Unit 19: Wrapper Class

After this unit, students should:

- be aware that Java provides wrapper classes around the primitive types
- be aware that Java will transparently and automatically box and unbox between primitive types and their corresponding wrapper classes

Writing General Code for Primitive Types

We have seen the following general code that takes in an array of `Object` objects, and searches if another object `obj` is in the given `array`.

```
1  // version 0.1 (with polymorphism)
2  boolean contains(Object[] array, Object obj) {
3    for (Object curr : array) {
4      if (curr.equals(obj)) {
5        return true;
6      }
7    }
8    return false;
9  }
```

Through polymorphism and overriding of the `equals` method, we can make sure that it is general enough to work on any reference type. But what about primitive types? Do we need to write a separate function for each primitive type, like this?

```
1  // version 0.4 (for int, a primitive type)
2  boolean contains(int[] array, int obj) {
3    for (int curr : array) {
4      if (curr == obj) {
5        return true;
6      }
7    }
8    return false;
9  }
```

Making Primitive Types Less Primitive

Java provides wrapper classes for each of its primitive types. A *wrapper class* is a class that encapsulates a *type*, rather than fields and methods. The wrapper class for `int` is called

`Integer`, for `double` is called `Double`, etc. A wrapper class can be used just like every other class in Java and behave just like every other class in Java. In particular, they are reference types and their instances can be created with `new`; instances are stored on the heap, etc.

For instance,

```
1  Integer i = new Integer(4);
2  int j = i.intValue();
```

The code snippet above shows how we can convert a primitive `int` value to a wrapper instance `i` of type `Integer`, and how the `intValue` method can retrieve the `int` value from an `Integer` instance.

With the wrapper type, we can reuse our `contains` method that takes in an `Object` array as a parameter to operate on an array of integers -- we just need to pass our integers into the method in an `Integer` array instead of an `int` array.

All primitive wrapper class objects are *immutable* -- once you create an object, it cannot be changed.

Auto-boxing and Unboxing

As conversion back-and-forth between a primitive type and its wrapper class is pretty common, Java provides a feature called auto-boxing/unboxing to perform type conversion between primitive type and its wrapper class.

For instance,

```
1  Integer i = 4;
2  int j = i;
```

The first statement is an example of auto-boxing, where the primitive value `int` of 4 is converted into an instance of `Integer`. The second statement converts an instance of `Integer` back to `int` (without affecting its value of 4).

Performance

Since the wrapper classes allow us to write flexible programs, why not use them all the time and forget about primitive types?

The answer: performance. Because using an object comes with the cost of allocating memory for the object and collecting garbage afterward, it is less efficient than primitive

types.

Consider the following two programs:

```
Double sum;
for (int i = 0; i < Integer.MAX_VALUE; i++)
{
    sum += i;
}
```

vs.

```
double sum;
for (int i = 0; i < Integer.MAX_VALUE; i++)
{
    sum += i;
}
```

The second one can be about 2 times faster. All primitive wrapper class objects are immutable -- once you create an object, it cannot be changed. Thus, every time the sum in the first example above is updated, a new `Double` object gets created. Due to autoboxing and unboxing, the cost of creating objects becomes hidden and is often forgotten.

The Java API in fact, provides multiple versions of the same method, one for all the reference types using `Object`, and one for each of the primitive types. This decision leads to multiple versions of the same code, but with the benefits of better performance. See the Arrays class for instance.