Head First Java: Chapter 10 Notes

August 30, 2019

Static Methods and Variables

Methods in a class can be flagged as *static*, which allows them to be used without an instantiated object of the class. A static method is called by using its class name rather than its object reference variable name. A good example is methods within the `Math` class. The constructor within `Math` class is flagged as `private` and therefore instants of a `Math` class cannot be instantiated. However, essentially all methods in `Math` are flagged as `static`, they can still be called by using the class name, such as `Math.abs()` and `Math.round()`. A class can have both static and non-static methods. Aside from the different way they are called, there are additional limitations with static method. Instance variables within a class are utilized by methods, but if the methods are flagged as `static`, they cannot use any instance variables within the class. Consider:

```
public class Duck {
    private int size;
    public static void main(String[] args) {
        System.out.println(‘‘Size of duck is ’’ + size);
        System.out.println(‘‘Size of duck in ’’ + getSize());
    }
    public void setSize(int s) {
        size = s;
    }
    public int getSize() {
        return size;
    }
}
```

In the listing above, the line of code `System.out.println(“Size of duck is ” + size);` will cause compilation failure. The reason being is that `main()` is flagged as a static method (as all `main()` method are and should be), and `size` is not a static variable, and thus cannot be called within `main()`. In this particular case, if the user desire to print the value of the `size` variable out, the `System.out.println()` method should be written in a separate non-static method, which can be called by declaring a object reference variable of the `Duck` class and assigning it to a new `Duck` object, and calling this new non-static method via the object reference variable. The code of line `System.out.println(“Size of duck is ” + size);` will also cause compilation failure, as `getSize()` is not flagged as static, and cannot be accessed by `main()`.

Like methods, variables can also be declared as `static`. Consider the code listing presented previously, where `size` is a `private` instance variable of the `int` type. Assuming the rest of the code body was corrected to allow for compilation, each instance of a `Duck` object would have its own `size` instance variable independent of each other. In other words, setting the value of `size` via `setSize()` on one `Duck` object would not have any affect on any other `Duck` objects. However, if the code for the `Duck` class is modified, such that `size` is now flagged as static (`private static int size;`), there will only be one `size` variable shared among all created `Duck` objects. Changing the value of `size` will now change the value of `size` for all other `Duck` objects. Flagging an instance variable as `static` removes ambiguity, and thus can be called by static methods. This also applies to methods flagged as `static` as well; they can be called by other static methods. Static variables are initialized whenever the class itself is *loaded*, which can be before any instance of the class is instantiated. Consider:

```
class Player {
    static int playerCount = 0;
    private String name;
    public Player(String n) {
        name = n;
        playerCount++;
    }
}
```

The variable `playerCount` in class `Player` is a static variable. Which means even before any instance of `Player` is instantiated, the variable `playerCount` can be assessed by calling `Player.playerCount`, which will have a value of `0`. The constructor of `Player` automatically increments `playerCount` by `1` each time a `Player` object is instantiated (*e.g.* Player p = new Player();), therefore the value of `playerCount` will be `1` (or however many) afterwards, can be be called either by `Player.playerCount` or using the object reference variable of the object instance (*e.g.* p.playerCount). The `name` variable in `Player` is not a static variable, and can only be accessed after an instance of `Player` is instantiated.

`Final` class, methods, and variables

Class, methods, and variables can be flagged as `final`, which prevents them to be modified. Classes that are flagged as `final` cannot be further extended, which means no subclasses can be created out of them. Methods, whether flagged as `static` or not, if flagged as `final` cannot be overridden by any subclasses. The values of any variables flagged as `final` cannot be changed. In the case of `static` variables, these are often used as *constants*, which by convention have names that are all-capitalized. Declared `static final` variables must be initialized within the class or method it was declared at, otherwise the code will not compiled (because it will be impossible to assign a value to the variable later as `finally` variable's value cannot be changed.) There are two equally valid way to initialize a `static final` variable:

```
public class Foo {
    public static final int SOME_NUMBER = 25;
}

// or

public class Foo {
    public static final int SOME_NUMBER;

    static {
        SOME_NUMBER = (int) 25 * Math.random();
    }
}
```

In the first example, the constant `SOME_NUMBER` is initialized as `25`, and can no longer be changed. The second example is also valid, and is useful the the value of the constant is to have some more complexity (*e.g.* requiring more than 1 line of code, etc.) The code within `static()` behaves like a static variable declaration; it runs as soon as the class it resides in loads. Therefore no instance of the class needs to be instantiated for the variable to be assigned value.

Other non-static variables, such as instance variables that aren't static, local variables, arguments/parameters, etc. can also be flagged as `final`, which simply prevents its value from changing once initialized.

Wrappers

In Java, primitives such as `boolean`, `int`, and `float` are not objects, which means additional processing needs to be done to them if any code or operation only applies to objects. This also has implication to memory management as only objects exists on the memory heap. In the current Java version, most of these operation, called *wrappting*, is done behind the scene by the compiler/JVM. A *wrapper* is an object that "wraps around" a corresponding primitive type, and enabling them to behave as if they are an object. The name of each wrapper is the capitalized name of their corresponding primitive type (*e.g.*, **Boolean** is the wrapper for `boolean`), except in the case for `int` and `char`, their corresponding wrappers are named `Integer` and `Character`. A wrapper is a class of objects that has its own methods, which makes them useful for methods to be called on primitives (which would be impossible otherwise as primitives are not objects and thus has no methods of their own.) Most primitive wrapping and unwrapping are done implicitly, but can also be done explicitly. For example:

```
int i = 288;
Integer iWrapped = new Integer(i);
int iUnwrapped = iWrapped.intValue();
```

Implicit wrapping and unwrapping in new versions of Java is most obvious when dealing with `ArrayList`, which only accepts objects into the array. For example:

```
ArrayList<Integer> listOfIntegers = new ArrayList<Integer>;
listOfIntegers.add(42);
int num = listOfIntegers.get(0);
```

In this above listing, `listOfIntegers` was declared as a parameterized `ArrayList` that only accepts `Integer` objects/wrappers. However, adding the integer primitive `42` to it is legal because the compiler automatically wraps `42` in an `Integer` wrapper. Using the `get()` method and plugging this `Integer` wrapper (containing `42`) into a primitive variable `num` is also legal as the compiler automatically unwraps the wrapper. Note that parameterizing `ArrayList` to primitives, such as `ArrayList<int>` is not legal.

Autoboxing, that is, auto-wrapping and auto-unwrapping, occurs elsewhere in Java as well. Here are a few examples:

- Method arguments — Consider the method `void takeNumber(Integer i) {}`, which requires an `Integer` wrapper argument. An `int` or `Integer` are both valid type of argument due to autoboxing. The reverse is true as well, if the method is declared to accept an `int` argument.
- Return values — Similar to above, in a method that was declared to return an `int`, an`Integer` wrapper can be returned instead, and `vice versa`.
- Boolean expressions — Boolean expressions in conditional statements such as `if` or `while` can accept either `boolean` primitives or `Boolean` wrappers.
- Mathematic operations — Despite the face that wrappers are objects, they can be treated as primitive numbers in Java mathematic operations. For example, if `i` is an `Integer` wrapper that contains the integer `4`, `i + 4` is legal in Java coding, despite the fact that i is technically an object reference variable, not a primitive.

- Assignments — During variable initialization, a wrapper or a primitive can be assigned to its corresponding variable type. For example, if `x` is an `int` with a value of 3, `Integer y = x;` is valid.

One key thing to note that wrappers are objects, and thus the complexity between objects and object reference variables, their location in memory (stack or heap) applies to them. In Java, strings are objects as well (hence `String`), and conversion between `String` and various primitives are performed with autoboxing in between. Methods that converts primitives to and from `String` are static methods within their corresponding wrapper class. For example, if a `String` object `s` has value of "2", to convert it to an `int` one can use the code `int x = Integer.parseInt(s);`. The applies to other primitives/wrappers as well, with the same method naming convention (*e.g.* `parseDouble()`, `parseBoolean()`). To convert a primitive to a string, the static method of each wrapper `toString()` can be used similarly.

String Formatting

String formatting in Java heavily utilizes the static method in the `String` class, `format()`. There is great flexibility available given the syntax:

```
String.format(‘‘%[argument number][flags][width][.precition]type’’, arg)
```

Each component in the syntax is as follows:

- `%` — Indicate the where formated argument should be inserted.
- `[argument number]` — Optional; used to specify argument to apply formatting to if there are more than one arguments.
- `[flags]` — Optional; indicate special formatting options, such as inserting commas in large numbers.
- `[width]` — Optional; specify minimum number of characters to use. If the argument is longer than the specified width it will be ignored.
- `[.precision]` — Optional; floating point precision (*i.e.* decimal place), always begins with a period.
- `type` — Mandatory; indicate argument type:
  - %d — Applies to integer arguments, will format to decimal integers (whole numbers).
  - %f — Applies to any floating point (`float` or `double`), will format to decimal integers.

- %e — Applies to any floating point, will format to scientific notation.
- %x — Applies to integer, will format to hexadecimal string.
- %c — Applies to single characters (`char` type), will format to Unicode character.
- %s — Applies to any data type, will format to `String` value.
- %t — Applies to date/time data types (including from class `Date` and `Calendar`), will format according to additional flag added after %t.

Based on this syntax, string formating can be use to insert variable values into string values:

```
int one = 20456654;
double two = 100567890.248907;
String s = String.format(''The rank is %,d out of %,.2f'', one, two);
```

The `%` specify where each variable value should be inserted. Since there are more than one arguments (two, in this case), Java will insert them in order. That is, the value of `one` to the first `%` and `two` to the second `%`. The `,` is a *flag* that tell Java to formate the numeric value with commas every three digits. The `d` tells Java to expect an `int` (or an `Integer`), and will format it just as a whole number. The `.2` tells Java to formate the floating point into a number with two decimal place. And finally the `f` tells Java to expect a floating point data type, and to formate it as a decimal number. The resulting string value of `s` is:

```
The rank is 20,456,654 out of 100,567,890.25
```

Time and Date in Java

`Date` is a class in Java where new instance of this object contains a time stamp value of the current time in this format:

```
Sun Nov 28 14:52:41 MST 2004
```

Each component of the time stamp is not fixed to a specific format. For example, `Nov` can be made to display as `November`. `Date` objects can be used in string formatting:

```
Date today = new Date();
String.format(''%tA, %tB %td'', today, today, today)
// this will display as: Sunday, November 28
```

Using the `%t` flag in string formatting, an additional flag must follow. In this case, `A` gives the day of the week (long form), `B` the full month name, and `d` the day of the month in roman numerals. To avoid making three separate instance of `today`, `String.format("%tA, %<tB %<td", today);` can be used instead. The < symbol tells Java to reuse the previous argument. There are many more flags that can be use for `%t` to achieve the desired results.

If more complex time and date manipulation beyond just using the current date/time is needed, then the `Calendar` class should be used. The `Calendar` class, like `ArrayList`, is in `java.util.*` and must be imported to be utilized. The `Calendar` class is an abstract class, so new instance of it cannot be created normally, but instead the static method `Calendar.getInstance()` should be used to assigned a new `Calendar` "instance" to a `Calendar` class object reference variable. The `Calendar` class overs much more flexibility compared to `Date`:

```
import java.util.Calendar;

Calendar c = Calendar.getInstance();
c.set(2004, 0, 7, 15, 40);
long day1 = c.getTimeInMillis();
day1 += 1000 * 60 * 60;
c.setTimeInMillis(day1);
c.add(c.DATE, 35);
c.roll(c.DATE, 35);
c.set(c.DATE, 1);
```

Each line of the above code illustrates an example on the flexibility of `Calendar`. Once an object of `Calendar` (really one of its subclass as itself is an abstract class), the time can be set with `set()`, which accepts exactly five arguments in the format of year, month (based in 0, like array index), day, hour(24-hour format), and minute. `Calendar` allows for precise time adding and subtracting by converting its time to milliseconds with `getTimeInMillis()`, adding/subtracting desired time from it (in units of milliseconds), then resetting the time in the `Calendar` object by using `setTimeInMillis()`. Or, individual component, such as the date can be accessed (in this case, via `c.DATE`), and manipulated with methods such as `add()` (add units of time, in this case days) or `roll()` (add units of time, in this case days, but do not increment the next greater unit of time, in this case adding 35 days but do not advance the month). The `set()` method is overloaded such that if a component field of the object (*e.g.* `c.DATE`) and a number is passed instead, it would set only that component field (in this case, setting c.DATE, the day of the month, to 1).