CS1007: Object Oriented Design and Programming in Java

Lecture #16

Mar 21

Shlomo Hershkop
shlomo@cs.columbia.edu

Announcements

- Will return midterms today
- Please make sure you understand the questions/answers, if not stop by OH

Outline

- Serializable
- Java implementation of Objects
- Types
- wrappers
- Testing types
- Object class
- Hashes
- Copy
- Covering chapter 6, 7-7.2
  - next time 7 - 7.5

- You've probably all heard of
- Web cache
- Memory cache
- Level 2 cpu cache

- What is a cache??

idea

- The general idea of the cache is to store frequently accessed data for rapid access

One application

- Say you have a file describing a group of people with various features
- You create a java object, and then create a custom horoscope……
- How to make it faster?

Idea:

- Allow the programmer to take a snapshot of live memory, and save it in a binary form…..
- No need to recreate classes

1. We need to tell java we want to save a certain class
2. Save the class

Example

```
public class student {
        private String name;
        private int age;

   …
        public String getName(){
                return name;
        }
   }
```

java.io.Serializable

```
public class student implements Serializable
   {
        private String name;
        private int age;

   …
        public String getName(){
                return name;
        }
   }
```

Save routine

```
public static void main(String args[]) {
    Student one = new Student….

    try{

    FileOutputStream fos = new FileOutputStream(“saved.data”);

    ObjectOutputStream out = new ObjectOutputStream(fos);

    out.writeObject(one);

    out.close;

    }catch(IOException ioe){ ..  }
```

Load Routine

```
try{

    FileInputStream fis = new FileInputStream(“saved.data”);

    ObjectInputStream in = new ObjectInputStream(fis);

    Student oldone = (Student)in.readObject();

    }catch(IOException ioe) {…}
```

Important note

- Only objects which extend serializable can be saved
- SO:
  - If your class has field variables which don't implement this….

Two options

1. Mark those non serializable as 'transient' this tells the jvm not to save those variables

2. Implement a custom `writeObject` and `readObject`

   can then choose which fields to save and load, and initialize any others

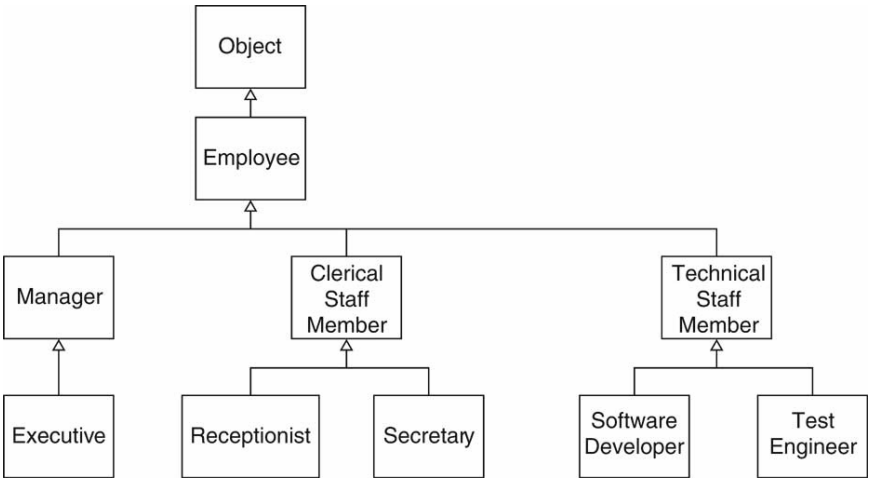
How to add read/write

```
private void writeObject(java.io.ObjectOutputStream
  out)      throws IOException{
  out.writeObject(name);
  out.writeInt(age);

}
 private void readObject(java.io.ObjectInputStream
  in) throws IOException, ClassNotFoundException{

  name = (String)in.readObject();
  age = in.readInt();
}
```

Inheritance

- Use the ‘extends’ keyword
- Allows you to reuse objects

- Some issues:
  - When do you extend?
  - When do you implement an interface?




```
public class Employee
{
   public Employee(String aName) { name = aName; }
   public void setSalary(double aSalary)
       { salary = aSalary; }
   public String getName() { return name; }
   public double getSalary() { return salary; }

   private String name;
   private double salary;
}
```

How do we specialize the class?

- Manager class adds new method: setBonus
- Manager class *overrides* existing method: getSalary
- Adds salary and bonus

Overriding methods

- methods setSalary, getname (inherited from Employee)
- method getSalary (overridden in Manager)
- method setBonus (defined in Manager)
- fields name and salary (defined in Employee)
- field bonus (defined in Manager)

- Why is Manager a subclass?
- Isn't a Manager superior?
- Doesn't a Manager object have more fields?
- The set of managers is a subset of the set of employees

Inheritance Hierarchies

- Real world: Hierarchies describe general/specific relationships
  - General concept at root of tree
  - More specific concepts are children
- Programming: Inheritance hierarchy
  - General superclass at root of tree
  - More specific subclasses are children

Substitution Principle

- Formulated by Barbara Liskov
- You can use a subclass object whenever a superclass object is expected

Example:

Employee e;

...

System.out.println("salary=" + e.getSalary());

- Can set e to Manager reference
- Polymorphism: Correct getSalary method is invoked

Dealing with superclass

- Can't access private fields of superclass

```
public class Manager extends Employee
{
   public double getSalary()
   {
      return salary + bonus;  // ERROR--private field
   }
   ...
}
```

- Be careful when calling superclass method

```
  public double getSalary()
   {
      return getSalary() + bonus;  // ERROR--recursive
   call
   }
```

super

- Use super keyword

```
public double getSalary()
 {
    return super.getSalary() + bonus;
 }
```

- super is not a reference
- super turns off polymorphic call mechanism

Super constructors

- Use super keyword in subclass constructor:

```
public Manager(String aName)
{
 super(aName); // calls superclass constructor
 bonus = 0;
}
```

- Call to super must be first statement in subclass constructor
- If subclass constructor doesn't call super, superclass must have constructor without parameters

Dealing with preconditions

- Precondition of redefined method at most as strong

```
public class Employee
{
   /**
      Sets the employee salary to a given value.
      @param aSalary the new salary
      @precondition aSalary > 0
   */
   public void setSalary(double aSalary) { ... }
}
```

- Can we redefine Manager.setSalary with precondition `salary > 100000`?
- No--Could be defeated:

```
Manager m = new Manager();
Employee e = m;
e.setSalary(50000);
```

Post conditions

- Postcondition of redefined method at least as strong
- Example: Employee.setSalary promises not to decrease salary
- Then Manager.setSalary must fulfill postcondition
- Redefined method cannot be more private. (Common error: omit public when redefining)
- Redefined method cannot throw more checked exceptions

Abstract classes

- Can create a class which is abstract i.e. can not be instantiated
- Can define abstract methods

- So when would you use Interface vs abstract class??

Abstract class

- You can define methods/variables in the abstract class
- Will be available to anyone extending the base abstract class
- No need to recode common methods

Re-use

- Most of the reuse with graphical programming will be through interface implementations
- Example: dealing with mouse actions

Mouse listeners

- Attach mouse listener to component
- Can listen to mouse events (clicks) or mouse motion events
- Anyone know how?

Interface!

```
public interface MouseListener
{
   void mouseClicked(MouseEvent event);
   void mousePressed(MouseEvent event);
   void mouseReleased(MouseEvent event);
   void mouseEntered(MouseEvent event);
   void mouseExited(MouseEvent event);
}

public interface MouseMotionListener
{
   void mouseMoved(MouseEvent event);
   void mouseDragged(MouseEvent event);
}
```

- Includes a lot
- What if you just want part of it?

Extend MouseAdapter

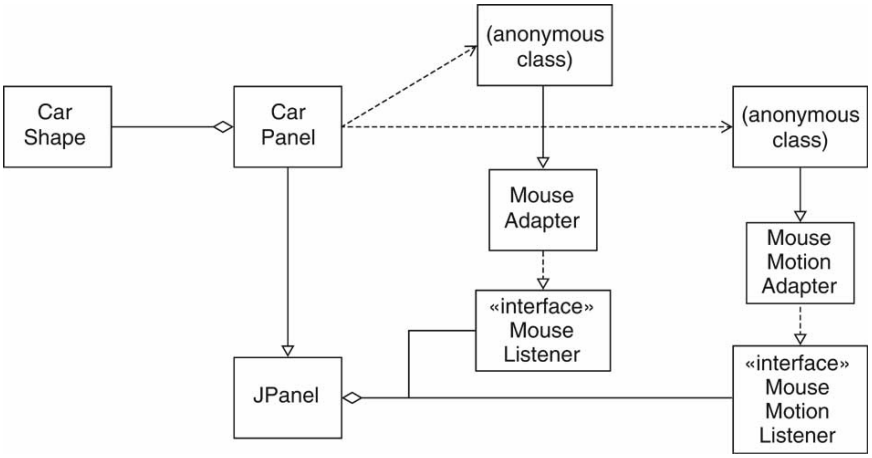
```
public class MouseAdapter implements MouseListener
{
   public void mouseClicked(MouseEvent event) {}
   public void mousePressed(MouseEvent event) {}
   public void mouseReleased(MouseEvent event) {}
   public void mouseEntered(MouseEvent event) {}
   public void mouseExited(MouseEvent event) {}
}
```

usage

```
addMouseListener(new
  MouseAdapter()
  {
    public void mousePressed(MouseEvent event)
    {
      mouse action goes here
    }
  });
```

Example: Car Mover Program

- Ch6/car/CarComponent.java
- Ch6/car/CarMover.java
- Ch6/car/CarShape.java



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 no such method
```

- 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

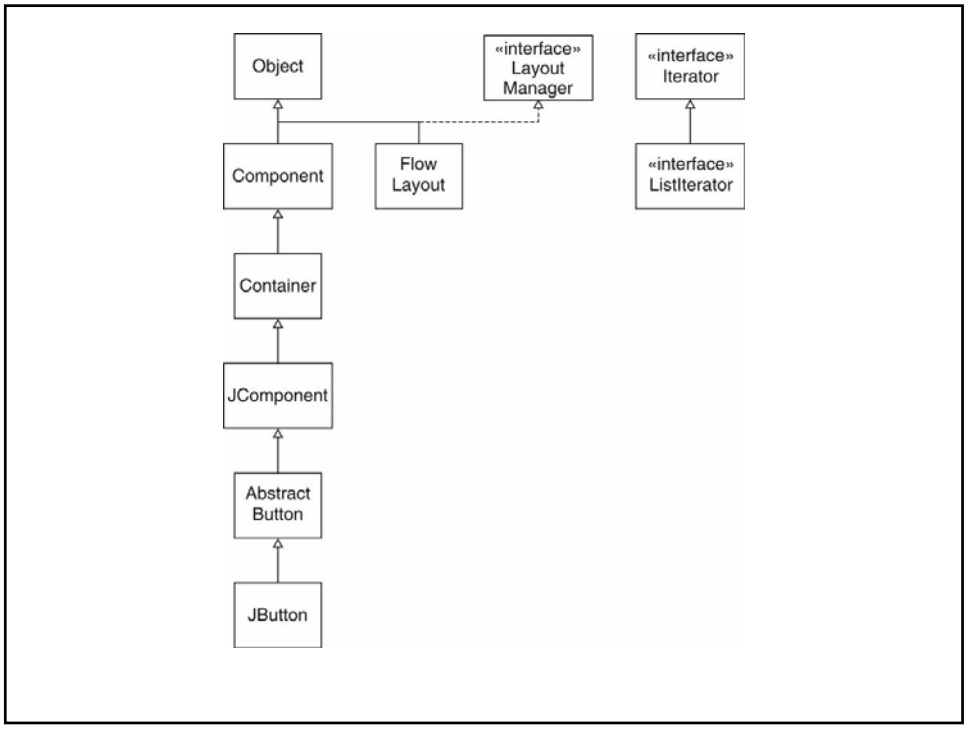
- value of primitive type
- reference to object of class type
- reference to array
- null
- Note: Can't have value of interface type

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[]


Object
«interface»
Layout
Manager
«interface»
Iterator
Component
Flow
Layout
«interface»
ListIterator
Container
JComponent
Abstract
Button
JButton