Lesson 51: Passing by Value vs. Reference (W17D1)

Balboa High School

Michael Ferraro

December 7, 2015

Do Now

Predict the output of the snippet below, and then test your prediction.

```
Fraction frxn = new Fraction();

for ( int i = 1 ; i <= 3 ; i++ ) {
    frxn = frxn.add(new Fraction(i, 10));
}

System.out.println( frxn.getValue() );
```

Afterward, figure out about how many `Fraction` objects were in memory just before your program terminated.

Aim

Students will learn the difference between passing primitives versus objects as parameters to methods.

Recall from last lesson...

Last lesson's "Do Now" was used to illustrate a point.

```
double a = -2.353;
double b = a;
a *= -1;

System.out.println("a = " + a);
System.out.println("b = " + b);
```

`a` and `b` retained distinct values. These **primitive variables** hold only **values**, not pointers/references to objects somewhere in memory. Therefore, changing one doesn't change the other, even though — at one time — one was set equal to the other (i.e., `double b = a`).

Recall from last lesson. . .

The situation for objects is different.

```
Fraction f1 = new Fraction( 2, 9 );
Fraction f2 = f1;

f2.num = 3;  //changes obj f1 is referring to, too!
f1.reduce(); //changes obj that f2 points to!

System.out.println("f1 = " + f1);
System.out.println("f2 = " + f2);
```

When it comes to objects, the variables don't hold values, but instead hold memory locations, or **references** to objects. So when one is set equal to another (i.e., `Fraction f2 = f1`), that causes both variables to point to the same object. Therefore, changing one really changes "both" (but there aren't really two objects!).

When Primitives are Parameters

Predict the output of the class below.

```
public class PrimitiveParameter {

    public static void signFlipper(int n) {
        n *= -1;
    }

    public static void main(String[] args) {
        int p = 9;
        signFlipper(p);
        System.out.println(p);
    }
}
```

When Primitives are Parameters

Why didn't `p`'s value change when we sent `p` to `signFlipper()` as an argument?

When Primitives are Parameters

Why didn't `p`'s value change when we sent `p` to `signFlipper()` as an argument?

Because we sent `p`'s *value*, not `p` itself! If you change the *value* 9, you aren't really changing `p`.

When Primitives are Parameters

Why didn't `p`'s value change when we sent `p` to `signFlipper()` as an argument?

Because we sent `p`'s *value*, not `p` itself! If you change the *value* 9, you aren't really changing `p`.

→ *When you send a* **primitive** *as an argument, you are passing the method its* **value***; this is called* **pass by value***.*

When Objects are Parameters

1. Create a new project called `Lesson51`.
2. Download and import `Coordinates*.java` from here.
3. Read over class `Coordinates` and its driver class; Ask any questions you have.
4. Predict the output of running `CoordinatesDriver`.
5. Run the driver class.
6. Prepare an explanation for why the output is as it is. Be sure to compare/contrast the result with what we observed earlier for primitive parameters.

When Objects are Parameters

```
main() {
    Coordinates pt = new Coordinates( 2.9,-1.8 );
    shiftRight( pt, 3 );
    System.out.println( pt );
}

shiftRight( Coordinates loc, int u) {
    loc.x += u;
}
```

(2.9 , -1.8)

When Objects are Parameters

```
main() {
    Coordinates pt = new Coordinates( 2.9,-1.8 );
    shiftRight( pt, 3 );
    System.out.println( pt );
}

shiftRight( Coordinates loc, int u) {
    loc.x += u;
}
```

(2.9 , -1.8)

When Objects are Parameters

```
main() {
    Coordinates pt = new Coordinates( 2.9,-1.8 );
    shiftRight( pt, 3 );
    System.out.println( pt );
}

shiftRight( Coordinates loc, int u) {
    loc.x += u;
}
```

(2.9 , -1.8)

When Objects are Parameters

```
main() {
    Coordinates pt = new Coordinates( 2.9,-1.8 );
    shiftRight( pt, 3 );
    System.out.println( pt );
}

shiftRight( Coordinates loc, int u) {
    loc.x += u;
}
```

(2.9 , -1.8)

When Objects are Parameters

```
main() {
    Coordinates pt = new Coordinates( 2.9,-1.8 );
    shiftRight( pt, 3 );
    System.out.println( pt );
}

shiftRight( Coordinates loc, int u) {
    loc.x += u;
}
```

(2.9 , -1.8)

When Objects are Parameters

```
main() {
    Coordinates pt = new Coordinates( 2.9,-1.8 );
    shiftRight( pt, 3 );
    System.out.println( pt );
}

shiftRight( Coordinates loc, int u) {
    loc.x += u;
}
```

(2.9 , -1.8)

5.9

When Objects are Parameters

```
main() {
    Coordinates pt = new Coordinates( 2.9,-1.8 );
    shiftRight( pt, 3 );
    System.out.println( pt );
}

shiftRight( Coordinates loc, int u) {
    loc.x += u;
}
```

(2.9 , -1.8)

5.9

When Objects are Parameters

When an object is an argument to a method, a *reference* to that object is sent. So when the method modifies the object via *its* reference, it's really changing the same object in memory that was sent.

We say that objects are **passed by reference** to methods. This is potentially dangerous!

When Objects are Parameters

Sending along a reference to an object could be dangerous if the method that is called *modifies* the object pointed to by the reference. How might you safeguard against that?

When Objects are Parameters

Sending along a reference to an object could be dangerous if the method that is called *modifies* the object pointed to by the reference. How might you safeguard against that?

Solution: Send along a reference to a *copy* of the original object — using a **copy constructor**!

When Objects are Parameters

Sending along a reference to an object could be dangerous if the method that is called *modifies* the object pointed to by the reference. How might you safeguard against that?

Solution: Send along a reference to a *copy* of the original object — using a **copy constructor**!

Exercise: Make an idential copy of the `pt` instance of `Coordinates` and send that copy to `shiftRight()`. You'll need to add a second constructor to `Coordinates`, however:

```
public Coordinates( Coordinates myPt )
```

Next. . .

- Make sure you're not falling behind on PS #9! You should have finished through §5 by now.
- Now you should work on these sections:
  - §6: Method Definitions (Litvin §10.5)
  - §7: Three Ways to Call Methods (Litvin §10.6)
  - §8: Pass by Value vs. Pass by Reference (Litvin §10.7)
  - §9: Use of `return` (Litvin §10.8)
- Make sure you finish the sections listed above within the next two classes (i.e., those sections are your **HW**).
- Next class: `SnackBar`!

HW

- §6: Method Definitions (Litvin §10.5)
- §7: Three Ways to Call Methods (Litvin §10.6)
- §8: Pass by Value vs. Pass by Reference (Litvin §10.7)
- §9: Use of `return` (Litvin §10.8)