Prepared by Linan Qiu <lq2137@columbia.edu>

Static

The keyword `static` seems to confuse a lot of people. It seems intimidating, but it is in fact really simple. Let's start with static variables.

Static Variables

Let's bring in our favorite class: `Pokemon`

```
// Pokemon.java

public class Pokemon {
  public int level;

  public Pokemon(int level) {
    this.level = level;
  }
}
```

Since `level` is a `public` variable, we can access a `Pokemon` instance's `level` directly.

```
// in a main method far far away

Pokemon pikachu = new Pokemon(5);
Pokemon mew = new Pokemon(43);

System.out.println(pikachu.level); // 5
System.out.println(mew.level); // 43
```

This is all cool. However, if we made the variable a `static` variable, things would be very different.

```
// Pokemon.java

public class Pokemon {
  public static int level;

  public Pokemon(int level) {
    this.level = level;
  }
}
```

All instances of a class share the same static variable. That means we'd see the following results:

```
// in a main method far far away

Pokemon pikachu = new Pokemon(5);
Pokemon mew = new Pokemon(43);

System.out.println(pikachu.level); // 43
System.out.println(mew.level); // 43

pikachu.level = 100;

System.out.println(pikachu.level); // 100
System.out.println(mew.level); // 100

mew.level = 1;

System.out.println(pikachu.level); // 1
System.out.println(mew.level); // 1

Pokemon bulbasaur = new Pokemon(50);
// remember the constructor changes the static int level
System.out.println(pikachu.level); // 50
System.out.println(mew.level); // 50
System.out.println(bulbasaur.level); // 50

System.out.println(Pokemon.level); // 50
```

`pikachu`, `mew`, `bulbasaur`, and any Pokemon *new*-ed would share the same static variable. So if you change one of them, all their levels change (because they're all the same variable and share the same location in the memory). This is unlike a normal instance variable (what we had before we made `level` static) when each instance has a separate variable in the memory. Back then, `pikachu.level` refers to a different `int` than `bulbasaur.level`. Now that they're static, they're all the same `int` in the memory.

This means that `pikachu.level` will be the same value as `mew.level` which will also be the same as `bulbasaur.level` and any other Pokemon's `level`. Then, it really makes no sense to write `pikachu.level` or `bulbasaur.level` since they're the same. In fact, Java allows us to write `Pokemon.level` as you can see in the last line of the code.

This also has another nuance – `static` variables exist in the class itself and not an instance. For example, the `level` variable belongs to the `Pokemon` **class** and not the specific instance of `Pokemon` like `pikachu` or `mew`. **Static variables exist within a class, not an instance.**

Should we use `static` variables for `level` in `Pokemon`? **NO. In fact, many students tried making variables in the `Rectangle` class for hw1 static, and this resulted in nonsensical answers**. `Pokemon` illustrates a case where declaring a variable `static` screws with your results. When do you use `static` then?

Static variables useful for declaring constants. For example,

```
//PhysicsReference.java

public class PhysicsReference {
  public static final double GRAVITATIONAL_CONSTANT = 9.81;
  public static final double PI = 3.41
}
```

Then, we can use the variables directly like `PhysicsReference.GRAVITATIONAL_CONSTANT` or `PhysicsReference.PI` instead of doing `PhysicsReference ref = new PhysicsRefeference()` then `ref.getGravity()`. After all, we expect the gravitational constant to be the same all the time, not specific to an instance of the reference. In fact an instance of the reference wouldn't even make sense.

Static Methods

The same idea of not being specific to instances and instead belonging to a class applies to static methods. Let's say that you're trying to build a calculator.

```
// Calculator.java

public class Calculator {
  public int power(int a, int exponent) {
    if (exponent < 0) {
      throw new IllegaOperationException();
    } else {
      int result = 1;
      for (int i = 0; i < exponent; i++) {
        result *= a;
      }
      return result;
    }
  }
}
```

It doesn't really make sense for us to code `Calculator` this way. After all, it'd require us to do this:

```java
// in a main method far far away

Calculator calc1 = new Calculator();
System.out.println(calc1.power(2, 4));

Calculator calc2 = new Calculator();
System.out.println(calc2.power(2, 5));
```

`calc1` and `calc2` are not going to be different ever. This is very unlike the case of `Pokemon` where the variables **should not** be static. Wouldn't it be nice if we could just call `power` from `Calculator` directly? Well turns out that's exactly what `static` methods do.

```java
// Calculator.java

public class Calculator {
  public static int power(int a, int exponent) {
    if (exponent < 0) {
      throw new IllegaOperationException();
    } else {
      int result = 1;
      for (int i = 0; i < exponent; i++) {
        result *= a;
      }
      return result;
    }
  }
}
```

Then, in your main method, you can do this:

```java
// in a main method far far away

System.out.println(Calculator.power(2, 4));
System.out.println(Calculator.power(2, 5));
```

In fact, we can move this main method into the Calculator class if we want to test it directly and conveniently.

```java
// Calculator.java

public class Calculator {
  public static int power(int a, int exponent) {
    if (exponent < 0) {
```

```
      throw new IllegaOperationException();
    } else {
      int result = 1;
      for (int i = 0; i < exponent; i++) {
        result *= a;
      }
      return result;
    }
  }

  public static void main(String[] args) {
    // since we are in the Calculator class, we don't need
    // to type Calculator.power. Only power is necessary
    // since Java can infer that we mean Calculator.power
    System.out.println(power(2, 4));

    // however, you can still do this
    System.out.println(Calculator.power(2, 5));
  }
}
```

By now, you should also realize why Java screams at you to declare methods static whenever you call it directly from the `main` method: if you're calling a method directly from the main method, you wouldn't have instantiated the class yet. Try reasoning this out for yourself:

```
public class Test {
  public static void printHanSolo() {
    System.out.println("Han Solo died");
  }

  public static void main(String[] args) {
    // you didn't do Test test = new Test() then test.printHanSolo()
    // instead, you called the method directly
    // hence, you're making use of a static method
    // and printHanSolo() has to be static
    printHanSolo();
  }
}
```

You can have `private static` method as well. These methods can only be called by other `public static` methods or within the class itself.

For example, here's a recursive algorithm for fast calculation of power (also called fast exponentiation.)

```java
// Calculator.java

public class Calculator {
  public static int power(int a, int exponent) {
    if (exponent < 0) {
      throw new IllegaOperationException();
    } else {
      innerPower(a, exponent);
    }
  }

  private static int innerPower(int a, int exponent) {
    if (exponent == 1) {
      return a;
    }
    if (exponent == 2) {
      return a * a;
    }
    if (exponent % 2 == 0) {
      return innerPower(innerPower(a, b / 2), 2);
    } else {
      return a * innerPower(innerPower(a, (b - 1) / 2), 2);
    }
  }

  public static void main(String[] args) {
    // we are inside Calculator now, so we can use both the
    // public and private methods
    System.out.println(power(2, 4));
    System.out.println(innerPower(2, 4));
  }
}
```

“‘java // Test.java

public class Test { public static void main(String[] args) { System.out.println(Calculator.power(2, 4)); // ok // we are no longer in the Calculator class, so // we can only access the public methods // so we can’t do this: // System.out.println(Calculator.innerPower(2, 4)); } }