Lecture 29

We look at further examples of polymorphism. In particular, we consider *generic data structures*. These are data structures that are so general that *any* class of object can be placed into them. For this, we use the class `Object`.

However, the power we gain has a price. We have to be quite disciplined in the way we use these data structures. And, due to Java's strong-typing, taking things out of the data structure is more complicated. (This is where we see why Java is only *mostly* statically-typed.)

Polymorphic data structures, again

- For real generality, we can define data structures that contain objects of class `Object`
  - sometimes called *generic data structures*
  - can put *any* kind of object into them, including mixtures
- E.g. an array of `Object`s:

```
Object[] a = new Object[3];
a[0] = new BankAccount(...);
a[1] = new BankAccount(...);
a[2] = new Cow();
```

- Of course, we can't put values of type `int` and the other primitive types into these data structures because they are not objects. This is why we have wrapper classes (see earlier lecture)

`Vector`: another generic data structure

- Arrays are fixed-length
- The Java library provides `Vector`s: array-like data structures but generic (i.e. hold `Object`s) and variable-length (capacity can grow as needed)
- See the on-line documentation for `java.util.Vector` for the full interface. Some of the methods:

```
public int size()
public boolean isEmpty()
public void elementAt(int index)
public Object lastElement()
public void remove(int index)
public void add(Object obj)
```

- `Vector` is much used by Java programmers wherever a list of unpredictable length is needed (despite its inefficiency!)

Observation I: the need for self-discipline

- Suppose we decide to use generic data structure (e.g. a `Vector`) but we only intend storing in it `LibraryBook`s

```
Vector stock = new Vector();
stock.add(new LibraryBook(...));
stock.add(new LibraryBook(...));
```

- We only want to store `LibraryBook`s, but we are allowed to store any `Object`. The fact that we're only intending to store `LibraryBook`s is not known and so cannot be checked for by Java. So nothing would stop us from inserting other kinds of objects, e.g.:

```
stock.add(new Cow());
```

- We have to enforce the restriction ourselves through disciplined programming

Observation II: taking objects out

- Suppose we have been inserting only `LibraryBook`s into a `Vector`
- *We know we only inserted* `LibraryBook`*s so the only objects we can take out of the* `Vector` *are* `LibraryBook`*s, but Java doesn't know this*
- The Java compiler will not be happy with this:

```
LibraryBook b =
   stock.elementAt(0); // error here
b.getNumOfCopiesOnLoan();
```

nor with this:

```
Object b = stock.elementAt(0);
b.getNumOfCopiesOnLoan(); // error here
```


Observation II, cont'd: the need for casting

- To persuade the Java *compiler* that it is OK for us to store the object (which we know is a `LibraryBook`) in the variable `b`, we must *cast*:

```
LibraryBook b =
    (LibraryBook) stock.elementAt(0);
b.getNumOfCopiesOnLoan();
```

Or to persuade it that it is OK to apply one of `LibraryBook`'s methods, we must cast:

```
Object b = stock.elementAt(0);
((LibraryBook) b).getNumOfCopiesOnLoan();
```

- By casting, we are promising that what comes out of the `Vector` will in fact be a `LibraryBook`
- Casting objects when you take them from polymorphic data structures is very common

Observation II, cont'd: run-time type errors

- So now what happens if the `Vector` is not exclusively `LibraryBook`s? (We lied to the compiler)

```
stock.add(new LibraryBook(...));
stock.add(new LibraryBook(...));
stock.add(new Cow());

LibraryBook b =
   (LibraryBook) stock.elementAt(2);
```

- We fooled the compiler
- But we get a run-time error (class cast error). (So, don't lie!)
- This is why Java is only *mostly* statically-typed: some type errors can only be discovered at run-time, when a cast fails

The reusability of generic data structures

- Despite the inconvenience of casting, generic data structures can make us more productive
- E.g. if we write a class definition for a stack of `Object`s, that class can be used in any program where we need stacks.

```
import java.util.Vector;
public class VectorBasedStack
{
   public VectorBasedStack()
   {  v = new Vector();
   }

   public int size()
   {  return v.size();
   }

   public boolean isEmpty()
   {  return v.isEmpty();
   }
```



```
   public void push(Object obj)
   {  v.add(obj);
   }

   public Object top()
   {  if (isEmpty())
      {  // deal with error
      }
      return v.lastElement();
   }

   public Object pop()
   {  if (isEmpty())
      {  // deal with error
      }
      Object elem = v.lastElement();
      v.remove(size() - 1);
      return elem;
   }

   private Vector v;
}
```


Generic data structures in libraries

- Class definitions for generic data structures are now widely available:
  - Java 1 standard library contains class definitions for `Hashtable`s of `Object`s, `Stack`s of `Object`s and `Vector`s of `Object`s
  - Java 2 standard library has even more (e.g. `ArrayList`, `LinkedList`, `TreeMap`, `TreeSet`, `HashMap`, `HashSet` and other hash tables)
  - There are other, better data structure libraries (e.g. the Java Generic Library, JGL, which is available for a nominal fee from `http://www.recursionsw.com`)
- Since we believe in *software reuse*, we should use these rather than coding our own from scratch