ReverseFile1

Write a program to print out lines of file in reverse order:

```
import java.util.Stack; import java.io.*;

public class ReverseFile1
{
   public static void main(String[] args)
   {
      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());
   }
}
```

(this will not compile)

Compiling ReverseFile1

```
fiji> javac ReverseFile1.java
ReverseFile1.java:8: Exception java.io.FileNotFoundException
 must be caught, or it must be declared in the throws clause
 of this method.
         = new BufferedReader(new FileReader(args[0]));
                               ^
ReverseFile1.java:12: Exception java.io.IOException must be
 caught, or it must be declared in the throws clause of this
 method.
      while ((line = in.readLine()) != null)
                                 ^
2 errors
fiji>
```


Exception Handling

Use *exceptions* instead of having functions return error status.

Why exceptions?

- When no return value possible (I.e. constructors).
- To separate error handling from general case code.

Non-`RuntimeExceptions` must be *handled* by caller.

- Caller can declare that exception as thrown as well.
- Caller can register an *exception handler*. Exception handlers are *dynamically scoped*.

What about `RuntimeException`s?

ReverseFile2

```
import java.util.Stack; import java.io.*;

public class ReverseFile2
{
   public static void main(String[] args) throws IOException
   {
      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);
}  }  }
```

ReverseFile3

```
import java.util.Stack; import java.io.*;

public class ReverseFile3
{
   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);
}  }  }
```


What could go wrong?

For `ReverseFile`, many things could go wrong:

- `args` could be empty, (`ArrayIndexOutOfBoundsException`, this is a subclass of `RuntimeException`)
- `args[0]` might not contain a file name (`FileNotFoundException`), or
- An IO error could happen reading the file (Any subclass of `IOException`).

Exceptions (more details)

We can handle different exceptions differently. Rewrite

```
catch (Exception e)
{
   e.printStackTrace(System.out);
}
```

as (`ReverseFile4`):

```
catch (FileNotFoundException e)
{
   System.err.println("File " + args[0] + " not found.");
}
catch (IOException e)
{
   System.err.println("Error reading from file: " + args[0]);
}
catch (ArrayIndexOutOfBoundsException e)
{
   System.err.println("USAGE: java ReverseFile <file-name>");
}
```

Order of `catch` clauses is important!

Inner Classes

//Java allows classes to be nested inside other classes.

Consider `Vector.elements()`. This returns an `Enumeration` that that can be used to loop through elements in the vector. E.g.

```
Vector v = new Vector();
...
printEnumeration(v.elements());
```

What class type is `v.elements()`?

Consider the `ConsecutiveIntegers` cass to represent a set of consecutive integers.

`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); }
}
```


Recall `IntegerRange`

```
public class IntegerRange implements Enumeration
{
   int min; int max;
   public IntegerRange(int minIn, int maxIn)
   {
      min = minIn;
      max = maxIn;
   }
   public boolean hasMoreElements()
   {
       return min <= max;
   }
   public Object nextElement()
   {
       return new Integer(min++);
   }
}
```


Inner Classes

Inner classes are private classes nested inside other classes.

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

Inner classes have access to private instance variables.

Suppose we did not have the `IntegerRange` class when writing `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 Inner Classes

Inner classes can be *anonymous* extending either a class 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 abstraction 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 the following extended example:

- Want system for filtering text files.
- Want to do line by line filters.
- Read from standard input and write to standard output.

```
public abstract class Filter
{
   // - take input line and return output line.
   // - call once with null at EOF for last minute output.
   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");
   }
}
```