Chapter 11
More About Classes

Inheritance

Polymorphism

- *It's Greek to me!*
- Objects can take on different forms:

```
ChequeAccount c = new ChequeAccount();
Account a = new ChequeAccount();
```

Polymorphism (cont.)

```
Account a = new ChequeAccount();
```

- The **`ChequeAccount`** object has taken on the *form* of an **`Account`** object

- It is still a **`ChequeAccount`** object

Polymorphism (cont.)

- Inheritance is the key:
  - Any object can take on the form of any of its superclasses

```
Throwable t = new FileNotFoundException();
```

Polymorphism (cont.)

- The type of the *reference variable* determines what we can do with object
  - Using polymorphism we lose access to more specialised features

```
Account a = new ChequeAccount();
```

- We cannot access any overdraft information using **`a`**

Why Do We Use Polymorphism?

- Generic methods
- Generic lists

Generic Methods

```
public void writeBalance (Account acc)
  { logFile.println(acc.accountNumber + ": R”
                    + acc.balance);
  } // writeBalance
```

- This method can work with **`Account`** objects or *any* subclass of the **`Account`** class:
  - **`ChequeAccount`**
  - **`FixedDepositAccount`**
  - **`CreditCardAccount`**
  - etc.

Generic Lists

```
Object list[] = new Object[100];
```

- This array can hold *any* object

```
list[0] = ”George”;
list[1] = new ChequeAccount();
list[2] = new Integer(-73);
list[3] = new double[5];
list[4] = System.in;
...
```

Chapter 12
An Example Program

A General Frequency Table

- Analyse frequencies of test marks!

- How many students scored marks
  - 0% – 9%
  - 10% – 19%
  - 20% – 29%
  - ...
  - 90% or above

Class Diagram

- Simple at this stage:

Client View

```
public static void main (String[] args)
  { FreqTable cs2 = new FreqTable();
    BufferedReader in = ...
    String s;
    int total = 0;
    while ((s = in.readLine()) != null)
      { cs2.addEntry(Integer.parseInt(s));
      }
    for (int k = 0; k < 10; k++)
      { System.out.println(”Hits in bin " +
                    k + " = " + cs2.hits(k));
      }
  } // main
```

Implementor View

```
public class FreqTable
  { private int bin[] = new int[10];

    public void addEntry (int entry)
      { int binNumber = entry / 10;
        if (binNumber < 0)
          binNumber = 0;
        if (binNumber >= 10)
          binNumber = 9;
        bin[binNumber]++;
      } // addEntry

    public int hits (int binNumber)
        { return bin[binNumber];
        } // hits
  } // class FreqTable
```

Adding More Flexibility

- Different numbers and sizes of “bins”

- Clients:

```
FreqTable cs2 = new FreqTable(0, 10, 10);

FreqTable temp = new FreqTable(6, 2, 17);
            // Temperatures 6 - 40 degrees,
                                // 2 degree bins
```

Adding More Flexibility (cont.)

- Problem:
  - mapping values to bins is not so easy

```
binNum = (value - lowerBound) / binSize
```

- Class Diagram:

The Improved **`FreqTable`** Class

- The data members:

```
public class FreqTable
  { private int bin[];
    private int lb;
                  // Lower bound of first bin
    private int n; // number of bins
    private int binSize;
              // Number of values in each bin
    . . .
  } // class FreqTable
```

The **`FreqTable`** Class (cont.)

- Constructor:

```
public FreqTable (int lowerBound,
                  int width,
                  int numBins)
// Constructor
  { bin = new int[numBins];
    lb = lowerBound;
    n = numBins;
    binSize = width;
  } // Constructor
```

The `FreqTable` Class (cont.)

- Methods:

```
public void addEntry (int entry)
  { int binNumber = (entry - lb) / binSize;
    if (binNumber < 0)
      binNumber = 0;
    if (binNumber >= n)
      binNumber = n-1;
    bin[binNumber]++;
  } // addEntry

public int hits (int binNumber)
  { return bin[binNumber];
  } // hits
```

The End!

For Now!