**CMPE212 – Reminders**

- Assignment 2 is posted. Due next Friday.

---

**Today**

  - Passing by Reference.
  - null Pointers.

- Start OOP, by considering “What is an Object?”

- Start Encapsulation.

---

**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
  ```

---

**Passing Parameters by Reference, Cont.**

- The rule for parameter passing into methods is:
  - Objects are passed by reference, primitive types are passed by value.

- See `PassingDemo.java` – Has a method with two parameters - an array and an `int` - which one(s) will stay changed?

- Instead of going element by element, if you re-assign the array to another array within the method, what happens?

- Does this rule apply to Strings, as well?

---

**Passing Arrays by Reference**

- Summary of `PassingDemo.java`:
  - Primitive types are passed by value.
  - Only element by element changes in arrays will “stick”.
  - Re-assigning the array to a pointer that has local scope in a method will not “stick”.
  - If you make element by element changes using an aliased local pointer (like the parameter), changes will “stick”.
  - Strings are immutable, so this does not apply. You cannot make elemental changes inside a String, even though a String is passed by reference.
  - StringBuilder objects are mutable.

---

**Passing Arrays by Reference, Cont.**

- So, mutable Objects (like arrays) can be passed into and out of a method through the parameter list. If a method changes the contents of a mutable Object passed into it – those changes “stick” even when the method is complete.
Aside - Comparing Objects

- Testing arrays and Objects for equality (with the "==" boolean operator) is also interesting:
  - This test will only give a true when both objects have been aliased, using the assignment operator ":=".
  - So, even if both arrays have identical contents, "==" will return false, unless both arrays point to the same location.
  - This means that comparing Objects with "==" only compares pointers, not contents.

Pointers – A Question

- So, which way is better to declare a 3 by 10000 two-dimensional array?:
  ```java
  int[][] wayOne = new int[3][10000];
  int[][] wayTwo = new int[10000][3];
  ```
- Or, it makes no difference?

null Pointer or null Reference

- null is not a keyword in Java – more like a literal constant.
- What is a null pointer?
- What is a null pointer error?
- Does null have a type?
- Can you test a pointer to see if it is null? How?
- Why would you want to?

null References, Cont.

- The idea of a null reference was first introduced into ALGOL W back in 1965 by C.A.R. Hoare (also known as the inventor of Quicksort).
- See: http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

Using null in Java

- You can test to see if a pointer is null. Use "== null"
- Sometimes, to widen the scope of a variable you need to declare it before you can instantiate it. Often, the compiler will insist that you assign the variable pointer to null.
- A NullPointerException is probably the most frustrating error to encounter in Java!

What Is An Object?

- A definition that would apply to any programming language:
  - An entity that exists in an operating computer program that has:
    - State
    - Behaviour
    - Identity

- The State of an Object is the collection of information held in that object. This information may change over time, as a result of operations carried out on the Object.
- The Behaviour is the collection of operations that an Object supports.
- The Identity of an Object allows the program access to a specific Object.


- An Object represents real or abstract entities.
- An Object representing a real entity, for example:
  - The State is awake and curious.
  - The Behavior is helping.
  - The Identity is minx. An instance of Cat.

What is a Class?

- You can have many (infinite!) different Objects with different values for each State category (or attribute!).
- But if each of these Objects has the same set of possible behaviours then you can group these Objects together into a Class and then use this Class as a "blueprint" for instances:
  - For the example, the class would be "Cat".
  - ginger, felix, sylvester, belle, etc. are all Objects and are instances of Cat.
  - Most (but not all) OO programming languages support the definition of classes.

What is a Class?, Cont.

- A class is defined in the source code of a program.
- It defines:
  - The operations that are allowed on instances of this class (the methods).
  - The possible categories of state that are allowed for instances of this class (the attributes).
- Instances of a class are created when the program is running.

What is a Class?, Cont.

- The State of an Object instance can be partly or completely defined when the instance is created.
- The State will likely change when operations are carried out on the instance.
- However, attributes cannot be added or removed and behavior cannot be added or removed. These are defined in the Class.

Object Categories

- In a program Objects will probably fall into one of these general categories:
  - Tangible things (ex: Cat)
  - Agents ( StringTokenizer)
  - Events and transactions (Mouse Event)
  - Users and roles ( Administrator)
  - Systems ( Mail System)
  - System interfaces and devices ( File)
  - Foundational classes ( String)
Object Extremes

- Two extremes of object structure:
  - **Utility classes**: All static methods and attributes
    - The Math class, for example.
    - You do not instantiate these classes – there is no point.
  - **Customizable classes**: All non-static methods and attributes.
    - Attribute values (some or all) must be set at the time of instantiation before the class can be used.
    - Scanner class for example.

Objects, Cont.

- And many classes fall in-between these two extremes:
  - A mix of static and non-static methods.
  - Static methods have nothing to do with the attributes and so can be used without instantiation of the class.
  - Non-static methods depend on the attributes which must be set through instantiation.
  - Wrapper classes for example: Double, Integer, etc.

Encapsulation

- Encapsulation is the process of defining a Class that has at least one customizable attribute.
- Encapsulation is about the abstraction or containment of the attributes defined in the class.
- In Java, methods and attributes must be encapsulated or contained in a class definition.
- We have not yet built any classes with attributes that must be set upon instantiation. Time to start!

Encapsulation, Cont.

- The Best Object Design:
  - Supports the re-usability of code.
  - Provides a well-defined interface to other objects and the user(s).
  - Builds into an Object the code that ensures the integrity of the data stored in the Object by:
    - Making sure that initial data values are "legal".
    - Controlling (or preventing) changes to data.
    - Preventing "privacy leaks" when data is returned out of an Object.
  - Works well with modular design practices making code easier to write, test and debug.

Encapsulation, Cont.

- This is the driving principle for the design of all Objects in the Java API!
  - So, maybe you should design your own Objects using the same principle?

An Example

- Consider the general model (or "pattern"...) of a database:
  - Each record is an instance of an Object, too.
An Example, Cont.

- For now, we will focus on just how to define one of the many objects needed for the database – the object that will be used to define the record structure – a “Foundational” Object.

- If each record only held one piece of data, we would not need an object for each record – the Collection could just be an array.

- But – more typically – each database record will hold several pieces of data.

An Example, Cont.

- Suppose you want to create a database to store records of Halloween visits.

- (Maybe we are going to build a model to predict the number of visitors for given weather conditions so we can predict how much candy to buy!)

- For each year, you wish to record:
  - The year.
  - Number of munchkins banging at your door.
  - Outdoor temperatures (one reading per hour) in deg C.
  - Weather condition – “rain”, “snow”, or “clear”.

An Example, Cont.

- What would you do without objects and encapsulation?:

- You could create four arrays, one for each of the fields.
  - Difficult to code when moving records around.
  - Suppose some record swap or delete is messed up and the records get out of sync – how would you know?

- It would be easier to design a Class that contains (or encapsulates) these four fields and then design another class to collect instances of this Class.
  - No way the fields could get out of sync since moving a record moves all fields at once.

Halloween1 Class

```java
public class Halloween1 {
    public int year;
    public int numMunchkins;
    public int[] temperatures;
    public String weatherCondition;
}
```

We can define the State of an instance of Halloween1, but it does not yet have any Behavior (methods)!

Halloween1, Cont.

```java
Halloween1[] hwDB = new Halloween1[100];
hwDB[0] = new Halloween1();
hwDB[0].year = 1992;
hwDB[0].numMunchkins = 200;
int[] temp = {10, 9, 7, 6};
hwDB[0].temperatures = temp;
hwDB[0].weatherCondition = "rain";
```

In a method in some other class:

```java
Halloween1[] hwDB = new Halloween1[100];
hwDB[0] = new Halloween1();
hwDB[0].year = 1992;
hwDB[0].numMunchkins = 200;
int[] temp = {10, 9, 7, 6};
hwDB[0].temperatures = temp;
hwDB[0].weatherCondition = "rain";
```
Halloween1, Cont.

- A question:
  Why not declare the attributes in Halloween1 static?

Every instance of a class shares static members – there is only one copy of each static member in memory. If the attributes were static then each instance of Halloween1 would have the same values!

Halloween1, Cont.

- Another question:
  What is wrong with the following?:

```java
hwDB[1] = new Halloween1();
hwDB[1].year = 3011;
hwDB[1].numMunchkins = -200;
int[] temp = {100, 200};
hwDB[1].temperature = temp;
hwDB[1].weatherCondition = "frogs";
```

Halloween1, Cont.

- The object, hwDB[1] is not particularly realistic.
- How can we prevent the creation of an object like this?
- Who is responsible for checking for realistic, or “legal” values for the attributes? The Halloween1 class or the class that uses it? Where is it easiest to put these checks?

Encapsulation, Cont.

- Attributes must be declared private, so that the class that owns them can control how they are set.
- If attributes are private then how can a user of the class assign the values of the attributes?
- Through methods, of course!
  - Constructor(s)
  - Mutator(s)

Encapsulation, Cont.

- Within these methods, you can write code to check the parameters for validity.
- What do you do if the values are not legal?
- Throw an exception!
- We need to start adding Behaviour to our Class.