2.8 Wrapper Classes

Wrapper classes

Wrapper classes are used to store primitive types inside of ordinary Java classes. `Integer` is a Java class, while `int` is not.

`Integer` has a single attribute storing the value of the `int` used to create the instance.

Constructors:

```
Integer i = new Integer(4);
```

```
Double d = new Double(2.718);
```

Getters:

```
int j = i.intValue();
```

```
double e = d.doubleValue();
```

Wrapper classes

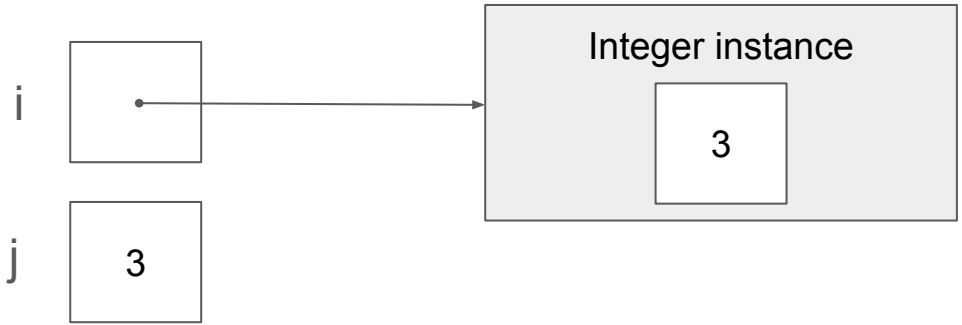
A variable whose type is a wrapper class like `Integer`, like any other class, is a reference to an object instance, not an object instance itself.

Using a wrapper class instead of a primitive type means more memory accesses, which is slower, and more memory used.

Wrapper object instances are also called "boxed primitives" – see the box?

```
Integer i = new Integer(4);

int j = 3;
```



Wrapper classes exist for every primitive type

Every primitive type in Java has a corresponding wrapper class.

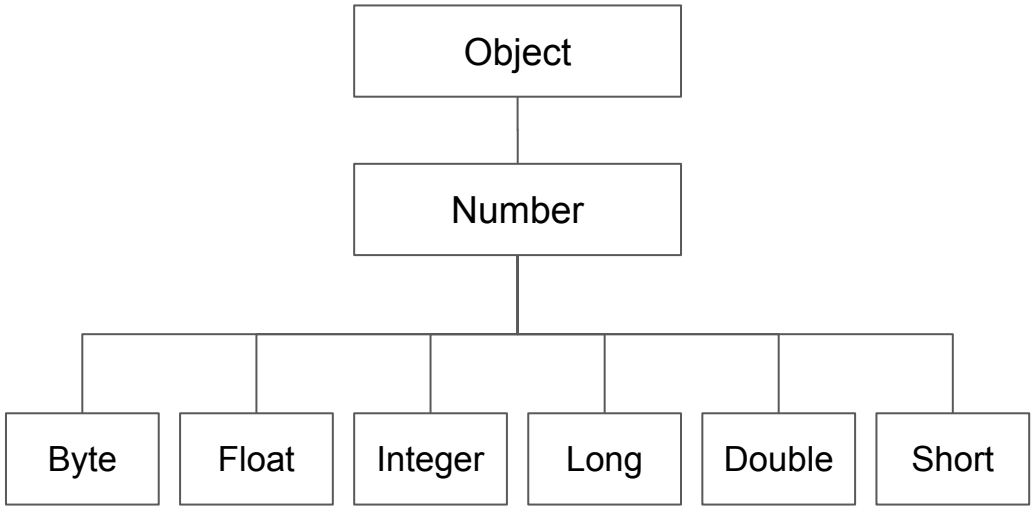
`boolean`	`java.lang.Boolean`
`byte`	`java.lang.Byte`
`char`	`java.lang.Character`
`double`	`java.lang.Double`
`float`	`java.lang.Float`
`int`	`java.lang.Integer`
`long`	`java.lang.Long`
`short`	`java.lang.Short`

java.lang.Number

Wrapper classes for the numeric types all share the same parent class, java.lang.Number, whose parent class is java.lang.Object.

From java.lang.Number, all of those wrapper classes get these methods:

`byte byteValue()`
`double doubleValue()`
`float floatValue()`
`int intValue()`
`long longValue()`
`short shortValue()`



Autoboxing and unboxing

Originally, Java programmers had to explicitly convert between primitive types and the equivalent wrapper objects.

In Java 1.5 (released 9/30/2004), **autoboxing** and **unboxing** were added.

Autoboxing:

Integer i = 4; is the same as Integer i = new Integer(4);

Unboxing:

int j = i; is the same as int j = i.intValue();

Wrapper classes are immutable

The wrapper classes are immutable, like `String`. Once you create an instance of `Integer`, you can't change the `int` inside it.

The wrapper classes have getter methods like `intValue()`, `doubleValue()`, `booleanValue()`, `floatValue()`, but no setter methods.

```
Integer i = 3;
```

`i = 5;` ← This is really creating a whole new `Integer` object instance, and reassigning the value of `i`.

Benchmarking int vs Integer

Main.java

```java
class Main {
  public static void main(String[] args) {
    final int N = 1000000000;
    long startTime = System.nanoTime();
    int sum = 0;
    for (int i=1; i<=N; i++) {
      sum += i;
    }
    long primitiveElapsed = System.nanoTime() - startTime;

    startTime = System.nanoTime();
    Integer wrapperSum = 0;
    for (Integer i=1; i<=N; i++) {
      wrapperSum += i;
    }
    long wrapperElapsed = System.nanoTime() - startTime;

    System.out.println("primitiveElapsed = " + (primitiveElapsed/1e9));
    System.out.println("wrapperElapsed = " + (wrapperElapsed/1e9));
  }
}
```

Benchmarking int vs Integer: Results

```
ggrossman@PK61V7VQ7Q wrappertest % javac Main.java
ggrossman@PK61V7VQ7Q wrappertest % java Main
primitiveElapsed = 0.398111584
wrapperElapsed = 2.581683042
ggrossman@PK61V7VQ7Q wrappertest % java Main
primitiveElapsed = 0.393371375
wrapperElapsed = 2.582966417
ggrossman@PK61V7VQ7Q wrappertest % java Main
primitiveElapsed = 0.393181291
wrapperElapsed = 2.579909
ggrossman@PK61V7VQ7Q wrappertest % java Main
primitiveElapsed = 0.399227708
wrapperElapsed = 2.655729167
ggrossman@PK61V7VQ7Q wrappertest %
```

Why even use wrapper classes?

1) They contain useful methods and attributes:
    a) Integer.parseInt(string) converts a string representation of an integer into an int
    b) Integer.MIN_VALUE and Integer.MAX_VALUE store the largest and smallest possible 32-bit integers your computer can store. These lower and upper bounds on all computable integers are very useful in algorithm development
2) Storing primitive types within classes enables us to use Java language constructs that can only be applied to classes…
    a) More on this later with Arrays and Maps

Java uses the wrapper classes for its own purposes

We learned that string concatenation will do string conversion, that is, convert other types to String before concatenating them. It does this using the wrapper classes.

```
String s = "Temperature: " + currentTemp + " Frozen: " + isFrozen();
```

This is equivalent to

```
String s = "Temperature: " + new Integer(currentTemp).toString() +
           "Frozen: " + new Boolean(isFrozen()).toString();
```

(The JVM probably optimizes this heavily but this is the idea.)

Reference: Java Language Specification, Section 5.1.11

Integer.MIN_VALUE, Integer.MAX_VALUE

Each numeric wrapper type has a MIN_VALUE, MAX_VALUE.

Integer.MIN_VALUE = -2147483648, Integer.MAX_VALUE = 2147483647

Why? int is 32-bit. The biggest binary number would be all 1's, but we reserve the MSB (Most Significant Bit) to represent positive/negative, called the **sign bit**.

b31 b0

0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}	2^{25}	2^{24}	2^{23}	2^{22}	2^{21}	2^{20}	2^{19}	2^{18}	2^{17}	2^{16}	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}

In a decimal number, the least place is multiplied by 10^0, the tens place by 10^1, hundreds by 10^2. In binary, you instead multiply by 2^0, 2^1, 2^2, ...

9	9	9	9
10^3	10^2	10^1	10^0

So the above number is $2^0+2^1+2^2+2^3+2^4+2^5+...+2^{30} = 2^{31}-1 = 2147483647$

Two's Complement (2^n-x)

Computers usually use this scheme for representing negative numbers. The MSB is the **sign bit**. To negate a number, you invert every bit and then add 1. So the number 1 as an int is

00000000000000000000000000000001

First, we invert all the bits.

11111111111111111111111111111110

Then, we add 1, and this is the binary representation of -1 in two's complement.

11111111111111111111111111111111

What happens if you negate again? (Invert bits and add 1)

Two's Complement: Why?

Two's complement turns out to be a good system for representing negative numbers, because positive and negative numbers represented this way can be added/subtracted just like any numbers, and the two's complement results work out as expected.

1 + (-1) = 0

```
  00000000000000000000000000000001
+ 11111111111111111111111111111111
----------------------------------
  00000000000000000000000000000000
```

Two's Complement: What happens when you overflow?

2147483647 + 1 = ????

```
 01111111111111111111111111111111
+00000000000000000000000000000001
--------------------------------

 10000000000000000000000000000000
```

Two's Complement: What happens when you overflow?

2147483647 + 1 = -2147483648

```
 01111111111111111111111111111111
+00000000000000000000000000000001
--------------------------------

 10000000000000000000000000000000
```

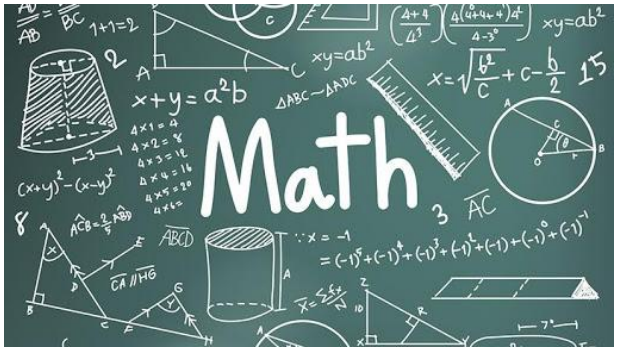
Integer.MAX_VALUE+1 == Integer.MIN_VALUE!

2.9: Using the `Math` Class

The `Math` Class

This class implements standard mathematical functions and constants.

Math has only static methods and attributes. It cannot be instantiated with the new operator… it has no public constructor!



The `Math` Class

You can prefix with `Math.` to access Math methods and attributes, or use **static imports** to bring some or all of Math into your code's default scope.

These two code samples are equivalent:

```
public class Main {
  public static void main(String args[]) {
    System.out.println(Math.PI);
    System.out.println(Math.sqrt(9));
  }
}
```

```
import static java.lang.Math.*;

public class Main {
  public static void main(String args[]) {
    System.out.println(PI);
    System.out.println(sqrt(9));
  }
}
```

Q: Why was no import statement required on the left?

Absolute Value

`static int abs(int x)`	Returns the absolute value of an `int` value
`static double abs(double x)`	Returns the absolute value of a `double` value

Remember from lesson 2.4:

Overloaded methods are two or more methods in the same class that have the same name but different parameters.

Calling `abs` method

Ex 1:
```
int a = Math.abs(-4);
```

Ex 2:
```
double b = Math.abs(-12.34);
```

Ex 3:
```
int favoriteNumber = 3;

    int c = Math.abs(favoriteNumber);
```

Power

`static double pow(double base, double exp)`

Returns $\texttt{base}^{\texttt{exp}}$.

Assumes `base` > 0, or `base` = 0 and `exp` > 0, or `base` < 0 and `exp` is an integer

(What happens if base < 0 and exp is not an integer? It returns NaN, which means "Not A Number." double can't represent imaginary numbers. Math.pow(-1, 0.5) is equivalent to Math.sqrt(-1), which also returns NaN since the answer is an imaginary number.)

P.S. Imaginary numbers can be represented in Java, but it's not built-in to the primitive data types or the standard library. You can use a third party library or write your own.

Calling `pow` method

Ex: `double d = Math.pow(5, 2);`

Square root

`static double sqrt(double x)`

Returns the positive square root of a double value.

(If `x` is negative, this will return `NaN` - Not A Number.)

Calling `sqrt` method

Ex: `double e = Math.sqrt(144);`

Random number

`static double random()`

Returns a `double` value greater than or equal to `0.0` and less than `1.0`

Example:

```
double randomValue = Math.random();

// Example output: 0.6573016382857277
```

Write code that generates a random `int` between 0 to 9

```
int random = (int) (Math.random() * 10);
```

Write code that generates a random `int` between 1 and 10

```
int random = (int) (Math.random() * 10) + 1;
```

To see all the `Math` methods, look at the `Java` documentation!

https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html

Practice on your own!

Go to replit.com and do the **section2.8** and **section2.9** exercises.

Try the CSAwesome lesson 2.8 exercises too.

Go to CSAwesome lesson 2.9 and complete the 5 multiple choice problems under **Check your understanding**

If you finish those, move on to the Programming Challenge towards the end of the lesson