Computer Programming – CS 6011
Lecture 5: Static - Exceptions

Fall 2023

Lecture 5

- Topics
  - Constants in Java
  - Static Keyword
  - Exceptions

Constants in Java

- Use the keyword *final**
  - `public final int MAX_SIZE = 100;`
  - `Similar to` **`const`** `in C++`

Static keyword (does not mean final)

- Static means “only one of”
  - Static Methods
  - Static Variables
- A *static* method is a function that does not have an object attached to it.
  - *Static* methods do not require an object to call.
- Sometimes we do want a “free floating” function. The best we can do in Java is to create a *static* class method.

Static Variables

- Static Variables
  - `public static int x;` //Global variable for the class
  - If you get a compiler hint to make a variable static, it probably means you are approaching the problem incorrectly.
  - The variable x is shared by all objects.
  - For normal member variables, each object has its own copy of the variable.

Static Method Examples

- `Math.sqrt( 10 );  // This is a static method within the Math class.`
- `Integer.parseInt()`
- `Integer.toString()`
- `Integer x = 100;`
- `String s = x.toString();`
- `Integer.toBinaryString()`
- `Integer.toHexString()`

- We don't need to create a Math object to use it:
  - `Math m = new Math();  //WRONG approach for static methods.`
  - `m.sqrt( 10 );  //We do not call static methods like this.`

How to create a Static Method?

```
class MyClass {
        public static void myStaticFunction( … ) {
        }
}
```

- To call a static method, we use:

```
1) MyClass.someStaticmethod() //or
2) someStaticmethod() // when inside the class

// Again, a static method does not "belong" to a specific object, it
belongs to the class.
```

Exceptions and Error Handling

Error Handling

- How do functions let the caller know that there is an error?
  - Return code.
- File openFile(String filename) { return null; }
- int readArrayElement (int index) { return -1;}
- We need another (better) way to send error conditions out of functions (method), constructors, etc.

Exceptions (Error Handling)

- Exceptions are a way of handling unexpected conditions / errors
- They are especially useful when an error is detected in a method, but there isn't enough information there to properly deal with it.
- ***throws an exception*** – means a piece of code has run into a problem and an exception has occurred and does not know what to do about it.
- ***catches an exception*** – means the code as seen than an exception was thrown and is going to attempt to handle it. (We also sometimes say: handle the exception.)
  - Catching an exception is done in a *try / catch* block.
  - Try / Catch blocks can have multiple sections in there are different types of exceptions you are going to handle.

Advantages of Exceptions

- Prevents the termination of the program when exception happens
- Separating Error-Handling Code from the remaining Code
- Grouping/Categorizing Error types
- Propagating Errors up the Call Stack

Separating Error-Handling Code from the remaining Code

```
open the file;
if (theFileIsOpen) {
      determine the length of the file;
       if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                 read the file into memory;
                 if (readFailed) {
                      errorCode = -1;
                   }
                } else {
                        errorCode = -2;
                }
         } else {
                errorCode = -3;
         }
         . . .
```

Separating Error-Handling Code from the remaining Code

```
readFile {
    try {
         open the file;
         read the file into memory;
         close the file;
    } catch (fileOpenFailed) {
       doSomething;
    } catch (readFailed) {
       doSomething;
    } catch (fileCloseFailed) {
       doSomething;
    }
}
```

Grouping/Categorizing Error types

```
try {
        // code that might cause an exception
}
catch( FileNotFoundException e ) {
        // Code to handle a specific exception of type File Not Found
}
catch( IoException e ) {
        // Code to handle a specific exception of type I/O Exception
}
catch ( Exception e ) {
        // Code to handle exceptions that are not of the above
}
```

Grouping/Categorizing Error types

```
try {
        // code that might cause an exception
}
catch( ArithmeticException e ) {
        // raised when an error occurs in arithmetic operations
}
catch(ClassNotFoundException e) {
        // when trying to access a class which is not found
}
catch(ArrayIndexOutOfBoundsException e) {
        // illegal index
}
```

Propagating Errors up the Call Stack

- Without exceptions, `g()` and `h()` are required to propagate the error codes returned by `readfile()` up the call stack until it reaches `f()`

```
void f() {
    g();
}

void g() {
   h();
   Error = call h();
   If error
      return error
   else
      proceed;
}
```

```
void h() {
   readfile();
   Error = call h();
   If error
      return error
   else
      proceed;
}
```

Propagating Errors up the Call Stack

- The methods that care about errors have to worry about detecting errors.

```
void f() {
    try {
          g();
    } catch (Exception e) {
      ProcessTheError or Report it
    }
}
void g() {
   h();
}
void h() {
   methodThatmighThrowException();
}
```

```
void f() {
    try {
          g();
    } catch (Exception e) {
      ProcessTheError or Report it
    }
}
void g() throws Exception {
   h();
}
void h() throws Exception{
   methodThatmighThrowException();
}
```

Exception Example

```
public static void main(String args[]) {
 System.out.print("Enter a number: ");
        Scanner sc = new Scanner(System.in);
        int num = sc.nextInt();
        try {
            int x = getElement(num);
            System.out.println("Result: " + x);
        } catch (IndexOutOfBoundsException e) {
            System.out.println("Exception message: " +
                    e.getMessage());
        }
        print();
    }
    static int getElement(int num) {
        ArrayList<Integer> ll = new ArrayList<>(num);
        return ll.get(0);
    }

    static void print() {
        System.out.println("Normal flow");
    }

}
```

$ Enter a number: 0

$ Exception message: Index 0 out of bounds for length 0

$ Normal flow

Checked Exception

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

    try {
        OpenFile();
    } catch (FileNotFoundException e) {
        System.out.println("Exception message: " + e.getMessage());
    }
}


 static void openFile() throws FileNotFoundException {//throws is mandatory because this type of exception is checked
    String str = "/users/somefile.txt";
        FileReader rd = new FileReader(str);
}

}
```

//**alternative:** use try-catch in openFile method instead of throws and remove it from main.

Unchecked Exception

```
public static void main(String args[]) {
 System.out.print("Enter a number: ");
        Scanner sc = new Scanner(System.in);
        int num = sc.nextInt();
        try {
            int x = getElement(num);
            System.out.println("Result: " + x);
        } catch (IndexOutOfBoundsException e) {
            System.out.println("Exception message: " +
                    e.getMessage());
        }
        print();
    }
    static int getElement(int num) throws IndexOutOfBoundsException { //throws is not mandatory because this type of exception is unchecked
        ArrayList<Integer> l1 = new ArrayList<>(num);
        return l1.get(0);
    }

    static void print() {
        System.out.println("Normal flow");
    }

}
```

Checked vs Unchecked Exceptions

- The main difference is that with checked exceptions, the compiler requires you to acknowledge at compile-time, while with unchecked exceptions it does NOT.

- Checked Exceptions:
  - FileNotFoundException, NoSuchFieldException, ClassNotFoundException, …

- Unchecked Exceptions:
  - ArithmeticException, NullPointerException, ArrayIndexOutofBoundsException, …

Useful Data Within An Exception

- Empty catch blocks means something wrong in your code
- Exceptions are objects, and thus have useful methods associated with them.
- Several of the useful methods are:
  - printStackTrace()
    - This method will display the call stack down to where the exception occurred (and is automatically called when an exception “escapes” from main (ie, when an exception is not handled by any part of the code).
  - getMessage()
    - This method returns the message stored in the exception (detailing what happened that caused the exception).

Friday Assignment(s)

- Code Reviews:
  - Fractions and Rainfall
  - Basic HTTP Server
- Lab - Exceptions
- Assignment – Server Refactoring