CS1007: Object Oriented Design and Programming in Java

Lecture #14
Nov 15
Shlomo Hershkop
shlomo@cs.columbia.edu

Outline

- Java implementation of Objects
- Types
- wrappers
- Testing types
- Object class
- Hashes
- Copy
- Reading 7-7.4
  - next time 7.4-7.8

Mistake

- I meant to skip chap 6, and do 7-7.4 last class.
- Oops, still need to know 6-6.4
- Will cover 7-7.4 today.

Announcement

- Homework 3 was released
- Due Nov 27 midnight
- Open ended….please adopt it to your needs
  - Need to document design
  - Need to fulfill basic requirements of assignment
Types

- A set of values and operations with those values.

Strongly typed language

- Strongly typed language: compiler and run-time system check that no operation can execute that violates type system rules
- Compile-time check
  Employee e = new Employee();
  e.clear(); // ERROR
- Run-time check:
  e = null;
  e.setSalary(200); // ERROR

Java view of Types

- Primitive types:
  int short long byte
  char float double boolean
- 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
Subtypes

- S is a subtype of T if:
  - S and T are the same type
  - S and T are both class types, and T is a direct or indirect superclass of S
  - S is a class type, T is an interface type, and S or one of its superclasses implements T
  - S and T are both interface types, and T is a direct or indirect superinterface of S
  - S and T are both array types, and the component type of S is a subtype of the component type of T
  - S is not a primitive type and T is the type Object
  - S is an array type and T is Cloneable or Serializable
  - S is the null type and T is not a primitive type

Examples

- Container is a subtype of Component
- JButton is a subtype of Component
- FlowLayout is a subtype of LayoutManager
- ListIterator is a subtype of Iterator
- Rectangle[] is a subtype of Shape[]
- int[] is a subtype of Object
- int is not a subtype of long
- long is not a subtype of int
- int[] is not a subtype of Object[]

Exception!

- Rectangle[] is a subtype of Shape[]
- Can assign Rectangle[] value to Shape[] variable:
  Rectangle[] r = new Rectangle[10];
  Shape[] s = r;
- Both r and s are references to the same array
- That array holds rectangles
- The assignment
  s[0] = new Polygon();
  compiles
- Throws an ArrayStoreException at runtime
- Each array remembers its component type
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

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

1.5

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

- Finite set of values
- Example: enum Size { SMALL, MEDIUM, LARGE }
- Typical use:
  Size imageSize = Size.MEDIUM;
  if (imageSize == Size.SMALL) . . .
- Safer than integer constants
  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
  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();
  }
- enum types are classes; can add methods, fields, constructors
- Enum API

Object testing

- Object O = ????

  How do we figure out what we are dealing with?

Type Inquiry

- Test whether e is a Shape:
  if (e instanceof Shape) . . .
- Common before casts:
  Shape s = (Shape) e;
- Don't know exact type of e
- Could be any class implementing Shape
- If e is null, test returns false (no exception)
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.forName method yields Class object:
  Class c = Class.forName("java.awt.Rectangle");
- .class suffix yields Class object:
  Class c = Rectangle.class; // java.awt prefix not needed
- Class is a misnomer, since also works for primitives
  int.class
  void.class
  Shape.class

An Employee Object vs. the Employee.class Object

Checking Type

- Test whether e is a Rectangle:
  if (e.getClass() == Rectangle.class) ... 
- Ok to use ==
- 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.awt.Rectangle[x=5,y=10,width=20,height=30]
- toString used by concatenation operator
  - aString + anObject
  - toString(obj).toString()
- Object.toString prints class name and object address
  - 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
- == tests for equal location
  - i.e. is it the same object (for classes)
  - Different than comparing two primitives
- Used in many standard library methods
- Example: ArrayList.indexOf
- Unique to your class implementation

Overriding equals

- Notion of equality depends on class
- Common definition: compare all fields
  public class Employee
  {
      public boolean equals(Object otherObject)
          // not complete--see below
      {
          Employee other = (Employee)otherObject;
          return name.equals(other.name)
                 && salary == other.salary;
      }
  }
  ...
  - Must cast the Object parameter to subclass
  - Use == for primitive types, equals for object fields

Equals in subclass

- Call equals on superclass
  public class Manager
  {
      public boolean equals(Object otherObject)
          {
              Manager other = (Manager)otherObject;
              return super.equals(other)
                     && department.equals(other.department);
          }
  }
Not so easy

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

```java
general 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);
}
```

Object.equals

- Object.equals tests for identity:

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

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

Requirements

- reflexive: x.equals(x)
- symmetric: x.equals(y) if and only if y.equals(x)
- transitive: if x.equals(y) and y.equals(z), then x.equals(z)
- x.equals(null) must return false

Employee.equals

- What does it mean?
simple

• Check for same name and salary?

• Check for id?

fixing

• Violates two rules
• Add test for null:
  if (otherObject == null) return false
• What happens if otherObject not an Employee
• Should return false (because of symmetry)
• Common error: use of instanceof
  if (t(otherObject instanceof Employee)) return false;
  // don’t do this for non-final classes
• 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
  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

Hashing
Hashing Components

- Hash table
- Hash function
- Collision
- Load

Hashing

- hashCode method used in HashMap, HashSet
- Computes an int from an object
- Example: hash code of String
  ```java
  int h = 0;
  for (int i = 0; i < s.length(); i++)
      h = 31 * h + s.charAt(i);
  ```
- Hash code of "eat" is 100184
- Hash code of "tea" is 114704

Hashing

- Must be compatible with equals:
  ```java
  if x.equals(y), then x.hashCode() == y.hashCode()
  ```
- Object.hashCode hashes memory address
- NOT compatible with redefined equals
- Remedy: Hash all fields and combine codes:
  ```java
  public class Employee
  {
      public int hashCode()
      {
          return name.hashCode() + new Double(salary).hashCode();
      }
      ...
  }
  ```

Shallow vs. Deep Copy

- Assignment (copy = e) makes shallow copy
- Clone to make deep copy
- Employee cloned = (Employee)e.clone();
Cloning

- `Object.clone` makes a new object and copies all fields.
- Cloning is subtle.
- `Object.clone` is protected.
- Subclass must redefine `clone` to be public.

```java
public class Employee
{
  public Object clone()
  {
    return super.clone(); // not complete
  }
  ...
}
```

Cloneable Interface

- `Object.clone` is nervous about cloning.
- Will only clone objects that implement Cloneable interface.

```java
public interface Cloneable
{
  ...
}
```

- Interface has no methods!
- Tagging interface—used in test
  if (x implements Cloneable
  - `Object.clone` throws `CloneNotSupportedException`
  - A checked exception.

```java
public class Employee
implements Cloneable
{
  public Object clone()
  {
    try
    {
      return super.clone();
    }
    catch(CloneNotSupportedException e)
    {
      return null; // won't happen
    }
  }
  ...
}
```
Deep cloning

- Why doesn't clone make a deep copy?
- Wouldn't work for cyclic data structures
- Not a problem for immutable fields
- You must clone mutable fields

```java
public class Employee implements Cloneable {
    public Object clone() {
        try {
            Employee cloned = (Employee)super.clone();
            cloned.hireDate = (Date)hireDate.clone();
            return cloned;
        }
        catch(CloneNotSupportedException e) {
            return null; // won't happen
        }
    }
    ...
}
```

Cloning and Inheritance

- Object.clone is paranoid
  - clone is protected
  - clone only clones Cloneable objects
  - clone throws checked exception
- You don't have that luxury
- Manager.clone must be defined if Manager adds mutable fields
- Rule of thumb: if you extend a class that defines clone, redefine clone
- Lesson to learn: Tagging interfaces are inherited. Use them only to tag properties that inherit
Next Time

• Continue reading

• Start homework