Writing Methods

Up until this point, you've written most of your code in the `main` method. As we continue to build out our code, however, we're going to need to create new methods that stand on their own. These methods will then be called by the `main` method. This might make you wonder: why have multiple methods instead of just one in the first place?

Procedural Abstraction allows us to name a block of code as a method and call it whenever we need it, thereby abstracting away the details of how it works. This serves to organize our code by allowing us to focus on how each block functions and reduce repetition. In addition, it helps with debugging and maintenance, since changes to that block of code only need to happen in one place. We can then call these methods wherever we need that functionality, rather than having to rewrite them. Here are some of the main reasons to use multiple methods in your programs:

- **Reducing Complexity:** Dividing one big problem into subproblems to solve it one piece at a time. This is often necessary for the kinds of problems you'll be solving on the job, so it's good to start practicing this way of thinking as soon as possible.
- **Reusing Code:** Not only should we avoid repeating ourselves in our code, but we should also strive to ensure that our code is reusable. The more atomic (indivisible) our logic is, the more likely it is that the code can be reused elsewhere.
- **Maintainability and Debugging:** Smaller methods are easier to debug and understand, but it also makes updating code easier. In the examples to come, try to think of how many errors could be accidentally introduced if repetitious code had to be edited everywhere it appears.

Let's look at an example with repetition. Then we'll create a method to reduce the redundant code:

```
public static void main(String args[]) {
System.out.println("I'm looking over a four-leaf clover");
System.out.println("That I overlooked before");
System.out.println("One leaf is sunshine, the second is rain");
System.out.println("Third is the roses that grow in the lane");
System.out.println();
System.out.println("No need explaining, the one remaining");
System.out.println("Is somebody I adore");
System.out.println("I'm looking over a four-leaf clover");
System.out.println("That I overlooked before");
}
```

The two-line chorus is repeated at the beginning and ending of the song.

When you see duplicate lines of code, that is a signal for you to make a new method! A method is a **named** set of statements. When we want to execute the statements, we call the method using its name. In a subsequent lesson, you will create

methods that are called using an object, referred to as **instance methods** or **object methods**. The methods in this unit are called without an object, so they are what we call **static methods**. Static methods are also referred to as **class methods** because they live in the class, not an object instantiated from that class.

InfoWarningTip
Static Methods: can be called without an object, as in `method()`
Instance Methods: must be called using the object they belong to, as in `object.method()`

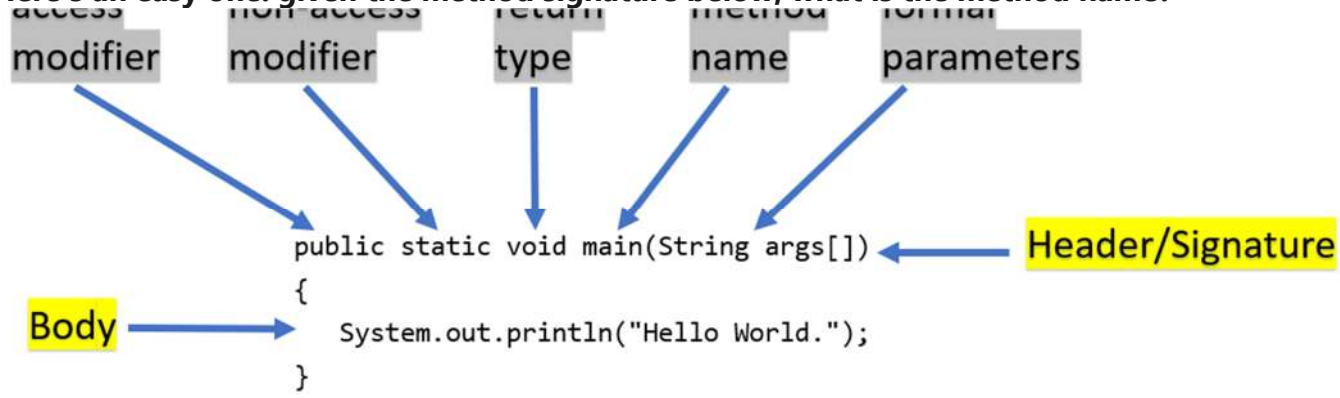
#Writing Static Methods

There are two steps to writing and using a static method:

- **Step 1: The Method Definition**
- **Step 2: The Method Call**

You define a method by writing the method's **header** and **body**. The header is also called a method **signature**. The parts of the main method header are shown in the figure below. These include an access modifier, static modifier, return type, name, and formal parameters. The method body consists of a set of statements enclosed in curly braces { }.

Here's an easy one: given the method signature below, what is the method name?



public

static

void

main

The code below contains a `chorus()` method definition that we could write to encapsulate the two lines that get repeated in the song from our first example.

```
// Step 1: define a new method named chorus
public static void chorus() {
System.out.println("I'm looking over a four-leaf clover");
System.out.println("That I overlooked before");
}
```

Whenever you want to use a method, you call it using the method name followed by a set of parentheses `()`. The method header `public static void chorus()` indicates the return type is `void` and there are no formal parameters between the parentheses, which means you can call the method as shown:

CopyC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdownPascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
// Step 2: call the chorus method
chorus();
```

Notice that we can just call the static method, we don't need to create an object to use for calling the method. The main method can call the chorus method multiple times to repeat the two repetitious lines of the song.

#Coding Exercise

Run the following code to see the song print out. Notice the first line of code in the main method is a call to the new method `chorus()`. Complete the challenge commented in the code:

Run

Song.java

```
public class Song {
  // The chorus method
  public static void chorus() {
     System.out.println("I'm looking over a four-leaf clover");
     System.out.println("That I overlooked before");
  }
  public static void main(String args[]) {
    chorus();
    System.out.println("One leaf is sunshine, the second is rain");
    System.out.println("Third is the roses that grow in the lane");
    System.out.println();
    System.out.println("No need explaining, the one remaining");
    System.out.println("Is somebody I adore");

    // Complete the song by calling the chorus() method again
```

CONSOLESHELL

#Check Your Understanding

A method definition consists of a method header and a method body. Click on the method header for the method named `greet` in the following code.

```
public class GreetingExample {
public static void greet() {
System.out.println("Hello!");
System.out.println("How are you?");
}
public static void main(String[] args) {
System.out.println("Before greeting");
greet();
System.out.println("After greeting");
}
}
```

Click on all statements contained within the `greet` method body.

```
public class GreetingExample {
public static void greet() {
System.out.println("Hello!");
System.out.println("How are you?");
}
public static void main(String[] args) {
System.out.println("Before greeting");
greet();
System.out.println("After greeting");
}
}
```

Click on the greet method call.

```
public class GreetingExample {
public static void greet() {
System.out.println("Hello!");
System.out.println("How are you?");
}
public static void main(String[] args) {
System.out.println("Before greeting");
greet();
System.out.println("After greeting");
}
}
```

Given the GreetingExample class in the previous problem, how many times is System.out.println called in total when the program runs? (Please provide a number, not the spelled-out word to answer this question.)

#Flow of Execution - Stack Diagrams

A class can contain several methods. It can be tempting to think that the methods are executed in the order they appear in the class, but this is not the case.

A program always begins at the first statement in the **main method**. Each statement in the main is executed one at a time until you reach a method call. A method call causes the program execution to jump to the first line of the called method. Each statement in the called method is then executed in order. When the called method is done, the program returns back to the main method to pick up where it left off.

How does the program keep track of all of this? The Java runtime environment keeps track of the method calls using a **call stack**. The call stack is made up of stack **frames**. Each time a method is called, a new frame is created and added to the stack. A frame contains the method's parameters and local variables, along with the number of the current line that is about to be executed.

The CodeLens Visualizer represents the call stack using a **stack diagram**, with each method frame drawn as a box. When a method is called, a new frame is added to the bottom of the stack diagram. You can tell which method is currently executing by looking at the **bottom** of the stack. Let's walk through an example in the tabbed window below.

InfoWarningTip
There's a link to the visualizer on the seventh tab below, but do go through the example one step at a time to read the information about them before you jump over to the visualizer. Understanding how the stack proceeds through your code is going to become increasingly important as your code becomes more complex.

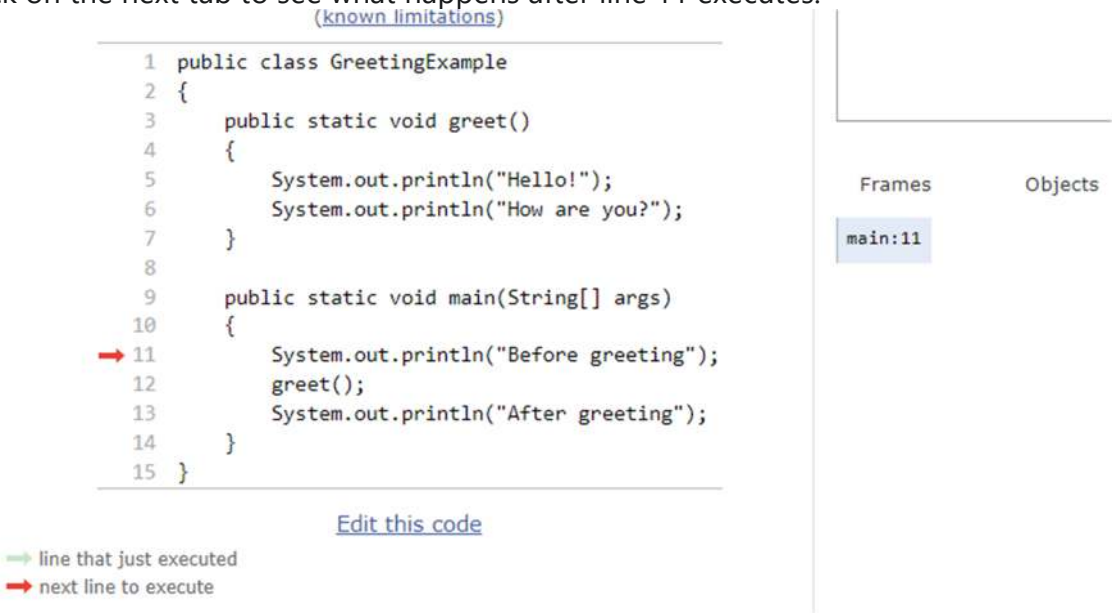
#Check your understanding

Click on each tab to observe the flow of control for the `GreetingExample` class.

The program starts at the first line of the main method. The red arrow shows that line 11 is next to execute.

The stack diagram is in the right portion of the screen print, below the print output section where it says "Frames". There is a single frame for the main method `main:11`, indicating line 11 is the current line in the method.

Click on the next tab to see what happens after line 11 executes.



#Check your understanding

What does the following code print?

```
public class LikeFood {

public static void fruit() {
System.out.println("apples and bananas!");
}

public static void consume() {
System.out.print("eat ");
}

public static void main(String[] args) {
System.out.print("I like to ");
consume();
consume();
consume();
fruit();
}
}
```

apples and bananas! eat I like to.

I like to consume consume consume fruit.

I like to apples and bananas! eat.

I like to eat eat eat apples and bananas!

What is the last thing printed when this program runs?

```
{
    public static void o()
    {
        System.out.println("called o");
        n();
        System.out.println("after n");
    }
    public static void n()
    {
        System.out.println("called n");
    }
    public static void m()
    {
        System.out.println("called m");
        n();
        System.out.println("after n");
    }
    public static void main(String[] args)
    {
        m();
        o();
    }
}
```

called n

called o

called m

after n

#Coding Exercise

A refrain is similar to a chorus, although usually shorter in length such as a single line that gets repeated. In the song below, the refrain is "The farmer in the dell". Add a method named `refrain` and update the main method to call the new method 3 times in

place of the duplicate print statements. Run your program to ensure the output is correct.

Run

```
public class FarmerSong {

  // New method to print the refrain
  public static void refrain() {
     System.out.println("The farmer in the dell");
  }

  public static void main(String args[]) {
     refrain(); // Call the new method
     refrain(); // Call the new method
     System.out.println("Heigh ho the derry-o");
     refrain(); // Call the new method
  }

}
```

#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.
- Write a **method definition** with a **method signature** like `public static void chorus()` and a **method body** that consists of statements nested within `{}`.
- Call the method using its name followed by parentheses, as in `chorus()`. The method call executes the statements in the method body.

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.

   CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdownPascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

   ```
   1
   2
   // 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:

   CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdownPascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

   ```
   1
   2
   3
   4
   5
   // 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();`

CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdownPascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
1
2
// 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?

#Video Example

You've done a lot of reading already today, so how about we mix it up a little? Follow along with the example in the video to see an instance method in action:

Open in new tab

#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?

CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdown PascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
amount += dollars;
return dollars;
```

CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdown PascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
dollars = amount;
return amount;
```

CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdown PascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
dollars += amount;
return dollars;
```

CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdown PascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
dollars = dollars + amount;
return amount;
```

CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdown PascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
amount = dollars + amount;
return dollars;
```

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;
}
```

I only

II only

III only

I and III only.

I, II, III

#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.

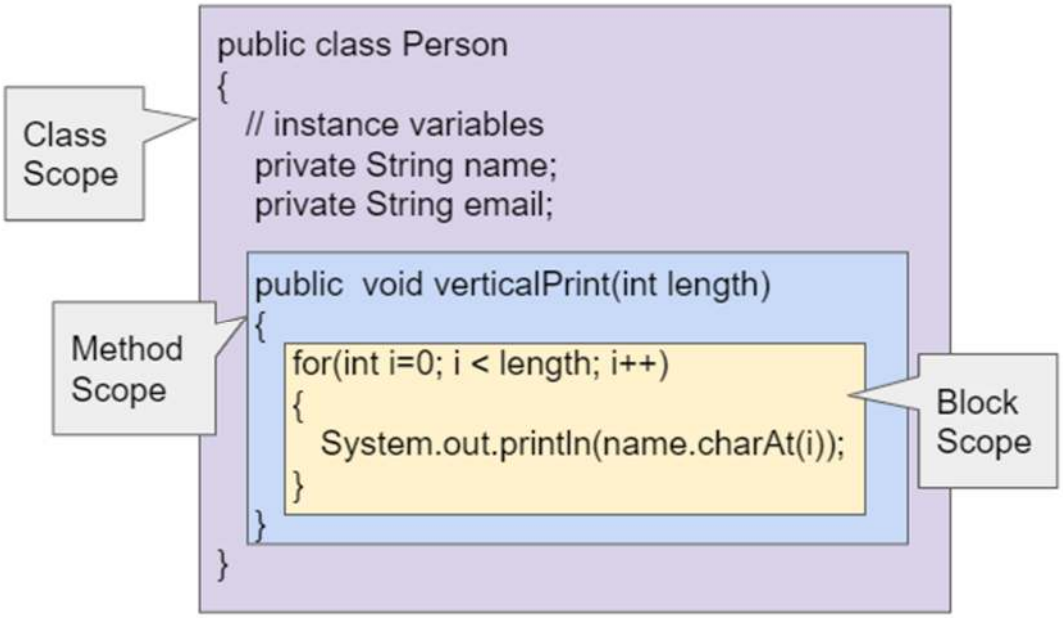
Scope and Access

The **scope** of a variable is defined as where a variable is accessible or can be used. The scope is determined by where you declare the variable when you write your programs. When you declare a variable, look for the closest enclosing curly brackets `{ }` – this is its scope.

Java has 3 levels of scope that correspond to different types of variables:

- **Class Level Scope** for **instance variables** inside a class.
- **Method Level Scope** for **local variables** (including **parameter variables**) inside a method.
- **Block Level Scope** for **loop variables** and other local variables defined inside of blocks of code with `{}`.

The image below shows these 3 levels of scope.



Class, Method, and Block Level Scope

#Check Your Understanding

Click on all the variable declarations that are at **Class** Level Scope. See the image above for help.

```
public class Name {

private String first;
public String last;

public Name(String theFirst, String theLast) {
String firstName = theFirst;
first = firstName;
last = theLast;
}
}
```

Click on all the variable declarations that are at **Method** Level Scope.Remember that the parameter variables and the local variables declared inside a method have Method Level Scope.

```
public class Name {

private String first;
public String last;

public Name(String theFirst, String theLast) {
String firstName = theFirst;
first = firstName;
last = theLast;
}
```

```
}
```

#Local Variables

Local variables are variables that are declared inside a method, usually at the top of the method. These variables can only be used within the method and do not exist outside of the method. Parameter variables are also considered local variables that only exist for that method. It's good practice to keep any variables that are used by just one method as local variables in that method.

Instance variables at class scope are shared by all the methods in the class and can be marked as public or private with respect to their access outside of the class. They have Class scope regardless of whether they are public or private.

Another way to look at scope is that a variable's scope is where it lives and exists. You cannot use the variable in code outside of its scope. The variable does not exist outside of its scope, but can be used inside of methods that share its scope.

#Coding Exercise

Try the following code to see that you cannot access the variables outside of their scope levels in the `toString()` method. Reach out to your team on Slack and explain to a teammate why you can't access these variables. Talk it through and answer each others' questions, then come back here. Try to fix the errors in the code by either using variables that are in scope or moving the variable declarations so that the variables have the required scope to function.

Run

Person.java

```
public class Person {
   private String name;
   private String email;
   public Person(String initName, String initEmail) {
      name = initName;
      email = initEmail;
   }
   public String toString() {
     for (int i=0; i < 5; i++) {
        int id = i;
     }
     // Problem 1: Variable 'i' and 'id' are defined within the for loop's sco
pe.
     // You can't access them outside the loop.
     System.out.println("i at the end of the loop is " + i);
```

CONSOLESHELL

If there is a local variable with the same name as an instance variable, the variable name will refer to the local variable instead of the instance variable, as seen below. We'll see in the next lesson, that we can distinguish between the local variable and the instance variable using the keyword `this` to refer to this object's instance variables.

Run

Person.java

```
public class Person {
   private String name;
   private String email;

   public Person(String initName, String initEmail) {
      name = initName;
      email = initEmail;
   }

   public String toString() {
     String name = "unknown";
     // The local variable name here will be used,
     //  not the instance variable name.
     return  name + ": " + email;
   }

   // main method for testing
   public static void main(String[] args) {
      // call the constructor to create a new person
      Person p1 = new Person("Sana", "sana@gmail.com");
      System.out.println(p1);
   }
}
```

#Programming Challenge : Debugging

Debug the following program that has scope violations. Then, add comments that label the variable declarations as class, method, or block scope.

Run

TesterClass.java

```
public class TesterClass {
   public static void main(String[] args) {
      Fraction f1 = new Fraction();
      Fraction f2 = new Fraction(1,2);
      System.out.println(f1);
      System.out.println(f2.numerator / f2.denominator);
   }
```

```
 }
/** Class Fraction */
class Fraction {
   //  instance variables
   private int numerator;
   private int denominator;
   // constructor: set instance variables to default values
```
CONSOLESHELL

#Practice

Consider the following class definitions. Which of the following best explains why the class will not compile?

```
public class Party {
private int boxesOfFood;
private int numOfPeople;

public Party(int people, int foodBoxes) {
numOfPeople = people;
boxesOfFood = foodBoxes;
}

public void orderMoreFood(int additionalFoodBoxes) {
int updatedAmountOfFood = boxesOfFood + additionalFoodBoxes;
boxesOfFood = updatedAmountOfFood;
}

public void eatFoodBoxes(int eatenBoxes) {
boxesOfFood = updatedAmountOfFood - eatenBoxes;
}
}
```

The class is missing an accessor method.

The instance variables boxesOfFood and numOfPeople should be designated public instead of private.

The return type for the Party constructor is missing.

The variable updatedAmountOfFood is not defined in eatFoodBoxes method.

The Party class is missing a constructor

Consider the following class definition.

```
public class Movie
{
private int currentPrice;
```

```
private int movieRating;

public Movie(int p, int r)
{
currentPrice = p;
movieRating = r;
}

public int getCurrentPrice()
{
int currentPrice = 16;
return currentPrice;
}

public void printPrice()
{
System.out.println(getCurrentPrice());
}
}
```

Which of the following reasons explains why the printPrice method is not functioning as intended and only ever prints out a value of `16`?

The private variables currentPrice and movieRating are not properly initialized.

The private variables currentPrice and movieRating should have been declared public.

The printPrice method should have been declared as private.

currentPrice is declared as a local variable in the getCurrentPrice method and set to 16, and will be used instead of the instance variable currentPrice.

The currentPrice instance variable does not have a value.

#Summary

- **Scope** is defined as where a variable is accessible or can be used.
- Local variables can be declared in the body of constructors and methods. These variables may only be used within the constructor or method and cannot be declared to be public or private.
- When there is a local variable with the same name as an instance variable, the variable name will refer to the local variable instead of the instance variable.
- Formal parameters and variables declared in a method or constructor can only be used within that method or constructor.