C10, Mechanisms for Software Reuse

Substitutability

- A variable declared as one type holds a value of another type
- In Java either:
  - Subclass:
    allPiles = new CardPile[13];
    allPiles[0] = new DeckPile(335,30);
  - Interface:
    FireButtonListener implements ActionListener …
    fire.addActionListener(new FireButtonListener);

Is-a rule, Has-a rule

- Is-a holds when the first concept is a specialization of the second concept:
  - a Florist is-a ShopKeeper
  - Rule: an X is-a Y "sounds correct"
- Has-a holds when the second concept is a component of the first concept:
  - a Car has-a Engine
  - Rule: an X has-a Y "sounds correct"
- Sometimes the distinction is not clear/easy

Inheritance of code, of behaviour

- Is-a achieved in two ways in Java:
  - If behavior (actual code) and/or data values can meaningfully be inherited -> use subclasses (CannonGame extends Frame)
  - If the specification of behaviors (method headers or "signatures") makes sense -> use interfaces (FireButtonListener implements ActionListener)

Composition and Inheritance

- Sometimes the distinction is not easy.
- Suppose we want to implement a Stack using class Vector (which supplies almost all necessary behavior anyway):

```
class Vector {
    public boolean isEmpty() {…}
    public int size() {…}
    public void addElement(Object value) {…}
    public Object lastElement() {…}
    public Object removeElementAt(int index) {…}
    …}
```

Using composition

```
class Stack {
    private Vector data;
    public Stack() { data = new Vector();}
    public boolean empty() {return data.isEmpty();}
    public push(Object item) {data.addElement(item); return
        item;}
    public Object peek() {return data.lastElement();}
    public Object pop() {
        Object result = data.lastElement();  // or: peek();
        data.removeElementAt(data.size()-1);
        return result;
```

- Clean code, e.g. Vector x = new Stack(); not possible!

Using Inheritance

```
class Stack extends Vector {
      public Object push(Object item) {
            addElement(item); return item;}
      public Object peek() {
            return elementAt(size()-1);}
      public Object pop() {
            Object obj = peek();
            removeElementAt(size()-1);
            return obj;
      }
}
```

Simpler code, but unwanted stuff possible:

```
Vector x = new Stack();
x.removeElementAt(17);
```

Compostion vs. Inheritance

- Not substitutable
- Simple, makes all methods explicit
- Easy to change underlying data-structure
- Can only access public stuff of the data-structure
- Longer code, but everything specified locally
- One additional indirection, (but JIT-inlining)

- Substitutable
- Allows for all, even inappropriate, parent class methods
- More tricky, especially if uses depend on parent methods
- Can access public and protected parent class stuff
- Shorter code, but must fully understand parent class, too
- Slight speed advantage, one indirection less

Combining Composition and Inheritance

- Subclasses of InputStream: ByteArrayInputStream, FileInputStream, …
- But:

```
class FilterInputStream extends InputStream {
      protected InputStream is;
      …
      }
```

- So InputStream is both a parent and a component of FilterInputStream, used to add functionality "around" some InputStream, called a "filter" or "wrapper"

Dynamic composition

- Inheritance: link fixed at compile-time
- Composition: more flexible, can be changed during runtime, especially useful to model internal "state change":

```
class Frog {
      private FrogBehavior b;
      public Frog() { b = new TadPoleBehavior();}
      public grow() {
            if (b.growUp()) b = new AdultFrogBehavior();
            b.grow(); b.swim(); }}
```

Inheritance of inner classes

- E.g. listener classes were constructed using inheritance, but are themselves components in the application class:

```
public class PinBallGame extends Frame {
      …
      private class MouseKeeper extends MouseAdapter { …}
      private class PinBallThread extends Thread {…}
}
```

Unnamed classes

- Syntactic suger as a shortcut for cases where we create only one instance of a class, and the class is relatively simple, e.g. FireButtonListener of CannonGame:

```
class CannonWorld extends Frame {
      …
      private class FireButtonListener implements ActionListener {
            public void actionPerformed(ActionEvent e) {…}
      }
      public CannonWorld() {
            fire.addActionListener(new FireButtonListener());
            …
            }
      }
```

"Unnamed" FireButtonListener

```
class CannonWorld extends Frame {
    …
    public CannonWorld() {
        fire.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {…}
            });
        …
    }
}
```

Summary

- Two common techniques for reuse:
  - Inheritance: "is-a"
  - Composition: "has-a"
- Distinction not always easy
- "Filters" (as found in the IO library) combine inheritance and composition