Agenda: Discussion Week 7
May 11, 2009

- Method signatures
- Static vs. instance
- compareTo
- Exceptions
- Interfaces
- 2-D arrays
- Recursion
- varargs
- enum
- Suggestions?
Method signatures

[protection] [static] [return type] [method name] ([argument 1], [argument 2], ...)

argument = [argument type] [argument name]

public class myClass {
    public void myMethod() {}
    private static void myMethod() {}
    public void myMethod(int i) {}
    private int myMethod() {return 1;}
    public boolean myMethod(int i, String s) {return true;}
}

Static vs. Instance: Variables

• Instance variables (default)
  – Associated with an object
  – Each object has distinct copies of instance variables

• Static variables (aka “class variables”)
  – Associated with the class, rather than with any object
  – Every instance of the class shares a class variable, which is in one fixed location in memory

• Reference
Static vs. Instance: Methods

• Instance methods (default)
  – Associated with an object
  – Can use the instance variables of that object

• Static methods
  – Associated with the class, rather than with any object
  – Can’t use the instance variables of any object of the class
compareTo

• method in Object class
• int compareTo(Object o)
• i.compareTo(j)
  – returns -1 if i < j  // “a”.compareTo(“b”)
  – returns 0 if i equals j  // “a”.compareTo(“a”)
  – returns 1 if i > j  // “b”.compareTo(“a”)

Exceptions

• try / catch / finally
  – Example: try{} catch (IOException ioe) {} finally {}  
• throw / throws  
  – Example: public void myMethod() throws IOException {}  
• Checked and unchecked exceptions  
  – Unchecked  
    • Compiler does not check for handlers (don’t need throws)  
    • RuntimeException, Error, and all their subclasses  
  – Checked  
    • Compiler checks for handlers (must use throws)  
    • All other exception classes
Interface (1)

• Defines a way to interact, but doesn’t define implementation
  – Has method headers, doesn’t have method bodies
  – `interface` keyword
• A class can implement an interface
  – Must define methods specified in interface
  – Can implement multiple interfaces
  – `implements` keyword

```java
public interface myInterface {
    public void myMethod();
}

public class myClass implements myInterface {
    public void myMethod() {}
}
```
public interface Hello {
    public void sayHello();
}

public class English implements Hello { // type English and type Hello
    public void sayHello() { System.out.println("Hello"); }
}

public class Spanish implements Hello { // type Spanish and type Hello
    public void sayHello() { System.out.println("Hola"); }
}

English english = new English(); english.sayHello(); // prints “Hello”
Spanish spanish = new Spanish(); spanish.sayHello(); // prints “Hola”;
Hello hello = new English(); hello.sayHello(); // prints “Hello”;
hello = new Spanish(); hello.sayHello(); // prints “Hola”;}
2-D arrays: Rectangular

- Rectangular arrays
  - can have a different number of elements in each row of a 2-D array

```java
int rows = 3;
int columns = 2;
String[][] rectangular2dArray = new String[rows][columns];
for (int r = 0; r < rows; r++) {
    for (int c = 0; c < columns; c++) {
        rectangular2dArray[r][c] = "(" + r + "," + c + ")";
    }
}
```

<table>
<thead>
<tr>
<th>Rows</th>
<th>Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>(0,1)</td>
</tr>
<tr>
<td>(1,0)</td>
<td>(1,1)</td>
</tr>
<tr>
<td>(2,0)</td>
<td>(2,1)</td>
</tr>
</tbody>
</table>
2-D arrays: Ragged

• Ragged arrays
  – can have a different number of elements in each row of a 2-D array

```java
int rows = 3;
String[][] ragged2dArray = new String[rows][];
for (int r = 0; r < rows; r++) {
    int columns = r + 1;
    ragged2dArray[r] = new String[columns];
    for (int c = 0; c < columns; c++) {
        ragged2dArray[r][c] = "(" + r + "," + c + ")";
    }
}
```

<table>
<thead>
<tr>
<th>Rows</th>
<th>Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td></td>
</tr>
<tr>
<td>(1,0)</td>
<td>(1,1)</td>
</tr>
<tr>
<td>(2,0)</td>
<td>(2,1)</td>
</tr>
</tbody>
</table>
Recursion

• Method reuse (method calls itself)
  – Base case: smallest part with known solution
  – Recursive case: break problem into smaller parts

• Loops and recursion
  – All recursive code can be implemented with loops
  – Looping: aka “iteration”
Sum of arithmetic sequence: Overview

- Sum of arithmetic sequence: $1+2+3+\ldots+n$
- Looping
  - Add to sum inside loop
  - After loop, return sum
- Recursion
  - Base case: When $n$ is 1, return 1
  - Recursive case:
    - Make recursive call on $n-1$
    - Return sum of $n$ and value returned from recursive call
Sum of arithmetic sequence: Looping

- Sum of arithmetic sequence: $1+2+3+\ldots+n$
- Looping
  - Add to sum inside loop
  - After loop, return sum

```java
def sumSequence(int n):
    sum = 0
    for i in range(1, n+1):
        sum += i
    return sum
```
Sum of arithmetic sequence: Recursion

- Sum of arithmetic sequence: $1+2+3+\ldots+n$
- Recursion
  - Base case: When $n$ is 1, return 1
  - Recursive case:
    - Make recursive call on $n-1$
    - Return sum of $n$ and value returned from recursive call

```java
public int sumSequence(int n) {
    if (n == 1) {  // Base case
        return 1;
    } else {  // Recursive case
        return n + sumSequence(n-1);
    }
}
```
Example: Binary search

• Binary search
  – Analogy: Finding a person’s number in the phone book

• Looping
  – Start at the beginning, look at each person’s name until you find a match

• Recursion
  – Base case
    • When size of section is 1, check if name matches
    • If it matches, return phone number, otherwise return -1
  – Recursive case
    • Divide section in half, then make recursive call to look left or right (of the section’s mid-point)
varargs

- Can pass in a variable number of arguments
  - Separate by commas
  - Passes arguments as an array

- Example
  - public static average(Double... grades)
  - average(grade0, grade1, grade2)
  - Passes in Double[] grades
    - grades[0] is grade0
    - grades[1] is grade1
    - grades[2] is grade2
enum type

• Enumeration
  – For fixed number of fixed values

• Example
  enum Day {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY} // Define enum
  Day Monday = Day.MONDAY; // Refer to element in enum
  for(Day day : Day.values()) { // for-each loop through enum
    System.out.println(day);
  }