
CS11 – Java

Fall 2009-2010

Lecture 3

Today's Topics

- Java Packages
 - Class Inheritance
 - Abstract Classes
 - Polymorphism
-

Java Packages

- Classes can be grouped into packages
 - A package is a collection of related types
 - Packages provide namespace management
 - Can't have two classes with same name in same package
 - Classes can have the same name, if they are in *different* packages
 - By default, a class is in the “default package”
 - Default package has no name!
 - Use **package** keyword to specify different package
 - `package cs11;`
 - Must be first statement in your `.java` file
 - Affects where `.java` and `.class` files must be placed!
 - For now, don't specify **package** keyword for your classes
-

Using Classes in Packages

- If a class is in a package:

- Must refer to class with *qualified* name:

```
javax.swing.JButton myButton =  
    new javax.swing.JButton("Crash Program");
```

- Must *import* the class itself:

```
import javax.swing.JButton;  
...
```

```
JButton myButton = new JButton("Crash Program");
```

- Must import entire package:

```
import javax.swing.*;  
...
```

```
JButton myButton = new JButton("Crash Program");
```

The Java API and Packages

- All Java API classes are in packages
- Classes in `java.lang` are automatically imported
 - Don't need to explicitly import anything in `java.lang`
- To import Java classes not in `java.lang` package:

```
import javax.swing.JButton;  
import javax.swing.JFrame;  
...
```

- Or:

```
import javax.swing.*;
```

- Importing a package is *not* recursive!
 - Importing `java.*` won't get you anywhere.
-

Class Inheritance

- A third of the “four big OOP concepts”
 - A class can extend another class to build on its functionality
 - Terminology:
 - Parent class, or superclass, or base class
 - Child class, or subclass, or derived class
 - Child classes inherit all methods and fields within parent class
 - Can add new functionality
 - Can also override parent-class methods
-

Class Inheritance (2)

- Class inheritance models an “is-a” relationship

- Example class hierarchy:

Vehicle

└ Wheeled Vehicle

└ Water Vehicle

└ Dump Truck

└ Sailboat └ Barge

- The child class is a specialization of the parent class
 - Child class also has characteristics of parent class
 - Can treat child class as if it were any parent-type
 - “A dump truck is a wheeled vehicle.”
 - “A sailboat is a vehicle.”
 - “A water vehicle is a vehicle.”
-

Class Inheritance (3)

- Example class hierarchy:

Vehicle

└ Wheeled Vehicle

└ Water Vehicle

└ Dump Truck

└ Sailboat └ Barge

- Sibling types do not model an “is-a” relationship!

- These statements are clearly false:

- “A dump truck is a water vehicle.”
- “A wheeled vehicle is a barge.”

- What about these statements?

- “A vehicle is a dump truck.”
- “A water vehicle is a sailboat.”

- Depends on the actual vehicle being considered!

- Need to examine a specific vehicle to verify the statement
-

Example Class Hierarchy

- The number classes in Java

 - `java.lang.Object`

 - ↳ `java.lang.Number`

 - ↳ `java.lang.Integer`

- `Integer` “is a” `Number`, “is an” `Object`
 - `Integer` extends `Number`, which extends `Object`
 - `Integer` inherits methods that `Object` defines
 - `boolean equals(Object o)`
 - `int hashCode()`
 - `String toString()`
 - `Class getClass()`
 - `Integer` overrides some of these methods too
-

Overriding `Object.toString()`

- Really useful idea, especially for debugging
- Used in string concatenation
 - You type this:

```
String msg = "Point is " + pt;
```

- Compiler automatically does this:

```
String msg = "Point is " + pt.toString();
```

- Simple to define:

```
public String toString() {  
    return "(" + xCoord + "," + yCoord + ")";  
}
```

Classes and Objects

- A class' parent-class methods can be called without any special syntax.

```
Integer intObj = new Integer(53);
```

```
...
```

```
Class c = intObj.getClass(); // Get type info
```

- **Integer** is also an **Object** – can call methods declared and/or implemented on **Object**
 - Child class can also provide its own methods

```
System.out.println("Value is " + intObj.intValue());
```
 - **Integer** *extends* **Object**'s functionality
 - `intValue()` returns an `int` version of the **Integer**
-

Reference Types

- Every reference has a class-type associated with it

```
Object obj;    // A reference of type Object
Integer val;   // A reference of type Integer
```

- The variable's type dictates what is accessible

- Example:

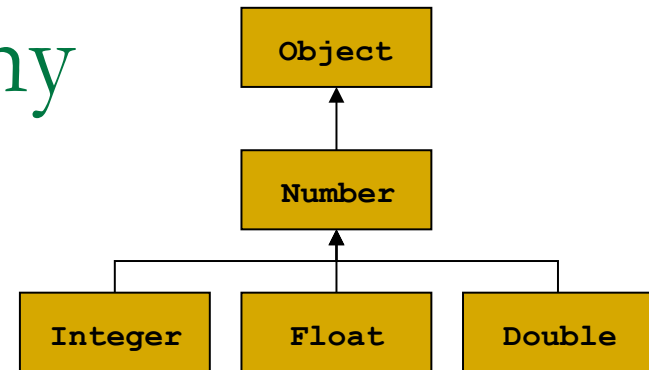
```
Object obj = new Integer(38);
```

```
...
```

```
System.out.println(obj.intValue());    COMPILE ERROR
```

- Compile error, because `Object` doesn't define `intValue()`
 - `intValue()` is declared in `Number` class (parent of `Integer`)
 - Even though `obj` refers to an `Integer` object, only the `Object` methods are visible
-

Navigating the Hierarchy



- Number hierarchy is like this:

- Moving down the hierarchy requires a run-time test.

```
Object obj = new Integer(453);
```

```
...
```

```
int i = ((Integer) obj).intValue(); // Cast obj
```

- You could also try this:

```
float f = ((Float) obj).floatValue(); // Runtime error
```

- This code compiles, but it will report an error when run
- Java can't assume the actual object-type at compile time!
 - (Even when it's obvious to a human...)
- So, we have a runtime type-check, and a potential error.

What Child Classes Don't Get

- Child classes cannot access **private** members in parent classes
 - **protected** access-modifier allows the child class to access parent-class' members
 - Only available within the class, and to subclasses
 - Looser than **private**, but still not **public**!
 - Child classes also don't inherit static fields and methods
 - They can be accessed, but they are not inherited
-

A Generic Task Class

```
public class Task {
    private String name;
    private boolean done;

    public Task(String taskName) {
        name = taskName;
        done = false;
    }

    /** Just record that the task is done. */
    public void doTask() {
        done = true;
    }

    /** Report if the task is done or not. */
    public boolean isDone() {
        return done;
    }
}
```

Making Useful Tasks

- Our **Task** class is *very* generic...
 - ...so generic that it's nearly useless!
- Extend **Task** class to provide useful tasks

```
public class FileUploadTask extends Task {  
    public FileUploadTask() {  
        // Call parent-class constructor  
        super("upload file");  
    }  
  
    ...  
}
```

- Parent-class constructors are *not* inherited!
 - If parent class doesn't have a default constructor, we must *explicitly* call one in the child class, using **super** keyword
-

Overriding Parent-Class Methods

- **FileUploadTask** should provide its own implementation of **doTask ()**

```
public class FileUploadTask extends Task {
    ...

    /** Perform the file-upload operation. */
    public void doTask() {
        ... // Open a connection, read the file, etc.
    }
}
```

- Method's signature is same as parent-class' method signature
 - This overrides Task's implementation of doTask()
-

Polymorphism

- Now we want to upload a file:

```
Task t = new FileUploadTask();  
t.doTask();
```

- Which implementation of `doTask()` does this call?
 - In Java, all instance-methods are virtual
 - Even though `t` is a `Task` reference, the `FileUploadTask` implementation is called
 - Reason: `t` refers to an object of type `FileUploadTask`
 - This is called polymorphism
 - The fourth “Big OOP Concept”
 - A statement’s behavior changes, depending on the type of the objects involved
-

Calling Parent-Class Methods

- Problem:

- `FileUploadTask.doTask()` doesn't set `done` to `true`
- Also, `done` is private!

- One solution:

- `FileUploadTask.doTask()` implementation can call the parent-class implementation:

```
/** Perform the file-upload operation. */  
public void doTask() {  
    ... // Open a connection, read the file, etc.  
  
    // All done!  
    super.doTask();  
}
```

The **Task** Abstraction

- Doesn't make sense for **Task** to have an implementation of **doTask ()**
 - Change **Task** to be an abstract class
 - An abstract class declares a set of behaviors, but only *partially* defines it.
 - Abstract classes cannot be instantiated
 - Child classes must be provided, that implement the missing functionality
 - Example: **FileUploadTask** must provide an implementation of **doTask ()**, that uploads a file.
-

The New, Abstract **Task** Class

- Our abstract Task class:

```
// A class that represents a generic task
public abstract class Task {
    private String name;
    private boolean done;

    public Task(String taskName) {
        name = taskName;
        done = false;
    }

    // Child classes implement this method.
    public abstract void doTask();

    ... // Rest of class
}
```

- Abstract classes can still have fields and non-abstract methods
-

The New **FileUploadTask**

- **FileUploadTask** doesn't "override" `doTask()`
 - There's nothing to override!
 - **FileUploadTask** *implements* `doTask()`

- Again, the signatures must match up

```
/** Implement doTask() to upload a file. */  
public void doTask() {  
    ... // Open a connection, read the file, etc.  
}
```

- (Without the **abstract** modifier, of course!)
 - Of course, we can't do `super.doTask()` anymore
 - Child class *must* provide an implementation of every abstract parent-class method
 - If not, child class must also be declared abstract.
-

Completing the Abstraction

- How can a task be marked as done?
 - A simple solution: set **done** to be protected
 - Another good solution:
 - **Task** can provide another protected method to do this:

```
protected void reportTaskDone() {  
    if (done) {  
        ... // Task was already done! Complain.  
    }  
    done = true;  
}
```
 - Now only child classes can report that the task is done
 - Which solution is more extensible?
 - Might want to add other processing when a task is finished
 - Can easily add this to **reportTaskDone()** later
-

Task References

- You can't instantiate the abstract **Task** class

```
Task t = new Task("send e-mail");    COMPILE ERROR
```

- The implementation of **Task** is incomplete!

- You *can* have a **Task**-reference

```
Task t = new FileUploadTask();  
t.doTask();    // Calls FileUploadTask.doTask()  
t = new SendEMailTask();  
t.doTask();    // Calls SendEMailTask.doTask()
```

- The correct implementation of **doTask()** gets called because of polymorphism

- APIs are made generic by using the base-class type

```
void enqueueTask(Task t) {  
    pendingList.store(t);  
}
```

Pitfalls With `equals ()` and Subclasses

- So, you have your `Point2d` class...

```
public boolean equals(Object obj) {  
    if (obj instanceof Point2d) {  
        Point2d other = (Point2d) obj;  
        if (xCoord == other.getX() &&  
            yCoord == other.getY()) {  
            return true;  
        }  
    }  
    return false;  
}
```

Works great!

- But something's missing...
-

Colorful Point2d

- Now you want a `Point2d` with a color.

```
public class ColorPoint2d extends Point2d {
    private Color ptColor;
    ...
    Color getColor() { return ptColor; }
}
```

- “Hmm, guess I’ll go ahead and implement `equals()` again...”

```
public boolean equals(Object obj) {
    if (obj instanceof ColorPoint2d) {
        ColorPoint2d other = (ColorPoint2d) obj;
        ... // Do your comparison.
    }
    return false;
}
```

Ouch, Now It's Broke!

- Now `Point2d.equals()` is less strict than `ColorPoint2d.equals()`

```
Point2d p = new Point2d(3, 5);  
ColorPoint2d cp = new ColorPoint2d(3, 5, Color.RED);  
System.out.println("p.equals(cp) = " + p.equals(cp));  
System.out.println("cp.equals(p) = " + cp.equals(p));
```

- Remember:
 - `a.equals(b)` should be same as `b.equals(a)`
- But in this case, the program prints:

```
p.equals(cp) = true  
cp.equals(p) = false
```

Moral

- Be careful about overriding `equals ()` on subclasses.
 - In general:
 - **Know the contracts on common methods!!!**
 - In this example, issues occur if `Point2d` and `ColorPoint2d` are used together in a program
 - If not, not so important to ensure proper `equals ()` behavior
 - If used together in same program, probably best to *not* derive `ColorPoint2d` from `Point2d`
 - Create a separate `ColorPoint2d` class with a `Point2d` field and a `Color` field
 - Provide accessors for `Point2d` and `Color` fields, and write a correct `equals ()` implementation
 - Everybody is happy. 😊
-

This Week's Homework

- ...may be different from previous terms
 - (I'll see what I come up with...)
 - Start playing with the Java Swing API
 - Very powerful and useful user interface API
 - A lot of features, and a lot of classes
 - Read more about it in the Java API Docs
 - Will also involve class hierarchies
 - Swing API has very large class hierarchies...
-