

CS231 Fall 2013 Lecture 3

Stephanie R Taylor

September 9, 2013

1 Administrative

- Return quizzes.

2 More Java Syntax

- *Comments*: Comment headers are important. If you are lazy like me, you just stick a simple comment at the top. If you are conscientious, like Kyle, you use Java doc comments.

They are multi-line comments that look like this:

```
/**
 * Takes a number and returns its square.
 * @param x The value to square.
 * @return The square of the given number.
 */
public float getSquare(float x) {
    ...
}
```

The other style of comment in Java is for an individual line:

```
// get the die rolling
d.roll();
```

- *Operators*: For your convenience, here is a list of binary operators in Java. (Notice there is no operator for exponentiation – you need to use `Math.pow`).

- `+` adds two numbers, concatenates two strings
- `-` subtracts one number from another
- `*` multiplies two numbers
- `/` divides one number into another.
- `%` performs the modulus operation (remainder after integer division)

- *Relational Operators*: Here they are:

- `==` equal to
- `!=` not equal to
- `>` greater than
- `>=` greater than or equal to
- `<` less than
- `<=` less than or equal to

- *while loops*. Here is an example. It tests the condition at the beginning of each iteration of the loop. If the condition is true, it executes the code in the body of the loop.

```

int count = 1;
while (count < 11) {
    System.out.println("Count is: " + count);
    count++;
}

```

[Construct a while loop to see how long it takes to get snake eyes.]

- *Compiling multiple files*: Note: 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

```

- *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

- What is printed in this for loop?

```
for(int i = 100; i > 0; i = i/2) {  
    System.out.println(i);  
}
```

- What happens in this loop?

```
for(int i = 0; true; i++) {  
    System.out.println(i);  
}
```

- Variables declared in init are local to the for loop

- All 4 parts are optional.

- * 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

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

- Import statements

- 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 is the Random class. You use it to choose random values [show them online API for the Random class]. 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: What if you create your own class called "Random" (for some weird reason)? This class now has the same name as the existing built-in class. How will the Java interpreter know which one to call when an object of that class name is created? Also, what if you want objects of both your class and the Java library class?
- This problem is called a "namespace" problem.
- Bad solution: require unique names for all classes by checking all over 3000 existing classes to make sure your name is unique.
- Good solution: break the library classes into groups called "packages". In that way two classes can have the same name without ambiguity if they are in different packages.
- In fact, all the Java library classes are in packages. The ArrayList and Random classes you'll see this afternoon is in the "java.util" package. (Actually, java is a package and util is a subpackage)
- How do you indicate this afternoon that you want the Random class from the java.util package and not an Random class from elsewhere? There are two ways:
 - * Always use the full official name of the class, which includes the package name. For example, the full name of the ArrayList class is java.util.ArrayList. So in your code you would write

```
java.util.Random rand = new java.util.Random();
System.out.println(rand.nextInt(6));
```

This is the approach you have to take if you want to use both your `Random` and the `java.util.Random` classes.

- * If you are only going to use the `java.util.Random` class, then you can use an "import" statement at the top of your file:

```
import java.util.Random;
```

This statement tells the Java compiler "from now on, when you see `Random` in my code, I mean `java.util.Random`". So now you can use `Random` without needing the package prefix:

```
Random rand = new Random();
System.out.println(rand.nextInt(6));
```

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

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

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

- You can also add your new classes to packages you create. We won't be doing it much or at all in this course, but it is easy. Just include the following line at the very top of your file:

```
package foo;
```

Now the official name of any class declared in that file begins with "foo."

- One last note: Some classes, such as the `System` class and the `String` class are in the `java.lang` package, but you can use it without the full official name and without importing `java.lang`. The reason is the `java.lang` package is a core part of the language and so it is implicitly imported for you automatically in all Java files.

- Import statements Summary

- Import statements are used to avoid namespace conflicts.
- Example: If you have a `Random` class that you declare and you want to create a `Random` object from your class and also a `Random` object from the class in the Java library, you need to do the following:

```
Random myRand = new Random(); //creates one of your Random objects  
java.util.Random rand = new java.util.Random(); //library Random
```

- If you don't have your own Random class, then you can use import statements at the top of your file:

```
import java.util.Random;
```

–or–

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

and then use Random without the package prefix, as in

```
Random rand = new Random(); //library Random
```

- Note: The java.lang package, including System, String, Math, and many other standard classes, is automatically imported.

- Generics

- The other class in the java.util package that you'll use this afternoon is the ArrayList class. An ArrayList is like a Python list in that it stores a list of data values. [Show them the API]

- Example code:

```
import java.util.*;  
ArrayList list = new ArrayList();  
list.add(3);
```

- 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:

```
import java.util.*;  
ArrayList<String> list1 = new ArrayList<String>();  
list1.add("Hello");  
ArrayList<Integer> list2 = new ArrayList<Integer>();  
list2.add(3);  
list2.add(4);  
System.out.println(list2.get(0) + list2.get(1));
```

- 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.

```
list.add("hi"); // ? ERROR!
```

- If you want a list that can store any kind of data, use <Object> as the generic type.
- Example:

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

● Primitive wrappers

- ArrayLists are like Python lists in that they can hold any kind of data in the list. So you can have statements such as

```
list.add("a string");
list.add(3);
```

- This works fine, except for one problem: ArrayLists actually store only objects and not primitives. But then how can we add an int?
- 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.
- When autoboxing, it wraps ints in Integers, booleans in Booleans, chars in Characters, and doubles in Doubles.
- Java is automatically unbox as well:

```
Integer ob = new Integer(3);
int x = ob; //auto unboxes the integer and assigns 3 to x
```

- Since it all happens behind the scenes, why am I bothering to tell you this?
- Answer: Occasionally, the wrapping rears its ugly head:

```
int x = list.get(0);
```

- This will crash since the ArrayList's get method has return type of Object, and int is not an object.

- The solution is to typecast the value returned by get:

```
int x = (Integer) list.get(0);
```

This statement tells 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.

Note: These notes were adapted from those of Dale Skrien.