Java Foundations Notes

Chapter 3 - Using Classes and Objects

CSC110

# More about Objects

## Objects

Objects have a state, which is the values of the data members (also called **fields** or **instance variables**), which is what the object knows about itself. In general, an object should not allow external entities to change its state.

Objects also have behaviors, which are the methods (also called **members** or **functions**) it contains. The behavior of an object describes what the object can do. The behavior of an object may change its state. In fact, objects should *only* allow their state to be changed through their behaviors.

Think of calling a method as “sending a message” that asks the object to do something. The message contains the operation’s name and arguments.

The client doesn’t care how the message is handled, only that it produces an expected result. Thinking this way will help you better design the classes in your program.

Creating an object is called **instantiation**. In Java, the new operator creates a new object. Each object is an **instance** of a particular class.

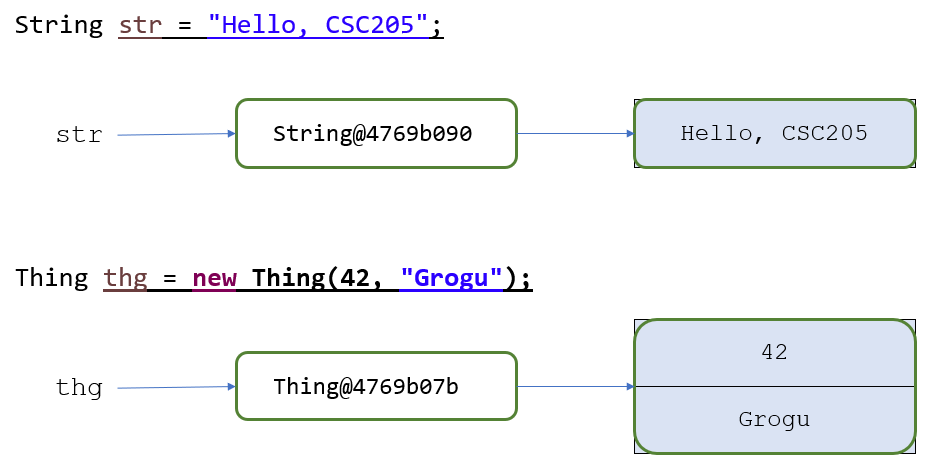
#### Objects vs Classes

A **class** represents a abstract concept, while an object is the realization of a class

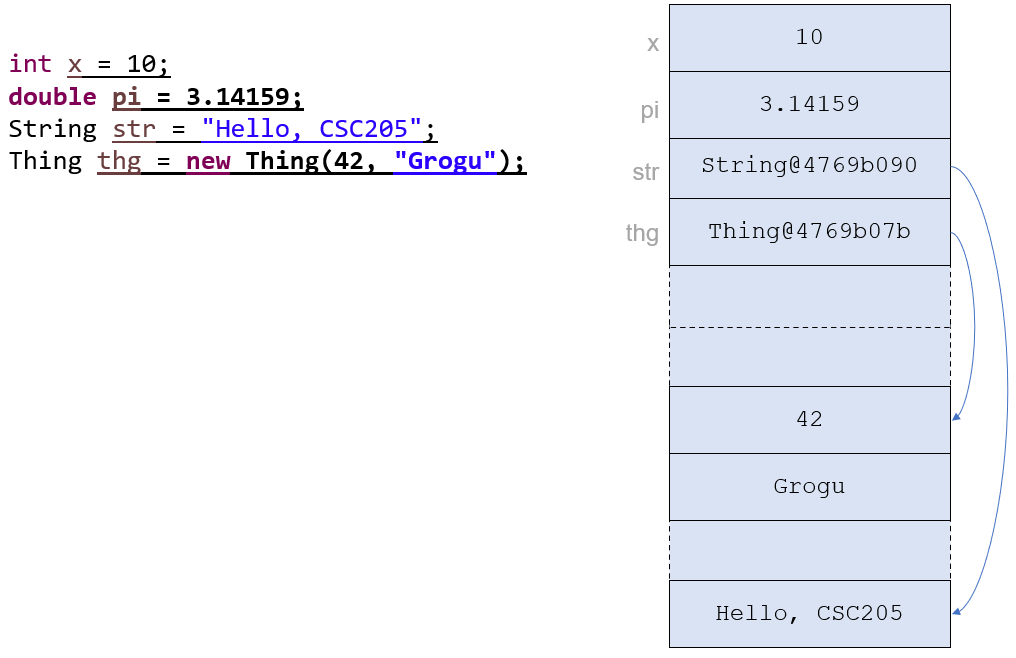
We **instantiate** an object of a specific class. There can be multiple objects of a given class, but each object is an instantiation of a single class

## Object Reference Variables

An **object reference variable** contains a reference to an object rather than the variable itself. A class name used as the type when declaring an object reference variable. The example below shows two object reference variables: str is a String object reference variable that refers to a string object with the contents “Hello, CSC205” and thg is a Thing object reference variable that refers to a Thing object with the contents {42, “Grogu”}. Note that the actual str & thg *variables* only hold a reference to the objects, *not the contents of the objects themselves!*



The following figure demonstrates what the previous 4 variables look like in memory. Note that the primitive variables hold *values* while the object reference variables hold *references* to objects located elsewhere in memory.



Note that when working with object reference variables, assignment statements like myVar = yourVar copy the *address*, *NOT the object*! Two or more references that refer to the same object are called **aliases** of each other. A single object can be accessed using multiple reference variables. Be careful - changing the object through one variable changes it for all of the aliases since they all point to the same object!

# Strings

Java String objects do not require the use of new, they can be instantiated with an assignment. So we can instantiate a string in either of the following ways:

String myStr1 = "Hello, World";

String myStr2 = new String("Hello, World");

Each string literal represents a String object. Note that in the case above, if we declared

String myStr3 = "Hello, World";

Then myStr1 and myStr3 would be aliases - they would both refer to the same string literal. However, myStr2 would not be an alias since we instantiated a new string with the same characters in it.

Also, String objects in Java are **immutable** - they cannot be changed. In cases where a string appears to be changed what actually happens is a *new String* is created with the changes and the reference variable is also updated to refer to that new string.

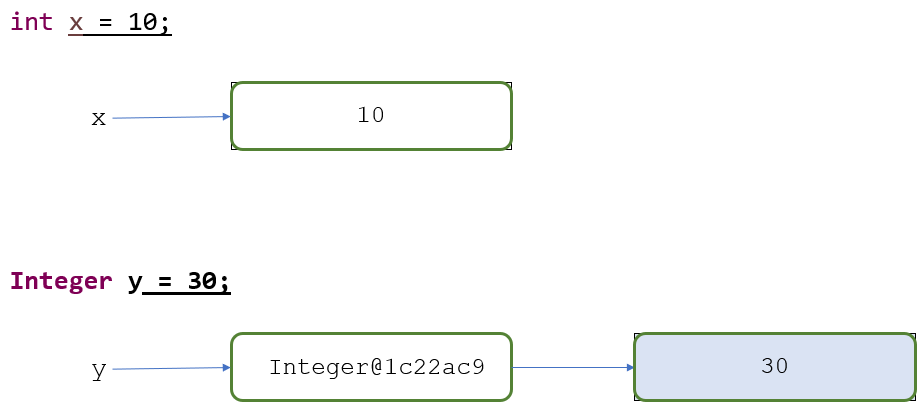
You should familiarize yourself with the Java String class & its methods.

# Wrapper Classes

Java has a corresponding **wrapper class** for each primitive type. For example:

* int → Integer
* char → Character
* double → Double

Below is an example of an int primitive variable and an Integer object reference variable.



These wrapper classes have static methods that help manage objects of that type. Java performs automatic **auto-boxing** to convert primitive types to the corresponding wrapper object type.

# Garbage Collection

When an object no longer has any valid references to it, it becomes garbage. Java performs automatic **garbage collection** where it frees up memory used by objects that no longer have references. Other languages, like C and C++, force the programmer to ensure garbage is handled correctly. In general, garbage collection is considered to be a good thing for a language to provide, though it does come at a performance overhead so is not always ideal in every situation. Also, keep in mind that you should not be profligate with objects just because of garbage collection - there is still a non-zero cost that can come back to bite you!

# The Random Class

An object of the Random class can be used to generate a stream of random numbers (technically, they are *pseudorandom* numbers, but that is beyond the scope of this course).

You can instantiate a Random object with no parameters, or with a long integer value to be used as a seed (an initial value to be used by the underlying pseudorandom number generator - but again that is beyond the scope of the course). If you instantiate two Random objects with the same seed they will generate the same sequence of random numbers.

## Random.nextInt

The nextInt method returns the next random integer in the random number generator’s sequence.

Now, suppose you want to generate integers in a certain range, say between MIN and MAX, inclusive. The formula for that is:

##### Random rand = new Random();

##### rand.nextInt(MAX - MIN + 1) + MIN;

Examples:

| Min Value | Max Value | nextInt formula |
| --- | --- | --- |
| 0 | 15 | nextInt(15 - 0 + 1) + 1 |
| 1 | 5 | **nextInt(5 - 1 + 1) + 1** |
| 1,000 | 2,000 | **nextInt(2000 - 1000 + 1) + 1000** |

## Random.nextDouble

The next method returns the next random double value between 0.0 and 1.0 in the random number generator’s sequence.

# Formatting Numbers

We often want to format numeric values in specific ways. For example, we may want to round a double to 4 decimal places, or display a float value like 0.25 as a percentage 25%. The Java API contains helpful classes that provide methods to format numbers in a variety of ways.

## The NumberFormat Class

The NumberFormat class can be used to format different types of numeric data, including currency & percentages.

The NumberFormat class is part of the java.text package, so when you are using NumberFormat you must include the following import statement at the top of your program.

##### import java.text.NumberFormat;

To create a NumberFormat object called cFmt that can be used to display a number as currency, you would instantiate an object as follows:

##### NumberFormat cFmt = NumberFormat.getCurrencyInstance();

Sidenote: you may notice that we don’t use new to instantiate the object here. That is because getCurrencyInstance is a special kind of method called a **factory method**. Factory methods return an object, so we can use them in place of new to instantiate an object. In this case, the getCurrencyInstance method returns a NumberFormat object that formats the number appropriately for the local currency. You don't *have* to understand what a factory method is, just understand that you can instantiate some types of objects using a factory method instead of new.

We can then use the format method from the NumberFormat class to display a number with appropriate formatting. In the following statement, amount is formatted as a dollar amount (in the US) with a leading dollar sign, commas where needed, and two digits to the right of the decimal place.

##### System.out.println("Highest bill : " + cFmt.format(amount));

Examples:

If amount is 344.19, then the following will be displayed:

##### Highest bill : $344.19

If amount is 89323.19, then the following will be displayed:

##### Highest bill : $89,323.19

If amount is 15, then the following will be displayed:

##### Highest bill : $15.00

If amount is 123456789.11, then the following will be displayed:

##### Highest bill : $123,456,789.11