CMPE212 – Reminders

- Assignment 2 is posted.
- Quiz 1 tomorrow. Described in last Thursday’s lecture. Quiz topic coverage ends on the material covered on Monday.

Today

- Exceptions, Catching and Throwing.
  - Aliasing Objects.
  - Passing by Reference.

Exceptions

- How can a method indicate that it is unable to return what it is supposed to return?
- How can a method deliver details about the error condition?
- How can you prevent the instantiation of an Object?

The limitation of only returning a single “thing” means that you either designate error values for the “thing” or you have some other way to return the indication of an error.

Exceptions - Cont.

- The designers of Java followed conventions used by many other OOP languages - they allowed for another way to get something out of a method. However, an exception is thrown, not returned.
- Exceptions are Objects (big surprise!).
- When an error condition is encountered, a method can throw an instance of a pre-defined exception Object.
- A method can throw several exceptions, one for each possible kind of error condition.

Exceptions - Cont., Catching

- This is done with a “try/catch” block (see the next slide for the syntax).
- The compiler will force you to use try/catch blocks when you invoke methods that throw exceptions.
- (Eclipse can help build a try/catch:
  - Select the code to be surrounded.
  - Right click and choose “Surround with”, then “Try/catch block.”
  - The wizard will automatically choose all the possible exceptions.)

Exceptions – Cont.

- Syntax of a “try-catch block”:

```java
try {
    // block of statements that might
    // generate an exception
} catch (exception_type identifier) {
    // block of statements
} catch (exception_type identifier) {
    // block of statements
}...[
}]] finally {
    // block of statements
}]
```
Exceptions – Cont.

- You must have at least one “catch block” after the “try block” (otherwise the try block would be useless!)
- You can have many catch blocks, one for each exception you are trying to catch.
- The code in the “finally” block, if you have one, is always executed, whether an exception is thrown, caught, or not.

Checked vs Unchecked Exceptions

- Checked exceptions must either be caught in a method or thrown from that method (using the throws clause in the method header).
  Examples:
  IOException, FileNotFoundException, ClassNotFoundException
- You should not catch an un-checked exception or an Error.
  Examples:
  OutOfMemoryError, StackOverflowError, VirtualMachineError

Checked vs Unchecked Exceptions, Cont.

- The compiler will ensure that checked exceptions are handled properly.
- Unchecked exceptions occur only at runtime and the compiler does not care about whether you try to catch them or not. But you should not catch them in any case!
- Unchecked exceptions are all sub-classes of java.lang.Error

Try With Resources

- This new syntax is very useful with Java 7 & newer versions’ improved file I/O syntax.
- More on this topic in exercise 4.

Try With Resources – Cont.

- Syntax of a “try-with-resources block”:

```java
try (instantiation; instantiation; ..) {
    // other statements that might // generate an exception
} catch (exception_type identifier) {
    // block of statements
} catch (exception_type identifier) {
    // block of statements
} finally {
    // block of statements
}
```

- The instantiation(s) inside the set of ( ) immediately after the try keyword are declared resources that must be local to the try/catch block.
- Note that there is no ; at the end of the list and that there does not have to be any catch blocks.
- These resources must all implement the AutoCloseable interface, which means that the try block can close the resource when it is finished.
- Resources will be closed whether or not an exception is thrown because their scope is forced to be in the try block only.
- As a result, all use of the resource must also take place in the try block.
Throwing an Exception

- Inside a method somewhere when you have detected a problem:
  
  ```java
  throw new ExceptionClass([message]);
  ```

- The optional message is a String.

- At the end of the method header, you must include the clause:
  ```java
  throws ExceptionClass
  ```

- ExceptionClass can be an exception we have built or one from the API.

Building an Exception Class

- We’ll do this when we start learning how to build our own classes.

One-Dimensional Arrays - Declaration

- As we have already seen - To create an array to hold 10 integers:

  ```java
  int[] testArray = new int[10];
  ```

- `testArray` now points to an area of memory that holds locations for 10 integers.

- It also points to one location that holds `testArray.length`, which is an attribute of the array, that is equal to the number of elements.

- Arrays are Objects in Java.

Aside – Java ≥10 Array Declaration

- This:

  ```java
  int[] testArray = new int[10];
  ```

- Can now be simplified to:

  ```java
  var testArray = new int[10];
  ```

- Reduces redundant declarations in the instantiation statement.

Modelling Pointers

- Use the array as an example to help model the behaviour of a pointer.

- Note that an array is an example of a mutable object – object contents can be changed by accessing the object’s contents through the pointer.

- Object design or class design determines mutability.

One-Dimensional Arrays - Declaration, Cont.

- As a “pointer”, `testArray` points to an area of memory that contains a series of int values as well as the attribute `length`.

One-Dimensional Arrays - Declaration, Cont.

- The java statement above can be split into two:

```java
int[] testArray;
testArray = new int[10];
```

- The first statement creates a variable of type `int[]` - that is to say that it will be an array of `int`'s.
- The variable, `testArray` is now an object of type `int[]` that contains an "object reference" or a pointer. The object reference is null after the first statement.

A “Null Pointer”

- After `int[] testArray;`

- After `testArray = new int[10];`

One-Dimensional Arrays - Cont.

- The array indices allow mutability. Memory locations are calculated by offsets from the base address:

```
testArray[0]
testArray[1]
testArray[2]
testArray[3]
testArray[4]
testArray[5]
testArray[6]
testArray[7]
testArray[8]
testArray[9]
```

Multi-Dimensional Arrays

- Consider:

```java
int[][] exArray = new int[3][5];
```

Multi-Dimensional Arrays - Cont.

- So `exArray` points to three one dimensional arrays:

```java
exArray[0]
exArray[1]
exArray[2]
```

Yes, you can refer to these arrays in code, just like this.

- Each of these arrays has the same length:

```java
exArray[2].length // returns 5
```
Multi-Dimensional Arrays - Cont.

```java
int[][] twoDArray = new int[10][20];
```

- The above is equivalent to:

```java
int[][] twoDArray;
twoDArray = new int[10][20];
```

- As shown above:

```java
twoDArray.length // gives 10
twoDArray[0].length // gives 20
```

Aliasing Objects - Array Example

```java
int[] first = {1, 2, 3, 4, 5};
int[] second = {10, 20, 30, 40, 50, 60, 70};
```

Aliasing Objects - Array Example, Cont.

```java
second = first; // Aliasing!
```

Aliasing Objects - Array Example, Cont.

```java
// after garbage collection
```

Aside – “Garbage Collection” in Java

- Some computer programming languages require you to indicate when you are done with variables so the memory they are occupying can be released back to the OS. Called “Garbage Collection”.

- (Fortunately!) Java has an automatic Garbage Collection system:
  - Variables are garbage collected once you move outside their scope.
  - Object contents are garbage collected when there are no pointers pointing to the contents.
The aliasing objects - array example, cont. 


Aliasing Objects - Array Example, Cont.

- So, setting one array to equal another as in:
  ```java
  array1 = array2;
  ```
  sets `array1` to point to the same data memory location that was (and still is) pointed to by `array2`.

- Now, changing the value of an element in `array2` will change that same element in `array1`, or visa-versa - this makes sense since both array Objects point to the same set of data values in memory!

Aliasing Objects - Array Example, Cont.

- Passing an Object into a method results in the method's parameter being aliased to the Object passed as an argument.
- Called “Passing by Reference”!

Aside – Does Java Actually Use “Pass by Reference”?

- It comes down to terminology.
- You could argue that since a memory address is just an integer that you are passing a pointer by value – you are passing the memory address as if it was just a primitive type value.
- The important thing to realize is that you are not making a copy of the object and passing that instead.
- Other languages (like C++) have both pointers and references and you can chose in code to pass by value, by reference or to pass a pointer. You don’t have a choice like this in Java!

Passing Parameters by Reference

- For example, in `main`:
  ```java
  int[] arrayA = {1, 2, 3, 4, 5};
  passArray(arrayA); // invoke passArray
  ```

- The `passArray` method:
  ```java
  public static void passArray(int[] arrayB) {
  // arrayB is aliased to arrayA from main
  // making elemental changes to arrayB will
  // also change elements in arrayA in main
  arrayB[3] = 400;
  } // end passArray
  // arrayA[3] is 400 in main
  ```