Instance Methods

A **static method** is also referred to as a **class method**, and it is called without an object. Because of this, static methods can't access any attributes (**instance variables**) or other non-static methods without first creating an object. If you want to access attributes and non-static methods, you must first *instantiate* an object.

Non-static methods, which we will refer to as **instance methods** or **object methods,** are called using an object, and therefore have access to that object's instance variables. Note that, for these methods, the method header will **not** include the keyword **static**.

There are three steps to creating and calling an instance method:

1. **Object of the Class**: Declare an object of your class in the main method or from outside the class.

```
// Step 1: declare an object in main or from outside the class

Classname objectName = new Classname();
```

2. **Method Definition**: write the method's **header** and **body** code as shown below:

```
// Step 2: Define the method in the class

// method header

public void methodName() {

// method body for the code

}
```

3. **Method Call**: whenever you want to use the method, call `objectName.methodName();`

```
// Step 3: call the object's method

objectName.methodName(); //Step 2
```

How is this different than calling a static method? Notice the **object reference and dot `.` notation** before the method name. Since the method may set and get instance variables, you must call the method on an actual instance of the class.

InfoWarningTip
Remember that **classes** don't hold instances of their variables. They're blueprints awaiting inputs. We can call their static methods and take advantage of their functionality, but their variables (and any methods that might require those variables to run) don't exist. A blueprint doesn't have all of the wood, concrete, and glass that it takes to build the house, right?

#Practice

Consider the following class, which uses the instance variable dollars to represent the money in a wallet in dollars.

```
public class Wallet {
private double dollars;

public double putMoneyInWallet(int amount) {
/* missing code */
}
}
```

The `putMoneyInWallet` method is intended to increase the `dollars` in the wallet by the parameter amount and then return the updated `dollars` in the wallet.

Which of the following code segments should replace *missing code* so that the `putMoneyInWallet` method will work as intended?

Consider the Liquid class below.

```
public class Liquid {
private int currentTemp;
private int boilingPoint;

public Liquid(int ct, int bp) {
currentTemp = ct;
boilingPoint = bp;
```

```
}

public boolean isBoiling(int amount) {
/* missing code */
}
}
```

The `isBoiling` method is intended to return `true` if increasing the `currentTemp` by the parameter amount is greater than or equal to the `boilingPoint`, or otherwise return `false`. Which of the following code segments can replace *missing code* to ensure that the `isBoiling` method works as intended?

```
I. if (currentTemp + amount < boilingPoint) {
return false;
} else {
return true;
}
II. if (amount > currentTemp) {
return false;
} else {
return currentTemp;
}
III. if (amount + currentTemp >= boilingPoint) {
return true;
} else {
return false;
}
```

#Summary

- **Procedural Abstraction** (creating methods) reduces the complexity and repetition of code. We can name a block of code as a method and call it whenever we need it, abstracting away the details of how it works.
- A programmer breaks down a large problem into smaller subproblems by creating methods to solve each individual subproblem.
- To write methods, write a **method definition** with a **method signature** like `public void chorus()` and a **method body** in `{}`. When you need to use your

method, write a method call using a `objectName.methodName` and passing in the arguments the method needs to do its job.

- To call an object's method, you must use the object's name and the dot `.` operator followed by the method name. For example **object.method();**
- When you call a method, you can give or pass in **arguments** or **actual parameters** to it inside the parentheses as in `object.method(arguments)`. The arguments are saved in local **formal parameter** variables that are declared in the method header, for example: `public void method(type param1, type param2) { … }`.
- Values provided in the arguments in a method call need to correspond to the **order** and **type** of the parameters defined in the method signature.
- When an actual parameter is a primitive value, the formal parameter is initialized with a copy of that value. Changes to the formal parameter have no effect on the corresponding actual parameter.
- When an actual parameter is a reference to an object, the formal parameter is initialized with a **copy** of that reference, **not** a copy of the object. The formal parameter and the actual parameter are then aliases (aka references), both referring to the same object at that particular location in the system's memory.
- When an actual parameter is a reference to an object, the method or constructor could use this reference to alter the state of the original object. However, it is good programming practice to **not** modify mutable (changeable) objects that are passed as parameters unless required in the specification.