Wrapper Classes

In Week One, we learned about the eight primitive types in Java. These eight primitive types are basic data types and not objects–but they're exceptions to the general rule in Java. Java is an object-oriented language, and much of the language revolves around the idea of treating everything as an object. Hence, oftentimes, we find it necessary to **convert** a primitive type into an object.

To facilitate this **conversion**, Java provides us with what is known as **wrapper classes**. Each primitive type in Java has a corresponding wrapper class. These wrapper classes contain several useful methods that we can use, just like the ones you learned about when studying Strings and Arrays.

The wrapper classes and their corresponding primitive data types are fairly predictable, but here they are in case you want to see them all laid out:

Primitive Data Type	Corresponding Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

InfoWarningTip
We'll cover the modern methods in a moment, however, we want to show you the old way first. In previous versions of Java, these conversions were a bit more wordy. You may see this kind of code in your work, so you should be ready to see it as shown in the following examples:

Converting from a primitive data type into a wrapper class object is straightforward. For example, to convert `int` into an `Integer` object, we can do the following:

CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdown PascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
Integer intObject = Integer.valueOf(100);
```

We declare and instantiate an `Integer` object by passing a primitive `int` value of 100. If we want to convert the `Integer` object back to an `int`, we can do the following:

CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdown PascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
int intPrimitive = intObject.intValue();
```

The `intValue()` method returns the value that we assigned to the `intPrimitive` variable. The type for this value is `int`, since that's what we're turning it back into.

#Autoboxing and Unboxing

Converting from a primitive data type to an object and vice versa is straightforward. However, in practice, it is even simpler than shown above. Since Java 1.4, Java has provided two mechanisms, autoboxing and unboxing, that allow for automatic conversion.

#Autoboxing

Autoboxing is a feature in Java that provides automatic conversion of primitive data types to their corresponding wrapper class objects. In other words, autoboxing automatically wraps the primitive values into their respective wrapper classes when Java requires the non-primitive versions.

For example, when you assign a primitive value to a variable of a wrapper class, autoboxing automatically converts the primitive value to an object of the wrapper class. This simplifies the process of working with both primitive types and objects.

So to convert from int to Integer, instead of writing:

CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdown PascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
Integer intObject = Integer.valueOf(100);
```

We can simply write:

CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdown PascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
Integer intObject = 100;
```

Doing so directly assigns a value of 100 to the variable. We don't even need to call the `valueOf()` method to convert the `int` to an Integer; Java does it for us behind the scenes. This process is known as **autoboxing**.

#Unboxing

Unboxing, on the other hand, is the process of converting a wrapper class object back to its corresponding primitive value. It extracts the primitive value from a wrapper object when the primitive is what we need.

To convert back from `Integer` to `int` instead of writing:

CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdown PascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
int intPrimitive = intObject.intValue();
```

We can just write:

CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdown PascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
int intPrimitive = intObject;
```

We do not need to use the `intValue()` method explicitly. When we assign an `Integer` object to an `int` variable, Java automatically converts the `Integer` object to an `int` type. This process is known as **unboxing**. It works for all of the other primitive types, too.

#Additional Benefits

Wrapper classes provide a convenient way to convert primitive types into objects and vice versa. Besides this purpose, wrapper classes also have one other major use–they provide us with methods for converting Strings into primitive types.

Suppose you want to convert a string into an int; you can use the `parseInt()` method in the Integer class as shown below:

CopyC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdown PascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
int num = Integer.parseInt("100");
```

InfoWarningTip

Remember that "100" is a String because it's contained in a set of double quotes. The `Integer.parseInt()` method returns the `int` value `100`, which we then assign to the `num` variable.

If the string cannot be converted into an int, the method will throw a `NumberFormatException` because it can't accomplish the job. We will be talking all about exceptions tomorrow, so consider this your first real exposure to the concept. The statement below will give us that error:

CopyC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdown PascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
int num = Integer.parseInt("abc");
```

#Knowledge Check

Which of the following is not a Wrapper Class?

Integer

BigInteger

Long

Character

Are all the Wrapper Classes Immutable?

Yes

No

Depends on the platform.

Indicate in which line of code Unboxing is happening.

```
public class Main {
public static void main(String[] args) {
int[] nums = { 5, 1, 7, 8, 3, 2};
for(Integer num: nums) {
System.out.println(num);
}
}
}
```

Line 5

Line 7

Line 8

No Unboxing is happening in this code.

Indicate in which line of code Autoboxing is happening.

```
public class Main {
public static void main(String[] args) {
int[] nums = { 5, 1, 7, 8, 3, 2};
for(Integer num: nums) {
System.out.println(num);
}
}
}
```

Line 5

Line 7

Line 8

No Unboxing is happening in this code.

What happens when the following code is run?

```
8
public class Main {
public static void main(String args[]) {
String s="130";
Byte b= new Byte(s);
System.out.println(b);
}
}
```

The program prints out 130.

Compilation error at line 5

A `NumberFormatException` is thrown.

The program prints out `true`.

#Extra Resources

When time permits, watch the following video to see Wrapper Classes in action :

Open in new tab

https://youtu.be/4MiEznM8y8Q