Java II

Walter Wes
UCSD Exten
Information Technology
Software Enginee

Module 3

- Viewing the Java API
- Packages
- Inheritance
- AWT Graphics
- Interfaces
- Static Methods
- Private Methods
- Another Way to Draw
- Creating New Classes

Viewing the Java API

- The Java Application Program Interface (API) is your window to the language.
- The API documentation for your version of Java is freely available at http://java.sun.com.
- You can view the documentation online, or you can download it to your local filesystem and view it offline.
- The documentation is formatted to support framed viewing within a web browser.
- The top-left window frame is the *package* frame.

Packages

- A *package* is a grouping of related classes.
- The *package* construct organizes classes and manages the complexity that arises from having vast numbers of classes.
- In much the same way that a book library organizes books according to field and genre, a Java class library is organized into *packages* of conceptually related classes.
- *Packages* are often organized hierarchically—a package may be broken down into a set of sub-packages, and those sub-packages may be broken down into a set of sub-sub-packages, etc.
- Programmers often develop their own *packages*, and the compiler writes them to the hard disk as a *directory/file hierarchy*. The *packages are represented as directories*, and the *classes are .class files within their respective directories*.

Packages

- Programmers create packages by placing a package statement at the beginning of a .java source code file. For example:

  package com.acme.area51.projectX.teleportation;

- The above package hierarchy specifies a set of *teleportation* classes, developed as part of *projectX*, within *area51*, by the *company acme* (com.acme is indicates an inverted internet domain name for the acme company, but the domain does not need to actually exist).
- Note that the series of names that make up the full package name are ordered from general to specific, as they move from left to right.
- A package statement at the beginning of a *.java* source code file tells the Java compiler to place the generated *.class* file within the specified package.

Packages

- To use a class defined as part of a *package*, an *import* statement is placed at the beginning of the *.java* source code file where the class needs to be used. For example:

  import com.acme.area51.projectX.teleportation.AntiMatter;

- The above *import* statement simplifies the use of the AntiMatter class. It allows for the creation of an *AntiMatter* object as follows:

  AntiMatter amObj = new AntiMatter();

- Alternatively, without the above *import* statement, you would be forced to create the object as follows:

  com.acme.area51.projectX.teleportation.AntiMatter amObj
  = new com.acme.area51.projectX.teleportation.AntiMatter();

Packages

- As you can see, the import statement saves you a lot of typing, since you need only type the class name, instead of the fully-qualified package name + class name.
- Another nifty trick is to use the * wildcard to specify all classes within the package. For example:

  *import com.acme.area51.projectX.teleportation.**

- The above statement *imports* (provides simplified typing for) the *AntiMatter* class, the *Matter* class, and all other classes that belong to the *teleportation package*.

Packages

- When you install Java on your system, you are provided with *packages* of numerous pre-built classes that collectively constitute your Java class environment.
- These common classes are provided as zipped up *package* hierarchies known as *.jar* (i.e, java archive) files.

Packages

- Now that you understand *packages*, you will better understand how to navigate the Java API using your browser.
- After opening the Java API Specification documentation with your browser, locate and select the following *package* in the upper-left frame of your browser:

  java.awt

- *AWT* is an acronym for *Abstract Window Toolkit*.
- The *AWT package* provides a set of classes that support fun drawing capabilities.
- The lower-left frame of your browser should show the classes that belong to the selected *package* (in this case, *java.awt*).
- Click on the class *Graphics*.
- You should now be able to scroll the main window and view the *Method Summary* section.

Inheritance

- *Inheritance* is the principle mechanism of reuse in object-oriented languages.
- When a class inherits from another class, it *gets* all the fields (instance variables) and methods of the class it inherits from.
- Therefore, when you want to develop a new class that it similar to an existing class, except for a few added fields and/or methods, then inheritance is an easy way of defining that new class.
- In this way, a *Vehicle* class will serve as a good basis for an *Airplane* class. The *Airplane* class inherits from the *Vehicle* class. An *Airplane IS-A Vehicle*. Perhaps the *Vehicle* class has an *engine* field and a *start()* method. The *Airplane* class then adds *wings* as a field, and *fly()* as a method.
- You could also have a *Car* class that inherits from the *Vehicle* class. A *Car IS-A Vehicle*, and it inherits the engine field and the *start()* method. The *Car* class adds a *drive()* method.
- A *SuperSonicPlane* class can now inherit from *Airplane*, and may add an *engageTurboBoosters()* method.

Inheritance

- Inheritance Terminology

super	*sub*
parent	*child*
base	*derived*

- The above terms are used in pairs to denote an inheritance relationship between two classes.
- A *subclass* inherits from its *superclass*.
- A *child* class inherits from its *parent* class.
- A *derived* class inherits from its *base* class.
- Although these paired terms are identical and interchangeable in meaning, they should not be mixed—e.g., it is not proper to say that a *child* class inherits from its *superclass*.

Inheritance

- Open the *Picture.java* file, and at the beginning of the class definition you will see the following:

  public class Picture extends SimplePicture

- The *extends* keyword means *inherits from*; a *subclass* is said to *extend* the functionality of its *superclass*.
- The above code statement defines *Picture* to be a class that *inherits from SimplePicture*.
- Open the *SimplePicture.java* file, and within it you will find the following statement:

  private BufferedImage bufferedImage;

- This *BufferedImage* is an object field within the *SimplePicture* object; in other words, a *SimplePicture* object *contains* (or *HAS-A*) *BufferedImage* object.
- Note that the *private* keyword ensures that this field cannot be accessed from outside of the *SimplePicture* object.

Inheritance

- Now that you understand Java *packages* and *inheritance*, you can understand how the insertion of the *addBox()* method to the *Picture* class (as presented in our textbook) works.

```
import java.awt.*;
.
public class Picture extends SimplePicture
{
.
  public void addBox()
  {
    Graphics g = this.getGraphics();
    g.setColor(Color.red);
    g.fillRect(150, 200, 50, 50);
  }
}
```

- The *import* statement enables us to type *Graphics* instead of *java.awt.Graphics*.
- We can invoke the *getGraphics()* method, because *Picture* inherits from *SimplePicture* and so *Picture* gets all of SimplePicture's fields and methods.

AWT Graphics

- The *SimplePicture getGraphics()* method simply returns the *Graphics* object of the underlying *BufferedImage* object. We can see this by looking at the code:

```
public Graphics getGraphics()
{
  return bufferedImage.getGraphics();
}
```

- In the early days of Java, the *Graphics* object was the only way to control drawing onto images. Soon, some enhanced two-dimensional drawing capabilities were introduced through a subclass of *Graphics* called *Graphics2D*.
- Now, instead of using *Graphics* objects, *Graphics2D* objects are always returned and used, but to use the enhanced geometric methods you must cast the *Graphics* object to a *Graphics2D* object.

AWT Graphics

- Casting a *Graphics* object to a *Graphics2D* object:

```
Graphics g = this.getGraphics();
Graphics2D g2 = (Graphics2D) g;  // enables use of Graphics2D methods
g2.setColor(Color.BLUE);
g2.fillRect(10, 40, 80, 60);
g2.setColor(Color.RED);
g2.drawString( "I  took the red pill!" );
```

- This type of casting is known as *downcasting*, because you are casting downward within the class hierarchy (down from Graphics to Graphics2D, in this case).
- *Downcasting* is generally a very *dangerous* thing to do, since you run the risk of downcasting to a class level that does not match the class that the object was actually instantiated from.
- In this case, our downcast is safe. Simple *Graphics* objects are no longer used; *Graphics2D* objects have supplanted *Graphics* objects within Java.
- If a *Graphics* object is returned to us from a method or is passed to us as a parameter, we know that we can safely downcast it to a *Graphics2D* object.

Interfaces

- *Interfaces* are class-like constructs that generally do not have data fields, and only define method declarations.
- *Interfaces* do not provide method bodies, they only provide method declarations.
- Objects cannot be instantiated from an *interface*.
- Objects can only be instantiated from classes.
- In Java, a class can only *extend* from one other class.
- Some languages (C++, for example) allow classes to *inherit* from more than one class; such languages are said to support *multiple inheritance*.
- Java does not support *multiple inheritance*.
- However, a Java class can *implement* multiple *interfaces*.
- The ability to *implement* multiple *interfaces* confers a *multiple inheritance-like* ability to Java classes.

Interfaces

- A class implements an *interface* by providing a method body for each of the methods declared by the *interfaces* that the class *implements*.
- If a class does not provide method bodies for all of its *interface* methods, the class is deemed to be an *abstract class*, which means that objects cannot be instantiated from that class.
- *Abstract classes* must have the *abstract* keyword as part of its class definition, otherwise the compiler will issue an error. For example:

```
public interface Peaceable  // within file Peaceable.java
{
   void governPeacefully();
}
public abstract class Utopia implements Peaceable  // in Utopia.java
{
   ...
}
```

Interfaces

- The *SimplePicture* class has a method called explore().

```
FileChooser.setMediaPath("/home/sifu/Desktop/PH_GUZDIAL/intro-
  prog-java/mediasources/");
Picture p = new Picture(FileChooser.getMediaPath("kitten.jpg"));
p.explore();
```

- You can see from the definition of the method that it creates a *PictureExplorer* object.
- Open the *PictureExplorer.java* file, and look at how the class definition begins:

```
public class PictureExplorer implements MouseMotionListener,
  ActionListener, MouseListener
{ ...
```

- The *PictureExplorer* class implements three *interfaces*: *MouseMotionListener*, *ActionListener*, and *MouseListener*.

Interfaces

- It is the implementations of the *interface* methods that enable the *PictureExplorer* object to respond to user mouse clicks, mouse drags, and button clicks.
- You can find more information about these *interface* methods within the Java API Specification document under the *java.awt package* within the *Interfaces* section.

Static Methods

- So far the methods we have been using are *non-static* methods.
- *Non-static* methods are methods that belong to various objects instantiated from a given class.
- *Non-static* methods are defined within a class definition, but they are invoked upon the objects that have been instantiated from that class.
- *Static* methods do not belong to objects, the belong to a given class—in a sense, they *stay* (hence, *static*) within the class.
- *Static* methods are invoked upon the class, using the class name rather than an object reference.

Static Methods

- Non-static example:

```
public class Account
{
   private double balance;
   public Account(double amount)
   {
      balance = amount;
   }
   public double getBalance()  // a non-static method
   {
      return balance;
   }
}
Account savings = new Account( 350.42 );   // Object must be created!
System.out.println( savings.getBalance() );  // Object reference used to invoke
```

Static Methods

- Static example:

```
public class Account
{
   private double balance;
   public Account(double amount)
   {
      balance = amount;
   }
   public static String getBankName()
   {
      return “Joe's Bank”;
   }
}
// Object does not need to be created!
System.out.println( Account.getBankName() );  // Class name used to invoke
```

Private Methods

- A *private method* is a method that can only be invoked from within the object it belongs to.
- *Private methods* are usually methods that serve as *utility methods*—methods that help other *public* methods do their job.
- Consider the following:

```
private boolean authenticated()  //Invoked only from inside the object
{
...
}
public Connection connect()      // Invoked from outside the object
{
  if (this.authenticated())
  {
   ...
  }
  ...
}
```

Private Methods

- It is a best practice to restrict visibility as much as possible.
- If a method doesn't need to be *public* then you should not make it *public*.
- If you can do what you need, even if a method is *private*, then that method should be *private*.

Another Way to Draw

- *SimplePicture* has *BufferedImage* as a field (instance variable).
- This underlying *BufferedImage* has its own *Graphics* (*Graphics2D*, actually) object that we obtain by calling the *getGraphics()* method.
- We have done our drawing by calling the various drawing methods on that *Graphics* object.
- There is another way that we can experiment with drawing.

Another Way to Draw

- *Swing*
  - *Swing* is a package that provides a rich set of user-interface classes.
  - *JFrame* is a *Swing* class that provides the enclosing window frame of a basic Java application.
  - A *JPanel* is a *Swing* class that has its own *Graphics* (*Graphics2D*) object, allowing you to draw within the panel.
  - The *JPanel* class has a *paintComponent(Graphics g)* method that renders the graphics within its panel.
  - The *paintComponent* method is invoked automatically by the *Swing* event-dispatch thread; you should never invoke *paintComponent*—it will be invoked for you.

Another Way to Draw

- The following code will give you another means to draw:

```
// DrawPanel.java
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JPanel;

public class DrawPanel extends JPanel
{
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g );  // invoke the superclass paintComponent
      this.setBackground( Color.WHITE );
      g.setColor( Color.CYAN );
      g.fillOval( 50, 50, 90, 60 );
   }
}
```

Another Way to Draw

```
// DrawApp.java
import java.awt.Color;
import javax.swing.JFrame;

public class DrawApp
{
   public static void main( String args[] )
   {
      JFrame frame = new JFrame( "Drawing on a JPanel" );
      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      DrawPanel drawPanel = new DrawPanel();
      drawPanel.setBackground( Color.WHITE );
      frame.add( drawPanel );
      frame.setSize( 200, 200 );
      frame.setVisible( true );
   }
}
```

Creating New Classes

- To create the classes *DrawPanel* and *DrawApp* as show previously, from within *DrJava* you can select...

  File -> New

- You can then cut and paste the class definition into the definitions pane.
- Clicking on the *Save* button will allow you to save the definition to a *.java* file.
- Clicking on the *Compile* button will compile the *.java* class definition and generate a *.class* file.
- After creating the *DrawPanel* and *DrawApp .class*s files, you can then *Run* the *DrawApp* application.

Creating New Classes

- Alternatively, you can use an IDE other than *DrJava*.
- You can also use a simple text editor—even *Notepad* will work fine (you won't have any nifty features like color-coding, however).
- If you use a text editor, then you will have to compile and run your code from the command line. To do so, launch a console or terminal application, and type the following:

  javac DrawPanel.java

  javac DrawApp.java

  java DrawApp

- The first two commands above compile the classes, and the third command runs the application.