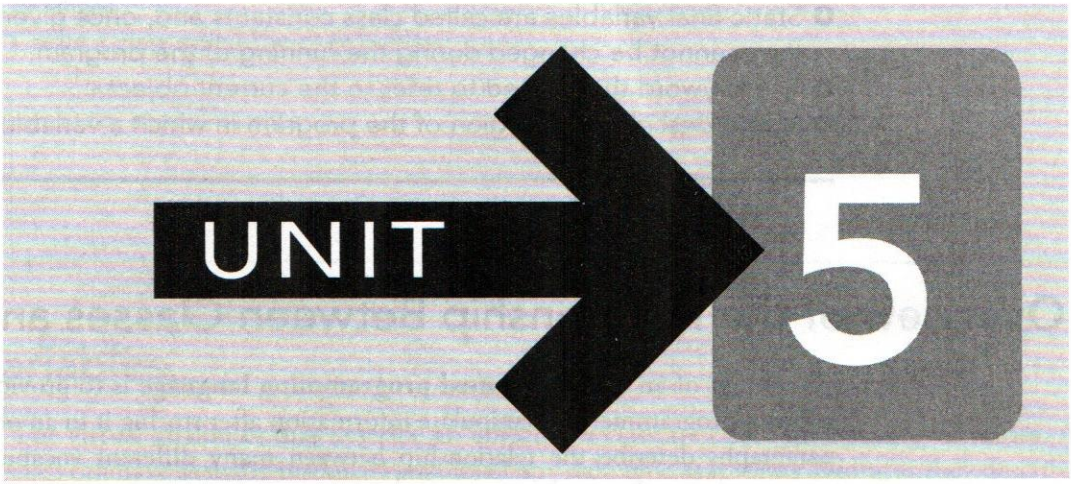


Writing Classes

IN THIS UNIT

Summary: This unit is the continuation of Unit 2, "Using Objects." In this unit you will learn how to design your own classes. Full understanding of designing classes, writing constructors, and writing methods are key to doing well on the AP Computer Science A Exam.

Key Ideas

- Java is an object-oriented programming language whose foundation is built on classes and objects.
- A class describes the characteristics of any object that is created from it.
- An object is a virtual entity that is created using a class as a blueprint.
- Objects are created to store and manipulate information in your program.
- An instance variable is used to store an attribute of an object.
- A method is an action that an object can perform.
- The keyword `new` is used to create an object.
- A reference variable stores the address of the object, not the object itself.
- Objects are passed by reference, while primitives are passed by value.
- Data encapsulation is a way of hiding user information.
- Overloaded constructors have the same name but different parameter lists.
- Overloaded methods have the same name but different parameter lists.
- Static variables are called class variables and are shared among all objects of the same class.

- Static final variables are called class constants and, once given a value, they cannot be changed during the running of the program.
- The keyword this is used to refer to the current object.
- Scope describes the region of the program in which a variable is known.

Overview of the Relationship Between Classes and Objects

The intent of an **object-oriented programming language** is to provide a framework that allows a programmer to manipulate information after storing it in an object. The following paragraphs describe the relationship between many different vocabulary terms that are necessary to understand Java as an object-oriented programming language. The terms are so related that I wanted to describe them all together rather than in separate pieces.

> **Warning: Proceed with Caution**
> The next set of paragraphs includes many new terms. You may want to reread this section many times. If you expect to earn a 5 on the exam, you must master everything that follows. Challenge yourself to learn it so well that you can explain this entire chapter to someone else.

Java classes represent things that are nouns (people, places, or things). A **class** is the blueprint for constructing all **objects** that come from it. The **attributes** of the objects from the class are represented using **instance variables**. The values of all of the instance variables determine the **state** of the object. The state of an object changes whenever any of the values of the instance variables change. **Methods** are the virtual actions that the objects can perform. They may either **return** an answer or provide a service.

Objects are created using the class that holds the blueprint of the object. The class contains a piece of code called a **constructor**, which tells the computer how to build (construct) an object from that class. The keyword **new** is used whenever the programmer wants to create an object from a class. When `new` is followed by the name of a constructor of a class, a new object is created. The action of constructing an object is also referred to as **instantiating** an object and the object is referred to as an **instance** of the class. In addition to creating the new object, a **reference** to the object is created. The **object reference variable** holds the memory address of the newly created object and is used to call methods that are contained in the object's class.

Writing user-defined classes will be discussed thoroughly in Unit 5. This unit introduces the use of the `String`, `Math`, `Integer`, and `Double` classes that are available.

The `class` Declaration

When you design your own class, you should create a file that contains the **class declaration**. The name of the class is identified in this declaration. By Java naming convention, a class name always starts with an uppercase letter and is written in camel case. Classes are designated public so users can create objects from that class.

Example
Create a class called the `Circle` class:

```
public class Circle                    // Class declaration
{

}
```

Instance Variables

The virtual attributes that describe an object of a class are called its **instance variables**. A class can have as many instance variables of any data type as it wants as long as they describe some characteristic or feature of an object of the class. Unlike local primitive variables, default values are assigned to primitive instance variables when they are created (int variables are 0, doubles are 0.0, booleans are false, and objects are null). The phrase **has-a** refers to the instance variables of a class as in: an objectName has-a instanceVariableName.

Example
Every `Circle` object has-a radius so create an instance variable called radius:

```
public class Circle                    // Class declaration
{
   private double radius;              // Instance variable
}
```

`private` Versus `public` Visibility

The words **private** and **public** are called **visibility modifiers** (or **access level modifiers**). Access to attributes should be kept internal to the class, so instance variables are designated as `private`. Using the word `private` in an instance variable declaration ensures that other classes do not have access to the data stored in the variable without asking the object to provide it. Using the word `public` in the class declaration ensures that any programmer can make objects from that class.

`private` Versus `public`
On the AP Computer Science A Exam, always give instance variables private access visibility. This ensures that the data in these variables is hidden.

Constructors

Constructors are the builders of the virtual objects from a class. They are used in combination with the keyword `new` to create an object from the class. Constructors have the same name as the class and are typically listed near the top of a class. Constructors are designated `public`. Constructors are used to set the initial state of an object, which should include initial values for all of its instance variables.

No-Argument Constructor

The constructor is the code that is called when you create a new object. When you define a new class, it comes with **no-argument constructor**. Depending on the IDE that you use, you may or may not see it. It is generally recommended that you write one anyway, especially when you are first learning classes.

> **The No-Argument Constructor**
>
> The no-argument constructor gets its name because no information is passed to this constructor when creating a new object. It is the constructor that allows you to make a generic object from a class.

Parameterized Constructors and Parameters

If you will know some information about an object prior to creating it, then you may want to write a **parameterized constructor** in your class. You use the parameterized constructor to give initial values to the instance variables when creating a brand-new object. The parameters that are defined in the parameterized constructor are placed in a **parameter list** and are on the same line as the declaration of the constructor.

The process of sending the initial values to the constructor is called **passing a parameter** to the constructor. The actual value that is passed to the constructor is called an **argument** (or **actual parameter**) and the variable that receives the value inside the constructor is called the **formal parameter**. You include a line of code in the parameterized constructor that assigns the value of the formal parameter to the instance variable.

If you choose to write a parameterized constructor for a class, then the default, no-argument constructor vanishes. This is why it is generally recommended that you simply write your own.

Example

Write the no-argument constructor and one parameterized constructor for the `Circle` class. The parameterized constructor must have a parameter variable for the radius of a circle.

```
public class Circle                          // Class declaration
{
   private double radius;                    // Instance variable

   public Circle()                           // No-argument constructor
   {
      radius = 0.0;
   }

   public Circle(double rad)                 // Parameterized constructor
   {
      radius = rad;                          // The radius is set to rad
   }
}
```

Parameters in the Parameter List

A parameter is a variable that is located in a parameter list in the constructor declaration. A pair of parentheses after the name of the constructor encloses the entire parameter list.

For example, the parameterized constructor for the `Circle` class has one parameter variable in its parameter list. The name of the parameter is rad (short for radius) and its data type is a double. The no-argument constructor for the `Circle` class does not have any parameters in its parameter list.

```
public Circle()                    // no parameters in the parameter list
{
  radius = 0.0;
}
public Circle(double rad)          // one parameter in the parameter list
{
   radius = rad;
}
```

Methods

The real power of an object-oriented programming language takes place when you start to manipulate objects. A **method** defines an action that allows you to do these manipulations. A method has two options. It may simply perform a service of some kind, or it may compute a value and give the answer back. The action of giving a value back is called **returning** a value.

Methods that simply perform some action are called **void** methods. Methods that return a value are called **return** (or **non-void**) methods. return methods must define what data type they will be returning and include a `return` statement in the method. A `return` method can return any kind of data type. In non-void methods, a return expression compatible with the return type is evaluated, and a copy of that value is returned. But if the return expression is a reference to an object, a copy of that reference is returned, not a copy of the object. This is referred to as "return by value."

Using the word **public** in the **method declaration** ensures that the method is available for other objects from other classes to use. A method declared **private** is only accessible within the class.

The `return` Statement on the AP Computer Science A Exam

Methods that are supposed to return a value must have a reachable `return` statement. Methods that are void do not have a `return` statement. This is a big deal on the Free-response questions of the AP Computer Science A Exam as you will lose points if you include a `return` statement in a `void` method or a constructor.

Accessor and Mutator Methods

When first learning how to write methods for a class, beginning programmers learn two types of methods:

1. Methods that allow you to access the values stored in the instance variables
2. Methods that allow you to modify the values of the instance variables

Methods that return the value of an instance variable are called **accessor** (or **getter**) methods. Methods that change the value of an instance variable are called **mutator** (or **modifier** or **setter**) methods. Methods that don't perform either one of these duties are simply called methods.

Is My Carbonated Soft Drink a Soda or a Pop?

They both refer to the same thing, it just depends on where you live. In a similar way, the methods that retrieve the instance variables of a class can be referred to as accessor or getter methods. The methods that change the value of an instance variable can be called modifier, mutator, or setter methods.

Declaring a Method

Methods are declared using a line of code called a **method declaration** statement. The declaration includes the access modifier, the return type, the name of the method, and the parameter list. The **method signature** only includes the name of the method and the parameter list.

General Form of a Method Declaration

```
accessModifier returnType methodName(parameterList)
```

One-Track Mind

Methods should have only one goal. The name of the method should clearly describe the purpose of the method.

`toString` Method

It's a good idea to override the `toString()` method that is provided by the `Object` class. By doing this, the class designer (that's you) has control over what is printed each time a user calls the `System.out.print()` method by displaying a meaningful representation of the attributes rather than simply a reference to the memory location (what good is that?).

In the `Circle` class below, if the `toString()` method was not overidden, then the call to `System.out.println(myCircle)` would print `Circle@7852e922` (or some other value referring to the location in memory).

Example

Using the `Circle` class already written, write an accessor method for the radius, a mutator method for the radius, a method that calculates and returns the area of the circle, and a `toString` method that returns a string describing the circles attributes.

```
public class Circle                          // Class declaration
{
   private double radius;                    // Instance variable

   public Circle()                           // No-argument constructor
   {
      radius = 0.0;
   }

   public Circle(double rad)                 // Parameterized constructor
   {
      radius = rad;                          // The radius is set to rad
   }

   public double getRadius()                 // Accessor method
   {
      return radius;
   }

   public void setRadius(double rad)         // Mutator method
   {
      radius = rad;                          // The radius is set to rad
   }

   public double getArea()                   // Method that calculates area
   {
      return 3.14 * radius * radius;         // A simple way to compute area
   }

   public String toString()
   {
      String str;
      str = "The circle has radius " + radius + " and area " +  getArea();
      return str;
   }
       // a string is built with all of the information about the circle
}
// End of Circle class
```

Java Ignores Spaces

Java ignores all spacing that isn't relevant. The compiler doesn't care how pretty the code looks in the IDE; it just has to be syntactically correct. On the AP Computer Science A exam, you will sometimes see code that has been streamlined to save space on the page.

For example, the `getRadius` method for the `Circle` class could appear like this on the exam:

```
public double getRadius(){
   return radius;    }
```

Putting It All Together: The `Circle` and `CircleRunner` Classes

I'm now going to combine the `Circle` class that we just created with another class to demonstrate code that follows an example of two classes interacting in an object-oriented setting. The `Circle` class defined how to make a `Circle` object. The `CircleRunner` class is where the programmer creates virtual `Circle` objects and then manipulates them. And in case you are wondering, the `CircleRunner` class doesn't have to be named that specifically, but it does convey the message that "hey, I'm the class that runs the program with circles".

Summary of the `Circle` Class

Every circle has-a radius, so the `Circle` class defines an instance variable called radius. The class also has two constructors as ways to build a `Circle` object. The no-argument constructor is used whenever you don't know the radius at the time that you create the `Circle` object. The radius gets the default value for a double, which is 0.0. The parameterized constructor is used if you know the radius of the circle at the time you construct the `Circle` object. You pass the radius when you call the parameterized constructor and the constructor assigns the value of its parameter to the instance variable.

The `Circle` class also has four methods:

1. The first method, `getRadius`, returns the radius from that object. Since the method is *getting* the radius for the programmer, it is called an accessor (or getter) method.
2. The second method, `setRadius`, is a `void` method that sets the value of the radius for the object. It is referred to as a modifier (or mutator or setter) method. This method has a parameter, which is a variable that receives a value that is passed to it when the method is called (or invoked) and assigns this value to the instance variable `radius`.
3. The third method, `getArea`, calculates and returns the area of the circle using the object's radius.
4. Finally, the fourth method, `toString` returns a string containing the radius and area of the circle.

Summary of the `CircleRunner` Class

The `Circle` class can't do anything all by itself. To be useful, `Circle` objects are created in a class that is *outside* of the `Circle` class. Each object that is created from a class is called an **instance of the class** and the act of constructing an object is called **instantiating** an object. A class that contains a **main method** is sometimes called a **runner** class. Other times, they are called **client** programs. The main method is the point of entry for a program. That means that when the programmer runs the program, Java finds the main method first and executes it. You will not have to write main methods on the AP Computer Science A exam.

Notice that the `Circle` class does not contain a **public static void main(String[] args)**. The purpose of the `Circle` class is to define how to build and manipulate a `Circle` object; however, `Circle` objects are created and manipulated *in a different class* that contains a main method.

How to Make a New Object from a Class

When you want to make an object from a class, you have to call one of the class's constructors. However, you may have more than one constructor, including the no-argument constructor or one or more parameterized constructors. The computer decides which constructor to call by looking at the parameter list in the declaration of each constructor. Whichever constructor matches the call to the constructor is used to create the object.

Examples

```
// This code uses the no-argument constructor (no parameters)
Circle circle1 = new Circle();

// This code uses the parameterized constructor  (1 parameter)
Circle circle2 = new Circle(4);

// This code causes a compile-time error because there isn't
// a Circle constructor that contains a String in its parameter list
Circle circle3 = new Circle("Fred");
```

Example

Write a runner class that instantiates three `Circle` objects. Make the first `Circle` object have a radius of 10, while the second and third Circles will get the default value. Manipulate the radii of the Circles and print their areas. Call the `toString` method with each of the three `Circle` objects to print each circle's radius and area.

```
// This class creates and manipulates three Circle objects
public class CircleRunner
{
   public static void main(String[] args)
   {
      Circle myCircle = new Circle(10);        // myCircle's radius = 10.0
      Circle hisCircle = new Circle();         // hisCircle's radius = 0.0
      Circle herCircle = new Circle();         // herCircle's radius = 0.0

      System.out.println(myCircle.getArea()); // prints 314.0
      System.out.println(hisCircle.getArea()); // prints 0.0

      hisCircle.setRadius(5);                  //hisCircle's radius = 5.0
      herCircle.setRadius(2 * hisCircle.getRadius());
                                               //herCircle's radius = 10.0

      System.out.println(hisCircle.getArea()); // prints 78.5
      System.out.println(herCircle.getArea()); // prints 314.0

      System.out.println(myCircle); // prints radius and area of myCircle
      System.out.println(hisCircle); // prints radius and area of hisCircle
      System.out.println(herCircle); // prints radius and area of herCircle
            // notice that the toString method was called by default
            //  this is because it overrode the Objects toString method.
   }
}
```

```
OUTPUT
314.0
0.0
78.5
314.0
The circle has radius 10.0 and area 314.0
The circle has radius 5.0 and area 78.5
The circle has radius 10.0 and area 314.0
```

Every Object from the Same Class Is a Different Object

Every object that is constructed from a class has its own instance variables. Two *different* objects can have the same **state**, which means that their instance variables have the same values.

In this example, the `CircleRunner` class creates three different `Circle` objects called `myCircle`, `hisCircle`, and `herCircle`. Notice that the `myCircle` object has a radius of 10. This is because when the `Circle` object created it, the computer looked for the parameterized constructor and when it found it, assigned the 10 to the parameter `rad` which in turn set the instance variable, radius, to 10.0. For the `hisCircle` and `herCircle` objects, the no-argument constructor was called and the instance variable, radius, was given the default value of 0.0.

Remember that `myCircle`, `hisCircle`, and `herCircle` all refer to different objects that were all created from the same `Circle` class and each object has its own radius. When each object calls the `getArea` method, it uses *its own* radius to compute the area.

The Dot Operator

The dot operator is used to call a method of an object. The name of the object's reference variable is followed by a period and then the name of the method.

```
objectReference.methodName();
```

Understanding the Keyword `new` When Constructing an Object

Beginning Java programmers often have a hard time understanding why the name of the class is used twice when creating an object, such as the `Circle` objects in `CircleRunner`. Let's analyze the following line of code to find out what's really happening.

```
Circle myCircle = new Circle(10);
```

Data type

Object reference variable

Call to the parameterized constructor

The keyword new tells the constructor to make an object

The first use of the word Circle is the data type of the variable. It's similar to how we defined an `int` or a `double` variable. The variable `myCircle` is called a **reference variable** of

type Circle. It's similar to a primitive variable in that it holds a value, but it's *way* different in the fact that it *does not* hold the `Circle` object itself. The value that `myCircle` holds is the memory address of the soon-to-be-created `Circle` object.

The second use of the word Circle(10) is a call to the parameterized constructor of the `Circle` class. The keyword **new** tells the computer, "Hey we are about ready to make a new object." The computer looks for a constructor that has a single parameter that matches the data type of the 10. Once it finds this constructor, it gives the 10 to the parameter, rad. The code in the constructor assigns the instance variable, radius, the value of the parameter rad. Then, voilá! The radius of the `Circle` object is 10.

Finally, the computer looks for a place to store the `Circle` object in memory (RAM) and when it finds a location, it records the memory address. The construction of the Circle is now complete, and the address of the newly formed `Circle` object is assigned to the reference variable, `myCircle`.

Now, anytime you want to manipulate the object that is referenced by `myCircle`, such as give it a new radius, you are actually telling `myCircle` to go find the object that it is referencing (also called, *pointing to*), and then set the radius for that object.

The Reference Variable

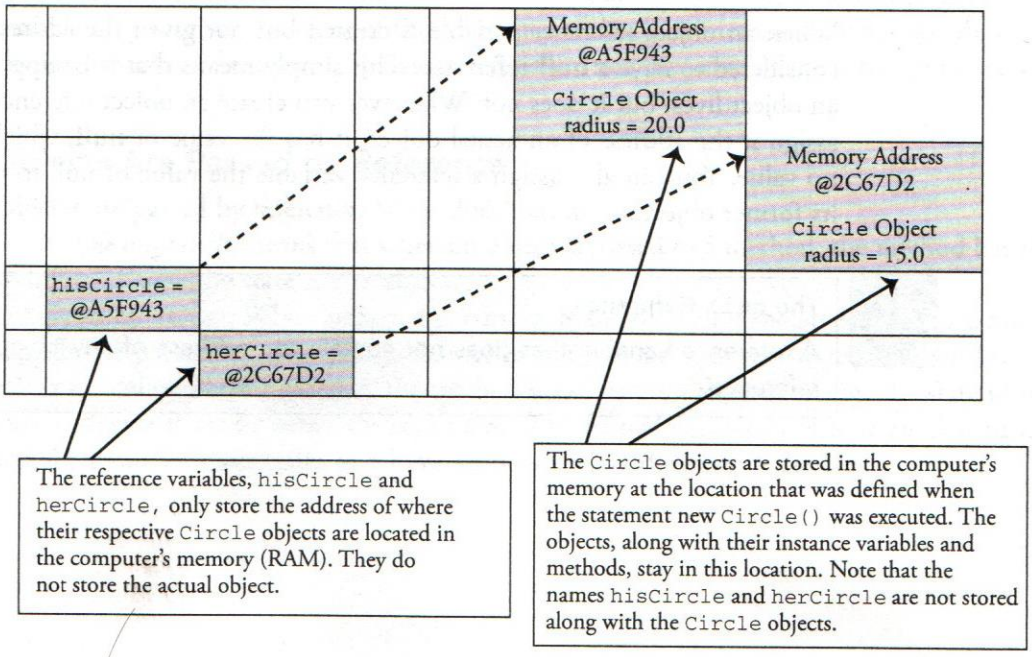
The reference variable does not hold the object itself. It holds the address of where the object is stored in RAM (random access memory).

The Reference Variable Versus the Actual Object

The following diagram is meant to help you visualize the relationship between object reference variables and the objects themselves in the computer's memory. Note: The memory addresses are simulated and only meant to serve as an example of addresses in the computer's memory.

Question 1: What happens after these two lines of code are executed?

```
Circle hisCircle = new Circle(20);
Circle herCircle = new Circle(15);
```

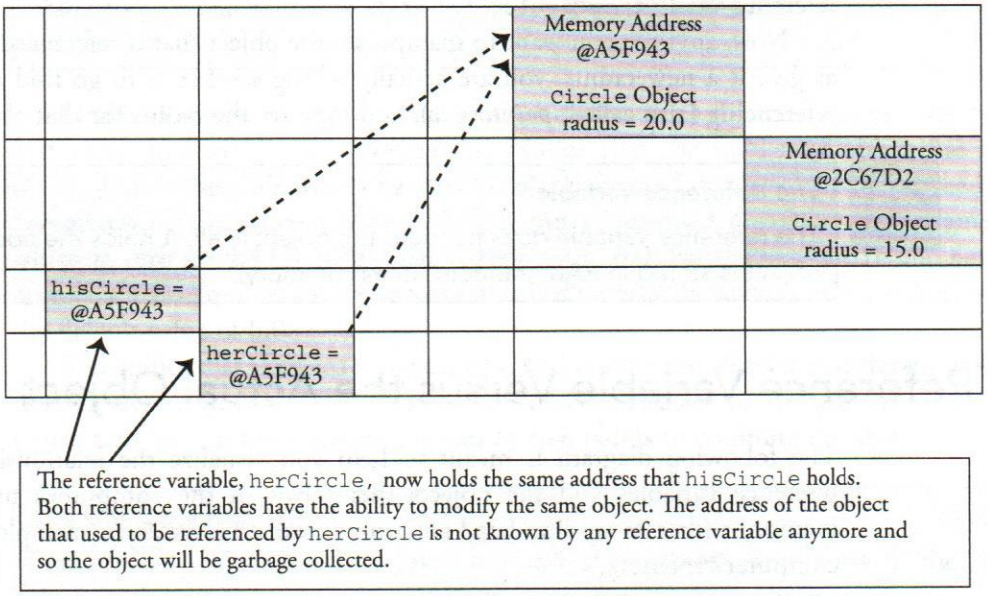


Question 2: What do you suppose happens after this line of code is executed?

```
herCircle = hisCircle;
```

If you follow the diagram, you will see that `herCircle` "takes on the value" of the address of the `hisCircle` reference variable. Now, `herCircle` contains the same address that `hisCircle` contains. This means that both reference variables contain the same address of the same object and are pointing to the same object. The word **aliasing** is used to describe the situation when two different object reference variables contain the same address of an object.

Look at the object that used to be referenced by `herCircle`. Nothing is pointing to it. It has been detached from the rest of the world and will be **garbage collected** by Java. This means that Java will delete it when it is good and ready.



The reference variable, `herCircle`, now holds the same address that `hisCircle` holds. Both reference variables have the ability to modify the same object. The address of the object that used to be referenced by `herCircle` is not known by any reference variable anymore and so the object will be garbage collected.

The `null` Reference

When an object reference variable is created but not given the address of an object, it is considered to have a **null reference**. This simply means that it is supposed to know where an object lives, but it does not. Whenever you create an object reference variable and don't assign it the address of an actual object, it has the value of **null**, which means that it has no value. You can also assign a reference variable the value of null to erase its memory of its former object.

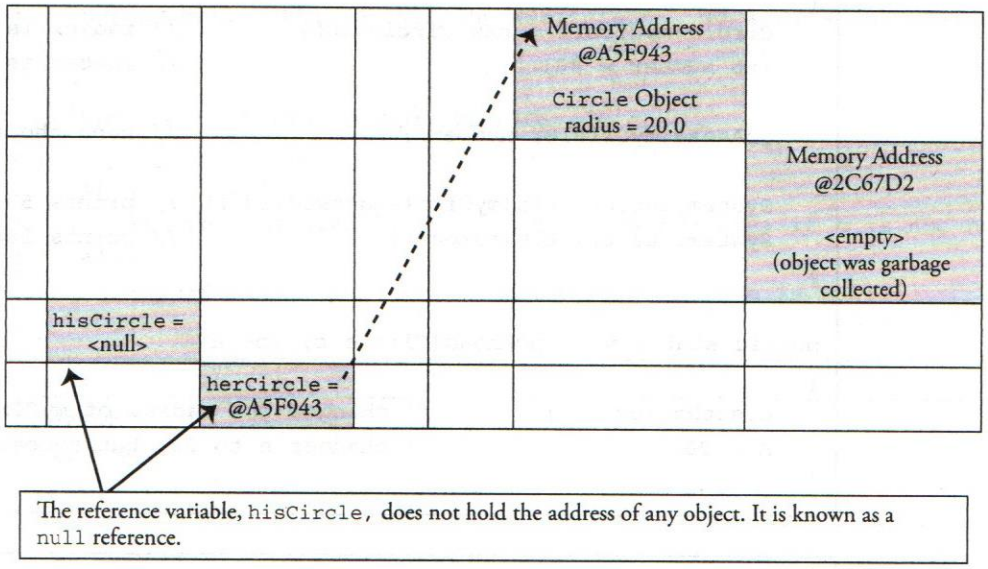
The `null` Reference

A reference variable that does not contain the address of any object is called a `null` reference.

Question 3: What do you suppose happens when this statement is executed?

```
hisCircle = null;
```

The reference variable `hisCircle` is now null; it has no value. This means that it does not contain the address of any object. The reference variable `herCircle` still holds the address of a `Circle` object and so it can access methods of that object.



Parameters

Primitives Are Passed by Value

Values that are passed to a method or constructor are called **actual parameters** (or **arguments**), while the variables in the parameter list of the method declaration are called **formal parameters**. When an actual parameter is of a primitive type, its value cannot be changed in the method because only the *value* of the variable is sent to the method.

Said another way, the formal parameter variable becomes a copy of the actual parameter. The method cannot alter the value of the actual parameter variable. The formal parameter variable only receives the *value* of the actual parameter. The method only has the ability to change the value of the formal parameter variable, not the value of the actual parameter.

Objects Are Passed by Reference

Objects are passed by reference. What does that mean?

Simple answer: It means that when an object is passed to a method, the method has the ability to change the state of the object.

Technical answer: When an actual parameter is an object, it is the object reference that is actually the parameter. This means that when the method receives the reference and stores it in a formal parameter variable, the method now has the ability to change the state of the object *because it knows where the object lives.* The formal parameter is now an alias of the actual parameter. Essentially, you have just given away the keys to the car.

Example

Demonstrate the difference between passing by reference and passing by value. The state of the object is changed in the method, while the state of the primitive variable is unchanged.

```
public static void main(String[] args)
{
   Circle myCircle = new Circle(100);        // radius is 100.0
   int number = 80;                          // number is 80

   goAhead(myCircle, number);                // send the arguments

   System.out.println(myCircle.getRadius()); // prints 50 (changed)
   System.out.println(number);               // prints 100 (not changed)
}

public static void goAhead(Circle c, int n)
{
   c.setRadius(50);        // changes the radius of myCircle to 50.0
   n = 20;                 // changes n to 20, but number stays 80
}
```

```
OUTPUT
50
80
```

Comparing Against a `null` Reference

Recall that a `nullPointerException` error occurs if you attempt to perform a method on a reference variable that is null. How can you tell if a reference variable is null so you won't get that error? The answer is to perform a **null check**. Compare the reference variable using the == comparator or != comparator. Just *don't use* the equals method to perform a null check. You'll get a `nullPointerException`!

Example

Demonstrate how to perform a null check using the `Circle` class. If the Circle reference is not null, then compute the area. If the Circle reference is null, then return -1 for the area.

```
public static void main(String[] args)
{
   Circle circle1 = new Circle(10);
   Circle circle2;                                  // circle2 is null

   System.out.println(doANullCheck(circle1);   // send a valid reference
   System.out.println(doANullCheck(circle2);   // send a null reference
}

public static double doANullCheck(Circle c)
{
   if (c != null)
      return c.getArea();          // return the area if c is not null
   else
      return -1;                   // return -1 if c is null
}
```

```
OUTPUT
314.15926
-1.0
```

Checking Against `null`

Never use the `equals` method to determine if an object reference is `null`. Only use `==` or `!=` to check if an object reference variable is `null`.

Overloaded Constructors

It is possible to have more than one constructor in a class. Constructors are always declared using the name of the class, but their parameter lists can differ. If there are two or more constructors in a class, they are all called **overloaded constructors**. The computer decides which of the overloaded constructors is the correct one to use based on the parameter list.

Overloaded constructors have the same name; however, they must differ by one of these ways:

- a different number of parameters
- the same number of parameters but at least one is a different type
- the same exact parameter types but in a different order

Example

Declare four constructors for a class called `Student`. The constructors have the same name as the class, but are different because their parameter lists differ.

```
public class Student
{
   private String firstName;
   private String lastName;
   private int age;

   public Student()                          // no-argument constructor
   {
      firstName = lastName = "";
      age = 0;
   }

   public Student(String first)              // constructor has 1 parameter
   {
      firstName = first;
      lastName = "";
      age = 0;
   }

   public Student(String first, String last)    // has 2 parameters
   {
      firstName = first;
      lastName = last;
      age = 0;
   }
   public Student(String first, String last, int yearsOld)  // has 3
   {
      firstName = first;
      lastName = last;
      age = yearsOld;
   }
}
```

Overloaded Methods

Methods that have the same name but different parameter lists are called **overloaded methods**. The computer decides which of the overloaded methods is the correct one to use based on the parameter list.

Overloaded methods must have the same name and may have a different return type; however, they must differ by one of these ways:

- a different number of parameters
- the same number of parameters but at least one is a different type
- the same exact parameter types but in a different order

Example

Declare four overloaded methods. The methods should have the same name, but different parameter lists as described above.

```
public void doSomething(int param)              // has 1 int parameter
{
   // Do something with param
}

public void doSomething(double param)           // has 1 double parameter
{
   // Do something with param
}

public void doSomething(double a, int b)        // has 2 parameters
{
   // Do something with a and b
}

public void doSomething(int a, double b) // has 2 params; different order
{
   // Do something with a and b
}
```

Blast from the Past

Recall that the **substring** method from the `String` class is overloaded. There are two versions of the method, but they are different because their parameter lists are not identical.

static, static, static

`static` Variables (Class Variables)

There are times when a class needs to have a variable that is shared by all the objects that come from the class. The variable is declared and assigned a value as though it were an instance variable; however, it uses the keyword **static** in its declaration. Static variables are also called **class variables** and can be declared private or public.

`static final` Variables (Class Constants)

Likewise, there are times when a class needs to have a **constant** that is used by all the objects of the class. As the name *constant* suggests, it *cannot* be reassigned a different value at any other place in the program. In fact, a compile-time error will occur if you try to give the constant a different value. Class constants are declared using the keywords `static` and `final` in their declaration and use all uppercase letters (and underscores, if necessary). These static final variables are also called class constants.

`static` Methods (Class Methods)

If a method is labeled `static`, it can only call other static methods and can only use or modify static variables. Static methods cannot access or change the values of the instance variables or cannot call `non-static` methods in the class. Class methods are associated with methods of the class, not individual objects of the class.

Example

Suppose you are hired by Starbucks to keep track of the total number of stores. You've already decided to have a class called `StarbucksStore`, but how can you keep track of the total number of stores? One solution is to make a static variable, called `storeCount`. Then, each time a store is constructed (pun intended), increment this class variable. Access will be given to the `storeCount` by creating a `static` method called `getStoreCount`. Let's also say that you decide to make a variable that holds the logo for all the stores. You choose to make this variable a constant because every store shares the logo, and none of them should be allowed to change the logo. For this we will make a static final variable called `STARBUCKS_LOGO` and initialize it to "`Mermaid`".

```
public class StarbucksStore
{
	private double coffeePrice;
	private double totalServed;

	public StarbucksStore(double price)
	{
		coffeePrice = price;
		storeCount++; // increments each time a new store is
				// constructed
	}

	public double getPrice()
	{
		return coffeePrice;
	}

	public void setPrice(double newPrice)
	{
		coffeePrice = newPrice;
	}

	public double getServed()
	{
		return totalServed;
	}

	public void addServed(double served)
	{
		totalServed += served;
	}

	public double sales()
	{
		return totalServed * coffeePrice;
	}

	// ------------ static variables and methods ---------------
	public static int storeCount = 0;
	public static final String STARBUCKS_LOGO = "Mermaid";

	public static int getStoreCount()
	{
		return storeCount;
	}
}
```

```
public class StoreRunner
{
      public static void main(String[] args)
      {
            StarbucksStore store1 = new StarbucksStore(3.95);
            StarbucksStore store2 = new StarbucksStore(4.95);
            StarbucksStore store3 = new StarbucksStore(5.29);

            System.out.println(StarbucksStore.getStoreCount());
            System.out.println(StarbucksStore.STARBUCKS_LOGO);

            store1.addServed(327);
            store2.addServed(450);
            store3.addServed(678);

            System.out.println(store1.sales());
            System.out.println(store2.sales());
            System.out.println(store3.sales());
      }
}
```

```
OUTPUT
3
Mermaid
1291.65
2227.5
3586.62
```

Did you notice how the class method `getStoreCount()` and constant `STARBUCKS_LOGO` were called from the main program? Since the method and constant belong to the class and not a particular object of that class, they were called by using the class name `StarbucksStore`. Does that look familiar? You've done something like this already when you called `Math.abs()` (a static method from the `Math` class) or `Integer.MAX_VALUE` (a static constant from the `Integer` class).

Data Encapsulation

Information Hiding

Protecting information by restricting its access is important in software development. Since classes are public, all the objects from the class are public. The way that we hide information or **encapsulate data** in Java is to use private access modifiers.

Encapsulated Data

It is expected on the AP Computer Science A Exam that all instance variables of a class are declared private. The only way to gain access to them is to use the accessor or mutator methods provided by the class.

What Happens If I Declare an Instance Variable `public`?

Let's suppose you just want to find out what happens if you declare an instance variable public. The result is that you can use the dot operator to access the instance variables.

Example

Declare the radius instance variable from the `Circle` class as public. Notice that the object has now allowed access to the public variable by anyone using the dot operator. Using the dot operator on a private variable produces a compile-time error.

```
// Pretend this code is in the Circle class
public double radius;  // public instance variable:  Don't ever do this!
```

```
// Pretend this code is in a runner class
Circle myCircle = new Circle(10); // radius is 10.0
myCircle.radius = 5.4;            // radius is easily changed when public
double result = myCircle.radius;  // result is 5.4
```

The Integer Class and Its `public` Fields

Remember how we accessed the largest and smallest Integer? We used the dot operator to just get the value. That's because MAX_VALUE and MIN_VALUE are public constant fields of the Integer class. You don't use an accessor method to get them.

```
int reallyBigInteger = Integer.MAX_VALUE; // MAX_VALUE is a public constant
```

Scope

The scope of a variable refers to the region of the program in which the variable is known. Attempting to use a variable outside of its scope produces a compile-time error.

Different types of variables have different scopes.

- The scope of a **local variable** is within the nearest pair of curly braces that encloses it.
- The scope of a **parameter** is the method or constructor in which it is declared.
- The scope of an **instance variable** is the class in which it is defined.

Documentation

Inline Comments

I've been using inline comments throughout and will continue to use them in the rest of the book. They begin with two forward slashes, //. All text that comes after the slashes (on the same line) is ignored by the compiler.

```
// This is an example of an inline comment.
```

Multiple-Line Comments

When your comment takes longer than one line, you will want to use a multiple-line comment. These comments begin with /* and end with */. The compiler ignores everything in between these two special character sequences.

```
/*
This is how to write a multiple-line comment. Everything typed here is
ignored by the compiler; even mi mysteaks is spelling, so be particularly
careful of your spelling when you are typing multiple-line comments!
*/
```

Javadoc Comments

You will see **Javadoc comments** (also called **documentation comments**) on the AP Computer Science A Exam, especially in the FRQ section. These comments begin with /** and end with */. The compiler ignores everything in between the two character sequences.

```
/**
* This is an example of a Javadoc comment.
* You will see these "documentation" comments on the AP CS exam.
*/
```

Javadoc

Javadoc is a tool that generates an HTML-formatted document that summarizes a class (including its constructors, instance variables, methods, etc.) in a readable format. Among programmers, this document itself is often referred to as a Javadoc. The summary is called the API (application program interface) for that set of classes and is extremely useful for programmers. You should refer to the Oracle Java website for a list of all Java classes and their methods (but only those listed on the Java Quick Reference will be tested on the AP Computer Science A exam).

Javadoc Tags

Similar to a hashtag, the **@param** is a tag that is used in Javadoc comments. When a programmer uses an @param tag, they state the name of the parameter and provide a brief description of it. Javadoc tags are automatically detected by Javadoc and appear in the API.

The **@return** is also a Javadoc tag. It is only used in return methods, and the programmer identifies the name of the variable being returned and a brief description of it.

> **Documentation Comments on the AP Computer Science A Exam**
>
> You won't have to write any comments on the AP Computer Science A exam. You just need to know how to read them.

Precondition and Postcondition

You know what happens when you assume, right? Well, a **precondition** states what you are allowed to assume about a parameter. It's a statement in the documentation comment before a constructor or method signature that describes the expectations of the parameter sent to the method. There is no expectation that the method will check to ensure preconditions are satisfied. If a method is called and the preconditions haven't been met, then the method most likely won't perform as expected. On the AP Computer Science A Exam, reading the preconditions can help you understand the nature of the parameters. A **postcondition** is a condition that must always be true after the execution of a section of program code. Postconditions describe the outcome of the execution in terms of what is being returned or the state of an object. Programmers must write method code to satisfy the postconditions when preconditions are met.

Example

This is the same `Circle` class that was created earlier; however, it now includes the Javadoc tags and precondition/postcondition statements as they may appear on the AP Computer Science A Exam. It also includes an improved way to calculate the area using the `Math` class.

```
public class Circle                    // Class declaration
{
   private double radius;              // Instance variable

   public Circle()                     // The no-argument constructor
   {
      // The radius gets a default value of 0.0
   }

   /**
   *  A parameterized constructor for a Circle
   *  @param r The radius of the Circle
   *         Precondition:  rad > 0
   */
   public Circle(double rad)
   {
      radius = rad;  // The radius gets the value of the parameter, rad
   }

   /**
   *  Accessor method that returns the radius of the Circle object
   *  @return radius of the Circle
   */
   public double getRadius()
   {
      return radius;
   }

   /**
   *  Mutator method that reassigns the radius of the Circle object
   *  @param r The radius of the Circle
   *         Precondition:  rad > 0
   */
   public void setRadius(double rad)
   {
      radius = rad;      // The radius is changed to be the parameter value
   }

   /**
   *  Method that calculates and returns the area of the Circle
   *  @return area of the Circle
   *          Postcondition:  area > 0
   */
   public double getArea()
   {
      return Math.PI * Math.pow(radius, 2); // uses PI for more precision
   }
}  // End of Circle class
```

The Keyword this

The keyword **this** is a reference to the current object, or rather, the object whose method or constructor is being called. It is also referred to as the **implicit** parameter.

Example 1

Pass the implicit parameter **this** to a different method in a class. This usage may be tested on the exam.

```
public static void main(String[] args)
{
   MyClass m1 = new MyClass("Batman");
   m1.doSomething();
}
```

```
public class MyClass
{
   private String name;
   public MyClass(String n)
   {   name = n;   }

   public String getName()
   {   return name;   }

   public void doSomething()
   {
      nowDoSomethingWith(this);        // Pass "this" to another method
   }

   public void nowDoSomethingWith(MyClass myObject) // myObject is m1
   {
      System.out.println("I am " + myObject.getName());
   }
}
```

```
OUTPUT
I am Batman
```

Example 2

A common place to use the keyword `this` is when the instance variables have the same name as the parameters in a constructor. The keyword `this` is used on the left side of the assignment statements to assign the instance variables the values of the parameters. The left side is the instance variable; the right side is the parameter variable. Using `this` here says, "Hey, I understand that we have the same name, so these are my instance variables and not the parameters."

```
public class Whatever
{
   private int a;
   private int b;

   public Whatever(int a, int b)    // Parameters have the same name
   {                                // as the instance variables.
      this.a = a;                   // Left side are the instance variables
      this.b = b;                   // Right side are the parameters
   }
}
```

IllegalArgumentException

If you pass an argument to a method and the value of the argument does not meet certain criteria required by the method, an **`IllegalArgumentException`** error may be thrown during run-time. Programmers may also choose to write a method that terminates with an `IllegalArgumentException` if it does not receive the expected input.

Rapid Review

Classes

- A class contains the blueprint, or design, from which individual objects are created.
- Classes tend to represent things that are nouns.
- A class declaration (or class definition) is the line of code that declares the class.
- By naming convention, class names begin with an uppercase letter and use camel case.
- The two access modifiers that are tested on the AP Computer Science A Exam are public and private.
- All classes should be declared public.
- A public class means that the class is visible to all classes everywhere.

Instance Variables

- Instance variables (or fields) are properties that all objects from the same class possess. They are the attributes that help distinguish one object from another in the same class.
- All instance variables should be declared private.
- A class can have as many instance variables as is appropriate to describe all the attributes of an object from the class.
- By naming convention, all instance variables begin with a lowercase letter and use camel case.
- All primitive instance variables receive default values. The default value of an int is 0, the default value of a double is 0.0, the default value for a boolean is false, and the default for reference is `null`.
- The current values of all of the instance variables of an object determine the state of the object.
- The state of the object is changed whenever any of the instance variables are modified.

Constructors

- Every class, by default, comes with a no-argument (or empty) constructor.
- Parameterized constructors should be created if there are values that should be assigned to the instance variables at the moment that an object is created.
- A class can have as many parameterized constructors as is relevant.
- The main reason to have a parameterized constructor is to assign values to the instance variables.
- Other code may be written in the constructor.
- A parameter is a variable that receives values and is declared in a parameter list of a method or constructor.

Methods

- The methods of a class describe the actions that an object can perform.
- Methods tend to represent things that are verbs.
- By naming convention, method names begin with a lowercase letter and use camel case.
- The name of a method should describe the purpose of the method.
- Methods should not be multi-purpose. They should have one goal.
- A method declaration describes pertinent information for the method such as the access modifier, the return type, the name of the method, and the parameter list.
- Methods that don't return a value are called `void` methods.
- Methods that return a value of some data type are called return methods.
- Methods can return any data type.
- return methods must include a reachable return statement.
- Methods that return the value of an instance variable are called accessor (or getter) methods.
- Methods that modify the value of an instance variable are called mutator (or modifier or setter) methods.
- The data type of all parameters must be defined in the parameter list.
- Methods declared as public can be used externally to the class.
- Methods declared as private can be used internally to the class.
- When designing a class programmers make decisions about what data to make accessible and modifiable from an external class.

Objects

- An object from a class is a virtual realization of the class.
- Objects store their own data.
- An instance of a class is the same thing as an object of a class.
- The word "instantiate" is used to describe the action of creating an object. Instantiating is the act of constructing a new object.
- The keyword `new` is used whenever you want to create an object.
- An object reference variable stores the memory address of where the actual object is stored. It can only reference one object.
- The reference variable does not store the object itself.
- Two reference variables are equal only if they refer to the exact same object.
- The word *null* means no value.
- A `null` reference describes a reference variable that does not contain an address of any object.

- You can create as many objects as you want from one class. Each is a different object that is stored in a different location in the computer's memory.

Passing Parameters by Value

- The parameter variables that are passed to a method or constructor are called actual parameters (or arguments).
- The parameter variables that receive the values in a method or constructor are called formal parameters.
- Passing by value means that only the value of the variable is passed.
- Primitive variables are passed by value.
- It is impossible to change the value of actual parameters that are passed by value.

Passing Parameters by Reference

- Passing by reference means that the address of where the object is stored is passed to the method or constructor.
- Objects are passed by reference (including arrays).
- The state of the object can be modified when it is passed by reference.
- It is possible to change the value of actual parameters that are passed by reference.

Overloaded

- Overloaded constructors may have (1) a different number of parameters, (2) the same number of parameters but of a different type, or (3) the same exact parameter types but in a different order.
- Overloaded methods have the same name; however, their method parameter lists are different in some way.
- Overloaded methods may have (1) a different number of parameters, (2) the same number of parameters but of a different type, (3) the same exact parameter types but in a different order.

`static`

- `Static` variables are also known as class variables.
- A class variable is a variable that is shared by all instances of a class.
- Changes made to a class variable by any object from the class are reflected in the state of each object for all objects from the class.
- Static final variables are also called class constants.
- Class constants are declared using both the keywords **static** and **final**.
- Constants, by naming convention, use uppercase letters and underscores.
- Constants cannot be modified during run-time.
- If a method is declared static, it can only call other `static` methods and can reference `static` variables.

Scope

- The scope of a variable refers to the area of the program in which it is known.
- The scope of a local variable is in the block of code in which it is defined.
- The scope of an instance variable is the class in which it is defined.
- The scope of a parameter is the method or constructor in which it is defined.

Documentation

- Documentation tags are used by Javadocs.
- The @param tag is used to describe a parameter.
- The @return tag is used to describe what is being returned by a method.
- A precondition describes what can be expected of the values that the parameters receive.
- A postcondition describes the end result criteria for a method.

Miscellaneous

- Data encapsulation is the act of hiding the values of the instance variables from other classes.
- Declaring instance variables as private encapsulates them.
- Use the keyword `this` when you want to refer to the object itself.
- An `IllegalArgumentException` error occurs when a parameter that is passed to a method fails to meet criteria set up by the programmer.

› Review Questions

Basic Level

1. The relationship between a class and an object can be described as:

(A) The terms *class* and *object* mean the same thing.
(B) A class is a program, while an object is data.
(C) A class is the blueprint for an object and objects are instantiated from it.
(D) An object is the blueprint for a class and classes are instantiated from it.
(E) A class can be written by anyone, but Java provides all objects.

2. Which of the following is a valid constructor declaration for the `Cube` class?

I. `public static Cube()`
II. `public void Cube()`
III. `public Cube()`
IV. `public Cube(int side)`

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) III and IV only

Questions 3–6 refer to the following class.

```
public class Student
{
   private String name;
   private double gpa;

   public Student(String newName, double newGPA)
   {
      name = newName;
      gpa = newGpa;
   }

   public String getName()
   {
      return name;
   }

   /*  Additional methods not shown  */
}
```

3. Which of the following could be used to instantiate a new `Student` object called `sam`?

(A) `Student sam = new Student();`
(B) `sam = new Student();`
(C) `Student sam = new Student("Sam Smith", 3.5);`
(D) `new Student sam = ("Sam Smith", 3.5);`
(E) `new Student(sam);`

4. Which of the following could be used in the `StudentTester` class to print the name associated with the `Student` object instantiated in problem 3?

(A) `System.out.println(getName());`
(B) `System.out.println(sam.getName());`
(C) `System.out.println(getName(sam));`
(D) `System.out.println(Student.name);`
(E) `System.out.println(getName(Student));`

5. Which of the following is the correct way to write a method that changes the value of a `Student` object's `gpa` variable?

(A)
```
public double setGpa(double newGpa)
{
   return gpa;
}
```
(B)
```
public double setGpa()
{
   return gpa;
}
```
(C)
```
public void setGpa ()
{
   gpa++;
}
```
(D)
```
public setGpa(double newGpa)
{
   gpa = newGpa;
}
```
(E)
```
public void setGpa(double newGpa)
{
   gpa = newGpa;
}
```

6. Consider the following code segment.

```
Student s1 = new Student("Brody Kai", 3.9);
Student s2 = new Student("Charlie Cole", 3.2);
s2 = s1;
s2.setGpa(4.0);
System.out.println(s1.getGpa());
```

What is printed as a result of executing the code segment?

(A) `3.2`
(B) `3.9`
(C) `4.0`
(D) Nothing will be printed. There will be a compile-time error.
(E) Nothing will be printed. There will be a run-time error.

Questions 7–8 refer to the following information.

Consider the following method.

```
public int calculate(int a, int b)
{
    a = a - b;
    int c = b;
    b = a * a;
    a = b - (a + c);
    return a;
}
```

7. The following code segment appears in another method in the same class.

```
int var = 9;
int count = 2;
System.out.println(calculate(var, count));
```

What is printed as a result of executing the code segment?

(A) `2`
(B) `9`
(C) `23`
(D) `40`
(E) `49`

8. The following code segment appears in another method in the same class.

```
int a = 9;
int b = 2;
int c = calculate(a, b);
System.out.println(a + " " + b + " " + c);
```

What is printed as a result of executing the code segment?

(A) `40 49 40`
(B) `9 2 40`
(C) `40 49 49`
(D) Run-time error: `a` and `b` cannot contain two values at once.
(E) Compile-time error; duplicate variable.

9. Consider the following calculation of the area of a circle.

```
double area = Math.PI * Math.pow(radius, 2);
```

`Math.PI` can best be described as:

(A) A method in the `Math` class.
(B) A method in the class containing the given line of code.
(C) A public static final double in the `Math` class.
(D) A private static final double in the `Math` class.
(E) A private static final double in the class containing the given line of code.

10. A programmer is designing a `BankAccount` class. In addition to the usual information needed for a bank account, she would like to have a variable that counts how many `BankAccount` objects have been instantiated. She asks you how to do this. You tell her:

(A) It cannot be done, since each object has its own variable space.
(B) She should use a class constant.
(C) She should call a mutator method to change the `numAccounts` variable held in each object.
(D) She should create a `static` variable and increment it in the constructor as each object is created.
(E) Java automatically keeps a tally. She should reference the variable maintained by Java.

11. Having multiple methods in a class with the same name but with a different number or different types of parameters is called:

(A) abstraction
(B) method overloading
(C) encapsulation
(D) method visibility
(E) parameterization

Advanced Level

12. Consider the following method.

```
public int mystery(int a, int b, int c)
{
   if (a < b && a < c)
   {
      return a;
   }
   if (b < c && b < a)
   {
      return b;
   }
   if (c < a && c < b)
   {
      return c;
   }
}
```

Which statement about this method is true?

(A) `mystery` returns the smallest of the three integers `a`, `b`, and `c`.
(B) `mystery` always returns the value of `a`.
(C) `mystery` always returns both values `a` and `c`.
(D) `mystery` sometimes returns all three values `a`, `b`, and `c`.
(E) `mystery` will not compile, because the return statement appears to be unreachable under some conditions.

13. The method from the previous problem is rewritten as shown.

```
public int mystery2(int a, int b, int c)
{
   if (a < b && a < c)
   {
      return a;
   }
   if (b < c && b < a)
   {
      return b;
   }
   return c;
}
```

Which statement about the method is true?

(A) `mystery2` returns the smallest of the three integers `a`, `b`, and `c`.
(B) `mystery2` always returns the value of `c`.
(C) `mystery2` returns either the values of both `a` and `c` or both `b` and `c`.
(D) `mystery2` sometimes returns all three values `a`, `b`, and `c`.
(E) `mystery2` will not compile, because the return statement appears to be unreachable under some conditions.

14. Consider the following class declaration.

```
public class AnotherClass
{
   private static int val = 31;
   private String data;

   public AnotherClass(String s)
   {
       data = s;
       val /= 2;
   }

   public void join(String s)
   {   data = data + s;   }

   public void setData(String s)
   {   data = s;   }

   public String getData()
   {   return data;   }

   public int getVal()
   {   return val;   }
}
```

The following code segment appears in the main method of another class.

```
AnotherClass word = new AnotherClass("Hello");
word.join("Hello");
AnotherClass sentence = new AnotherClass(word.getData());
sentence.join("Hi");
word.setData(sentence.getData());
word.join("Hello");
sentence.join("Hi");
sentence.setData("Hi");
System.out.println(word.getVal());
```

What is printed as a result of executing the code segment?

(A) `30`
(B) `15`
(C) `7.75`
(D) `7.5`
(E) `7`

15. Consider the following method declaration.

```
public int workaround(int a, double b)
```

Which of these method declarations would correctly overload the given method?

I. `public double workaround(int a, double b)`
II. `public int workaround(double b, int a)`
III. `public int workaround(String s, double b)`
IV. `public double workaround(int a, double b, String s)`
V. `public int workaround(int b, double a)`

(A) I only
(B) IV only
(C) II and III only
(D) II, III, and IV only
(E) All of the above

16. Free-Response Practice: Dream Vacation Class

Everyone fantasizes about taking a dream vacation.

Write a full `DreamVacation` class that contains the following:

a. An instance variable for the name of the destination
b. An instance variable for the cost of the vacation (dollars and cents)
c. A no-argument constructor
d. A parameterized constructor that takes in both the name of the vacation and the cost
e. Accessor (getter) methods for both instance variables
f. Modifier (setter) methods for both instance variables

17. Free-Response Practice: The `Height` Class

The purpose of the `Height` class is to hold a measurement in feet and inches, facilitate additions to that measurement, and keep the measurement in simplest form (inches < 12). A `Height` object can be instantiated with a measurement in feet and inches, or in total inches.

Write the entire `Height` class that includes the following:

a. Instance variables `int feet` for the number of feet and `int inches` for the number of inches. These instance variables must be updated anytime they are changed so that they hold values in simplest form. That is, inches must *always* be less than 12 (see method described in part C below).
b. Two constructors, one with two parameters representing feet and inches, and one with a single parameter representing inches. The parameters do not necessarily represent the measurement simplest form.
c. A method called `simplify()` that recalculates the number of feet and inches, if necessary, so that the number of inches is less than 12.
d. An `add(int inches)` method that takes the number of inches to add and updates the appropriate instance variables.
e. An `add(Height ht)` method that takes a parameter that is a `Height` object and adds that object's inches and feet to this object's instance variables, updating if necessary.
f. Accessor (getter) methods for the instance variables `inches` and `feet`.