CS231: Lecture 3 (More Java)

- **Compiling several files.** If your program is broken into several Java files because you created several different classes A.java, B.java, C.java, you need to compile all of them that you change since last compilation before running. So do:

  javac A.java B.java C.java
  javac *.java
  java A //if A is the class with the main method you are running

- **Arithmetic Operators**
  + Additive operator (also used for String concatenation)
  - Subtraction operator
  * Multiplication operator
  / Division operator
  % Remainder operator

- **Equality and Relational Operators**
  == Equal to
  != Not equal to
  > Greater than
  >= Greater than or equal to
  < Less than
  <= Less than or equal to

- **For Loops**
  - Most languages have for loops as well as while loops. While loops can do everything a for loop can, so why add for loops to a language? [convenience, if you know how many times you will be looping or exact set of data you'll be looping through].

  - Example:
    for(int i = 0; i < 1000; i++) {
      System.out.println(i);
    }

  - for loop format:
    for(<init>; <condition>; <update>) {
      <body>
    }

  - Here's what happens:
    <init> is executed
    while <condition> is true
    <body> is executed
    <update> is executed

  - Vars declared in init are local to the for loop
  - Some of the 4 parts are optional (i.e., could be left blank). **Guess which ones.** [All 4 parts are optional.]
i. You could declare i outside of loop. The only difference is then i is not local to the for loop and instead is local to the main method.

ii. You can leave out the condition, which means that the condition is always true (as if true were used as the condition).

- Example

```java
public class TestForLoop {
    public static void main(String args[]) {
        for(int x = 10; x < 20; x = x + 1) {
            System.out.print("value of x: "+x);
            System.out.print("\n");
        }
    }
}
```

- While loops.
  o A while loop statement in Java programming language repeatedly executes the code in the body of the loop as long as a given condition is true. When the condition becomes false, program control passes to the line immediately following the loop.
  o Format:
    ```java
    while(Boolean_expression) {
        // Statements
    }
    ```
  o Example

```java
public class TestWhileLoop {
    public static void main(String args[]) {
        int y = 10;
        while( y < 20 ) {
            System.out.print("value of y: "+y);
            y++;
            System.out.print("\n");
        }
    }
}
```

- Import statements
  o Java has a huge library of classes (more than 3 thousand of them) that it has already created that are ready for your use. This is wonderful.
For example, there are the Math, String, and System classes you used in Die.java. They are part of a built-in package (like the __built-in__ package in Python) called java.lang.

There are other libraries as well that, like in Python, you need to import.

An example is the Random class. You use it to choose random values [https://docs.oracle.com/javase/8/docs/api/java/util/Random.html]. Note that it does much more than the Math.random() method does and so is the preferred way of dealing with randomness except if you want a random double between 0 and 1.

Problem: [Create a Java class that creates a Random object and calls one of its methods. You get an error message]

```java
public class TestRandom {
    public static void main( String args[] ){
        // create random object
        Random generator = new Random();

        // check next int value
        System.out.println("Next int value: " + generator.nextInt(1000));
    }
}
```

Solution: Look in API for info on Random class. Note that it is in the java.util package, so you can fix the problem by adding an "import java.util.Random;" at the beginning of the file. (You are telling javac to go into the java.util package and find the Random class and import it.)

Add `import java.util.*;`

In summary, all the Java library classes are in packages. The classes in the java.lang package are automatically imported. The java.lang package is a core part of the language and so it is implicitly imported for you automatically in all Java files. In contrast, all classes in all other packages need to be imported. Use the documentation to find out which package your class is in. Most classes we will use are in the default java.lang or java.util.

You can also import a whole package instead of just one class from that package:

```java
import java.util.*;
```

Then you can use all the classes in that package without worrying about the package prefix.

**Generics**

Before generics, we can store any type of objects in collection i.e. non-generic. Now generics, forces the java programmer to store specific type of objects.
A class like `ArrayList<E>` is a generic type. It has a type parameter `E` that represents the type of the elements stored in the list. Instead of just using an `ArrayList`, not saying anything about the type of elements the list contains, we can use a `ArrayList<String>` or a `ArrayList<Integer>`, thereby specifying that we mean a list of strings or integral values respectively.

The other class in the `java.util` package that you’ll use is the `ArrayList` class. An `ArrayList` is like a Python list in that it stores a list of data values.

Example [API](https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html)

Example code:

```java
import java.util.*;
ArrayList list = new ArrayList();
list.add(3);
int x = (int) list.get(0);
```

But I don’t want you to write code like this. When you use an `ArrayList`, you are almost always storing data of one particular kind, like integers or strings or Dice. In those cases, I want you to declare that your `ArrayList` will be storing that kind of data.

Example code:

```java
import java.util.*;
ArrayList<Integer> list1 = new ArrayList<Integer>();
list1.add(3);
list1.add(4);
int x = list1.get(0); // no typecast necessary
System.out.println(list2.get(0) + list2.get(1));
```

```java
ArrayList<String> list2 = new ArrayList<String>();
list2.add("Hello");
String s = list2.get(0); // no typecast necessary
```

**Note:** Casting is an operation that allows us to change the type of a value. We can take a value of one type and cast it into an equivalent value of another type.

```java
double d1 = 3.2;
double d2 = 3.9999;
int i1 = (int) d1; // i1 gets value 3
int i2 = (int) d2; // i2 gets value 3
double d3 = (double) i2; // d3 gets value 3.0
```

The advantage of doing it this way is that (a) it is easier to understand and (b) the Java compiler will catch errors where you accidentally try to add a value of the wrong type to the `ArrayList`.

`list1.add("hi"); // ← ERROR!`

Always use generic types. Don’t ever create an `ArrayList` without the brackets.
If you want a list that can store any kind of data, use <Object> as the generic type.  
Example:  

```java
ArrayList<Object> list = new ArrayList<Object>();
list.add(3); //no error
list.add("Hi"); //no error
```

- **Generic Types Summary**  
  o Generic types are used to restrict lists and other data structures to storing particular kinds of data.  
  o If you want a list that can store any kind of data, use <Object> as the generic type.

- **Primitive wrappers**  
  o ArrayList<Object> can hold any kind of data in the list. So you can have statements such as  
    ```java
    list.add("a string");
    list.add(3);
    ```
  o This works fine, except for one problem: ArrayLists actually store only objects and not primitives. But then how can we add an int?  
  o Answer: Behind the scenes, Java "wraps" the primitive in an object. It actually converts the primitive 3 to an Integer object that stores the value 3 and this object is what is stored in the list.  
  o **Autoboxing** is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an int to an Integer, a double to a Double, and so on. If the conversion goes the other way, this is called **unboxing**.  
  o When autoboxing, it wraps ints in Integers, booleans in Booleans, chars in Characters, and doubles in Doubles.  
  o Java automatically unbox as well:  
    ```java
    Integer ob = new Integer(3);
    int x = ob;  //auto unboxes the integer and assigns 3 to x
    ```
  o Occasionally, the wrapping rears its ugly head:  
    ```java
    int x = list.get(0);
    ```
  o This will crash since the ArrayList's get method has return type of Object, and so the method could return any type of object, not just an int. Therefore, the assignment is illegal.  
  o The solution is to typecast the value returned by get:  
    ```java
    int x = (Integer) list.get(0);
    ```
    or  
    ```java
    int x = (int) list.get(0);
    ```
  These statements are equivalent and tell the Java compiler 4 things:  
  1. Call the get method of the list  
  2. The 0th element is actually an Integer object
3. Unwrap the Integer to just the int it contains
4. Assign that int to a new variable x.