Lecture 29

We look at the idea of interfaces again, and relate these to abstract data types.

We then look at *Java interfaces*: a way of explicitly telling Java about an interface. These act as 'specifications'.

This leads to a further example of polymorphism in Java. We can use this to write code that is much more maintainable than it might otherwise be.

Our approach to design: determine the interface

- Before writing a class definition, first determine the class interface
  - the set of services (methods) we offer to clients
  - e.g. `Time` offers constructor, `isEqualsTo`, `isBefore`, `getIntervalInSecs`, `getClockTime` and `getMilitaryTime`
- Similarly, if designing data structures, first determine the interface
  - e.g. stacks support constructor, `push`, `pop`, `size`, `isEmpty` and `top`
  - e.g. queues offer constructor, `enqueue`, `dequeue`, `size`, `isEmpty` and `front`
  - but data structures people sometimes refer to the interface as the *abstract data type*

Data abstraction

If the interface remains unchanged, clients needn't be changed, even if the implementation changes

```
public class Time
{  // various public methods

   // various helper methods

   private int timeInSecs;
}

public class Time
{  // the same public methods
   // but with different bodies

   // different helper methods

   private int hours;
   private int mins;
   private in secs;
}
```


Java interfaces

- We can take this a stage further: record the interface as a *Java interface*
- A Java interface is just a collection of abstract methods (i.e. we give the signatures but not the bodies)

```
public interface Stack
{
   public int size();

   public boolean isEmpty();

   public Object top();

   public void push (Object element);

   public Object pop();
}
```


Java interfaces v. class definitions

- Contrast:
  - a class definition can contain instance/class variables, instance/class methods (comprising both their signatures and bodies, unless abstract)
  - a Java interface can contain only public instance method signatures (and named constants)
- A Java interface acts like a specification
  - it specifies, e.g., what it means to be a stack
  - to be a stack, an object must offer (at least) these methods in its public interface

Implementing an interface

When you write a class definition, you can declare that it 'implements an interface', i.e. it meets the specification:

```
import java.util.Vector;

public class VectorBasedStack
   implements Stack
{
   public VectorBasedStack()
   {  v = new Vector();
   }

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

   // definitions of isEmpty, push, top
   // & pop as per previous lecture

   private Vector v;
}
```

Implementing an interface

```
public class ArrayBasedStack
   implements Stack
{  public ArrayBasedStack(int theCapacity)
   {  capacity = theCapacity;
      s = new Object[capacity];
   }
   public int size()
   {  return (top + 1);
   }

   // definitions of isEmpty, push, top
   // & pop as per KH's course

   private int capacity;
   private Object s[];
   private int top = -1;
}
```

When you declare that your class definition implements an interface, then you are saying that the class definition contains definitions of every method specified in the Java interface. The compiler will check up on you.

More polymorphism & dynamic method binding

- Polymorphism:
  - A variable of a superclass type may refer to objects of that class or its subclasses
  - A variable of a Java interface type may refer to objects from any class that implements the Java interface

    ```
    Stack s;
    s = new VectorBasedStack();
    s = new ArrayBasedStack(10);
    ```

- Dynamic method binding is deciding *at run-time* which of several methods to run, based on the object being referenced:

  ```
  Stack s;
  s = new VectorBasedStack();
  s.push(new Cow());
  ```


Java interfaces: what's the point?

- Using Java interfaces polymorphically gives you client code that is more easily modified.
- Consider how much error-prone effort is involved in changing from using an `ArrayBasedStack` to a `VectorBasedStack` in client code that doesn't use Java interfaces:

```
public void method1()
{  ArrayBasedStack s = new ArrayBasedStack(1
   s.push(new Cow());
   method2(s);
   method3(s);
}
public void method2(ArrayBasedStack st)
{  System.out.println(st.top());
}
public void method3(ArrayBasedStack st)
{  st.push(new Pig());
}
```


Java interfaces: what's the point? cont'd

- See how easy it is to change from using an `ArrayBasedStack` to using a `VectorBasedStack` in client code that does use Java interfaces:

```
public void method1()
{  Stack s = new ArrayBasedStack(10);
   s.push(new Cow());
   method2(s);
   method3(s);
}
public void method2(Stack st)
{  System.out.println(st.top());
}
public void method3(Stack st)
{  st.push(new Pig());
}
```

(Considering how often we want to change the data structures a program uses, this is a significant increase in program maintainability.)