Chapter 5: Java Gotchas and Subtleties

In this chapter we'll highlight a few aspects of Java that sometimes get overlooked when starting out with the language.

5.1. Shadowing

Variable shadowing occurs when the same variable name is used in two different scopes. In Python, one example would be:

```
def f() -> None:
    x = 10

    def g() -> None:
        x = 20
        print(f"g()'s x = {x}")

    print(f"f()'s x = {x}")
```

This would print out the following:

```
g()'s x = 20
f()'s x = 10
```

Each of the functions have their own stack frame on the call stack where their `x` variable is stored. The stack frame for the call to `f()` contains both its `x` with the value of `10` along with the definition for the function `g()`. The stack frame for the call to`g()`'s stack frame would simply contain its `x` with the value `20` — once `g()` finishes execution, that stack frame would disappear and the stack frame for `f()` would still be unchanged. You can open PyCharm and step through with the debugger for yourself to see this.

In Java, we have a similar concept. For example, consider the following code:

```
public class ShadowExample {
    private int shadowedVariable = 10;

    public void shadowingMethod(){
        int shadowedVariable = 20;
        System.out.println(shadowedVariable);
        System.out.println(this.shadowedVariable);
    }
}
```

When we call `ShadowExample.shadowingMethod()`, the following would be printed:

```
20
10
```

Since both variables have the same name, we must use `this` to refer to the instance variable, while just `shadowedVariable` is used for the local variable. This is similar to what we saw previously with a constructor taking in a parameter with the same name as the instance variable it was assigned to (i.e., `this.name = name`).

5.2. Array Copy

In Python, we could copy lists by creating a slice of them. For example:

```
lst = [1, 2, 3]
lst_copy = lst[:]
lst_alias = lst
```

Both `lst` and `lst_copy` would contain the same items but have different memory addresses: modifying one would not modify the other. `lst_alias`, however, would be an alias to `lst`: if we modify one, we modify the other.

The same concept applies to Java, except arrays have a `clone` method. For example:

```
int[] lst = {1, 2, 3};
int[] lst_copy = lst.clone();
int[] lst_alias = lst;
```

The relationships between `lst` and `lst_copy` along with `lst` and `lst_alias` are the same as what we had in our Python example.

Furthermore, nested lists in Python behave the same as nested arrays in Java. In Python, if we had:

```
nested_lst = [[1, 2], [3, 4]]
nested_lst_copy = nested_lst[:]
nested_lst_copy[1] = [5, 6]
nested_lst_copy[0][0] = 7
```

Then the inner nested list would be an alias, but the outer list wouldn't. In this example, we would get the following contents for each list:

```
>>> nested_lst
[[7, 2], [3, 4]]
>>> nested_lst_copy
[[7, 2], [5, 6]]
```

To make a deeper copy without any aliasing, we would need to make copies of every inner list.

Java behaves in exactly the same way: using `clone()` creates a copy of the outermost arrays, but not copies of inner arrays. To make a deeper copy, we would need to `clone()` all inner arrays.

5.3. Autoboxing

In Java, we have to define types and adhere to our type declarations, otherwise our code will not compile. However, **autoboxing** is a conversion that the Java compiler makes automatically between primitive types and their corresponding object wrapper class and vice versa (e.g. `int` and `Integer`). When converting from the wrapper class to the primitive it is referred to as **unboxing**.

For instance, we can do:

```
int x = 4;
Integer y = new Integer(x); // equivalently Integer y = x
int z = y;
```

In the second assignment statement, we could just write `Integer y = x;` and the 4 will get autoboxed! In the third line, `y` is unboxed and `z` gets the value 4. With this autoboxing feature of Java, we are able to simply work with the primitive type and let Java autobox and unbox as needed since this simplifies the code writing process for you.

5.3.1. Value Comparison

Consider comparing `int` values and `Integer` objects. If **one operand is a primitive**, Java will **unbox** the `Integer` and compare values:

```
Integer a = 100;
int b = 100;

System.out.println(a == b); // true — compares values; a is unboxed
```

This makes sense, since if `b` had been autoboxed instead, then we might be in for a surprise as we'll see next!

5.3.2. Reference Comparison

If **both operands are `Integer` objects**, `==` compares **references**, not values:

```
Integer x = new Integer(100);
Integer y = new Integer(100);

System.out.println(x == y); // false — different objects
System.out.println(x.equals(y)); // true — same value
```

This can be a source of bugs, so just as we saw with strings and objects in general in Java, one should make sure to always use `equals` when comparing objects. Luckily, your IDE will warn you since this is such a common source of bugs.

5.3.3. Integer Caching

Similar to string interning that we saw previously, Java caches `Integer` values from **-128 to 127** for efficiency, so:

```
Integer a = 100;
Integer b = 100;
System.out.println(a == b); // true — same cached object

Integer c = 200;
Integer d = 200;
System.out.println(c == d); // false — not cached
```

5.3.4. Null Safety

Unboxing a `null Integer` causes a `NullPointerException` at runtime:

```
Integer a = null;
int b = 100;

System.out.println(a == b); // throws NullPointerException
```

In order to compare `a` and `b`, as we saw above, `a` will be unboxed. Unboxing an `Integer` corresponds to replacing `a` with `a.intValue()` when the code is compiled.

> If you were to run this example, the error message reveals the exact error:
> `java.lang.NullPointerException: Cannot invoke "java.lang.Integer.intValue()" because "a" is null`

For more details on autoboxing and the corresponding wrappers for each primitive class, see the official Java tutorial!

You may be wondering what the purpose of wrapper classes are for the primitives, as the above seemed to just suggest that using wrappers can lead to bugs if used carelessly. The following chapter discussing Generics in Java should make this clear.