core WEB programming

Basic Java Syntax

Agenda

- **Creating, compiling, and executing simple Java programs**
- **Accessing arrays**
- **Looping**
- **Using if statements**
- **Comparing strings**
- **Building arrays**
  - One-step process
  - Two-step process
- **Using multidimensional arrays**
- **Manipulating data structures**
- **Handling errors**

Getting Started

- **Name of file must match name of class**
  - It is case sensitive, even on Windows
- **Processing starts in main**
  - `public static void main(String[] args)`
  - Routines usually called "methods," not "functions."
- **Printing is done with System.out**
  - System.out.println, System.out.print
- **Compile with "javac"**
  - Open DOS window; work from there
  - Supply full case-sensitive file name (with file extension)
- **Execute with "java"**
  - Supply base class name (no file extension)

Example

- **File: HelloWorld.java**

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello, world.");
  }
}
```

- **Compiling**

```
DOS> javac HelloWorld.java
```

- **Executing**

```
DOS> java HelloWorld
Hello, world.
```

More Basics

- **Use + for string concatenation**
- **Arrays are accessed with []**
  - Array indices are zero-based
  - The argument to `main` is an array of strings that correspond to the command line arguments
    - args[0] returns first command-line argument
    - args[1] returns second command-line argument
    - Etc.
- **The length field gives the number of elements in an array**
  - Thus, `args.length` gives the number of command-line arguments
  - Unlike in C/C++, the name of the program is not inserted into the command-line arguments

Example

- **File: ShowTwoArgs.java**

```
public class ShowTwoArgs {
  public static void main(String[] args) {
    System.out.println("First arg: " +
                       args[0]);
    System.out.println("Second arg: " +
                       args[1]);
  }
}
```

Example (Continued)

- **Compiling**

```
DOS> javac ShowTwoArgs.java
```

- **Executing**

```
DOS> java ShowTwoArgs Hello World
First args Hello
Second arg: Class

DOS> java ShowTwoArgs
[Error message]
```

Looping Constructs

- **while**

```
while (continueTest) {
  body;
}
```

- **do**

```
do {
  body;
} while (continueTest);
```

- **for**

```
for(init; continueTest; updateOp) {
  body;
}
```

While Loops

```
public static void listNums1(int max) {
  int i = 0;
  while (i <= max) {
    System.out.println("Number: " + i);
    i++; // "++" means "add one"
  }
}
```

Do Loops

```
public static void listNums2(int max) {
  int i = 0;
  do {
    System.out.println("Number: " + i);
    i++;
  } while (i <= max);
                    // ^ Don't forget semicolon
}
```

For Loops

```
public static void listNums3(int max) {
  for(int i=0; i<max; i++) {
    System.out.println("Number: " + i);
  }
}
```

Aside: Defining Multiple Methods in Single Class

```
public class LoopTest {
  public static void main(String[] args) {
    listNums1(5);
    listNums2(6);
    listNums3(7);
  }

  public static void listNums1(int max) {…}
  public static void listNums2(int max) {…}
  public static void listNums3(int max) {…}
}
```

Loop Example

- **File ShowArgs.java:**

```
public class ShowArgs {
  public static void main(String[] args) {
    for(int i=0; i<args.length; i++) {
      System.out.println("Arg " + i +
                         " is " +
                         args[i]);
    }
  }
}
```

If Statements

- **Single Option**

```
if (boolean-expression) {
  statement;
}
```

- **Multiple Options**

```
if (boolean-expression) {
  statement1;
} else {
  statement2;
}
```

Boolean Operators

- **==, !=**
  - Equality, inequality. In addition to comparing primitive types, == tests if two objects are identical (the same object), not just if they appear equal (have the same fields). More details when we introduce objects.
- **<, <=, >, >=**
  - Numeric less than, less than or equal to, greater than, greater than or equal to.
- **&&, ||**
  - Logical AND, OR. Both use short-circuit evaluation to more efficiently compute the results of complicated expressions.
- **!**
  - Logical negation.

Example: If Statements

```
public static int max2(int n1, int n2) {
  if (n1 >= n2)
    return(n1);
  else
    return(n2);
}
```

Strings

- **String is a real class in Java, not an array of characters as in C and C++.**
- **The String class has a shortcut method to create a new object: just use double quotes**
  - This differs from normal objects, where you use the new construct to build an object
- **Use equals to compare strings**
  - Never use ==

Strings: Common Error

```
public static void main(String[] args) {
  String match = "Test";
  if (args.length == 0) {
    System.out.println("No args");
  } else if (args[0] == match) {
    System.out.println("Match");
  } else {
    System.out.println("No match");
  }
}
```

- **Prints "No match" for *all* inputs**
  - Fix:

```
if (args[0].equals(match))
```

Building Arrays: One-Step Process

- **Declare and allocate array in one fell swoop**

```
type[] var = { val1, val2, ... , valN };
```

- **Examples:**

```
int[] values = { 10, 100, 1000 };
Point[] points = { new Point(0, 0),
                   new Point(1, 2),
                   ... };
```

Building Arrays: Two-Step Process

- **Step 1: allocate an array of references:**
  type[] var = new *type*[size];
- **Eg:**
  int[] values = new int[7];
  Point[] points = new Point[someArray.length];
- **Step 2: populate the array**
  points[0] = new Point(...);
  points[1] = new Point(...);
  ...
  Points[6] = new Point(…);
- **If you fail to populate an entry**
  - Default value is 0 for numeric arrays
  - Default value is null for object arrays

Multidimensional Arrays

- **Multidimensional arrays are implemented as arrays of arrays**

```
int[][] twoD = new int[64][32];

String[][] cats = { { "Caesar",  "blue-point" },
                    { "Heather", "seal-point" },
                    { "Ted",     "red-point"  } };
```

- **Note: the number of elements in each row (dimension) need not be equal**

```
int[][] irregular = { { 1 },
                      { 2, 3, 4},
                      { 5 },
                      { 6, 7 } };
```

TriangleArray: Example

```
public class TriangleArray {
  public static void main(String[] args) {

    int[][] triangle = new int[10][];

    for(int i=0; i<triangle.length; i++) {
      triangle[i] = new int[i+1];
    }

    for (int i=0; i<triangle.length; i++) {
      for(int j=0; j<triangle[i].length; j++) {
        System.out.print(triangle[i][j]);
      }
      System.out.println();
    }
  }
}
```

TriangleArray: Result

```
> java TriangleArray

0
00
000
0000
00000
000000
0000000
00000000
000000000
0000000000
```

Data Structures

- **Java 1.0 introduced two *synchronized* data structures in the `java.util` package**
  - Vector
    - A strechable (resizeable) array of `Objects`
    - Time to access an element is constant regardless of position
    - Time to insert element is proportional to the size of the vector
    - In Java 2 (eg JDK 1.2 and later), use ArrayList
  - Hashtable
    - Stores key-value pairs as `Objects`
    - Neither the keys or values can be `null`
    - Time to access/insert is constant
    - In Java 2, use HashMap

Useful Vector Methods

- **addElement/insertElementAt/setElementAt**
  - Add elements to the vector
- **removeElement/removeElementAt**
  - Removes an element from the vector
- **firstElement/lastElement**
  - Returns a reference to the first and last element, respectively (without removing)
- **elementAt**
  - Returns the element at the specified index
- **indexOf**
  - Returns the index of an element that equals the object specified
- **contains**
  - Determines if the vector contains an object

Useful Vector Methods

- **elements**
  - Returns an `Enumeration` of objects in the vector

```
Enumeration elements = vector.elements();
while(elements.hasMoreElements()) {
  System.out.println(elements.nextElement());
}
```

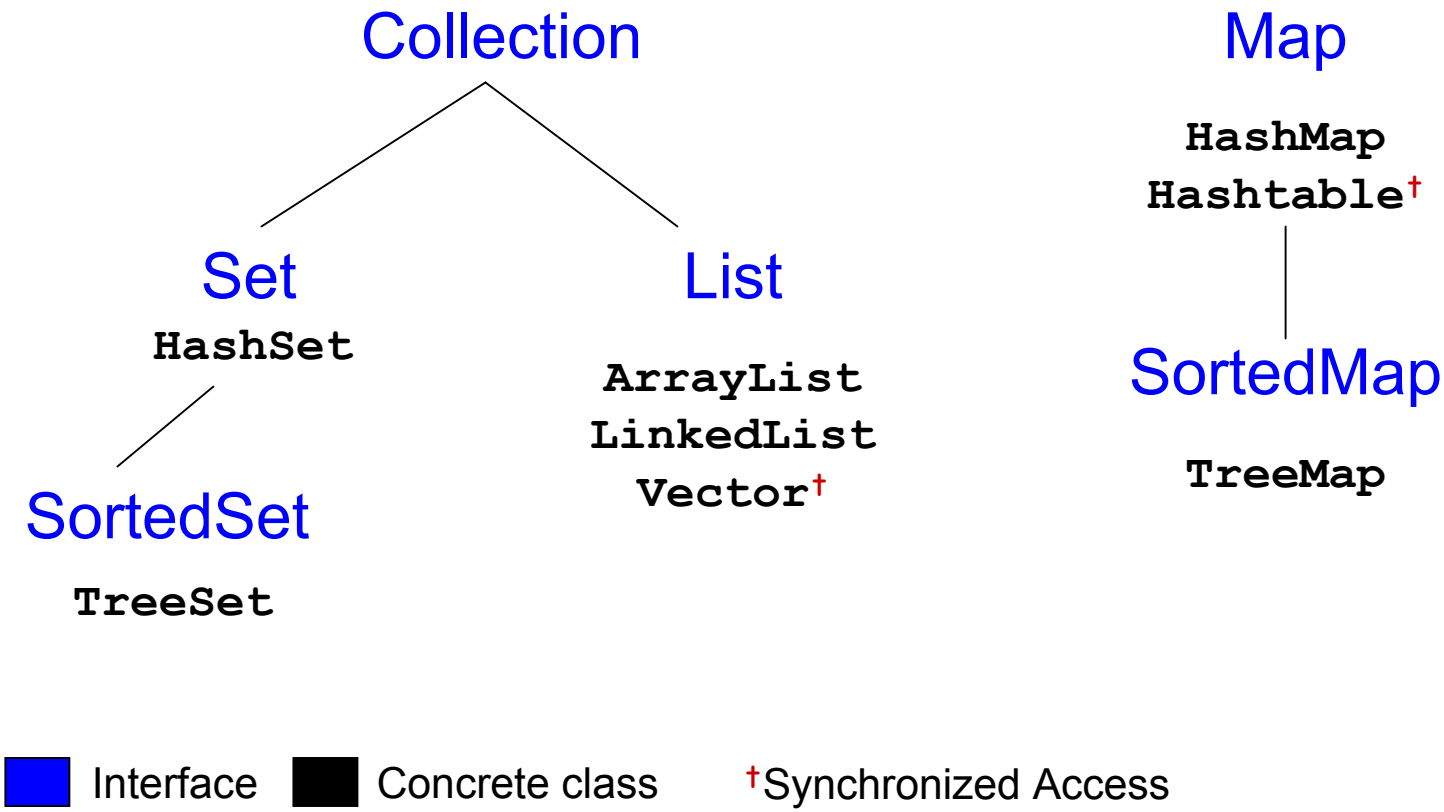
- **size**
  - The number of elements in the vector
- **capacity**
  - The number of elements the vector can hold before becoming resized

Useful Hashtable Methods

- **put/get**
  - Stores or retrieves a value in the hashtable
- **remove/clear**
  - Removes a particular entry or all entries from the hashtable
- **containsKey/contains**
  - Determines if the hashtable contains a particular key or element
- **keys/elements**
  - Returns an enumeration of all keys or elements, respectively
- **size**
  - Returns the number of elements in the hashtable
- **rehash**
  - Increases the capacity of the hashtable and reorganizes it

Collections Framework

- **Additional data structures added by Java 2 Platform**



Collection Interfaces

- **Collection**
  - Abstract class for holding groups of objects
- **Set**
  - Group of objects containing no duplicates
- **SortedSet**
  - Set of objects (no duplicates) stored in ascending order
  - Order is determined by a `Comparator`
- **List**
  - *Physically* (versus logically) ordered sequence of objects
- **Map**
  - Stores objects (unordered) identified by unique keys
- **SortedMap**
  - Objects stored in ascending order based on their key value
  - Neither duplicate or `null` keys are permitted

Collections Class

- **Use to create synchronized data structures**

```
List list = Collection.synchronizedList(new ArrayList());

Map map =  Collections.synchronizedMap(new HashMap());
```

- **Provides useful (static) utility methods**
  - sort
    - Sorts (ascending) the elements in the list
  - max, min
    - Returns the maximum or minimum element in the collection
  - reverse
    - Reverses the order of the elements in the list
  - shuffle
    - Randomly permutes the order of the elements

Wrapper Classes

- **Each primitive data type has a corresponding object (wrapper class)**

Primitive Data Type	Corresponding Object Class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

  - The data is stored as an immutable field of the object

Wrapper Uses

- **Defines useful constants for each data type**
  - For example,

```
Integer.MAX_VALUE
Float.NEGATIVE_INFINITY
```

- **Convert between data types**
  - Use `parseXxx` method to convert a `String` to the corresponding primitive data type

```
try {
  String value = "3.14e6";
  Double d = Double.parseDouble(value);
} catch (NumberFormatException nfe) {
  System.out.println("Can't convert: " + value);
}
```

Wrappers: Converting Strings

Data Type	Convert String using either …
byte	Byte.parseByte(*string*)
	new Byte(*string*).byteValue()
short	Short.parseShort(*string*)
	new Short(*string*).shortValue()
int	Integer.parseInteger(*string*)
	new Integer(*string*).intValue()
long	Long.parseLong(*string*)
	new Long(*string*).longValue()
float	Float.parseFloat(*string*)
	new Float(*string*).floatValue()
double	Double.parseDouble(*string*)
	new Double(*string*).doubleValue()

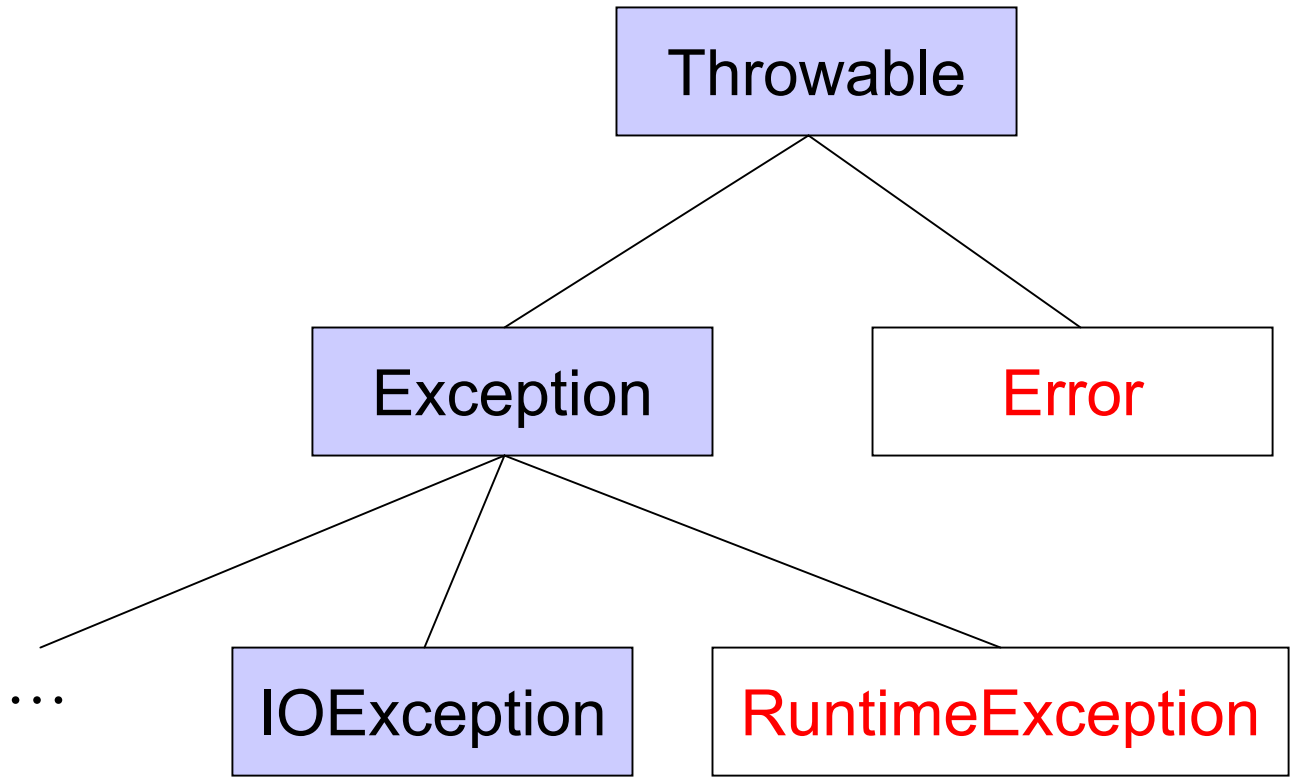
Error Handling: Exceptions

- **In Java, the error-handling system is based on exceptions**
  - Exceptions must be handed in a `try`/`catch` block
  - When an exception occurs, process flow is immediately transferred to the `catch` block
- **Basic Form**

```
try {
  statement1;
  statement2;
  ...
} catch(SomeException someVar) {
  handleTheException(someVar);
}
```

Exception Hierarchy

- **Simplified Diagram of Exception Hierarchy**



Throwable Types

- **Error**
  - A non-recoverable problem that should not be caught (OutOfMemoryError, StackOverflowError, …)
- **Exception**
  - An abnormal condition that should be caught and handled by the programmer
- **RuntimeException**
  - Special case; does not have to be caught
  - Usually the result of a poorly written program (integer division by zero, array out-of-bounds, etc.)
    - A `RuntimeException` is considered a bug

Multiple Catch Clauses

- **A single `try` can have more that one `catch` clause**

```
try {
  ...
} catch (ExceptionType1 var1) {
  // Do something
} catch (ExceptionType2 var2) {
  // Do something else
}
```

  - If multiple `catch` clauses are used, order them from the most specific to the most general
  - If no appropriate `catch` is found, the exception is handed to any outer `try` blocks
    - If no `catch` clause is found within the method, then the exception is thrown by the method

Try-Catch, Example

```
...
BufferedReader in = null;
String lineIn;
try {
  in = new BufferedReader(new FileReader("book.txt"));
  while((lineIn = in.readLine()) != null) {
    System.out.println(lineIn);
  }
  in.close();
} catch (FileNotFoundException fnfe ) {
  System.out.println("File not found.");
} catch (EOFException eofe) {
  System.out.println("Unexpected End of File.");
} catch (IOException ioe) {
  System.out.println("IOError reading input: " + ioe);
  ioe.printStackTrace(); // Show stack dump
}
```

The finally Clause

- After the final `catch` clause, an optional `finally` clause may be defined
- The `finally` clause is always executed, even if the `try` or `catch` blocks are exited through a `break`, `continue`, or `return`

```
try {
  ...
} catch (SomeException someVar) {
  // Do something
} finally {
  // Always executed
}
```

Thrown Exceptions

- **If a potential exception is not handled in the method, then the method must declare that the exception can be thrown**

```
public SomeType someMethod(...) throws SomeException {
  // Unhandled potential exception
  ...
}
```

  - Note: Multiple exception types (comma separated) can be declared in the `throws` clause

- **Explicitly generating an exception**

```
throw new IOException("Blocked by firewall.");

throw new MalformedURLException("Invalid protocol");
```

Summary

- **Loops, conditional statements, and array access is the same as in C and C++**
- **`String` is a real class in Java**
- **Use `equals`, not `==`, to compare strings**
- **You can allocate arrays in one step or in two steps**
- **`Vector` or `ArrayList` is a useful data structure**
  - Can hold an arbitrary number of elements
- **Handle exceptions with `try/catch` blocks**

core WEB programming

Questions?