**Interfaces**

C++ is among those languages offering multiple inheritance: a class may inherit from more than one direct superclass. This adds flexibility, but it also brings problems.

- A class may inherit two or more methods of a given name along different paths. Which method should take precedence?

Many strategies have been developed for this, but they often produce surprising results.

- Multiple inheritance imposes considerable implementation difficulties on a language, and may make it take longer to call a method.

In view of these difficulties, Java adopted a different approach:

- A class is allowed to inherit methods from more than one direct superclass, but
- all the methods inherited must be *abstract*, with the exception of the methods inherited from a *single* class.

This solves the problems with multiple inheritance:

- Since only a single method is inherited, there can't be any conflict about which method predominates.
- Since the inheritance graph is a tree, implementation is comparatively simple and fast.

The entirely-abstract classes inherited from are called *interfaces*.

Let's take a look at an interface.

```java
interface Drivable {
    boolean startEngine();
}
```
void stopEngine();
float accelerate(float acc);
boolean turn(Direction dir);
}

The keyword interface replaces the word class in the class definition, and all methods must be abstract.

Note that these methods, being abstract, have no body defined here. Instead, they end with a semicolon.

They could be defined to be abstract, but that’s not necessary.

Since interfaces name capabilities, it’s often a good idea to name them after those abilities.

Any class that implements all the methods can then declare that it implements the interface.

class Automobile implements Drivable {
    ...
    boolean startEngine() {
        if (notTooCold) engineRunning = true;
        ...
    }

    void stopEngine() {
        engineRunning = false;
    }

    etc.
}

Other classes, like lawn mower, could also implement Drivable.

An interface can also serve as a type specifier. You can write procedures that take objects of the interface type as parameters:

    Drivable myVehicle;
But the main purpose of an interface is to impose requirements on the class(es) that want to implement it.

Example: The java.util.Enumeration interface.
This interface can be used by any kind of set to provide serial access to its elements.

An object that implements the Enumeration interface must provide two methods,

- `nextElement()`, which returns an Object type so it can be used with any kind of collection, and
- `hasMoreElements()`.

Any kind of object can implement the Enumeration interface and then use it to provide access to its elements.

This means we can write general looping code, to which we later plug in the appropriate object types.

```java
Enumeration e = ...  
while (e.hasMoreElement()) {  
    String s = (String) e.nextElement();  
    System.out.println(s);  
}
```

**Modularization of Java programs**

In Visualworks\Smalltalk, related classes are combined into the same category.

This is the only structuring feature available in Smalltalk.

Java has two different mechanisms for grouping classes.

- Multiple classes may be defined in the same file.
  In this case, however, only one of these classes may be public—the one whose name matches the name of the file.
- Classes from multiple files may be grouped into a `package`.
  In this case, an identical `package` statement is placed at the beginning of each file.

```java
package csc517.project.gui;  
public class Window {  
    ...  }
```
package csc517.project.gui;
public class DialogBox extends Window {
    ...
}

Package names.

- The components of package names always correspond to the components at the end of the pathname where they are stored.
- The beginning of the pathname is taken from the CLASSPATH environment variable.

So, the full pathname could be—

/ncsu/efg/csc517/project/gui/Window.java

Protection of variables and methods: So far, we have seen three of Java’s visibility levels, public, ________, and ________.

Java has another visibility level:

- default, i.e., declared without a visibility specifier
  
  int i;

Here is a table that summarizes the visibility:

<table>
<thead>
<tr>
<th>Situation</th>
<th>public</th>
<th>protected</th>
<th>Default</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherited by subclass in same package?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Inherited by subclass in different package?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Accessible to other classes in same package?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Accessible to classes in different packages?</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Note that it is possible to have public access methods, but private instance variables.

This allows the implementor to change the implementation of the class without affecting clients of the class, as long as the signature of the methods doesn’t change.

Also note the difference between “inherited” and “accessible.” Subclasses inherit protected fields, but they can’t access them in instances of the superclass.

The same applies to protected fields when the subclass is not in the same package as the superclass.

File streams

A stream is a sequence of data.

- The stream that flows from your keyboard to your program is the standard input stream.
- The stream that flows from your program to the display is the standard output stream.

To read data, you create an input stream that connects an input file to your program.

In order to use input (or output) streams, you need to inform Java that you want to use classes from the input-output package:

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
```

or simply

```java
import java.io.*;
```

Then, you create an instance of FileInputStream, and assign it to a particular file.

```java
FileInputStream inputFile =
new FileInputStream("/ncsu/efg/input.data");
```
We can use `inputFile.read()` to read a single byte. However, it is usually more convenient to read complete numbers and strings.

To do this, one converts a `FileInputStream` into an instance of `StreamTokenizer`.

```java
StreamTokenizer tokens =
   new StreamTokenizer(inputFile);
```

Tokenizers treat whitespace as delimiters. Thus, a file containing the following characters is viewed as a stream of seven numbers:

```
1.0 2 3.1415926536 -42
5.15e6 678 0
```

The `nextToken` method moves the tokenizer from one token to the next.

Each time it moves to a token, it assigns the token it moves past to its instance variable `nval`.

What are the values that `nval` takes on on successive calls to `nextToken`?

The `nval` variable is of type `double`, so if you want to change it to any other value, you need to cast it:

```java
(int) tokens.nval
```

When the tokenizer reaches the end of the stream, `nextToken` returns a special value, which is the same as the value in the tokenizer’s `TT_EOF` instance variable.

```
Thus, to test for end-of-file, you can use
while (tokens.nextToken() != TT_EOF) {
   ...
   i = (int) tokens.nval ...
   ...
}
```

Before the end of the file is reached, `nextToken` returns

- `TT_NUMBER` whenever it reads a number (and it assigns the value of the number to `nval`), and
• **TT_WORD** whenever it reads a string (and it assigns the value of the string to **sval**).

Output streams are created in a similar fashion:

```java
FileOutputStream outputFile =
   new FileOutputStream("output.data");
```

You can write a byte at a time to the output stream with the **write(byte[] b)** method. However, it is usually more convenient to use an instance of **PrintStream**.

```java
PrintStream output =
   new PrintStream(outputFile);
```

You can write to **PrintStreams** using the **print** and **println** methods.

After you have finished reading from or writing to a file, you should close it:

```java
inputFile.close();
outputFile.close();
```

**Characters and strings**

We have seen that tokenizers can be used to read data from input files.

The **sval** instance variable receives a string that is scanned by the tokenizer.

Tokenizers can also be used to read quoted strings and to treat end-of-line as significant.

The length of a string can be determined with the **length()** message:

```java
String s = "Hello, world!";
System.out.println(s.length);
```

Strings are constants. You can’t change a string, although you can concatenate two to produce a longer string.
A single character can be fetched from a string with the `charAt` method.

`s.charAt[0] returns`

Java strings differ from C++ strings.

- In C++, strings are arrays of characters. They are terminated by null characters.
- Java strings are instances of the `String` class. They are not terminated by null characters.
  (Actually, due to bounds-checking, it is impossible to access the character after the end of a string.)

Characters in Java are surrounded by single quotes, e.g., 'E', and are declared like—

```java
char c = 'E';
```

The standard default of a character is the null character, denoted by '\u000'

**Arrays**

In Java, an array is declared by giving the type and the variable name:

```java
int scores[];
```

Note that the declaration doesn’t tell how large the array is. The size is determined when the array is created, with the `new` operator:

```java
scores = new int [classSize];
```

One may combine these two statements:

```java
int scores[] = new int [classSize];
```

The first element in each array has subscript 0. Thus, to create an array of month lengths, where the length of the `i`th month is stored in the `i`th element, we must declare it like this:
static int month_days[] = new int [13];

Instead of explicitly giving the size of the array, one may use an array initializer, which implicitly specifies it:

    static int month_days[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

Array elements are accessed just as in C or C++:

    System.out.println(month_days[2]);

Each array has an instance variable called length:

    month_days.length

It is also possible to declare arrays of class types:

    Circle circles [] = new Circle [4];

One may create a new object to put into an array:

    circles[3] = new Circle(5.5);

And we can access the fields and methods of array elements:

    circles[3].radius
    circles[3].area()

Overwriting an array element can cause an instance to be garbage collected:

    circles[3] = new Circle(1.2);

Now, if the old value of circle[3] hasn’t been stored anywhere else, the circle at 5.5 may be garbage-collected.

IMPLEMENTATION OF ARRAYS

Arrays of primitive types are laid out with—

- 4 bytes for the length instance variable.
- Enough bytes in each element to hold an item of the type (in the case of int, 4 bytes).
Arrays of class types are laid out with—

- 4 bytes for the length instance variable.
- Enough bytes in each element to hold the *address* of the corresponding array element:

![Array diagram]

Because addresses of class instances are stored in arrays, it is possible to put subclass elements in the same array as elements of their parent class.

In essence, arrays contain pointers, but in Java—as in Smalltalk—the programmer never has to manipulate pointers explicitly.

**Multidimensional arrays**

Multidimensional arrays are represented as arrays of arrays:

```java
int twoDArray[][] = new int[256][16];
```

Since multidimensional arrays are allocated as arrays of arrays, they do not necessarily have to be rectangular.

You may create an array with the last several dimensions unspecified, and later create “rows” that have different lengths:

```java
int triangular[][] = new int[4][];
triangular[0] = new int[1];
triangular[1] = new int[2];
triangular[2] = new int[3];
triangular[3] = new int[4];
```

These can also be created with nested initializers:

```java
int triangular[][] = {{0},{0,1},{0,1,2},{0,1,2,3}};
```

When an array reference is out of bounds, the `ArrayIndexOutOfBoundsException` exception is thrown.
**Arrays and methods**

When an array is a parameter or a return value, empty brackets are placed after the type specification:

```java
public Shape[] translate(Shape shapes[]) { ...
}
```

In a called method, it is even possible to *create* an array, and return it from the method:

```java
class CreateDemo {
    public static void main(String args[]) {
        int[] box;
        box = create(4, 2, 17);
        System.out.println(box[2]);
    }

    protected static int[] create(int length, int subscript, int new_value) {
        int[] out_array;
        out_array = new int[length];
        out_array[subscript] = new_value;
        return out_array;
    }
}
```

What does this program print?

Now let’s look at the argument to `procedure main(...)`

It is an array of `Strings`. This array contains the command-line arguments.

This program displays all of the command-line arguments:

```java
public class Demonstrate {
    public static void main(String argv[]) {
        int counter;
        int max = argv.length;
        for (counter = 0; counter < max; ++counter) {
            System.out.println(argv[counter]);
        }
    }
}
```
Collection classes in Java

Java has standard collection classes, which often have something in common with their Smalltalk counterparts. They are found in java.util.*.

Vectors: A vector is similar to a Smalltalk OrderedCollection.

Like arrays, vectors are declared and then created:

    Vector v = new Vector();

Vectors can contain an element of class Object. This means that they can contain an instance of any class at all.

To add elements to the rear of a vector, use the addElement method.

    v.addElement(e);

An element can be inserted in a particular place using the insertElementAt method. Arguments are

    • an instance, and
    • an integer index, saying where the instance is to be added.

Thus, v.insertElementAt(e, 0) inserts element e at the head of the vector.

The converse, v.removeElementAt is useful for deleting elements.

    v.removeElementAt(0)

There are also v.firstElement() and v.lastElement().

What methods can be used to represent a queue?

    •
    •

What methods can you use to represent a stack?
Elements of primitive data types are *not* class instances, and they *cannot* be stored in vectors. There is a good reason for this. What is it?

However, Java provides *wrapper* classes to allow primitive values to be converted back and forth to class instances.

For *ints*, the wrapper is called *Integer*. An *Integer* instance is created the same way as any other instance:

```
Integer i = new Integer(n);
```

Here is how to store integers in a vector and read them back again:

```java
import java.util.*;
class VectorDemo {
    public static void main(String args[]) {
        Vector v = new Vector();
        v.addElement(new Integer(1));
        v.addElement(new Integer(2));
        Integer i = (Integer) v.lastElement();
        System.out.println(i);
    }
}
```

What else do you notice about this program that we haven’t seen before?
Why is this necessary here? Well, we said that a vector contains instances of what type?

So, when elements are fetched from vectors, an instance of what type is fetched?

We need to convert this object into an instance of the class it was before it was put in the vector.

Unlike in C and C++, casts in Java—

- are legal only between instances in the same inheritance hierarchy, and
- are type-checked at compile time and run time.

A cast can be used to narrow the type of a reference (make it something that’s less general). That’s what we needed to do with the Integer cast in the program above.

**Hashtables**

A hashtable is similar to a Smalltalk Dictionary.1

```java
Hashtable dates = new Hashtable();
dates.put("Christmas", new Date("25 Dec 1996"));
dates.put("Independence", new Date("4 July 1997"));
```

The first argument to put is the

The second argument is the

To get a value out, the get method is used:

```java
Date d = (Date) dates.get("Christmas");
```

A null value is returned if no value is found for the specified key.

Also,

- remove removes an element.

---

1Actually, there is a class Dictionary in Java. It is an abstract superclass of Hashtable.
containsKey tells whether a particular key is found in the dictionary.

Summary

- Interfaces provide many of the benefits of multiple inheritance, while avoiding most of the costs.
- Packages and multiple compilation units make it possible to organize a large program.
- Characters and strings are similar to, but not identical to, those in C++.
- File streams are used for input and output. Utility classes such as tokenizers and print streams facilitate their use.
- The exception-handling mechanism of Java relies upon “throwing” exceptions from one place in the code to be dealt with by an exception-handler elsewhere.
- Arrays in Java are zero-based and created on the heap.
- Java has a set of collection classes, which differ in detail from Smalltalk’s.