CS1007: Object Oriented Design and Programming in Java

Lecture #17
Mar 23
Shlomo Hershkop
shlomo@cs.columbia.edu

Outline

• Objects and types
• Understanding what is happening in the background
• Understanding the program

• Reading: 7-7.4
Announcements

• Posted last class notes

• Code from the book:
  – Check the resource webpage

• Homework 3 will be released this weekend

Understanding variables

• To understand what is going on with variables in any programming language, need to understand
• Which types are support
• Which values can be assigned to them
Java view of Types

- Primitive types:
- Class types
- Interface types
- Array types
- The null type
- Note:
  - void is not a type

Values

- value of primitive type
- reference to object of class type
- reference to array
- null
- Note: Can't have value of interface type
Array types

• Although arrays can be thought of as a collection of types, it actually is its own type

Example of inheritance

• Interface java.awt.Shape
  – Represents a two dimensional shape

• Some implementations:
  – Rectangle
  – Polygon
• So I can say:

```java
Shape shapeobj;
Rectangle rec = new Rectangle();
Polygon poly = new Polygon();
shapeobj = rec;

System.out.println("shape is now: " + shapeobj);
```

Careful

```java
Rectangle[] r = new Rectangle[10];
Shape[] s = r;

• This assignment will compile fine
s[0] = new Polygon();

• But Throws an ArrayStoreException at runtime
```
Why primitives

- If java is so object oriented

- How do primitives fit in?
Upgrading

- Can always upgrade a primitive to an equivalent class:

- Integer i = new Integer(5);

- Why would you want to upgrade to object?

- Should be aware of memory overhead

Wrapping

- Primitive types aren't classes
- Use wrappers when objects are expected
- Wrapper for each type:

  Integer Short Long Byte
  Character Float Double Boolean
Before java 1.5

Integer A = new Integer(5);
...
Int x = A.intValue();

Boxing

• Auto-boxing and auto-unboxing
• Integer X = 5;
ArrayList<Integer> numbers = new ArrayList<Integer>();
    numbers.add(13);
int n = numbers.get(0);
In between types

- Enum is a type with a preset number of values
- Great for keeping track of states
- Enum types are classes; can add methods, fields, constructors
- Enum API

Enumerated

```java
enum Size { SMALL, MEDIUM, LARGE }
```

- Typical use:
  ```java
  Size imageSize = Size.MEDIUM;
  if (imageSize == Size.SMALL) . .
  ```
• Safer than integer constants
  
  ```java
  public static final int SMALL = 1;
  public static final int MEDIUM = 2;
  public static final int LARGE = 3;
  ```

---

Typesafe Enumeration

• enum equivalent to class with fixed number of instances

  ```java
  public class Size
  {
      private Size() { }
      public static final Size SMALL = new Size();
      public static final Size MEDIUM = new Size();
      public static final Size LARGE = new Size();
  }
  ```
Object testing

- Many methods will return an Object object.

- Object Obj = ????

- How do we figure out what we are dealing with?

Type Inquiry

- Test whether e is a Shape:
  
  ```java
  if (e instanceof Shape) …
  ```

- Good idea before doing a cast:
  
  ```java
  Shape s = (Shape) e;
  ```
• Remember: we don't know exact type of e

• WHY??

• Note:
  – If e is null, test returns false (no exception)

Confusion

• If Object class isn’t confusing enough

• There is also a type of class called:
  • Class
Plain old class

- getClass method gets class of any object
- Returns object of type Class
- Class object describes a type
  Object e = new Rectangle();
  Class c = e.getClass();
  System.out.println(c.getName()); // prints java.awt.Rectangle

- .class suffix yields Class object:
  Class c = Rectangle.class;
- Class is not exactly a class since also works for primitives
  int.class
  void.class
  Shape.class
• Use `Class.forName` method to yields a `Class` object:

```java
Class c =
    Class.forName("java.awt.Rectangle");
```

### An Employee Object vs. the Employee.class Object

- **: Employee**
  - `name = "Jane Doe"
  - `salary = 50000`

- **: Class**
  - `name = "Employee"
  - `superclass = null`

- **: Class**
  - `name = "java.lang.Object"
  - `superclass = null`
Checking Type

• Test whether e is a Rectangle:
  if (e.getClass() == Rectangle.class) . . .

• Why can we use the ==

• A unique Class object for every class
• Test fails for subclasses
• Use instanceof to test for subtypes:
  – if (e instanceof Rectangle) . . .
Array Types

• Can apply getClass to an array
• Returned object describes an array type
  double[] a = new double[10];
  Class c = a.getClass();
  if (c.isArray())
      System.out.println(c.getComponentType());

• getName produces strange names for array types
  [Z for boolean[]
  [D for double[])
  [[java.lang.String; for String[][]

SUPERclass

• All classes extend Object
• Most useful methods:
  – String toString()
  – boolean equals(Object otherObject)
  – Object clone()
  – int hashCode()
**toString**

- Returns a string representation of the object
- Useful for debugging
- Example: Rectangle.toString returns something like
  ```java
  java.awt.Rectangle[x=5,y=10,width=20,height=30]
  ```
- toString used by concatenation operator
- `aString + anObject` means
  `aString + anObject.toString()`

**Default**

- `Object.toString()`
  - Prints class name and object address
    ```java
    System.out.println(System.out) yields
    java.io.PrintStream@d2460bf
    ```
- Implementor of PrintStream didn't override `toString:`
Overriding toString

• Format all fields:
public class Employee
{
    public String toString()
    {
        return getClass().getName()
            + "[name=" + name
            + ",salary=" + salary
            + "]";
    }
    ...
}

• Typical string:
Employee[name=Harry Hacker,salary=35000]

Subclass toString

• Format superclass first
public class Manager extends Employee
{
    public String toString()
    {
        return super.toString()
            + ";[department=" + department + "]";
    }
    ...
}

• Typical string
Manager[name=Dolly Dollar,salary=100000][department=Finance]
Equals()

- equals tests for equal contents
- Used in many standard library methods
- Example: ArrayList.indexOf
  - Will trigger a equals call on your object in the array
- Unique to your class implementation

```java
/**
 * Searches for the first occurrence of the given argument, testing for equality using the equals method.
 * @param elem an object.
 * @return the index of the first occurrence of the argument in this list; returns -1 if the object is not found.
 */
public int indexOf(Object elem)
{
    if (elem == null)
    {
        for (int i = 0; i < size; i++)
            if (elementData[i] == null) return i;
    }
    else
    {
        for (int i = 0; i < size; i++)
            if (elem.equals(elementData[i])) return i;
    }
    return -1;
}
```
Object.equals

- Object.equals tests for identity:

```java
public class Object {
    public boolean equals(Object obj) {
        return this == obj;
    }
    ...
}
```

- Override equals if you don’t want to inherit that behavior

Requirements Rules

1. reflexive: x.equals(x)
2. symmetric: x.equals(y) if and only if y.equals(x)
3. transitive: if x.equals(y) and y.equals(z), then x.equals(z)
4. x.equals(null) must return false
Employee.equals

• What does it mean?

Overriding equals

• Notion of equality depends on class, YOU need to define this
• Example: compare all fields
  public class Employee
  {
    public boolean equals(Object otherObject)
      // not complete yet
    {
      Employee other = (Employee)otherObject;
      return name.equals(other.name)
        && salary == other.salary;
    }
    ...
  }
• Must cast the Object parameter to subclass
• Can use == for primitive types, equals for object fields
Rules?

• What rules are being violated?

fixing

• Add test for null:
  if (otherObject == null) return false

• What happens if otherObject not an Employee?
• Common error: use of instanceof

```java
if (!(otherObject instanceof Employee)) return false;
```

• Which type of classes is this valid for?

• Violates symmetry: Suppose e, m have same name, salary
  e.equals(m) is true (because m instanceof Employee)
  m.equals(e) is false (because e isn't an instance of Manager)
  • Remedy: Test for class equality
    ```java
    if (getClass() != otherObject.getClass())
        return false;
    ```
Best practice

• Start with these three tests:

```java
public boolean equals(Object otherObject) {
    if (this == otherObject) return true;
    if (otherObject == null) return false;
    if (getClass() != otherObject.getClass())
        return false;
    ...
}
```

• First test is an optimization

Equals in subclass

• Call equals on superclass

```java
public class Manager {
    public boolean equals(Object otherObject) {
        Manager other = (Manager)otherObject;
        return super.equals(other) &&
        department.equals(other.department);
    }
}
```
Not always straight forward

- **Two sets** are equal if they have the same elements in **some** order

```java
public boolean equals(Object o) {
    if (o == this) return true;
    if (!(o instanceof Set)) return false;
    Collection c = (Collection) o;
    if (c.size() != size()) return false;
    return containsAll(c);
}
```

Hashing

- Goal want to quickly locate elements

- Need to use `.equals()` method to know if I’ve located what I’m looking for
Next time

• Read 7.4-7.7

• Check for homework 3 tomorrow