Module 9

Text Applications

Objectives

- Write a program that uses command-line arguments and system properties
- Write a program that reads from *standard input*
- Describe the C-type formatted input and output
- Write a program that can create, read, and write files
- Describe the basic hierarchy of collections in the Java 2 Software Development Kit (Java 2 SDK)
- Write a program that uses sets and lists
- Write a program to iterate over a collection
- Write a program that uses generic collections

Relevance

- It is often the case that certain elements of a program should not be hard-coded, such as file names or the name of a database. How can a program be coded to supply these elements at runtime?
- Simple arrays are far too static for most collections (that is, a fixed number of elements). What Java technology features exist to support more flexible collections?
- Besides computation, what are key elements of any text-based application?

Command-Line Arguments

- Any Java technology application can use command-line arguments.
- These string arguments are placed on the command line to launch the Java interpreter, after the class name:

```
java TestArgs arg1 arg2 "another arg"
```

- Each command-line argument is placed in the `args` array that is passed to the static `main` method:

```
public static void main(String[] args)
```

Command-Line Arguments

```
1   public class TestArgs {
2     public static void main(String[] args) {
3       for ( int i = 0; i < args.length; i++ ) {
4         System.out.println("args[" + i + "] is '" + args[i] + "'");
5       }
6     }
7   }
```

Example execution:

```
java TestArgs arg1 arg2 "another arg"
args[0] is 'arg1'
args[1] is 'arg2'
args[2] is 'another arg'
```

System Properties

- System properties are a feature that replaces the concept of *environment variables* (which are platform-specific).
- The `System.getProperties` method returns a `Properties` object.
- The `getProperty` method returns a `String` representing the value of the named property.
- Use the `-D` option to include a new property.

The `Properties` Class

- The `Properties` class implements a mapping of names to values (a `String` to `String` map).
- The `propertyNames` method returns an `Enumeration` of all property names.
- The `getProperty` method returns a `String` representing the value of the named property.
- You can also read and write a properties collection into a file using `load` and `store`.

The `Properties` Class

```
1   import java.util.Properties;
2   import java.util.Enumeration;
3
4   public class TestProperties {
5     public static void main(String[] args) {
6       Properties props = System.getProperties();
7       Enumeration propNames = props.propertyNames();
8
9       while ( propNames.hasMoreElements() ) {
10        String propName = (String) propNames.nextElement();
11        String property = props.getProperty(propName);
12        System.out.println("property '" + propName
13                           + "' is '" + property + "'");
14      }
15    }
16  }
```

The `Properties` Class

Here is an example test run of this program:

```
java -DmyProp=theValue TestProperties
```

Here is the (partial) output:

```
property 'java.version' is '1.5.0-rc'
property 'java.compiler' is 'NONE'
property 'path.separator' is ':'
property 'file.separator' is '/'
property 'user.home' is '/home/basham'
property 'java.specification.vendor' is 'Sun Microsystems Inc.'
property 'user.language' is 'en'
property 'user.name' is 'basham'
property 'myProp' is 'theValue'
```

Console I/O

- The variable `System.out` enables you to write to *standard output*.

  It is an object of type `PrintStream`.
- The variable `System.in` enables you to read from *standard input*.

  It is an object of type `InputStream`.
- The variable `System.err` enables you to write to *standard error*.

  It is an object of type `PrintStream`.

Writing to Standard Output

- The `println` methods print the argument and a newline character (`\n`).
- The `print` methods print the argument without a newline character.
- The `print` and `println` methods are overloaded for most primitive types (`boolean`, `char`, `int`, `long`, `float`, and `double`) and for `char[]`, `Object`, and `String`.
- The `print(Object)` and `println(Object)` methods call the `toString` method on the argument.

Reading From Standard Input

```
1   import java.io.*;
2
3   public class KeyboardInput {
4     public static void main (String args[]) {
5       String s;
6       // Create a buffered reader to read
7       // each line from the keyboard.
8       InputStreamReader ir
9         = new InputStreamReader(System.in);
10      BufferedReader in = new BufferedReader(ir);
11
12      System.out.println("Unix: Type ctrl-d to exit." +
13                         "\nWindows: Type ctrl-z to exit");
```

Reading From Standard Input

```
14      try {
15        // Read each input line and echo it to the screen.
16        s = in.readLine();
17        while ( s != null ) {
18          System.out.println("Read: " + s);
19          s = in.readLine();
20        }
21
22        // Close the buffered reader.
23        in.close();
24      } catch (IOException e) { // Catch any IO exceptions.
25        e.printStackTrace();
26      }
27    }
28  }
```

Simple Formatted Output

- You can use the formatting functionality as follows

```
out.printf("name count\n");
String s = String.format("%s %5d%n", user, total);
```

- Common formatting codes are listed in this table.

Code	Description
`%s`	Formats the argument as a string, usually by calling the `toString` method on the object.
`%d %o %x`	Formats an integer, as a decimal, octal, or hexadecimal value.
`%f %g`	Formats a floating point number. The `%g` code uses scientific notation.
`%n`	Inserts a newline character to the string or stream.
`%%`	Inserts the `%` character to the string or stream.

Simple Formatted Input

- The Scanner API provides a formatted input function.
- A Scanner can be used with console input streams as well as file or network streams.
- You can read console input as follows:

```
1   import java.io.*;
2   import java.util.Scanner;
3   public class ScanTest {
4     public static void main(String [] args) {
5       Scanner s = new Scanner(System.in);
6       String param = s.next();
7       System.out.println("the param 1" + param);
8       int value = s.nextInt();
9       System.out.println("second param" + value);
10      s.close();
11    }
12  }
```

Files and File I/O

The `java.io` package enables you to do the following:

- Create `File` objects
- Manipulate `File` objects
- Read and write to file streams

Creating a New `File` Object

The `File` class provides several utilities:

- `File myFile;`
- `myFile = new File("myfile.txt");`
- `myFile = new File("MyDocs", "myfile.txt");`

Directories are treated just like files in Java; the `File` class supports methods for retrieving an array of files in the directory, as follows:

```
File myDir = new File("MyDocs");
myFile = new File(myDir, "myfile.txt");
```

The `File` Tests and Utilities

- File information:

```
String getName()
String getPath()
String getAbsolutePath()
String getParent()
long lastModified()
long length()
```

- File modification:

```
boolean renameTo(File newName)
boolean delete()
```

- Directory utilities:

```
boolean mkdir()
String[] list()
```

The `File` Tests and Utilities

- File tests:

```
boolean exists()
boolean canWrite()
boolean canRead()
boolean isFile()
boolean isDirectory()
boolean isAbsolute();
```

File Stream I/O

- For file input:
  - Use the `FileReader` class to read characters.
  - Use the `BufferedReader` class to use the `readLine` method.
- For file output:
  - Use the `FileWriter` class to write characters.
  - Use the `PrintWriter` class to use the `print` and `println` methods.

File Stream I/O

A file input example is:

```
1   import java.io.*;
2   public class ReadFile {
3     public static void main (String[] args) {
4       // Create file
5       File file = new File(args[0]);
6
7       try {
8         // Create a buffered reader
9         // to read each line from a file.
10        BufferedReader in
11          = new BufferedReader(new FileReader(file));
12        String s;
13
```

File Stream I/O

```
14          // Read each line from the file and echo it to the screen.
15          s = in.readLine();
16          while ( s != null ) {
17            System.out.println("Read: " + s);
18            s = in.readLine();
19          }
20          // Close the buffered reader
21          in.close();
22
23        } catch (FileNotFoundException e1) {
24          // If this file does not exist
25          System.err.println("File not found: " + file);
26
27        } catch (IOException e2) {
28          // Catch any other IO exceptions.
29          e2.printStackTrace();
30        }
31      }
32    }
```

File Output Example

```
1   import java.io.*;
2
3   public class WriteFile {
4     public static void main (String[] args) {
5       // Create file
6       File file = new File(args[0]);
7
8       try {
9         // Create a buffered reader to read each line from standard in.
10        InputStreamReader isr
11          = new InputStreamReader(System.in);
12        BufferedReader in
13          = new BufferedReader(isr);
14        // Create a print writer on this file.
15        PrintWriter out
16          = new PrintWriter(new FileWriter(file));
17        String s;
```

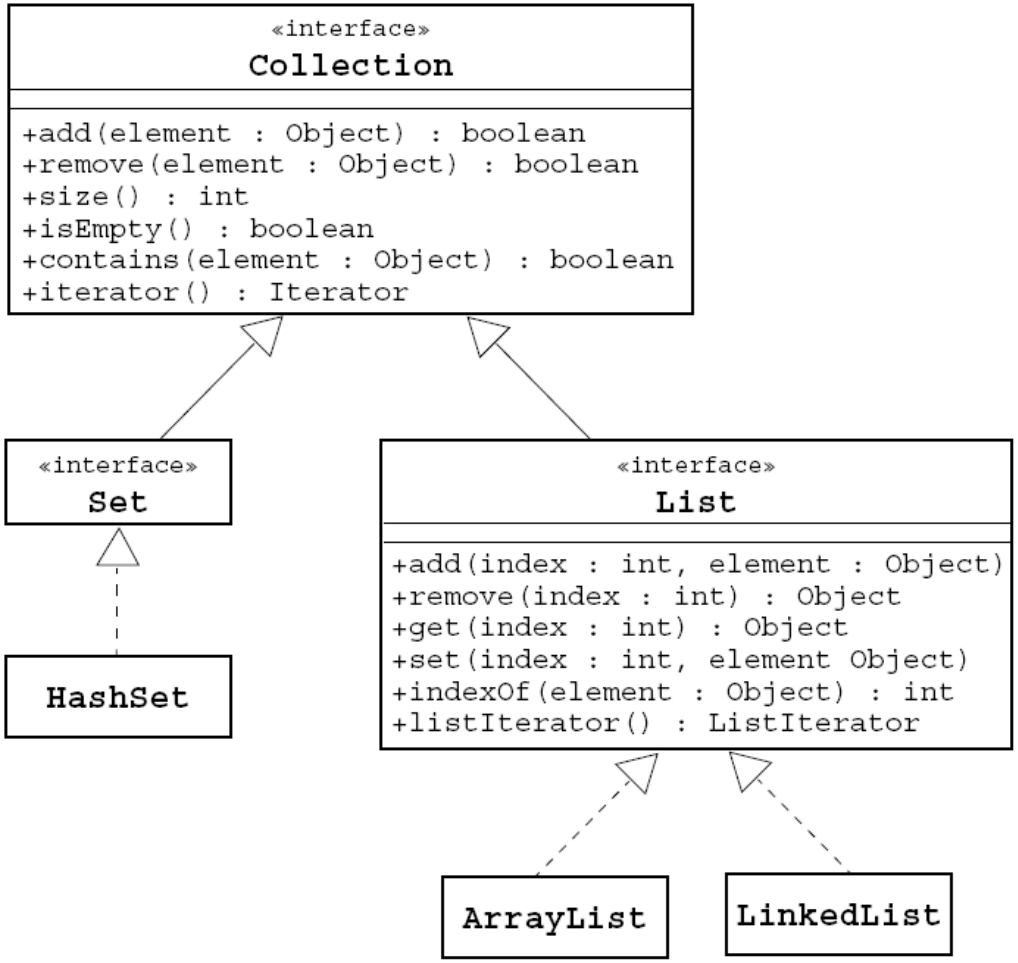
File Output Example

```
18
19          System.out.print("Enter file text.  ");
20          System.out.println("[Type ctrl-d to stop.]");
21
22          // Read each input line and echo it to the screen.
23          while ((s = in.readLine()) != null) {
24            out.println(s);
25          }
26
27          // Close the buffered reader and the file print writer.
28          in.close();
29          out.close();
30
31        } catch (IOException e) {
32        // Catch any IO exceptions.
33          e.printStackTrace();
34        }
35      }
36  }
```

The Collections API

- A *collection* is a single object representing a group of objects known as its elements.
- The Collections API contains interfaces that group objects as one of the following:
  - `Collection` – A group of objects called elements; any specific ordering (or lack of) and allowance of duplicates is specified by each implementation
  - `Set` – An unordered collection; no duplicates are permitted
  - `List` – An ordered collection; duplicates are permitted

The Collections API



A `Set` Example

```
1   import java.util.*;
2
3   public class SetExample {
4     public static void main(String[] args) {
5       Set set = new HashSet();
6       set.add("one");
7       set.add("second");
8       set.add("3rd");
9       set.add(new Integer(4));
10      set.add(new Float(5.0F));
11      set.add("second");          // duplicate, not added
12      set.add(new Integer(4));    // duplicate, not added
13      System.out.println(set);
14    }
15  }
```

The output generated from this program is:

```
[one, second, 5.0, 3rd, 4]
```

A `List` Example

```
1   import java.util.*
2
3   public class ListExample {
4     public static void main(String[] args) {
5       List list = new ArrayList();
6       list.add("one");
7       list.add("second");
8       list.add("3rd");
9       list.add(new Integer(4));
10      list.add(new Float(5.0F));
11      list.add("second");          // duplicate, is added
12      list.add(new Integer(4));    // duplicate, is added
13      System.out.println(list);
14    }
15  }
```

The output generated from this program is:

```
[one, second, 3rd, 4, 5.0, second, 4]
```

Collections in JDK™ Version 1.1

Collections in the Java Development Kit (JDK™) include:

- The class `Vector` implements the `List` interface.
- The class `Stack` is a subclass of `Vector` and supports the `push`, `pop`, and `peek` methods.
- The class `Hashtable` implements the `Map` interface.
- The `Enumeration` interface is a variation on the `Iterator` interface.

  An enumeration is returned by the `elements` method in `Vector`, `Stack`, and `Hashtable`.
- Classes are thread-safe, and therefore, *heavy weight*.
- These classes also support generics.

Generics

Generics are described as follows:

- Provides compile-time type safety
- Eliminates the need for casts

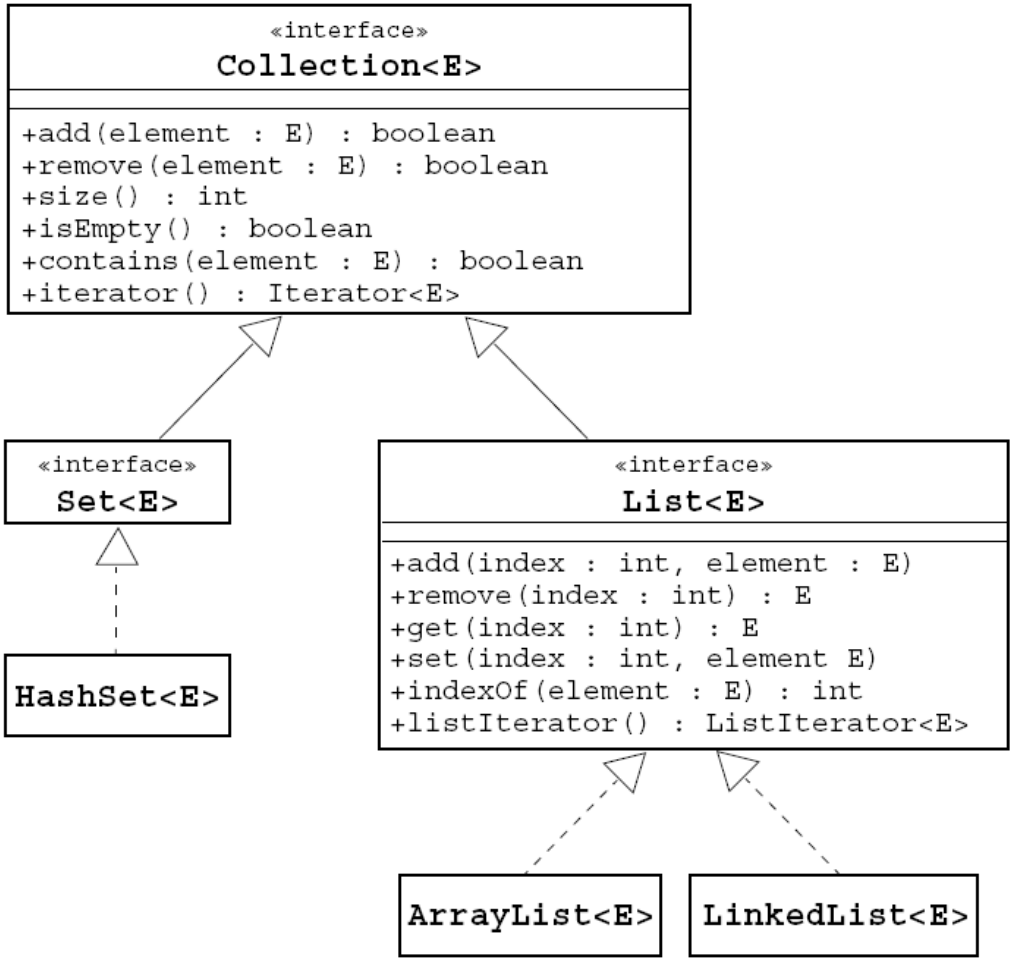
Before Generics

```
ArrayList list = new ArrayList();
list.add(0, new Integer(42));
int total = ((Integer)list.get(0)).intValue();
```

After Generics

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, new Integer(42));
int total = list.get(0).intValue();
```

Generic Collections API



Compiler Warnings

```
1   import java.util.*;
2   public class GenericsWarning {
3     public static void main(String[] args) {
4       List list = new ArrayList();
5       list.add(0, new Integer(42));
6       int total = ((Integer)list.get(0)).intValue();
7     }
8   }
```

javac GenericsWarning.java

```
Note: GenericsWarning.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

javac -Xlint:unchecked GenericsWarning.java

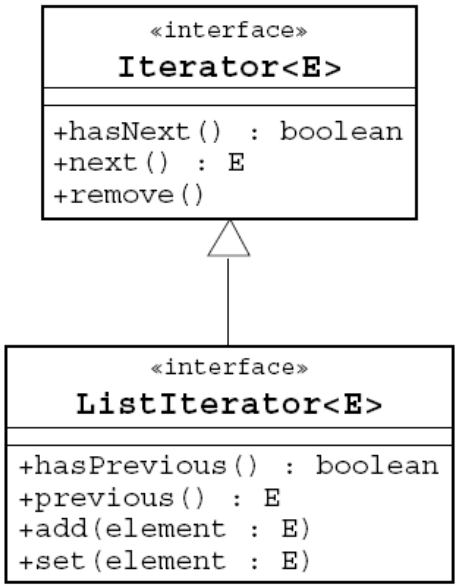
```
GenericsWarning.java:7: warning: [unchecked] unchecked call to add(int,E)
as a member of the raw type java.util.ArrayList
    list.add(0, new Integer(42));
            ^
1 warning
```

Iterators

- Iteration is the process of retrieving every element in a collection.
- An `Iterator` of a `Set` is unordered.
- A `ListIterator` of a `List` can be scanned forwards (using the `next` method) or backwards (using the `previous` method).

```
List list = new ArrayList();
// add some elements
Iterator elements = list.iterator();
while ( elements.hasNext() ) {
  System.out.println(elements.next());
}
```

The Iterator Interface Hierarchy



Enhanced `for` Loop

The enhanced `for` loop has the following characteristics:

- Simplified iteration over collections
- Much shorter, clearer, and safer
- Effective for arrays
- Simpler when using nested loops
- Iterator disadvantages removed

Iterators are error prone:

- Iterator variables occur three times per loop.
- This provides the opportunity for code to go wrong.

Enhanced `for` Loop

An enhanced `for` loop can look like this:

- Using iterators:

```
public void deleteAll(Collection<NameList> c){
  for ( Iterator<NameList> i = c.iterator() ; i.hasNext() ; ){
    NameList nl = i.next();
    nl.deleteItem();
  }
}
```

- Using enhanced `for` loop in collections:

```
public void deleteAll(Collection<NameList> c){
  for ( NameList nl : c ){
    nl.deleteItem();
  }
}
```

Enhanced `for` Loop

- Using enhanced `for` loop in arrays:

```
public int sum(int[] array){
  int result = 0;
  for ( int element : array ) {
    result += element;
  }
  return result;
}
```

- Using enhanced `for` loop in nested loops:

```
List<Subject> subjects=...;
List<Teacher> teachers=...;
List<Course> courseList = new ArrayList<Course>();
for ( Subject subj : subjects ) {
  for ( Teacher tchr : teachers ) {
    courseList.add(new Course(subj, tchr));
  }
}
```