example:
  – g.drawLine(0, 50, 100, 50);

Example…

• In Meter example, drawLine appears in the paint method:

```java
public void paint (Graphics g) {
    g.drawLine(0, 50, 100, 50);
    ...
}
```
Containers and Components

• Remember, containers contain components, and containers are components
• A Display might contain:
  – JRootPane
    • JLayeredPane
    • JMenuBar
    • JPanel (The Content Pane)
      – various components

Adding Elements

• Get the content pane
• Add elements
  
  on frame, 
getContentPane().add("Center", meter");

Example

```java
import javax.swing.*;
import java.awt.event.*;
public class MovieApplication extends JFrame {
  public static void main (String argv []) {
    new MovieApplication("Movie Application");
  }
  // Declare instance variables:
  private Meter meter;
  // Define constructor
  public MovieApplication(String title) {
    super(title);
    meter = new Meter(0, 30); getContentPane().add("Center", meter);
    addWindowListener(new LocalWindowListener());
    setSize(350, 150); show();
  }
  // Define window adapter
  private class LocalWindowListener extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
}
```

Layout Managers

• Layout of objects controlled by Layout Manager
• There is always a ‘default’ Layout Manager (in this case, BorderLayout)
Example

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

public class MovieApplication extends JFrame {
    public static void main(String argv[]) {
        new MovieApplication("Movie Application");
    }
    private Meter meter;
    public MovieApplication(String title) {
        super(title);
        meter = new Meter(0, 30);
        getContentPane().setLayout(new BorderLayout);
        getContentPane().add("Center", meter);
        addWindowListener(new LocalWindowListener());
        setSize(350, 150);
        show();
    }
}
```

How to Draw Text in Windows

-39-

How to Write Text in Windows

-40-

How to Use the Model-View Approach to GUI Design

-41-
Applets

- Applets provide a mechanism for accessing a Java program from a web browser
- Two components:
  - Java program
  - Web page framework (next section)

JApplet

- Applets extend the JApplet class
- Differences from a standalone program:
  - No main method – applet instantiated by browser via zero-argument constructor
  - Applet’s size determined by browser/web page
  - Applet has no close button
Example

```java
import javax.swing.*;
import java.awt.event.*;
import java.util.*;
public class MovieApplication extends JApplet {
    // Declare instance variables:
    private Meter meter;
    private Movie movie;
    // Define constructor
    public MovieApplication () {
        // Create model
        getMovie();
        // Create and connect view to application
        getContentPane().add("Center", getMeter());
    }
    // Define getters and setters
    public Meter getMeter () {
        if (meter == null) {
            setMeter(new Meter(0, 30));
        }
        return meter;
    }
    public Movie getMovie () {
        if (movie == null) {
            setMovie(new Movie (10, 10, 10, "On to Java"));
        }
        return movie;
    }
    public void setMeter (Meter m) {
        meter = m;
        meter.addMouseListener(new MeterListener(this));
    }
    public void setMovie (Movie m) {
        if (movie == m) {return;}
        if (movie instanceof Movie) {movie.deleteObservers();}
        if (m instanceof Movie) { movie = m; movie.addObserver(new MovieObserver(this)); movie.changed(); }
    }
}
```

Example...

- First, zero argument constructor is invoked
- Also, init method
  - Inherited init method does nothing
- Other applet methods:
  - start – when page is first (re)visited
  - stop – when page is replaced, or before destroy
  - destroy – when page is abandoned (e.g. browser is shutting down)

Running Applets

- Since applets are run in web browsers, they must be anchored in web pages
- Web pages are marked up in HTML (hypertext markup language)
HTML

```html
<html>
<head>
<title>Welcome to a simple HTML file</title>
</head>
<body>
<hr>
This text can be viewed by a web browser.
<p>
It consists of only text, arranged in two paragraphs, between horizontal rules.
</p>
</body>
</html>
```

Applet Directive

```html
<html>
<head>
<title>Welcome to the Meter Applet Demonstration</title>
</head>
<body>
<hr>
<applet code="MovieApplication.class" width=350 height=150>
</applet>
<hr>
</body>
</html>
```

Running

- Place applet class file and html file in your web server’s filespace
- Access using a browser and URL, as any other web page
- Note: Java also provides a tool for running applets independently - appletviewer

Appletviewer

- When using appletviewer, you must still use an HTML file with applet directives
- You will only see the embedded applet, not the web page
- example:
  - `appletviewer file:///d:/phw/onto/java/test.html`
Advanced Control

- It is possible to exercise much more control over applets
  - Pass parameters to applets from HTML
  - Control Applets with JavaScript
  - Control JavaScript from your Applet
  - ...

- Read:
- Core WEB Programming, Marty Hall & Larry Brown
- http://www.corewebprogramming.com

How to Use Resource Locators

How to Use Choice Lists to Select Instances

How to Bring Images Into Applets
How to Use Threads to Implement Dynamic Applets

-48-

Terminology

• Process – running computer program
• Current values for a given process constitute it’s context
• Multiprocessing system maintains context for individual processes, and allocates time slices
• Each process has it’s own allocated section of memory, or address space

Threads

• Like processes, but threads share the same address space (and are therefore lighter)
• Threads in Java are supported by the Thread class

Thread Class

• To create a Java thread:
  – Define a subclass of the Thread class
  – Include a definition for the run method
  – Create an instance of your subclass
  – Call the start method on your subclass instance (which invokes run)
    • run is never called directly – it is like main
Threads…

• After calling start:
  – Your current program thread will continue on, and
  – Your new thread will begin execution

Example

```java
import java.lang.*;
public class DemoThread extends Thread {
  public static void main (String argv []) {
    DemoThread thread = new DemoThread();
    thread.start();
  }
  public void run () {
    while (true) {
      System.out.println("Looping...");
    }
  }
}
```

Example…

• Note, it is not necessary to have your main class (started by main) extend thread – it has its own thread of execution
• You may want to create a class that can be started directly (via main), or as a subthread

Creating a Thread

• One common approach is to:
  – Define a run method
  – Create a constructor
    • Basic initialization code
    • Call to start()

• Thread is started upon creation
Sleeping

- You can cause a thread to pause execution for some time
- `sleep(n)`
  - `n` is in milliseconds
- Must catch an `InterruptedException`

Example

```java
import java.lang.*;
public class DemoThread extends Thread {
    public static void main(String argv[]) {
        DemoThread thread = new DemoThread();
        thread.start();
    }
    public void run() {
        while (true) {
            System.out.println("Looping...");
            try {
                sleep(200);
            } catch (InterruptedException e) { }
        }
    }
}
```

Stopping a Thread

- Can no longer stop a thread directly (this is an unsafe practice)
- Add a flag to the threads run loop, which can be set internally or externally
  - if flag has a certain value, exit the loop

Example

```java
import java.lang.*;
public class DemoThread extends Thread {
    boolean execute = true;
    public void Stop() { execute = false; }
    public void run() {
        while (execute) {
            System.out.println("Looping...");
            try {
                sleep(200);
            } catch (InterruptedException e) { }
        }
    }
}
```
Synchronization

- Multithreaded applications require low-level synchronization support to control thread interaction
- Java associates ‘locks’ with objects
- Only one thread can possess a lock at a time
- Use the ‘synchronized’ keyword to associate methods with locks

Synchronization…

class stack {
    ...
    public synchronized void push (Object o) {
        stack.insertElementAt(o,0);
    }
    public synchronized Object pop () {
        Object o = stack.elementAt(0);
        stack.removeElementAt(0);
        return o;
    }
}

How to Create Forms and to Fire Your Own Events

-49-

How to Display Menus and Dialog Windows

-50-
How to Develop Your Own Layout Manager

-51-

How to Implement Dynamic Tables

-52-

How to Activate Remote Computations

-53-

How to Collect Information Using Servlets

-54-
How to Construct JAR Files for Program Distribution

-55-