CMT219 Algorithms, Data Structures and Programming

Session 4

Mathematical Methods
String Manipulation
Generics and Collections
Sorting
Class Design

- **Mathematical Methods**
- String Manipulation
- Generics and Collections
- Sorting
- Class Design

Random Numbers

- The **Random** class of the Java library implements a **random number generator** that produces numbers that appear to be completely random.
- To generate random numbers
  - construct an object of the **Random** class
  - apply one of the following methods
    - **nextInt(n)** returns a random integer between **0** (inclusive) and **n** (exclusive)
    - **nextDouble( )** returns a random floating-point number between **0** (inclusive) and **1** (exclusive)

Example: **Dice.java**

```
import java.util.Random;

public class Dice
{
   public static void main( String[] args )
   {
      Random generator = new Random();
      for ( int i = 1; i <= 24; i++ )
      {
         int number = Math.abs( generator.nextInt() ) % 6 + 1;
         System.out.print( number + " " );
      }
      System.out.println( "" );
   }
}
```

Result (24 random numbers between 1 and 6)

```
6 5 2 3 1 2 6 5 5 4 5 5 2 5 1 3 6 3 2 6 4 6 4 6
```

The Power and Sqrt Function

- E.g. the value of **2^5 = 2 * 2 * 2 * 2 * 2 = 32**
- Use **Math.pow**(x, y) static method to calculate **x^y**

- E.g. the square root of 49 is 7, i.e. **7 * 7 = 49**.
- Use **Math.sqrt**(x) to obtain the square root of **x**

- See JDK references for methods of **Math** class

Example: **Power.java**

```
public class Power
{
   // calculation of a value raised to a power

   // main method
   public static void main( String[] args )
   {
      System.out.println( "The cube of 2.5 = "
                               + Math.pow( 2.5, 3 ) );
   }
}
```

Result (2.5^3)

```
The cube of 2.5 = 15.625
```

Factorials

- The **factorial** of the number **6** is written as **6!**
- **6! = 6 * 5 * 4 * 3 * 2 * 1 = 720**
- In general, n! = n * (n-1) * (n-2) * … * 3 * 2 * 1

- Why do we use `long` as the return type in the following example?

Example: **Factorial.java**

```
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;
public class Factorial
{
   // main method
   public static void main( String[] args ) throws IOException
   {
      BufferedReader reader = new BufferedReader(
            new InputStreamReader( System.in ) );
      System.out.print( "Please enter a number: " );
      String inputLine = reader.readLine();
      int n = Integer.parseInt(inputLine);
      System.out.println( n + "! = " + factorial( n ) );
      System.exit( 0 );
   }
```

Example: **Factorial.java** (cont.)

```
   // calculation of the factorial of a number
   public static long factorial( int n )
   {
      if ( n == 0 || n == 1)
         return 1;
      else
      {
         long result = 1;
         for (int i = 2; i <= n; i++ )
              result = result * i;
         return result;
      }
   }
}
```

Example: **Factorial.java** (cont.)

- Result:

```
Please enter a number: 3
3! = 6

Please enter a number: 20
20! = 2432902008176640000
```

Recursion

- **Recursion** occurs when a method calls itself.
- For recursion to be successful
  - Every recursive call must ***simplify*** the computation in some way.
  - There must be ***special cases*** to handle the simplest computation(s).

Recursive Definition of Factorial

- ***n! = n * (n–1) * … * 2 * 1***
- Alternatively we may write
  - ***n! = n * (n–1)!***
  - the special case, which prevents us from trying to repeatedly use the formula for ever

    0! = 1

- Note that to ***avoid endless repetition*** we must test for and apply the special case(s) ***before*** the general case.

Example: **Factorial.java**

```java
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;

public class Factorial
{
   // main method
   public static void main( String[] args )
                               throws IOException
   {
      BufferedReader reader = new BufferedReader(
            new InputStreamReader( System.in ) );
      System.out.print( "Please enter a number: " );
      String inputLine = reader.readLine();
      int n = Integer.parseInt(inputLine);
      System.out.println( n + "! = "
                              + factorial( n ) );
      System.exit( 0 );
   }
```

Example: **Factorial.java** (cont.)

```
   // calculation of the factorial of a number
   public static long factorial( int n )
   {
      if ( n == 0 )
         return 1;
      else
      {
         long result = n * factorial( n - 1 );
         return result;
      }
   }
}
```

Example: **Factorial.java** (cont.)

- **The sequence of calls and return values for factorial(6) is:**

```
factorial(6) calls factorial(5)
   factorial(5) calls factorial(4)
      factorial(4) calls factorial(3)
         factorial(3) calls factorial(2)
            factorial(2) calls factorial(1)
               factorial(1) calls factorial(0)
                  factorial(0) returns 1
               factorial(1) returns 1 (1x1)
            factorial(2) returns 2 (2x1)
         factorial(3) returns 6 (3x2)
      factorial(4) returns 24 (4x6)
   factorial(5) returns 120 (5x24)
factorial(6) returns 720 (6x120)
```

Recursion

- Recursion can be useful for solving more complicated problems, e.g. sorting (put numbers in order) etc.
- Programming efficiency should also be taken into account.

- Mathematical Methods
- **String Manipulation**
- Generics and Collections
- Sorting
- Class Design

StringBuffer Class

- **String** is an immutable class.
  - Once a **String** object is created, its contents never change.
- The **StringBuffer** class should be used when the contents of the string are to be changed.
- The default **StringBuffer** constructor creates a **StringBuffer** with
  - **no** characters in it
  - an initial capacity of **16** characters
- Methods
  - length(): returns the number of characters currently in a **StringBuffer**
  - capacity(): returns the number of characters that can be stored in a **StringBuffer** without allocating more memory
  - toString( )**:** converts the data in the **StringBuffer** to a **String**

StringBuffer class (cont.)

- Alternative constructors
  - StringBuffer(int capacity): creates a **StringBuffer** object with
    - **no** characters in it
    - an initial capacity of characters as specified
  - StringBuffer(String initString): creates a **StringBuffer** object with
    - The content of the string in it
    - an initial capacity of the length of the String plus 16 characters

Example: **CreateBuffers.java**

```
public class CreateBuffers
{
   public static void main( String[] args )
   {
      // Create a StringBuffer with a capacity of 16
      // characters
      StringBuffer buf1 = new StringBuffer();
      System.out.println( "Number of characters = "
                                      + buf1.length() );
      System.out.println( "Space in buffer = "
                                    + buf1.capacity() );
      System.out.println( "Contents = " + "\""
                             + buf1.toString() + "\"" );
      // Create a StringBuffer with a capacity of 10
      // characters
      StringBuffer buf2 = new StringBuffer( 10 );
      System.out.println( "Number of characters = "
                                      + buf2.length() );
      System.out.println( "Space in buffer = "
                                    + buf2.capacity() );
      System.out.println( "Contents = " + "\""
                             + buf2.toString() + "\"" );
```

```
Number of characters = 0
Space in buffer = 16
Contents = ""
Number of characters = 0
Space in buffer = 10
Contents = ""
```

Example: **CreateBuffers.java** (cont.)

```
      // Create a StringBuffer containing specified
      // String and an extra 16 unused characters.
      StringBuffer buf3 = new StringBuffer( "cat" );
      System.out.println( "Number of characters = "
                              + buf3.length() );
      System.out.println( "Space in buffer = "
                              + buf3.capacity() );
      System.out.println( "Contents = " + "\""
                        + buf3.toString() + "\"" );
   }
}
```

```
Number of characters = 3
```

```
Space in buffer = 19
```

```
Contents = "cat"
```

Example: **CreateBuffers** (cont.)

- Result

```
Number of characters = 0
Space in buffer = 16
Contents = ""
Number of characters = 0
Space in buffer = 10
Contents = ""
Number of characters = 3
Space in buffer = 19
Contents = "cat"
```

StringBuffer Methods

- **StringBuffer** provides methods to manipulate the current string content it contains:
  - **insert**(int offset, String str): insert a string at the specified offset in this **StringBuffer.**
  - **append**(String str)**:** append a string to this **StringBuffer.**
  - **reverse**(): replace the character sequence contained in this **StringBuffer** by the reverse of the sequence.
  - **delete**(int start, int end): removes the characters in a substring of this **StringBuffer**, start inclusive, end exclusive, 0-based.
  - **replace**(int start, int end, String str): replaces the characters in a substring of this **StringBuffer**.

StringBuffer Methods (cont.)

- **toString**(): returns the **String** representation of this **StringBuffer**.
- **substring**(int start, int end): returns the **String** representation of a subsequence of characters currently contained in this **StringBuffer**. If only start is specified, the substring continues to the end of the **StringBuffer**, start inclusive, end exclusive, 0-based.
- **charAt**, **setCharAt** and **deleteCharAt:** get, change and delete the character at a specified position in this **StringBuffer**.
- **getChars**(int srcBegin, int srcEnd, char[ ] dst, int dstBegin): copies a subsequence of this **StringBuffer** to a character array.

StringBuffer Methods (cont.)

- **setLength**(int newLength): set the length of the current **StringBuffer** object
  - If newLength is less than the current length, the content is truncated
  - If newLength is larger than the current length, null characters (\u0000) are used to fill the extra space
- **ensureCapacity**(int newCapacity): minimum capacity of the current object is set to the greater of
  - newCapacity
  - twice the old capacity plus 2

Example: **BufferSize.java**

```
public class BufferSize
{
   public static void main( String[] args )
   {
      // Create a StringBuffer containing specified String and an
      // extra 16 unused characters.
      StringBuffer buffer = new StringBuffer( "Mike Evans" );
      char[] surName = new char[ buffer.length() ];
      System.out.println( "Number of characters in buffer = "
                                              + buffer.length() );    10
      System.out.println( "Space in buffer = "
                                            + buffer.capacity() );    26
      System.out.println( "Contents of buffer = " + "\""
                                     + buffer.toString() + "\"" ); "Mike Evans"

      buffer.getChars( 5, 10, surName, 0 );
      System.out.println( "Contents of character array is "
                      + "\"" + String.valueOf( surName ) + "\"" ); "Evans"
```

Example: **BufferSize.java** (cont.)

```
      buffer.setLength( 4 );
      System.out.println( "Number of characters in buffer = "
                                              + buffer.length() );    4
      System.out.println( "Space in buffer = "
                                            + buffer.capacity() );    26
      System.out.println( "Contents of buffer = " + "\""
                                     + buffer.toString() + "\"" );   "Mike"

      // Minimum capacity of StringBuffer is set to the greater of
      // (1)    the minimumCapacity argument
      // (2)    twice the old capacity plus 2
      buffer.ensureCapacity( 27 );
      System.out.println( "Space in buffer = "
                                            + buffer.capacity() );    54
      buffer.ensureCapacity( 150 );
      System.out.println( "Space in buffer = "
                                            + buffer.capacity() );   150
   }
}
```

Example: **BufferSize.java** (cont.)

- Result

```
Number of characters in buffer = 10
Space in buffer = 26
Contents of buffer = "Mike Evans"
Contents of character array is "Evans"
Number of characters in buffer = 4
Space in buffer = 26
Contents of buffer = "Mike"
Space in buffer = 54
Space in buffer = 150
```

Example: **Reverse.java**

```
public class Reverse
{
   public static void main( String[] args )
   {
      // Create a StringBuffer containing specified
      // String and an extra 16 unused characters.
      StringBuffer buffer = new StringBuffer( "cat" );
      buffer.reverse();
      System.out.println( buffer.toString() );
   }
}
```

Result (Reversed string)

```
tac
```

Example: **Manipulate.java**

```
public class Manipulate
{
   public static void main( String[] args )
   {
      // Create a StringBuffer containing specified String
      // and an extra 16 unused characters.
      StringBuffer buffer = new StringBuffer( "cat" );

      // Display a character in the StringBuffer
      char ch = buffer.charAt( 1 );
      System.out.println( String.valueOf( ch ) );  a

      // Change a character in the StringBuffer
      buffer.setCharAt( 1, 'o' );
      System.out.println( buffer.toString() );     cot

      // Insert a character in the StringBuffer
      buffer.insert( 2, 's' );
      System.out.println( buffer.toString() );     cost
```

Example: **Manipulate.java** (cont.)

```
        // Add characters to the StringBuffer
        buffer.append( "ly" );
        System.out.println( buffer.toString() );          costly

        // Display a portion of the StringBuffer
        System.out.println( buffer.substring( 0, 3 ) );   cos

        // Delete characters from the StringBuffer
        buffer.deleteCharAt( 2 );
        buffer.delete( 3, 5 );
        System.out.println( buffer.toString() );          cot

        String str = "art";
        buffer.replace( 1, 3, str );
        System.out.println( buffer.toString() );          cart
    }
}
```

Example: **Manipulate.java** (cont.)

- Result

```
a
cot
cost
costly
cos
cot
cart
```

StringBuilder class

- **StringBuilder** is similar to **StringBuffer** with virtually the same methods
  - **StringBuffer** is synchronised, thread-safe (more later)
  - **StringBuilder** is faster than **StringBuffer** but does not work directly when multiple threads may access the same object at the same time
  - **Both** are much faster than directly use **String** objects when strings are manipulated (see the performance comparison demo)

Performance Comparison

- ConcatPerf.java
- Compare direct String concatenation, StringBuffer, StringBuilder
- Result (may vary from different computers)

```
Iterations: 100000
Buffer    : 16
concatStrBuff  -> length: 100000 time: 17
concatStrBuild -> length: 100000 time: 4
concatStrAdd   -> length: 100000 time: 7724
```

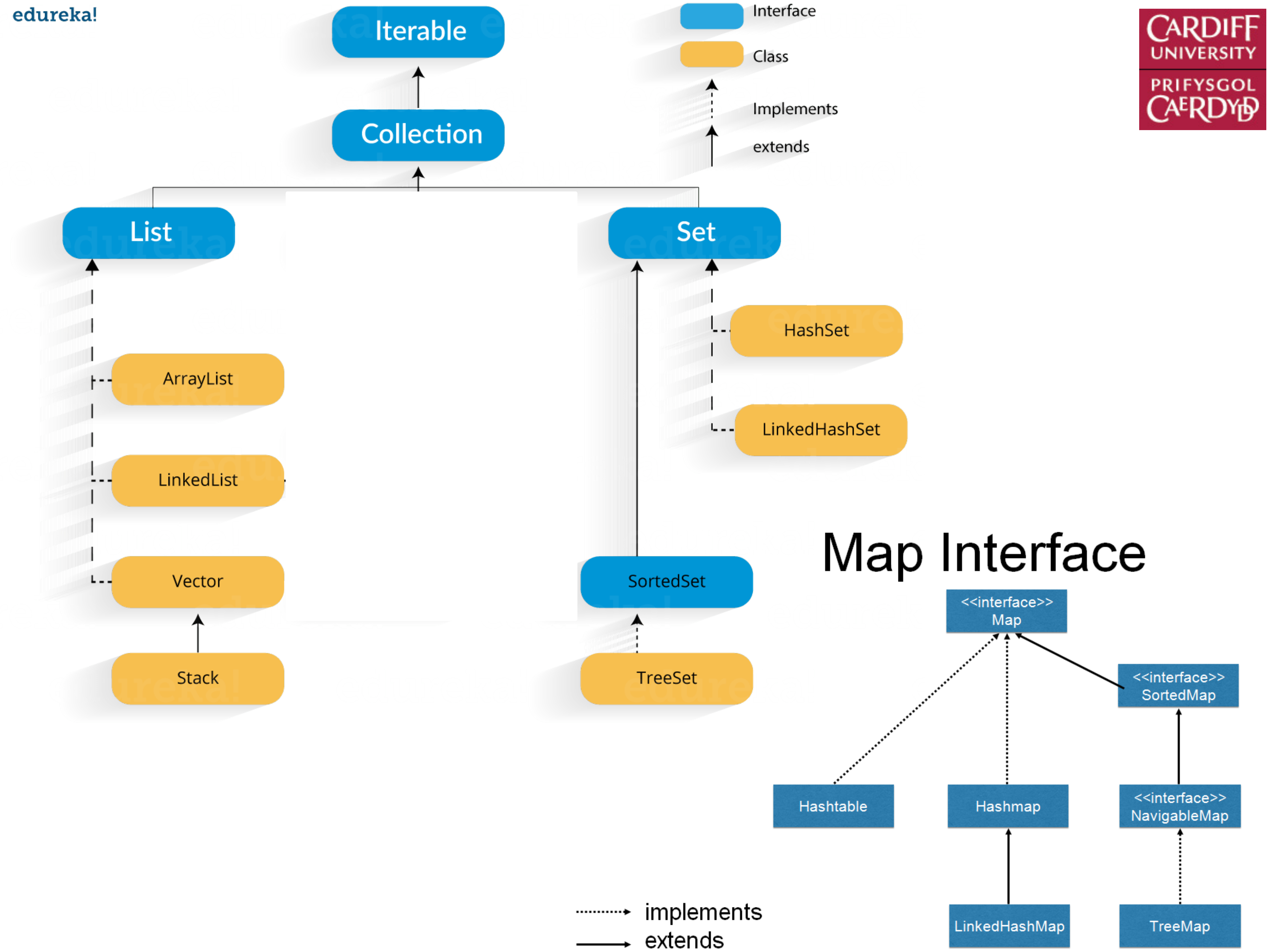
- Mathematical Methods
- String Manipulation
- **Generics and Collections**
- Sorting
- Class Design

Java Collections Framework

- Arrays are efficient and useful, but have problems:
  - Pre-defined, fixed size
  - No useful pre-defined methods
- Java Collections Framework
  - Help manage data collections using typical data structures
  - Provide relevant associated algorithms to manipulate the data
  - Defined in the **java.util** package

Java Collections Framework

- Collections are organised using the following **interfaces**, each with different **concrete classes**
  - **List**: ordered list of objects, e.g. jobs for a printer
  - **Set**: objects with no repetition and order is not important, e.g. registration numbers for permitted cars in a car park
  - **Map**: pairs of objects, the first is a **key**, the second its associated **value**, like a look-up table, e.g. user information for a network (username & password pairs)


Iterable
Collection
Interface
Class
Implements
extends
List
Set
ArrayList
LinkedList
Vector
Stack
HashSet
LinkedHashSet
SortedSet
TreeSet
Map Interface
<<interface>> Map
<<interface>> SortedMap
Hashtable
Hashmap
<<interface>> NavigableMap
LinkedHashMap
TreeMap
implements
extends

Generics

- In Java, all the classes are directly or indirectly derived from **Object** class
- For a collection class to handle general objects, they can be handled as **Object** instances. Problems:
  - When an instance is retrieved from the collection, it needs to be explicitly cast to the actual class
  - There is no proper type check
- **Generics**
  - Allows to send one or more types to a class (or interface) as a parameter
  - All the Java collections use generics

Generics

- Example (create an **ArrayList** of type **String**):

```
ArrayList<String> printQ = new ArrayList<String>();
```

  - <…> is used to specify the parameter types
  - If multiple types are involved, separate them with a comma, e.g. `Map<String, String>`
  - **ArrayList<String>** is a specialised version of **ArrayList** which holds **String** elements. `ClassName <TypeList>` should be treated as a whole when used

- Advantages
  - Allows the compiler to check items are of correct type
  - Removes the need for type casting when retrieving items from the collection
  - Avoids the possibility of `ClassCastException`

Wrapper Classes

- Primitive types are not classes, so cannot be used as types for generics
  - So generic collections cannot store e.g. **int** or **double**
  - Wrapper classes are introduced for this purpose: **Integer** class for **int**, **Double** class for double, e.g.

```
ArrayList<Integer> nums = new ArrayList<Integer>();
```

  - Autoboxing and unboxing automate the process of moving from a primitive type to its associated wrapper

List Interface

- `List` interface
  - Defines the common behaviour of an ordered list of objects
  - Concrete classes:
    - **ArrayList**
    - **Vector**
    - **LinkedList**
  - Internally, **ArrayList/Vector** are based on arrays, **LinkedList** is based on linked lists, but the same List interface applies

List Interface (cont.)

- `List` interface methods
  - **add**(object): adds an object to the end of the list
  - **add**(index, object): inserts an object at the specified index
  - **set**(index, object): sets the item at the specified index
  - **get**(index): retrieves the item at the specified index
  - **remove**(index): removes the item at the specified index
  - **toString()**: converts the list to a String representation
  - **size()**: returns the number of items in the list
- If the index is out of the range, an `IndexOutofBoundsException` will be thrown.

Example: **ListTest.java**

```
import java.util.*;

public class ListTest
{
    // Simulating printer queue
    public static void main(String[] args)
    {
        ArrayList<String> printQ = new ArrayList<String>();
        printQ.add("myLetter.doc");
        printQ.add("myPhoto.jpg");
        printQ.add("results.xls");
        System.out.println(printQ);

        printQ.add(0, "importantMemo.doc"); // inserts into the front
        System.out.println(printQ);

        printQ.set(3, "newChapter.doc");    // sets an item
        System.out.println(printQ);

        printQ.remove(2);                   // removes an item
        System.out.println(printQ);

        System.out.println("Item at index 1: " + printQ.get(1));
    }
}
```

```
[myLetter.doc, myPhoto.jpg, results.xls]
[importantMemo.doc, myLetter.doc, myPhoto.jpg, results.xls]
[importantMemo.doc, myLetter.doc, myPhoto.jpg, newChapter.doc]
[importantMemo.doc, myLetter.doc, newChapter.doc]
Item at index 1: myLetter.doc
```

Example: **ListTest.java**

- Output:

```
[myLetter.doc, myPhoto.jpg, results.xls]
[importantMemo.doc, myLetter.doc, myPhoto.jpg, results.xls]
[importantMemo.doc, myLetter.doc, myPhoto.jpg, newChapter.doc]
[importantMemo.doc, myLetter.doc, newChapter.doc]
Item at index 1: myLetter.doc
```

`Set` Interface

- `Set` interface
  - Defines the common behaviour of a collection of objects
    - With no repetition
    - Ordering is unimportant
  - Which of the following can be considered as a set?
    - A queue of people waiting to see a doctor?
    - The winners of an annual competition over the last 10 years?
    - Car registration numbers allocated parking permits?
  - Concrete class: e.g. `HashSet`

Set Interface (cont.)

- Set interface methods
  - **size**(): returns the number of items in the set
  - **add**(object): adds an item to the set. If the item already exists, the set is unchanged.
  - **remove**(object): removes an item from the set
  - **toString**(): converts a set to a String representation
- Note
  - **add** and **remove** returns a boolean to indicate if the operation was successful.
  - Items in the set are unordered
  - No duplicated items

Example: **SetTest.java**

```
import java.util.*;
public class SetTest
{
    // use set to record car registrations with parking permit
    public static void main(String[] args)
    {
        Set<String> regNums = new HashSet<String>();
        regNums.add("CK14EAD");
        regNums.add("DV59CDE");
        regNums.add("CA61VAE");
        System.out.println(regNums);                  [DV59CDE, CK14EAD, CA61VAE]

        System.out.println(regNums.add("DV59CDE"));                  false
        System.out.println(regNums);                  [DV59CDE, CK14EAD, CA61VAE]

        System.out.println(regNums.remove("DV59CDE"));               true
        System.out.println(regNums);                  [CK14EAD, CA61VAE]

        System.out.println(regNums.remove("DV59CDE"));               false
        System.out.println(regNums);                  [CK14EAD, CA61VAE]
    }
}
```

Example: **SetTest.java** (cont.)

- Output

```
[DV59CDE, CK14EAD, CA61VAE]
false
[DV59CDE, CK14EAD, CA61VAE]
true
[CK14EAD, CA61VAE]
false
[CK14EAD, CA61VAE]
```

`Map` Interface

- The `Map` interface defines the methods to process a collection consisting of *pairs* of objects
  - Like a **look-up** table
  - The **key** object is used to look up (access) an associated **value** object in the table
  - E.g. (username, password)
- `Map` interface is a generic type that requires two element types: the type for **key** and the type for **value**
- Concrete classes: e.g. HashMap

Map Interface (cont.)

- `Map` interface methods:
  - **put**(key, value): add the (*key*, *value*) pair to the map; if the *key* exists already, it will overwrite previous entry with the same *key*.
  - **containsKey**(key): checks whether the *key* exists in the map
  - **get**(key)**:** retrieves the *value* associated with the given *key*, or **null** if the key does not exist
  - **toString**(): converts the map to a String representation

Example: **MapTest.java**

```
import java.util.*;
public class MapTest
{
    // check if the username, password exists in the map
    private static void check_user(String username, String password)
    {
        if (!users.containsKey(username))
            System.out.println("User " + username + " does not exist");
        else if (users.get(username).equals(password))
            System.out.println("User " + username + " logged in");
        else
            System.out.println("User " + username + " exists but the password is wrong.");
    }
    public static void main(String[] args)
    {
        // add username, password pairs to the map
        users = new HashMap<String, String>();
        users.put("laura", "monkey");
        users.put("bobby", "monkey");
        users.put("lucy", "velvet");

        // print the map
        System.out.println(users);
```

```
{laura=monkey, bobby=monkey, lucy=velvet}
```

```
        // check users (username, password)
        check_user("bob", "monkey");
        check_user("laura", "hello");
        check_user("laura", "monkey");
    }
    private static Map<String, String> users; // map for user accounts
}
```

```
User bob does not exist
User laura exists but the password is wrong.
User laura logged in
```

Example: **MapTest.java** (cont.)

- Output

```
User bob does not exist
User laura exists but the password is wrong.
User laura logged in
```

Enhanced **for** Loops

- Enhanced **for** can be used to iterate through a collection in a consistent way
  - **for** (type item: collection_object) statement
  - *type* should be the type of the items in the collection
  - The variable *item* will take each item in the collection and run the statement (or compound statement).
  - Sometimes called **for each** loops

Example: **ForEach.java**

```
import java.util.*;

public class ForEach
{
    // Simulating printer queue
    public static void main(String[] args)
    {
        ArrayList<String> printQ = new ArrayList<String>();
        printQ.add("myLetter.doc");
        printQ.add("myPhoto.jpg");
        printQ.add("results.xls");
        printQ.add("memo.doc");

        // find all the .doc files
        for (String item: printQ)
        {
            if (item.endsWith(".doc"))
                System.out.println(item);
        }
    }
}
```

```
myLetter.doc
memo.doc
```

Example: **ForEach.java** (cont.)

- Output

```
myLetter.doc
memo.doc
```

Accessing Collection using `Iterator`

- An alternative way to access items in a collection
  - Collection classes implement **Iterable** interface which defines **iterator()** method that returns an **Iterator** object
  - **Iterator** object has the following methods to allow access items in the collection
    - **hasNext**(): returns if there are more elements in the collection
    - **next**(): retrieves the next element from the collection
    - **remove**(): removes from the collection

Example: **IteratorTest.java**

```
import java.util.*;

public class IteratorTest
{
    // Simulating printer queue
    public static void main(String[] args)
    {
        ArrayList<String> printQ = new ArrayList<String>();
        printQ.add("myLetter.doc");
        printQ.add("myPhoto.jpg");
        printQ.add("results.xls");
        printQ.add("memo.doc");

        // find all the .doc files
        Iterator<String> files = printQ.iterator();
        while (files.hasNext())
        {
            String filename = files.next(); // gets the next file
            if (filename.endsWith(".doc"))
                System.out.println(filename);
        }
    }
}
```

Example: **IteratorTest.java** (cont.)

- Output

```
myLetter.doc
memo.doc
```

- Mathematical Methods
- String Manipulation
- Generics and Collections
- **Sorting**
- Class Design

Sorting

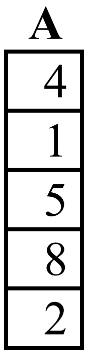
- Sorting
  - A process of arranging items in a specific order
  - Typical orders: ascending, descending
  - A common operation needed by lots of algorithms
- Many sorting algorithms have been developed
  - Simple sorting methods: bubble sort, insertion sort, selection sort
  - Other methods: Quick sort, merge sort etc.
  - Different algorithms may have varied performance for input data of different nature.
- We show how to use Java
  - For simple sorting (e.g. Bubble sort)
  - Use Java standard library for sorting collections

Bubble Sort

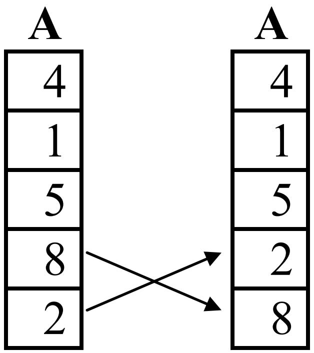
- The steps involved in a **bubble sort** are:
  - ***Compare*** the numbers in the last two elements of the array and ***exchange*** the numbers in these elements if they are not in ascending order.
  - ***Compare*** the numbers in the two elements above the last element of the array and ***exchange*** the numbers if they are not in ascending order.
  - Continue this process until the ***smallest number*** has ***bubbled up*** to be the ***first element*** of the array.
  - ***Repeat*** the above process ***using all elements*** in the array ***except the first element*** so that the next smallest number is in second element of the array.
  - ***Continue this process*** until the array is sorted.

Bubble Sort: Example

- Suppose an unsorted array **A** is

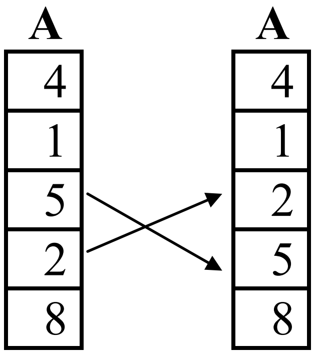


- The first step of the **bubble sort** produces

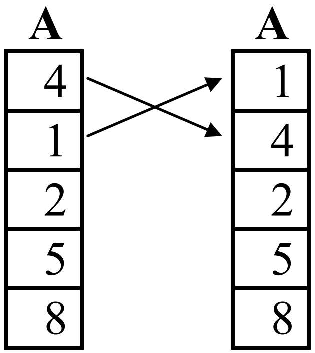


Bubble Sort: Example

- The second step of the **bubble sort** produces

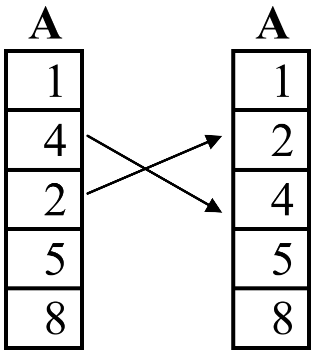


- The third step of the **bubble sort** produces



Bubble Sort: Example

- The final step of the **bubble sort** produces



- Note:
  - We omitted steps that do not incur item swap in the previous illustration.
  - Even if no swap happens, comparisons still need to be performed.

Example: **BubbleSort.java**

```
public class BubbleSort
{
   public static void main( String[] args )
   {
      int marks[] = { 23, 17, 59, 86, 10 };
      System.out.println( "Unsorted Array" );
      System.out.println( "--------------" );
      for ( int i = 0; i < marks.length; i++ )
         System.out.println( marks[ i ] );
      // sort array
      bubble( marks );
      System.out.println( "Sorted Array" );
      System.out.println( "------------" );
      for ( int i = 0; i < marks.length; i++ )
         System.out.println( marks[ i ] );
   }
```

Example: **BubbleSort.java** (cont.)

```
   // method to sort an array using a bubble sort
   public static void bubble( int[] marks )
   {
      int i, j, temp;
      for ( i = 0; i < marks.length - 1; i++ )
      {
         for ( j = marks.length - 1; j > i; j-- )
         {
            if ( marks[ j ] < marks[ j - 1 ] )
            {
               temp = marks[ j ];
               marks[ j ] = marks[ j - 1 ];
               marks[ j - 1 ] = temp;
            }
         }
      }
   }
}
```

Sorting using Java API

- You can sort arrays in a descending order using simple sorting by changing the comparison
- Simple sorting methods are ***not efficient*** for large amounts of data.
- There are built in ***sorting*** methods in the ***Java API***, and you should use those if you really need to do some sorting.
  - The static method of **Arrays.sort**() for sorting an array
  - The static method of **Collections.sort**() for sorting a List (including e.g. ArrayList)
  - Check JDK documentation for more detail

Example: **ArraySort.java**

```
import java.util.Arrays;

public class ArraySort
{
    public static void main(String[] args)
    {
        int[] nums = {10, 2, -3, 1, 4};
        Arrays.sort(nums);
        for (int item : nums)
        {
            System.out.print(item + " ");
        }
    }
}
```

Output:

```
-3 1 2 4 10
```

Note:

- **Arrays.sort**() is defined in the **java.util** package
- The enhanced **for** loop for accessing all the elements in the array

Example: **CollectionSort.java**

```
import java.util.*;
public class CollectionSort
{
    public static void main(String[] args)
    {
        ArrayList<Integer> list = new ArrayList<Integer>();
        list.add(10);
        list.add(2);
        list.add(-3);
        list.add(1);
        list.add(4);
        Collections.sort(list);
        System.out.println(list);
    }
}
```

```
Output: [-3, 1, 2, 4, 10]
```

Note

- **Collections.sort**() is defined in **java.util** package
- This also applies to other **List** collections

- Mathematical Methods
- String Manipulation
- Generics and Collections
- Sorting
- **Class Design**

Designing and building a software system from scratch

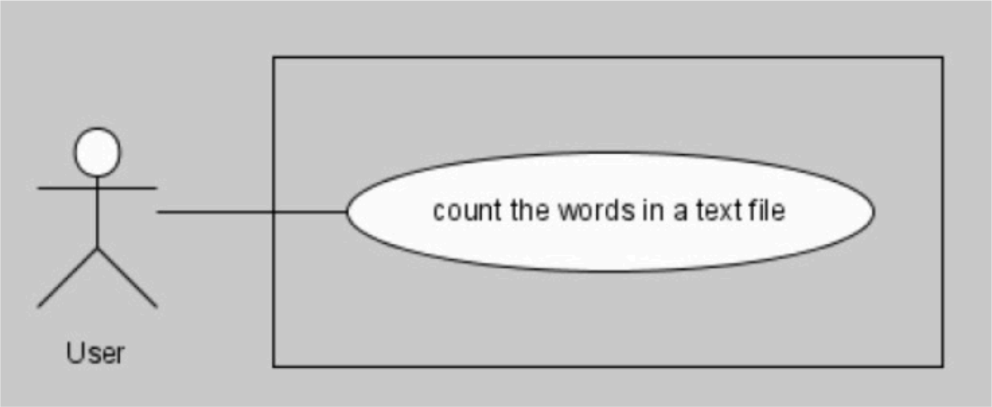
- Typical steps involved in OO design and implementation
  - Develop a precise specification of the system
  - Determine the classes, their responsibilities and collaborators
  - Determine the precise protocol or interface of the classes
  - Implement the classes

Example Project

- Initial requirements of **word frequency counter** application
  - when the Java program starts, it analyses the specified text file. It constructs a summary report regarding each word that occurs in the file and the number of times that word occurs, sorted from most frequently occurring word to least frequently occurring.
- What needs to be elaborated with the specification?

Use Cases

- A **use case** is a sequence of steps indicating how the program is to behave in a certain situation to achieve a particular goal.
- Create a use case for each of the ways the system is expected to behave.

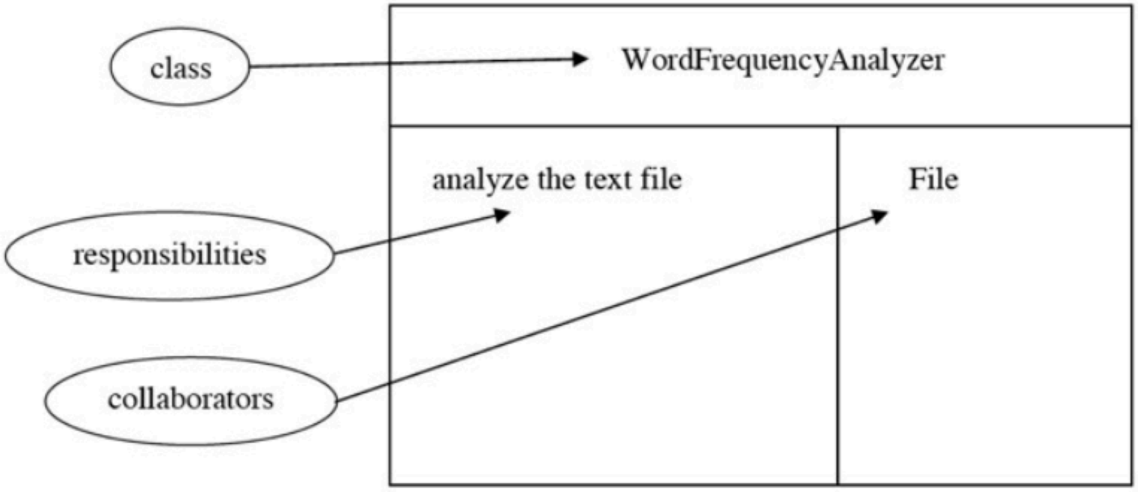


Finding the Classes (First Attempt)

- Let each noun in the specification corresponds to a class and each verb to a method of a class
  - Nouns: program, user, file, word, number, times, report, pathname …
  - Some correspond to useful classes, e.g. Program, File, Report; but some may not
- Use concepts from the application domain

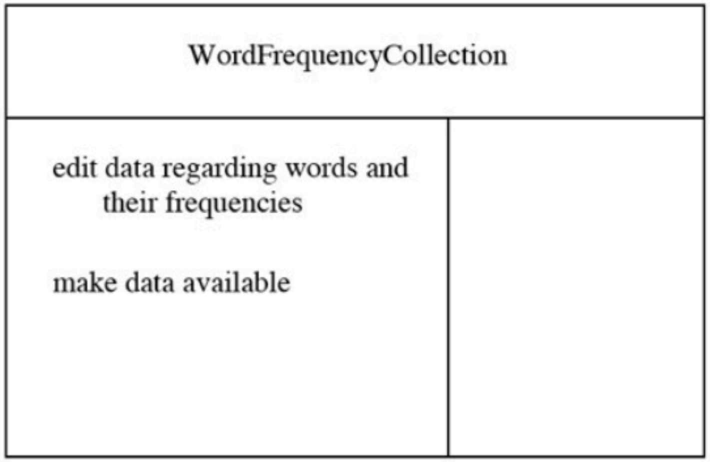
CRC Cards

- CRC
  - stands for “Class, Responsibilities and Collaborators”
  - Use one note card for each class
  - Use role playing to determine responsibilities and collaborators
- A CRC card for **WordFrequencyAnalyzer** class

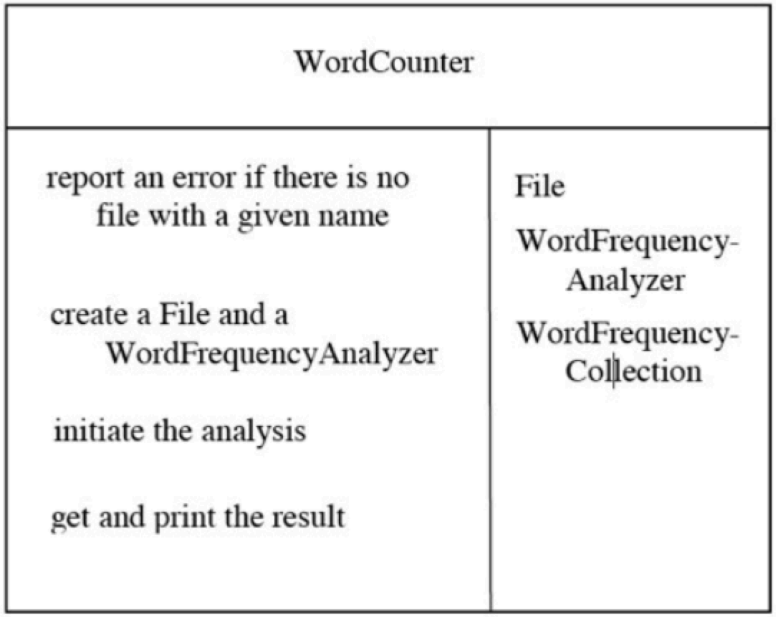


CRC Cards (cont.)

- **WordFrequencyCollection** and **WordCounter** classes



WordFrequencyCollection	
edit data regarding words and their frequencies make data available	



WordCounter	
report an error if there is no file with a given name	File
create a File and a WordFrequencyAnalyzer	WordFrequency-Analyzer WordFrequency-Collection
initiate the analysis	
get and print the result	

Class Protocols

- The CRC cards provide potential classes, their responsibilities and collaborators
- Next step is to construct the **protocols**, or **public interfaces**, of all the classes.

Class Protocols: Direct Map

- Direct map from CRC cards

```
public class WordCounter
{
   public WordCounter(); //constructor
   public void checkFileExistence(String filename);
   public File createFile();
   public WordFrequencyAnalyzer createAnalyzer();
   public void initiateAnalysis();
   public getAndPrintResult();
}
public class WordFrequencyAnalyzer
{
   public WordFrequencyAnalyzer(); //constructor
   public void analyzeText(File file);
   public WordFrequencyCollection getResults();
}
public class WordFrequencyCollection
{
   public WordFrequencyCollection(); //constructor
   public void editCollection();
   public String toString();
}
```

Class Protocols: Refinement

- Methods in **WordCounter** class are specific steps and can be made private (i.e. not part of the public interface)

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

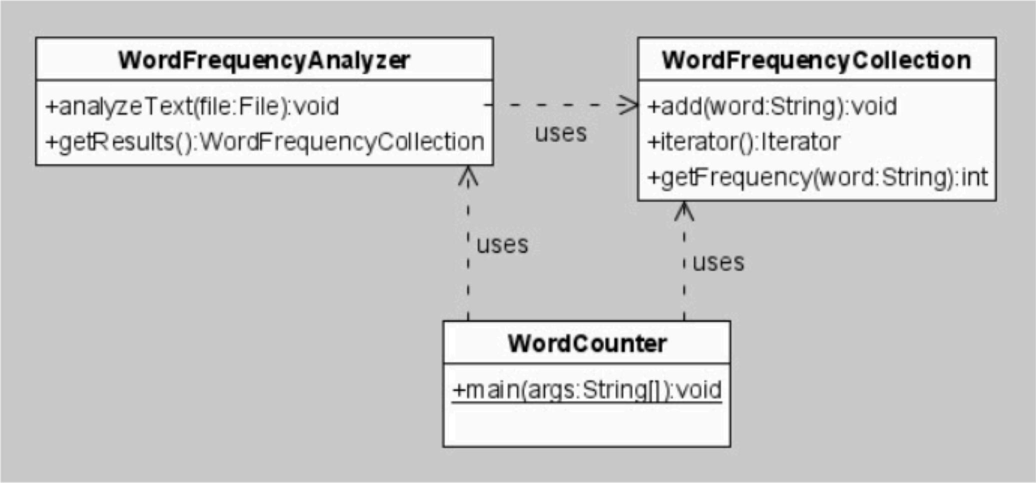
Class Protocols: Refinement (cont.)

- **WordFrequencyCollection** class
  - **editCollection** too vague ➔ **add** method
  - toString: too generic for returning the result ➔ use an iterator (which allows accessing each element)

```
public class WordFrequencyCollection implements
Iterable<String>
{
   public WordFrequencyCollection(); //constructor
   public void add(String word);
   public Iterator<String> iterator();
   public int getFrequency(String word);
}
```

The overall class diagram

- Gives the big picture of classes and their relationship


WordFrequencyAnalyzer
+analyzeText(file:File):void
+getResults():WordFrequencyCollection
uses
WordFrequencyCollection
+add(word:String):void
+iterator():Iterator
+getFrequency(word:String):int
uses
uses
WordCounter
+main(args:String[]):void

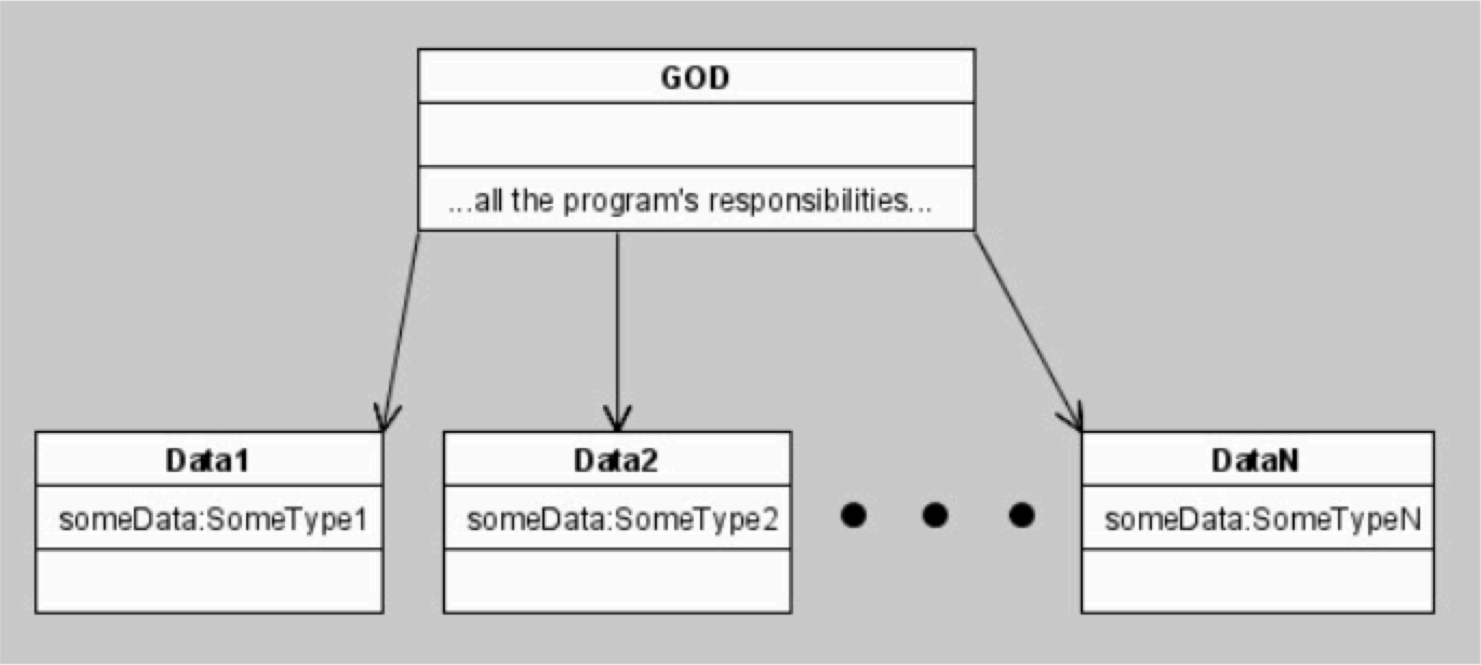
- Guideline: When working in the early high-level design phase, keep the discussion at a high level. In particular, avoid wasting time on implementation details and low level data representations.

Maximise Class Cohesion

- A class should model one concept
- All its methods should be related to and appropriate for that concept
- Promotes understanding and reusability of the class
- Example: The Java `String` class
- Guideline: Every class should be responsible for doing one thing only and doing it well.

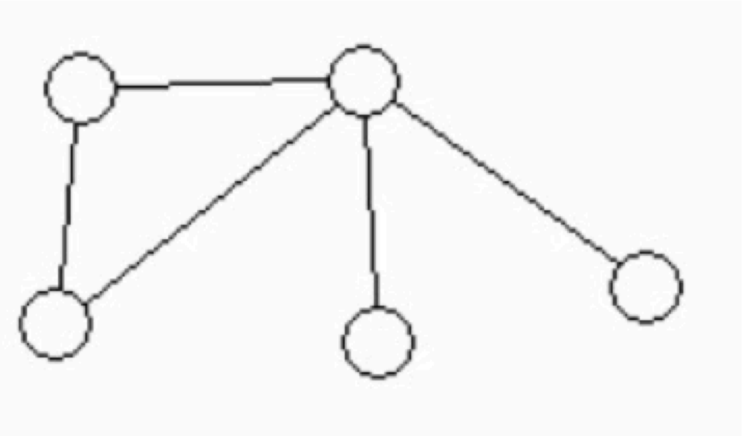
An ill-formed example

- An ill-formed God class with all the responsibilities and slave (data-only) classes



Separation of Responsibilities

- A class should do one thing only => different kinds of responsibilities should be delegated to different classes
- Example: traversing a Graph object
- Who keeps track of which nodes of the Graph have already been visited?
  - The nodes themselves
  - The Graph object
  - A different object
- Imagine multiple concurrent traversals



Separation of Responsibilities (cont.)

- Guideline (Expert Pattern)
  - The object that contains the necessary data to perform a task should be the object that performs the task.
  - “Ask not what you can do to an object, ask what an object can do to itself.”

Avoid Duplication

- Duplication can occur in many forms:
  - duplicate copies of same data
  - duplicate code within a method or between methods
  - duplicate processes (duplicate execution of a piece of code)
- Code with duplication is less readable and less maintainable than code without duplication.
- **Guideline**: Only **one** class should be responsible for knowing and maintaining a set of data, even if that data is used by many other classes.

Behaviour of a Class

- Should a class have only the minimum necessary behaviour to accomplish its tasks?
- Should we add more behavior to increase the reusability of the class?
- Should we add lots of auxiliary methods to make the class easier to use?
- What should the protocol be for each method?

Behaviour of a Class (cont.)

- **Guideline**: Give classes complete and consistent interfaces.
- **Examples**
  - GUI components could have `setSelected`() and `setUnselected`() methods, but better to have `setSelected(boolean b)` and `isSelected`() methods.
  - If there is a `getXXX` method, consider a `setXXX` method unless the XXX property is read only.

Behaviour of a Class (cont.)

- Consistency: the methods that do similar things should be laid out similarly
- Example:
  - `setAge(int age, int index)`
  - `setName(String name, int index)`
- An inconsistency example:
  - To get the size of a Collection
    - For an array A: A.length (public instance variable)
    - For an ArrayList v: v.size()
    - For a String s: s.length()
  - Avoid this kind of inconsistency as much as possible.