Agenda

• Overloading
• Designing “real” classes
• Inheritance
• Advanced topics
  – Abstract classes
  – Interfaces
  – Understanding polymorphism
  – Setting a CLASSPATH and using packages
  – Visibility modifiers
  – Creating on-line documentation using JavaDoc
Example 4: Overloading

class Ship4 {
    public double x=0.0, y=0.0, speed=1.0, direction=0.0;
    public String name;

    public Ship4(double x, double y, 
        double speed, double direction, 
        String name) {
        this.x = x;
        this.y = y;
        this.speed = speed;
        this.direction = direction;
        this.name = name;
    }

    public Ship4(String name) {
        this.name = name;
    }

    private double degreesToRadians(double degrees) {
        return(degrees * Math.PI / 180.0);
    }
}
... public void move() {
    move(1);
}

public void move(int steps) {
    double angle = degreesToRadians(direction);
    x = x + (double)steps * speed * Math.cos(angle);
    y = y + (double)steps * speed * Math.sin(angle);
}

public void printLocation() {
    System.out.println(name + " is at (" + x + "," + y + ").");
}
}
Overloading: Testing and Results

```java
public class Test4 {
    public static void main(String[] args) {
        Ship4 s1 = new Ship4("Ship1");
        Ship4 s2 = new Ship4(0.0, 0.0, 2.0, 135.0, "Ship2");
        s1.move();
        s2.move(3);
        s1.printLocation();
        s2.printLocation();
    }
}
```

- **Compiling and Running:**
  ```
javac Test4.java
java Test4
```

- **Output:**
  ```
Ship1 is at (1,0).
Ship2 is at (-4.24264,4.24264).
```
Overloading: Major Points

• **Idea**
  − Allows you to define more than one function or constructor with the same name
  • Overloaded functions or constructors must differ in the number or types of their arguments (or both), so that Java can always tell which one you mean

• **Simple examples:**
  − Here are two `square` methods that differ only in the type of the argument; they would both be permitted inside the same class definition.

    ```java
    // square(4) is 16
    public int square(int x) { return(x*x); }
    
    // square("four") is "four four"
    public String square(String s) {
        return(s + " " + s);
    }
    ```
Example 5: OOP Design and Usage

/** Ship example to demonstrate OOP in Java. */

public class Ship {
    private double x=0.0, y=0.0, speed=1.0, direction=0.0;
    private String name;

    /** Get current X location. */
    public double getX() {
        return(x);
    }

    /** Set current X location. */
    public void setX(double x) {
        this.x = x;
    }
}

www.corewebprogramming.com
Example 5: Major Points

- **Encapsulation**
  - Lets you change internal representation and data structures *without users of your class changing their code*
  - Lets you put constraints on values *without users of your class changing their code*
  - Lets you perform arbitrary side effects *without users of your class changing their code*

- **Comments and JavaDoc**
  - See later slides (or book) for details
Example 6: Inheritance

```java
public class Speedboat extends Ship {
    private String color = "red";

    public Speedboat(String name) {
        super(name);
        setSpeed(20);
    }

    public Speedboat(double x, double y,
                   double speed, double direction,
                   String name, String color) {
        super(x, y, speed, direction, name);
        setColor(color);
    }

    public void printLocation() {
        System.out.print(getColor().toUpperCase() + " ");
        super.printLocation();
    }

    ...
}
```
public class SpeedboatTest {
    public static void main(String[] args) {
        Speedboat s1 = new Speedboat("Speedboat1");
        Speedboat s2 = new Speedboat(0.0, 0.0, 2.0, 135.0,
                                      "Speedboat2", "blue");
        Ship s3 = new Ship(0.0, 0.0, 2.0, 135.0, "Ship1");
        s1.move();
        s2.move();
        s3.move();
        s1.printLocation();
        s2.printLocation();
        s3.printLocation();
    }
}
Inheritance Example: Result

• Compiling and Running:
  
  javac SpeedboatTest.java
  
  – The above calls javac on Speedboat.java and Ship.java automatically
  
  java SpeedboatTest

• Output
  
  RED Speedboat1 is at (20,0).
  BLUE Speedboat2 is at (-1.41421,1.41421).
  Ship1 is at (-1.41421,1.41421).
Example 6: Major Points

• Format for defining subclasses
• Using inherited methods
• Using super(...) for inherited constructors
  – *Only* when the zero-arg constructor is not OK
• Using super.someMethod(...) for inherited methods
  – *Only* when there is a name conflict
Inheritance

• Syntax for defining subclasses
  ```java
  public class NewClass extends OldClass {
      ...
  }
  ```

• Nomenclature:
  – The existing class is called the superclass, base class or parent class
  – The new class is called the subclass, derived class or child class

• Effect of inheritance
  – Subclasses automatically have all public fields and methods of the parent class
  – You don’t need any special syntax to access the inherited fields and methods; you use the exact same syntax as with locally defined fields or methods.
  – You can also add in fields or methods not available in the superclass

• Java doesn’t support multiple inheritance
Inherited constructors and super(...)

• When you instantiate an object of a subclass, the system will automatically call the superclass constructor first
  – By default, the zero-argument superclass constructor is called unless a different constructor is specified
  – Access the constructor in the superclass through

    super(args)

  – If super(...) is used in a subclass constructor, then super(...) must be the first statement in the constructor

• Constructor life-cycle
  – Each constructor has three phases:
    1. Invoke the constructor of the superclass
    2. Initialize all instance variables based on their initialization statements
    3. Execute the body of the constructor
Overridden methods and super.method(...) 

- When a class defines a method using the **same name, return type, and arguments** as a method in the superclass, then the class **overrides** the method in the superclass
  - Only non-static methods can be overridden

- If there is a locally defined method and an inherited method that have the same name and take the same arguments, you can use the following to refer to the inherited method
  
  ```java
  super.methodName(...)
  ```

  - Successive use of `super` (super.super.methodName) will not access overridden methods higher up in the hierarchy; `super` can only be used to invoke overridden methods from within the class that does the overriding
Advanced OOP Topics

- Abstract classes
- Interfaces
- Polymorphism details
- CLASSPATH
- Packages
- Visibility other than public or private
- JavaDoc details
Abstract Classes

• **Idea**
  – Abstract classes permit declaration of classes that define only part of an implementation, leaving the subclasses to provide the details

• **A class is considered abstract if at least one method in the class has no implementation**
  – An abstract method has no implementation (known in C++ as a pure virtual function)
  – Any class with an abstract method must be declared abstract
  – If the subclass overrides all the abstract methods in the superclass, than an object of the subclass can be instantiated

• **An abstract class can contain instance variables and methods that are fully implemented**
  – Any subclass can override a concrete method inherited from the superclass and declare the method abstract
Abstract Classes (Continued)

• An abstract class cannot be instantiated, however references to an abstract class can be declared

```java
public abstract ThreeDShape {
    public abstract void drawShape(Graphics g);
    public abstract void resize(double scale);
}
```

```java
ThreeDShape s1;
ThreeDShape[] arrayOfShapes = new ThreeDShape[20];
```

• Classes from which objects can be instantiated are called concrete classes
Interfaces

• Idea
  – Interfaces define a Java type consisting *purely* of constants and abstract methods
  – An interface does not implement any of the methods, but imposes a design structure on any class that uses the interface
  – A class that implements an interface must either provide definitions for all methods or declare itself abstract
• **Modifiers**
  - All methods in an interface are implicitly abstract and the keyword `abstract` is not required in a method declaration.
  - Data fields in an interface are implicitly `static final` (constants).
  - All data fields and methods in an interface are implicitly `public`.

```java
public interface Interface1 {
    DataType CONSTANT1 = value1;
    DataType CONSTANT2 = value2;

    ReturnType1 method1(ArgType1 arg);
    ReturnType2 method2(ArgType2 arg);
}
```
• **Extending Interfaces**
  – Interfaces can extend other interfaces, which brings rise to sub-interfaces and super-interfaces
  – Unlike classes, however, an interface can extend more than one interface at a time

  ```java
  public interface Displayable extends Drawable, Printable {
    // Additional constants and abstract methods
    ...
  }
  ```

• **Implementing Multiple Interfaces**
  – Interfaces provide a *form* of multiple inheritance because a class can implement more than one interface at a time

  ```java
  public class Circle extends TwoDShape implements Drawable, Printable {
    ...
  }
  ```
Polymorphism

• “Polymorphic” literally means “of multiple shapes” and in the context of object-oriented programming, polymorphic means “having multiple behavior”

• A polymorphic method results in different actions depending on the object being referenced
  – Also known as late binding or run-time binding

• In practice, polymorphism is used in conjunction with reference arrays to loop through a collection of objects and to access each object's polymorphic method
public class PolymorphismTest {
    public static void main(String[] args) {

        Ship[] ships = new Ship[3];

        ships[0] = new Ship(0.0, 0.0, 2.0, 135.0, "Ship1");
        ships[1] = new Speedboat("Speedboat1");
        ships[2] = new Speedboat(0.0, 0.0, 2.0, 135.0,
                                    "Speedboat2", "blue");

        for(int i=0; i<ships.length ; i++) {
            ships[i].move();
            ships[i].printLocation();
        }
    }
}
Polymorphism: Result

• Compiling and Running:

```java
javac PolymorphismTest.java
java PolymorphismTest
```

• Output

```
RED Speedboat1 is at (20,0).
BLUE Speedboat2 is at (-1.41421,1.41421).
Ship1 is at (-1.41421,1.41421).
```
CLASSPATH

• The CLASSPATH environment variable defines a list of directories in which to look for classes
  – Default = current directory and system libraries
  – Best practice is to not set this when first learning Java!

• Setting the CLASSPATH
  set CLASSPATH = .;C:\java;D:\cwp\echoserver.jar
  setenv CLASSPATH .:~/java:/home/cwp/classes/
  – The period indicates the current working directory

• Supplying a CLASSPATH
  javac -classpath .;D:\cwp WebClient.java
  java -classpath .;D:\cwp WebClient
Creating Packages

- A **package** lets you group classes in subdirectories to **avoid accidental name conflicts**
  
  - To create a package:
    1. Create a subdirectory with the same name as the desired package and place the source files in that directory
    2. Add a package statement to each file
      
      ```java
      package packagename;
      ```
    
    3. Files in the main directory that want to use the package should include
      
      ```java
      import packagename.*;
      ```
  
- The package statement must be the **first statement** in the file

- If a package statement is omitted from a file, then the code is part of the default package that has no name

www.corewebprogramming.com
Package Directories

• The package hierarchy reflects the file system directory structure

   - The root of any package must be accessible through a Java system default directory or through the CLASSPATH environment variable
Visibility Modifiers

• public
  – This modifier indicates that the variable or method can be accessed anywhere an instance of the class is accessible
  – A class may also be designated public, which means that any other class can use the class definition
  – The name of a public class must match the filename, thus a file can have only one public class

• private
  – A private variable or method is only accessible from methods within the same class
  – Declaring a class variable private "hides" the data within the class, making the data available outside the class only through method calls
Visibility Modifiers, cont.

- **protected**
  - Protected variables or methods can only be accessed by methods within the class, within classes in the same package, and within subclasses
  - Protected variables or methods are inherited by subclasses of the same or different package

- **[default]**
  - A variable or method has default visibility if a modifier is omitted
  - Default visibility indicates that the variable or method can be accessed by methods within the class, and within classes in the same package
  - Default variables are inherited only by subclasses in the same package
Protected Visibility: Example

- Cake, ChocolateCake, and Pie inherit a calories field
- However, if the code in the Cake class had a reference to object of type Pie, the protected calories field of the Pie object could not be accessed in the Cake class
  - Protected fields of a class are not accessible outside its branch of the class hierarchy (unless the complete tree hierarchy is in the same package)
• **Even through inheritance, the fat data field cannot cross the package boundary**
  
  – Thus, the fat data field is accessible through any Dessert, Pie, and Cake object within any code in the Dessert package
  
  – However, the ChocolateCake class does not have a fat data field, nor can the fat data field of a Dessert, Cake, or Pie object be accessed from code in the ChocolateCake class
## Visibility Summary

<table>
<thead>
<tr>
<th>Data Fields and Methods</th>
<th>public</th>
<th>protected</th>
<th>default</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessible from same class?</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Accessible to classes (nonsubclass) from the same package?</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Accessible to subclass from the same package?</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Accessible to classes (nonsubclass) from different package?</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Accessible to subclasses from different package?</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Inherited by subclass in the same package?</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Inherited by subclass in different package?</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Other Modifiers

• **final**
  – For a class, indicates that it cannot be subclassed
  – For a method or variable, cannot be changed at runtime or overridden in subclasses

• **synchronized**
  – Sets a lock on a section of code or method
  – Only one thread can access the same synchronized code at any given time

• **transient**
  – Variables are not stored in serialized objects sent over the network or stored to disk

• **native**
  – Indicates that the method is implement using C or C++
Comments and JavaDoc

• **Java supports 3 types of comments**
  - // Comment to end of line.
  - /* Block comment containing multiple lines. Nesting of comments in not permitted. */
  - /** A JavaDoc comment placed before class definition and nonprivate methods. Text may contain (most) HTML tags, hyperlinks, and JavaDoc tags. */

• **JavaDoc**
  - Used to generate on-line documentation
    javadoc Foo.java Bar.java
  - JavaDoc 1.4 Home Page
    • http://java.sun.com/j2se/1.4/docs/tooldocs/javadoc/
Useful JavaDoc Tags

- **@author**
  - Specifies the author of the document
  - Must use `javadoc -author ...` to generate in output
    ```
    /** Description of some class ...
     *  
     *   @author <A HREF="mailto:brown@lmbrown.com">Larry Brown</A>
     */
    ```

- **@version**
  - Version number of the document
  - Must use `javadoc -version ...` to generate in output

- **@param**
  - Documents a method argument

- **@return**
  - Documents the return type of a method
Useful JavaDoc Command-line Arguments

- **-author**
  - Includes author information (omitted by default)

- **-version**
  - Includes version number (omitted by default)

- **-noindex**
  - Tells javadoc not to generate a complete index

- **-notree**
  - Tells javadoc not to generate the tree.html class hierarchy

- **-link, -linkoffline**
  - Tells javadoc where to look to resolve links to other packages

  -link http://java.sun.com/j2se/1.3/docs/api
  -linkoffline http://java.sun.com/j2se/1.3/docs/api
c:\jdk1.3\docs\api
/** Ship example to demonstrate OOP in Java.
 * @author <A HREF="mailto:brown@corewebprogramming.com">Larry Brown</A>
 * @version 2.0 */

public class Ship {
    private double x=0.0, y=0.0, speed=1.0, direction=0.0;
    private String name;

    /** Build a ship with specified parameters. */
    public Ship(double x, double y, double speed, double direction, String name) {
        setX(x);
        setY(y);
        setSpeed(speed);
        setDirection(direction);
        setName(name);
    }
    ...
}
JavaDoc, Example

```bash
> javadoc -linkoffline http://java.sun.com/j2se/1.3/docs/api
c:\jdk1.3\docs\api
  -author -version -noindex -notree Ship.java
```
JavaDoc: Result

Class Ship

```java
java.lang.Object
 |   --Ship
```

public class Ship
extends Object

Ship example to demonstrate OOP in Java.

Version:
2.0
Author:
Larry Brown

Constructor Summary

```java
Ship(double x, double y, double speed, double direction,
    String name)
    Build a ship with specified parameters.

Ship(String name)
    Build a ship with default values (x=0, y=0, speed=1.0, direction=0.0).
```

Method Summary
Summary

- Overloaded methods/constructors, except for the argument list, have identical signatures
- Use `extends` to create a new class that inherits from a superclass
  - Java does not support multiple inheritance
- An inherited method in a subclass can be overridden to provide custom behavior
  - The original method in the parent class is accessible through `super.methodName(...)`
- Interfaces contain only abstract methods and constants
  - A class can implement more than one interface
Summary (Continued)

• With polymorphism, binding of a method to an object is determined at run-time
• The CLASSPATH defines in which directories to look for classes
• Packages help avoid namespace collisions
  – The package statement must be first statement in the source file before any other statements
• The four visibility types are: public, private, protected, and default (no modifier)
  – Protected members can only cross package boundaries through inheritance
  – Default members are only inherited by classes in the same package
Questions?