Java II

Walter Wes
UCSD Exten
Information Technology
Software Enginee

Module 2

- Arrays
- Fields and Dot Notation
- The For-Each Loop
- The While Loop
- Hierarchical Decomposition
- Scope
- The For Loop
- Nested Loops
- Method Return Values
- Method Overloading
- Conditional Execution

Arrays

- An array is a data structure composed of multiple elements.
- Array elements are contiguous in memory.
- Array elements are of homogeneous data type (they are each of the same type or class).
- Since array elements are of homogeneous data type, by definition they have the same size.
- Array elements are retrievable by index.

Arrays

- int[] number = {8, 42, 31, 6, -7, 2, 58, 256, 0, -400};
- The above declaration will have the following memory map:

number[0]	8
number[1]	42
number[2]	31
number[3]	6
number[4]	-7
number[5]	2
number[6]	58
number[7]	256
number[8]	0
number[9]	-400

Arrays

- int[] number = {8, 42, 31, 6, -7, 2, 58, 256, 0, -400}

```
int x;
int y;
x = number[0];  // assigns   8 to x
y = number[1];  // assigns 42 to y

int[] pair = new int[2];  // creates an array with size of two ints.
pair[0] = x;  // assigns   8 to element 0 of pair.
pair[1] = y;  // assigns 42 to element 1 of pair.
```

Arrays

- The previous example can be thought of as being one data column with multiple rows—this is known as a *one-dimensional array*.
- You can also have an array with multiple columns as well as multiple rows—yielding a *two-dimensional array*.
- int[][] sudoku = new int[9][9];

  ...declares a *two-dimensional int array* named *sodoku*.

  ...a *9 by 9* array object is created and assigned to the *array reference sudoku*.

Field and Dot Notation

- Arrays have a public field named length, which give you the length of the dimension.
- int[] number = {8, 42, 31, 6, -7, 2, 58, 256, 0, -400};
- System.out.println(number.length);

  ...displays *10*, which is the *length* of the *one-dimensional array*.
- double[][] grade = new double[30][5];
- System.out.println(grade.length);

  ...displays *30*, which is the *length* of the first dimension of *grade*.
- System.out.println(grade[0].length);

  ...displays *5*, which is the *length* of the second dimension of *grade*.

The For-Each Loop

- Since the media class *Picture* represents a picture as an array of *Pixel*s, the entire array can be *traversed* (or *iterated*) by means of a *for-each loop*.
- A public method named *adjustRed(int deltaFactor)* can be added to the *Picture* class:

```
public void adjustRed(int deltaFactor)
{
  Pixel[] pixelArray = this.getPixels();
  int value = 0;
  for (Pixel pixel : pixelArray)
  {
    value = pixel.getRed();
    value = (int) (value * deltaFactor);  // adjust red by deltaFactor
    pixel.setRed(value);
  }
}
```

The While Loop

- Using a *while loop* instead, we would have:

```
public void adjustRed(int deltaFactor)
{
  Pixel[] pixelArray = this.getPixels();
  int value = 0;
  int pixIndex = 0;
  boolean done = false;
  while(!done)
  {
    value = pixelArray[pixIndex].getRed();
    value = (int) (value * deltaFactor);  // adjust red by deltaFactor
    pixelArray[pixIndex].setRed(value);
    pixIndex++;
    if (pixIndex == pixelArray.length)
    {
      done = true;
    }
  }
}
```

The While Loop

- If you are iterating through an entire array or collection, the for-each loop is preferable to the while loop.
- However, if you may want to stop midway through an interation, then a while loop may be a better choice.
- A good example of when a while loop is preferable is when you are reading an input stream (the keyboard, perhaps) and you want to continue processing the stream until the user provides a special input (the letter 'q', perhaps) indicating the desire to quit.

Hierarchical Decomposition

- Generally, a method should be as *cohesive* as possible.
- This means that *a method should do one thing and only one thing really well*.
- *Cohesive methods foster clarity and reusability*.
- Cohesive methods also facillitate the establishment of a *hierarchy of goals*.
- Major goals are broken down into sub-goals, and sub-goals are broken down into sub-sub-goals, and... etc.
- This process of continual subordination is known as *top-down refinement* or *problem decomposition*.
- In many cases, an inverted, *bottom-up* approach is preferable. Although the directionality of construction may be different, the advantages of composition/decomposition are similar.

Scope

- As the name suggest, *scope* is about *what you can see*; *scope* is about *visibility*.
- When a *local variable is declared* within a method, the *variable comes into scope*, i.e. *it becomes visible at that point*.
- *When the method ends, local variables are no longer in scope*,i.e. *they are no longer visible*.
- It is a best practice to *grant only as much visibility as is necessary to a variable*. This is an example of an important software engineering principle known as the *principle of least privilege*.
- Within DrJava, any variables declared within the interactions pane are only visible within that pane—the pane constrains their scope.

The For Loop

- The *for loop* construct is similar to the *for-each loop*, the difference being the support of *index variable declaration, initialization and incrementation or decrementation*. Again, the for-each loop is preferred for iterating over an entire array or collection.
- Implementing our *adjustRed(int deltaFactor)* method using a *for loop* we would have:

```
public void adjustRed(int deltaFactor)
{
  Pixel[] pixelArray = this.getPixels();
  int value = 0;
  for (int i = 0; i < pixelArray.length; i++)
  { // Although this could be one line of code, we break it into three for clarity.
    value = pixelArray[i].getRed();
    value = (int) (value * deltaFactor);  // adjust red by deltaFactor
    pixelArray[i].setRed(value);
  }
}
```

Nested Loops

- It is often the case that you need loops *nested* within loops *nested* within loops *nested* within... etc.
- Here's a simple example of a *for loop nested within another for loop*:

```
for (int x = 0; x < getWidth(); x++)
{
  for (int y = 0; y < getHeight(); y++)
  {
    pixel = getPixel(x, y);
    // do stuff to color...
    pixel.setColor(aColor);
  }
}
```

- Note that each loop has its own *scope*, which means that *while the inner loop can see x, the outer loop* ***cannot*** *see y!*

Method Return Values

- If a method does not return a value, then its return type should be declared as *void*.

```
public void foo();
{
  // do stuff
}
```

- Constructor methods do not follow this rule, however. *Constructors*, by definition, *have no return type* and *must be named the same as the class* to which they belong.

```
public class Bar
{
  // private fields (instance variables) here
  public Bar()  // Constructor name is the same as the class name
  {
    // do stuff
  }
}
```

Method Return Values

- If a method does return a value, then its return type should be declared to be of the appropriate type.

```
public int foofaraw();
{
  int num;
  // do stuff
  return num;
}
```

Method Overloading

- Methods are said to be *overloaded* if they have the *same name*, but *differ* in the *number of arguments*, the *type of arguments*, or a *combination of number and type* of arguments.
- The term *overload* has the connotation of a *loading up* on the *method name*.

  int foo() {...}

  int foo(int num) {...}

  int foo(int num, int offset) {...}

  int foo(int num, float average) {...}

- Note that overloaded methods are *never differentiated by return type*, so the following will not compile:

  int foofaraw(int num) {...}

  float foofaraw(int num) {...} // Differs from the above only by return type!

Conditional Execution

- Using the relational operators presented in Module 1, we can create many conditional expressions that control when and what transformations we apply to our objects and data.
- Conditional expressions are used within logical control structures, which include the following patterns:

```
if (boolean expression)  // one logical option
{
  // do stuff
}


if (boolean expression)  // two logical options
{
  // do stuff
}
else
{
  // do different stuff
}
```

Conditional Execution

- Another useful logical control structure is the nested if...else if construction:

```
if (boolean expression)  // three or more logical options
{
  // do stuff
}
else if (boolean expression)  // use as many “else if”s as you need
{
  // do different stuff
}
else  // this last else is optional; use as needed
{
  // do completely different stuff
}
```

Conditional Execution

- You should step through the textbook material covering *conditionally modifiying pixels*, and work through all of the examples.

Deconstructing the Picture Class

- If you inspect the Picture.java file, you will see the source code for the media class Picture.
- *public class Picture extends SimplePicture*

  ...establishes an *inheritance relationship* between Picture and SimplePicture.

  - This means that Picture *is a* SimplePicture.
  - All fields (instance variables) and methods defined by SimplePicture become the basis of Picture.
  - Additional fields and methods can then be added by Picture.
  - In this case, we see that Picture adds only methods.

Deconstructing the Picture Class

- There are 4 overloaded constructors.

  public Picture () {...}

  public Picture(String fileName) {...}

  public Picture(int width, int height) {...}

  public Picture(Picture copyPicture) {...}
- Each constructor invokes a constructor of its *parent class* (SimplePicture) via the statement *super(...)*, passing argument along as necessary.
- The toString() method provides a String representation of Picture object. This allows you to output text to the console that may provide you feedback and assist you in debugging.

Deconstructing the SimplePicture Class

- The *SimplePicture* class is structurally similar to the *Picture* class.
- *private BufferedImage bufferedImage;*

  ...declares a *field* named *bufferedImage* to be of type *BufferedImage*. *Private* data can only be seen with the object instantiated from the class. All fields should be declared *private* in this way, thereby enforcing *encapsulation*.
- The *BufferedImage* class allows you to create color and greyscale images.
- *bufferedImage = new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);*

  ...creates a BufferedImage object with x and y dimensions of width and height, using an RGB color scheme (i.e. Red, Green, and Blue color components).

Deconstructing the SimplePicture Class

- *private PictureFrame pictureFrame;*

  ...declares a *field* named *pictureFrame* to be of type *PictureFrame*.

- The *PictureFrame* is another media class; it provides a frame for a *Picture*.
- *pictureFrame = new PictureFrame(this);*

  ...create a *PictureFrame* object, passing a reference to the *Picture* object (this object that we are) to the *constructor*. The *PictureFrame* will thereby be able to refer back to the *Picture* object that it frames.

- *bufferedImage = ImageIO.read(new File(this.fileName));*

  ...creates a *BufferedImage* object and loads the image data from the image file into the *BufferedImage* object.

Deconstructing the SimplePicture Class

- Let's see it work!
  - From within the *Interactions pane*, type the following as a single line (*your_install_location* should be the place where you installed the textbook CD):

    SimplePicture simplePicture = new SimplePicture("your_install_location/PH_GUZDIAL/intro-prog-java/mediasources/gorge.jpg");

  - It will seem like nothing happened, but you created a *SimplePicture* object. You just haven't made it visible yet. To make it visible, type the following:

    simplePicture.setVisible(true);

Deconstructing the SimplePicture Class

- The *main method*
  - As a convenience, you might want to create a *main method* within the SimplePicture class that contains the code you just type into the Interactions pane.
  - The *main method* is the entrance to your running application. When a Java class has a main method, you can then run that class.
  - Often you only have one class with a *main method*—i.e., your *application class*.
  - Placing a *main method* in other classes allows you to run and test them. It's a common Java trick to perform a simple form of *unit testing*—testing the small units that you use to build your application.

Deconstructing the SimplePicture Class

- Let's try this technique with the SimplePicture class. Within the SimplePicture.java file (and inside the class definition), add the following:

  public static void main(String args[])

  {

  SimplePicture simplePicture = new SimplePicture("your_install_location/PH_GUZDIAL/intro-prog-java/mediasources/gorge.jpg");

  simplePicture.setVisible(true);

  }

- Now, you should be able to *Save*, *Compile*, and *Run*.
- Using a *main method* is a convenient alternative to repeatedly typing statements into the *Interactions pane*.

Deconstructing the SimplePicture Class

- You should devote some time to review the SimplePicture class.
- Within DrJava, you can right-click on a statement and select *Toggle Breakpoint*. The statement will be highlighted in red to indicate that you have set a *breakpoint* at that location.
- From the *DrJava* menu bar, selecting *Debugger->Debug Mode* will *toggle the debug mode*. If debug mode is set, then as the java code is executing, *hitting a breakpoint location will cause the execution to halt*. You will then have the option of stepping through your code, one line at a time, or continuing (resuming) execution thereafter.
- Note that the *SimplePicture* class implements the *DigitalPicture interface*.

    public class SimplePicture implements DigitalPicture

- We will look at *interfaces* in the next module.
- *Interfaces* are covered in greater detail in the *UCSD Extension Java III course*.

Deconstructing the SimplePicture Class

- You should also be sure to familiarize yourself with the Java Application Programming Interface (API) documentation.
- By exploring the Java API you will learn a great deal about the inner workings of key Java classes.
- Be sure that API you access is the one that corresponds to the Java Development Kit (JDK) version you are using.
- Both the JDK and the associated API Documentation can be located at www.sun.com.
- The API can be accessed online (through web-site access), or you can download it and install it onto your local file system, in which case you can access it offline.