Subclasses and Interfaces

Subclassing:

- Inheritance of code
- Polymorphism

Inferfaces:

- Polymorphism

Abstract Classes

Sometimes it does not make sense ever instanciate a class, E.g. `UWPerson`.

Sometimes we want to specify part of an implementation but not all of it.

Classes declared `abstract` can not be instantiated.

Abstract classes may declare abstract methods that non-abstract subclasses must implement.

InputStream

```
public abstract class InputStream
{
   // simple default implementations.
   public int available() { return 0; }
   public void close() {}
   public boolean markSupported() { return false; }
   public void reset() {}

   // abstract read() method, overridden by subclass
   // depending on its input source.
   public abstract int read() throws IOExcption;

   // default implementations that use read()
   public int read(byte[] b, int off, int len)
   {
      ...
      int b = read();
      ...
   }
   public int read(byte[] b) throws IOException
      { return read(b, 0, b.length); }
   public long skip(long n)
      { ... }
}
```


StringInputStream

```
public class StringInputStream extends InputStream
{
   private byte[] data;
   private int length;
   private int cursor;

   public StringInputStream(String in)
   {
      data = in.getBytes();
      length = in.length();
      cursor=0;
   }

   public int read()
   {
      if (cursor >= length)
        return -1;

      return (int)data[cursor++];
   }
}
```

Abstract Classes – Summary

Abstract `InputStream` class made `StringInputStream` easy to create.

`StringInputStream` only had to implement `int read()` method.

`StringInputStream` inherited `read(byte[])` and `read(byte[], int, int)`.

Inherited `read(byte[],int,int)` used `StringInputStream`'s `int read()` method.

Generic Collections

All classes are subclasses of `Object`.

To make a collection of any type use `Object` as type.

E.g. `java.util.Vector` and `java.util.Hashtable`.

It is easy to insert elements:

```
Vector v = new Vector();
v.add(new Bird());
```

But, cast is required to remove elements:

```
Bird b = (Bird)v.elementAt(0);
```

Java does not provide generic support like C++'s templates (only arrays).

Primitive Wrappers

Any `Object` can be put in a `Vector`.

Java primitive `int`s are not `Objects`.

Can not have a `Vector` of `int`.

Can wrap `int` in `Integer` class:

```
Vector v = new Vector();
v.add(new Integer(5));

int i = ((Integer)(v.lastElement())).intValue();
```

Wrapper classes also have useful class functions and class variables: E.g. `Integer.parseInt(String)` and `Integer.MAX_VALUE`.

ReverseFile

```
public class ReverseFile
{
   public static void main(String[] args)
   {
      try
      {
         BufferedReader in
            = new BufferedReader(new FileReader(args[0]));
         Stack s = new Stack();

         String line;
         while ((line = in.readLine()) != null)
            s.push(line);

         while (!s.empty())
            System.out.println((String)s.pop());
      }
      catch (Exception e)
      {
         e.printStackTrace(System.out);
      }
   }
}
```

Exception Handling

Can use exceptions instead of having functions return some error status.

Constructors can throw exceptions if they fail (no return value possible).

Can seperate error handling code from general case code.

For `ReverseFile` make things could go wrong:

- `args` could be empty,
- `args[0]` might not contain a file name, or
- an IO error could happen reading the file.

`ConsecutiveIntegers` Class

```
public class ConsecutiveIntegers1
{
   int min, max;
   public ConsecutiveIntegers1(int minIn, int maxIn)
      { min = minIn; max = maxIn; }

   public int size() { return(max - min + 1); }

   public int intAt(int i)
                 throws ArrayIndexOutOfBoundsException
   {
      if (i < 0 || i >= size())
         throw new ArrayIndexOutOfBoundsException();

      return min+i;
   }

   public Object elementAt(int i)
                 throws ArrayIndexOutOfBoundsException
      { return new Integer(intAt(i)); }

   public Enumeration elements()
      { return new IntegerRange(min,max); }
}
```


Inner Classes

Inner classes are private classes nexted inside other classes.

Inner classes have “outer” class’s instance variables and methods in their scope.

Inner classes have access to private instance variables.

Reconsider `ConsecutiveIntegers.elements()`

`ConsecutiveIntegers.elements()`

```
public class ConsecutiveIntegers2
{
   int min, max;

   ...

   private class OurEnumeration implements Enumeration
   {
      int current = 0;
      public boolean hasMoreElements()
      {
         return (current < size());
      }

      public Object nextElement()
      {
         return elementAt(current);
      }
   }

   public Enumeration elements()
      { return new OurEnumeration(); }
}
```

Anonymous Classes

Inner classes can be anonymous extending either an object or an interface:

```
public class ConsecutiveIntegers3
{
   int min, max;
   ...
   public Enumeration elements()
   {
      return(new Enumeration() {
         int current = 0;
         public boolean hasMoreElements()
         {
            return (current < size());
         }

         public Object nextElement()
         {
            return elementAt(current);
         }
      });
   }
}
```


Reflection

Java reflection is the ability to dynamically refer to the Java class system inside a Java program.

A language construct is first class if it can be stored in variables, passed to functions, and returned from functions.

Java reflection via `java.lang.Class` and `java.lang.reflect.*`
(We will discuss other OO metaclass systems later)

Two dynamic ways to get a `Class` object:

```
Object someObject = "hello";
String className = "java.lang.String";

Class c1 = Class.forName(className);
Class c2 = someObject.getClass();
```

One static way to get a `Class` object:

```
Class c3 = String.class;
```


Using a `Class`

Some useful functions are:

`Static Class forName(String name)` Returns `Class` object for `name`.

`Class[] getInterfaces()` Returns array of interfaces implemented by this class.

`String getName()` Returns complete class name:
E.g. `String.class.getName()` returns `"java.lang.String"`.

`boolean isInstance(Object o)` Returns `true` if `o` is an instance of this class. A dynamic version of `instanceof`.

`boolean isAssignableFrom(Class c)` Returns `true` if `c` is more specific than this class.
E.g. `Bird.class.isAssignableFrom(Owl.class)` is `true`.

`Object newInstance()` Creates a new instance of this class by calling the default constructor.

`java.lang.reflect.*`

java.lang.reflect.Method Provides abstraction for method selection and invocation. Can get `Method` objects.

java.lang.reflect.Field Provides abstration for field setting and getting.

java.lang.reflect.Constructor Provides abstraction for invoking constructors (especially non-default constructors).

Advantages of Reflection

Applications that can talk about themselves are highly customizable. Consider:

```
public abstract class Filter
{
   public abstract String filterString(String in);
}
```


FilterMain

```
public abstract class FilterMain
{
   public static void main(String[] args)
   {
      try
      {
         Filter f = (Filter)Class.forName(args[0]).newInstance();
         BufferedReader in
            = new BufferedReader(new InputStreamReader(System.in));

         String line;
         while((line = in.readLine()) != null)
         {
            System.out.print(f.filterString(line));
         }
         System.out.print(f.filterString(null));
      }
      catch (Exception e)
      {
         e.printStackTrace(System.err);
      }
   }
}
```


FirstTenFilter

```
public class FirstTenFilter extends Filter
{
   int count = 0;
   public String filterString(String line)
   {
      if (count++ < 10)
         return line + "\n";
      return "";
   }
}
```

Running `java FilterMain FirstTenFilter` is the same as running `head -10` from the UNIX prompt.

WordCountFilter

```
public class WordCountFilter extends Filter
{
   int numWords = 0, numLines = 0, numChars = 0;

   public String filterString(String line)
   {
      if (line != null)
      {
         numLines++;
         numChars += line.length() + 1;
         numWords += (new StringTokenizer(line)).countTokens();
         return "";
      }
      else
         return("\t" + numLines + "\t" + numWords
                     + "\t" + numChars + "\n");
   }
}
```

Threads

A modern operating system runs many processes (running programs) at once.

Processes can communicate through IPC (interprocess communication).

These "heavy weight" processes are protected from each other by the operating system.

Often times it is useful to write programs as if they were multipule processes. But protection of operating system is not needed and overhead is too much.

Threads are light weight processes.

A process can have many running threads. Theads share the processes memory. Each thread has its own call stack.

Theads vs State-Machines

Consider animation of unfriendly creatures in Doom-like game.

state-machine: At each clock tick animate one step of creature movement. Must have kept state to determine where creature currently is. Then must generate next movement.

```
Creature[] creatures = ...;

while(alive())
{
   for(int i=0; i < creatures.length; i++)
      creature.animate();
}
```

This works, but gets complicated very quickly as creatures need more diverse sequences of moves.

Threads vs State-Machines (cont)

threads: Each creature is run by its own thread. At each clock tick creature thread gets awakened. Creature thead can do whatever it wants. Creature thread puts itself to sleep when it has finished its turn. Creature's state is stored on the creature thread's call stack.

```
Creature[] creatures = ...;
for(int i=0; i < creatures.length i++)
   creatures.start();  // start creature threads.

while(alive()) {
   nofityAll();  // this notifies all threads waiting
}                // on 'this' object.
```

This assumes that each creature puts itself to sleep by waiting on `this` object.

Threads: Synchronization

When there are multipule threads in the system running at once, sequentially written code might not work as it originally did.

Threads can trample on each other. Given a code block:

```
x = x + 1;
```

Say $x = 10$ and two threads run this block at the same time.

x	Thread 1	Thread 2
10	`x+1` $\Rightarrow$ 11	
10		`x+1` $\Rightarrow$ 11
10		`x = 11`
11		
11	`x = 11`	
11		

The code executed twice and the value of `x` is only 11. It should be 12.

Synchronization (cont)

(learn more about synchronization in Operating Systems)

A method can be synchronized on `this`. A block of code can be synchronized on any object.

The object that a block of code or method is synchronized on holds the lock on that block of code. No other thread can enter a synchronized block that uses the same object as its lock.

```
public syncronized int incX()
{
   x = x + 1;
   return x;
}
```

or

```
synchronized (<expression>)
{
   <code>
}
```


Threads and Methods

Threads of control are an orthogonal issues to classes. Threads make method invocations on objects.

Any thread can invoke any method on any object.

Where to use Threads

Good places to use Threads include:

- GUI Animation.
- Server Applications.
- IO intensive Applications.

Why? Because threads allow you to easily do multiple things roughly at the same time.