Lecture Set 6: Static Methods & Variables and Exceptions

1. Parameter Passing
2. Static variables and static methods
3. Exceptions

Parameter Passing

- Parameter List
- Names of Parameters
- Primitive type parameters
- Reference type parameters

- (See parameter passing example in CVS)

Why Have Static Variables / Methods?

- Sometimes info needs to be shared data among all objects of a specific class type
  - e.g. How many objects in a class have been created?
  - A constant that needs to be the same for all objects of that type
- Sometimes it is useful to have methods that are in a class that can be invoked without first creating objects of that type
- Static components help for these types of things

Declaring Static Methods (and variables and constants)

- Static methods

```
public static void main (…) { … }
public static void drawFlag(MyGrid grid, int
  Ccode) { … }
```

- How do we call static methods?

```
FlagMaker.drawFlag(grid, 1);
```

- Can have static variables and constants too

```
public static int numStudents = 0;
public static final int MAX_ENROLLMENT = 10;
```

- How do we use static variables and constants?

```
StudentRoster.numStudents
StudentRoster.MAX_ENROLLMENT
```

When To Use Static Methods?

- When a method should be invocable without object creation
- When a method should not change instance variables
  - A static method can only change static variables
  - Instance variables can only be changed by non-static methods

Default Values

- Static and instance variables
  - initialized
    - most types to 0
    - char to value 0 (non-printable character)
    - strings are assigned to null
- Local variables do not have a default value and you get a error from eclipse

Calling one method from another – static and non-static

- static methods
  - when running they ARE NOT associated with a specific instance
  - you do NOT have a "current object" but you do have a current class
  - are usually called with: ClassName.sMethodName()
  - if you are already in a static method, since you have a class name understood as the default, you can just use sMethodName()
- non-static methods
  - when running they ARE associated with a specific instance
  - you do have a "current object"
  - are called with: objectName.nsMethodName()
  - if you are already in a non-static method, since you have a current object assumed, you can just use nsMethodName() to call it on that current object
  - since that non-static method must also be in the class, the class name is also understood as the default so you can use sMethodName() to call the static member of that class

Exceptions

- Programs can generate errors
  - Arithmetic
    Divide by zero, overflows, …
  - Object / Array
    Using a null reference, illegal array index, …
  - File and I/O
    Nonexistent file, attempt to read past the end of the file, (we'll see more about file I/O later in course), …
  - Application-specific
    Errors particular to application (e.g., attempt to remove a nonexistent customer from a database)
- In Java: error = exception
- What to do when an error occurs?
  1. Basically ignore it: Print an error message and terminate?
  2. Have the method handle it internally: Handle error in the code where the problem lies as best you can.
  3. Have the method pass it off to someone else to handle: Return "error code" so that whoever called this function can handle it.
  4. Modern language approach: Cause "exception" to be thrown (and caught (or processed) by any function up the stack trace)

Exception Behavior

- If program generates ("throws") exception then default behavior is:
  - Java clobbers ("aborts") the program
  - Stack trace is printed showing where exception was generated (red and blue in Eclipse window)
- Example

```
public static int mpg(int miles, int
gallons){
  return miles/gallons;
}
```

- Throws an exception and terminates the program.

Throwing Exceptions Yourself

- To throw an exception, use throw command:

```
throw e;
```

  e must evaluate to an exception object
- You can create exceptions just like other objects, e.g.:

```
RuntimeException e = new RuntimeException("Uh oh");
```

  - `RuntimeException` is a class
  - Calling new this way invokes constructor for this class
  - `RuntimeException` generalizes other kinds of exceptions (e.g. `ArithmeticException`)

Exceptions, Classes and Types

- Exceptions are objects
- Some examples from the Java class library (mostly java.lang):
  - `ArithmeticException`: Used e.g. for divide by zero
  - `NullPointerException`: attempt to access an object with a null reference
  - `IndexOutOfBoundsException`: array or string index out of range
  - `ArrayStoreException`: attempting to store wrong type of object in array
  - `EmptyStackException`: attempt to pop an empty Stack (java.util)
  - `IOException`: attempt to perform an illegal input/output operation (java.io)
  - `NumberFormatException`: attempt to convert an invalid string into a number (e.g., when calling Integer.parseInt( ))
  - `RuntimeException`: general run-time error (subsumes above)
  - `Exception`: The most generic type of exception

Java Exceptions in Detail

- Exceptions are (special) objects in Java
  - They are created from classes
  - The classes are derived ("inherit") from a special class, Throwable
  - We will learn more about inheritance, etc., later
- Every exception object / class has:
  - `Exception(String message)`
    Constructor taking an explanation as an argument
  - `String getMessage()`
    Method returning the embedded message of the exception
  - `void printStackTrace()`
    Method printing the call stack when the exception was thrown

Handling Exceptions

- Aborting program not always a good idea
  - E-mail: can't lose messages
  - E-commerce: must ensure correct handling of private info in case of crash
  - Antilock braking, air-traffic control: must recover and keep working
- Java includes provides the programmer with mechanisms for recovering from exceptions

Java Exception Terminology

- When an anomaly is detected during program execution, the JVM throws a particular type of exception
  - There are built-in exceptions
  - Users can also define their own (more later)
- To avoid crashing, a program can catch a thrown exception (if it isn't caught – you see the red and blue messages – stack trace)
- An exception generated by a piece of code can only be caught if the program is alerted. This process is called trying the piece of code.

Exception Propagation

- In previous example:
  - Exception thrown in one method …
    … but caught in another
  - Java uses exception propagation to look for exception handlers
  - When an exception occurs, Java pops back up the call stack to each of the calling methods to see whether the exception is being handled (by a try-catch block). This is exception propagation
  - The first method it finds that catches the exception will have its catch block executed. Execution resumes normally in the method after this catch block
  - If we get all the way back to main and no method catches this exception, Java catches it and aborts your program

Exception Handling: Example

- `DateReader.java`
  - Prompts user for a date in mm/dd/yyyy format
  - Prints year
- Program uses:
  - `substring` method
    May throw `IndexOutOfBoundsException`
  - `Integer.parseInt` method
    May throw `NumberFormatException`
  - `getYear` method (if d is null)
    May throw NullPointerException
- How do we know about these exceptions? Javadoc!

`http://java.sun.com/j2se/1.5.0/docs/api/java/lang/package-summary.html`

Javadoc Documentation Standard

- When documenting a method, list exceptions that method can throw
  - Use `@exception` tag
  - Be sure to include unhandled exceptions that operations in method may throw
- Example:

```
/**
 * Returns the year part of a date string
 * @param d date string in mm/dd/yyyy format
 * @return an integer representing the date
 * @exception IndexOutOfBoundsException
 * @exception NumberFormatException
 */
public static int getYear(String d) {
  …
}
```