Chapter 10

Object-Oriented Programming

Chapter Introduction

In this chapter, you will learn about:

- The principles of object-oriented programming
- Classes
- Public and private access
- Ways to organize classes
- Instance methods
- Static methods
- Using objects

10-1 Principles of Object-Oriented Programming

Object-oriented programming (OOP) (A programming technique that focuses on objects, or "things," and describes their attributes and behaviors.) focuses on an application's data and the methods you need to manipulate that data. With OOP, you consider the objects that a program will manipulate—for example, a customer invoice, a loan application, or a menu from which a user selects an option. You define the characteristics of those objects and the methods that each object will use; you also define the information that must be passed to those methods.

OOP uses all of the familiar concepts of modular procedural programming, such as variables, methods, and passing arguments. Methods in object-oriented programs continue to use sequence, selection, and looping structures and make use of arrays. However, OOP adds several new concepts to programming and involves a different way of thinking. A considerable amount of new vocabulary is involved as well. First, you will read about OOP concepts in general, and then you will learn the specific terminology.

Five important features of object-oriented languages are:

- Classes
- Objects
- Polymorphism

- Inheritance
- Encapsulation

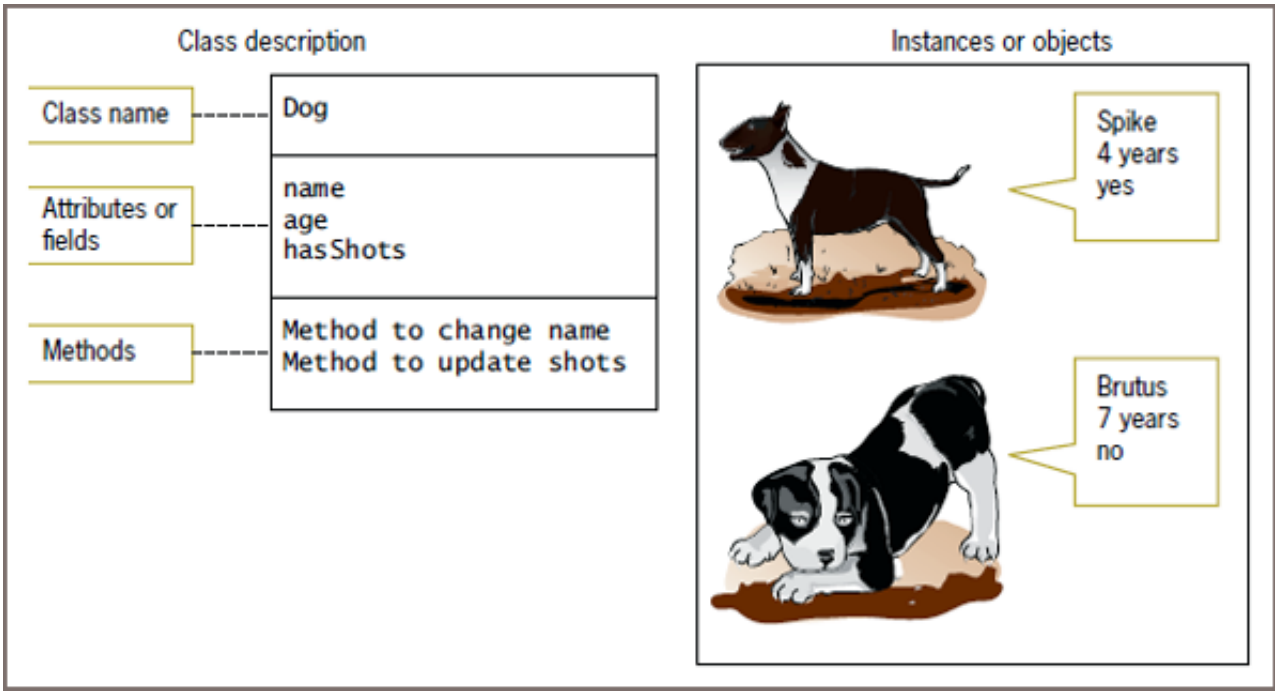
Chapter 10: Object-Oriented Programming: 10-1a Classes and Objects
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

10-1a Classes and Objects

In object-oriented terminology, a **class** (A group or collection of objects with common attributes.) describes a group or collection of objects with common attributes. An **object** (One tangible example of a class; an instance of a class.) is one **instance** (An existing object or tangible example of a class.) of a class. Object-oriented programmers sometimes say an object is one **instantiation** (An instance of a class.) of a class; this word is just another form of *instance*. For example, your `redChevroletAutomobileWithTheDent` is an instance of the class that is made up of all automobiles, and your `goldenRetrieverDogNamedGinger` is an instance of the class that is made up of all dogs. A class is like a blueprint from which many houses might be constructed, or like a recipe from which many meals can be prepared. One house and one meal are each an instance of their class; countless instances might be created eventually. For example, Figure 10-1 depicts a `Dog` class and two instances of it.

Figure 10-1

A `Dog` Class and Two Instances



Since the beginning of this book, you have created application classes that could be used without creating instances. In this chapter, you will create additional types of classes that client programs will **instantiate** (To create an object.); that is, instances of the classes will be created.

Objects both in the real world and in object-oriented programming contain attributes and methods. **Attributes** (One field or column in a database table or a characteristic that defines an object as part of a class.) are the characteristics that define an object as part of a class. For example, some of your automobile's attributes are its make, model, year, and purchase price. Other attributes include whether the automobile is currently running, its gear, its speed, and whether it is dirty. All automobiles possess the same attributes, but not the same values for those attributes. Similarly, your dog has the attributes of its breed, name, age, and whether its shots are current. Methods are the actions that can be taken on an object; often they alter, use, or retrieve the attributes. For example, an automobile has methods for changing and viewing its speed, and a dog has methods for setting and finding out its shot status.

Thinking of items as instances of a class allows you to apply your general knowledge of the class to its individual members. A particular instance of an object takes its attributes from the general category. If your friend purchases an `Automobile`, you know it has a model name, and if your friend gets a `Dog`, you know the dog has a breed. You might not know the current status of your friend's `Automobile`, such as its current speed, or the status of her `Dog`'s shots, but you do know what attributes exist for the `Automobile` and `Dog` classes, which allows you to imagine these objects reasonably well before you see them. You know enough to ask the `Automobile`'s model and not its breed; you know enough to ask the `Dog`'s name and not its engine size. As another example, when you use a new application on your computer, you expect each component to have specific, consistent attributes, such as a button being clickable or a window being closable. Each component gains these attributes as a member of the general class of GUI (graphical user interface) components.

Most programmers employ the format in which class names begin with an uppercase letter and multiple-word identifiers are run together, such as `SavingsAccount` or `TemporaryWorker`. Each new word within the identifier starts with an uppercase letter. In Chapter 2, you learned that this format is known as *Pascal casing*.

Much of your understanding of the world comes from your ability to categorize objects and events into classes. As a young child, you learned the concept of *animal* long before you knew the word. Your first encounter with an animal might have been with the family dog, a neighbor's cat, or a goat at a petting zoo. As you developed speech, you might have used the same term for all of these creatures, gleefully shouting "Doggie!" as your parents pointed out cows, horses, and sheep in picture books or along the roadside on drives in the country. As you grew more sophisticated, you learned to distinguish dogs from cows; still later, you learned to distinguish breeds. Your understanding of the class `Animal` helps you see the similarities between dogs and cows, and your understanding of the class `Dog` helps you see the similarities between a Great Dane and a Chihuahua. Understanding classes gives you a framework for categorizing new experiences. You might not know the term *okapi*, but when you learn it's an animal, you begin to develop a concept of what an okapi

might be like.

When you think in an object-oriented manner, everything is an object. You can think of any inanimate physical item as an object—your desk, your computer, and your house are all called *objects* in everyday conversation. You can think of living things as objects, too—your houseplant, your pet fish, and your sister are objects. Events also are objects—the stock purchase you made, the mortgage closing you attended, and your graduation party are all objects.

Everything is an object, and every object is a member of a more general class. Your desk is a member of the class that includes all desks, and your pet fish is a member of the class that contains all fish. An object-oriented programmer would say that the desk in your office is an instance, or one tangible example, of the `Desk` class, and your fish is an instance of the `Fish` class. These statements represent **is-a relationship** (An is-a relationship exists between an object and its class.) because you can say, "My oak desk with the scratch on top *is a* `Desk` and my goldfish named Moby *is a* `Fish`." Your goldfish, my guppy, and the zoo's shark each constitute one instance of the `Fish` class.

Object-oriented programmers also use the term *is-a* when describing inheritance. You will learn about inheritance later in this chapter and in Chapter 11.

The concept of a class is useful because of its reusability. For example, if you invite me to a graduation party, I automatically know many things about the party object. I assume that there will be attributes such as a starting time, a number of guests, some quantity of food, and gifts. I understand parties because of my previous knowledge of the `Party` class, of which all parties are members. I don't know the number of guests or the date or time of this particular party, but I understand that because all parties have a date and time, then this one must as well. Similarly, even though every stock purchase is unique, each must have a dollar amount and a number of shares. All objects have predictable attributes because they are members of certain classes.

The data components of a class that belong to every instantiated object are the class's **instance variables** (The data components that belong to every instantiated object.) . Also, object attributes often are called **fields** (are object attributes or data.) to help distinguish them from other variables you might use. The set of all the values or contents of an object's instance variables is known as its **state** (The set of all the values or contents of a class's instance variables.) . For example, the current state of a particular party might be 8 p.m. and Friday; the state of a particular stock purchase might be $10 and five shares.

In addition to their attributes, objects have methods associated with them, and every object that is an instance of a class possesses the same methods. For example, at some point you might want to issue invitations for a party. You might name the method `issueInvitations()`, and it might display some text as well as the values of the party's

date and time fields. Your graduation party, then, might possess the identifier `myGraduationParty`. As a member of the `Party` class, it might have data members for the date and time, like all parties, and it might have a method to issue invitations. When you use the method, you might want to be able to send an argument to `issueInvitations()` that indicates how many copies to print. When you think of an object and its methods, it's as though you can send a message to the object to direct it to accomplish a particular task—you can tell the party object named `myGraduationParty` to print the number of invitations you request. Even though `yourAnniversaryParty` also is a member of the `Party` class, and even though it also has an `issueInvitations()` method, you will send a different argument value to `yourAnniversaryParty`'s `issueInvitations()` method than I send to `myGraduationParty`'s corresponding method. Within an object-oriented program, you continuously make requests to an object's methods, often including arguments as part of those requests.

In grammar, a noun is equivalent to an object and the values of a class's attributes are adjectives—they describe the characteristics of the objects. An object also can have methods, which are equivalent to verbs.

When you program in object-oriented languages, you frequently create classes from which objects will be instantiated. You also write applications to use the objects, along with their data and methods. Often, you will write programs that use classes created by others; at other times, you might create a class that other programmers will use to instantiate objects within their own programs. A program or class that instantiates objects of another prewritten class is a **class client** (A program or class that instantiates objects of another prewritten class.) or **class user** (A program or class that instantiates objects of another prewritten class.) . For example, your organization might already have a class named `Customer` that contains attributes such as `name`, `address`, and `phoneNumber`, and you might create clients that include arrays of thousands of `Customer`s. Similarly, in a GUI operating environment, you might write applications that include prewritten components from classes with names like `Window` and Button.

Chapter 10: Object-Oriented Programming: 10-1b Polymorphism
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

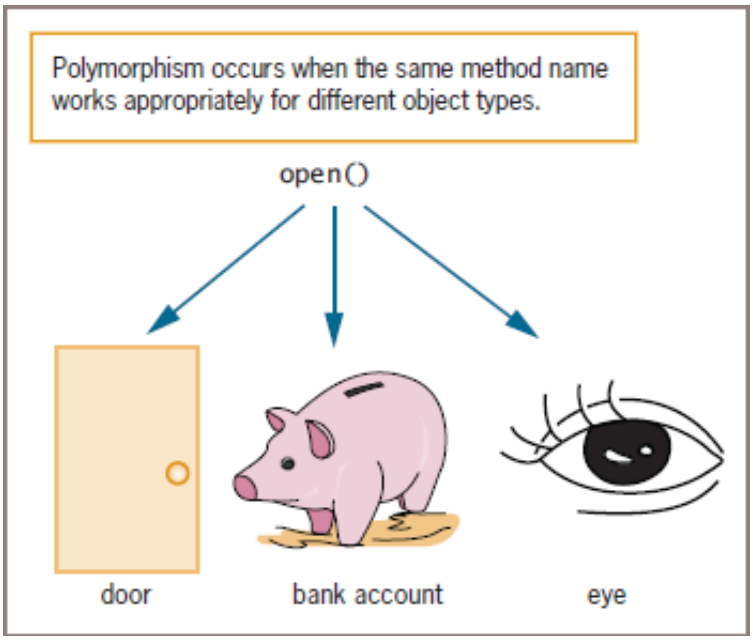
10-1b Polymorphism

The real world is full of objects. Consider a door. A door needs to be opened and closed. You open a door with an easy-to-use interface known as a doorknob. Object-oriented programmers would say you are *passing a message* to the door when you tell it to open by turning its knob. The same message (turning a knob) has a different result when applied to your radio than when applied to a door. As depicted in Figure 10-2, the procedure you use

to open something—call it the "open" procedure—works differently on a door than it does on a desk drawer, a bank account, a computer file, or your eyes. However, even though these procedures operate differently using the various objects, you can call each of these procedures "open."

Figure 10-2

Examples of Polymorphism



Within classes in object-oriented programs, you can create multiple methods with the same name, which will act differently and appropriately when used with different types of objects. In Chapter 9, you learned that this concept is called *polymorphism*, and you learned to overload methods. For example, you might use a method named `print()` to print a customer invoice, loan application, or envelope. Because you use the same method name to describe the different actions needed to print these diverse objects, you can write statements in object-oriented programming languages that are more like English; you can use the same method name to describe the same type of action, no matter what type of object is being acted upon. Using the method name `print()` is easier than remembering `printInvoice()`, `printLoanApplication()`, and so on. Object-oriented languages understand verbs in context, just as people do.

As another example of the advantages to using one name for a variety of objects, consider a screen you might design for a user to enter data into an application you are writing. Suppose that the screen contains a variety of objects—some forms, buttons, scroll bars, dialog boxes, and so on. Suppose also that you decide to make all the objects blue. Instead of having to memorize the method names that these objects use to change color—perhaps `changeFormColor()`, `changeButtonColor()`, and so on—your job would be easier if the creators of all those objects had developed a `setColor()` method that works appropriately with each type of object.

Purists find a subtle difference between overloading and polymorphism.

Some reserve the term *polymorphism* (or **pure polymorphism** (The situation in which one method implementation can be used with a variety of arguments in object-oriented programming.)) for situations in which one method body is used with a variety of arguments. For example, a single method that can be used with any type of object is polymorphic. The term *overloading* is applied to situations in which you define multiple methods with a single name—for example, three methods, all named `display()`, that display a number, an employee, and a student, respectively. Certainly, the two terms are related; both refer to the ability to use a single name to communicate multiple meanings. For now, think of overloading as a primitive type of polymorphism.

Chapter 10: Object-Oriented Programming: 10-1c Inheritance
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

10-1c Inheritance

Another important concept in object-oriented programming is **inheritance** (The process of acquiring the traits of one's predecessors.) , which is the process of acquiring the traits of one's predecessors. In the real world, a new door with a stained glass window inherits most of its traits from a standard door. It has the same purpose, it opens and closes in the same way, and it has the same knob and hinges. As Figure 10-3 shows, the door with the stained glass window simply has one additional trait—its window. Even if you have never seen a door with a stained glass window, you know what it is and how to use it because you understand the characteristics of all doors. With object-oriented programming, once you create an object, you can develop new objects that possess all the traits of the original object plus any new traits you desire. If you develop a `CustomerBill` class of objects, there is no need to develop an `OverdueCustomerBill` class from scratch. You can create the new class to contain all the characteristics of the already developed one, and simply add necessary new characteristics. This not only reduces the work involved in creating new objects, it makes them easier to understand because they possess most of the characteristics of already developed objects.

Figure 10-3

An Example of Inheritance

 Watch the video *An Introduction to Object-Oriented Programming*.

Chapter 10: Object-Oriented Programming: 10-1d Encapsulation
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

10-1d Encapsulation

Real-world objects often employ encapsulation and information hiding. **Encapsulation** (The act of containing a task's instructions and data in the same method.) is the process of combining all of an object's attributes and methods into a single package; the package includes data that is frequently hidden from outside classes as well as methods that are available to outside classes to access and alter the data. **Information hiding** (The concept that other classes should not alter an object's attributes—only the methods of an object's own class should have that privilege.) is the concept that other classes should not alter an object's attributes—only the methods of an object's own class should have that privilege. (The concept is also called **data hiding** (The concept that other classes should not alter an object's attributes—only the methods of an object's own class should have that privilege.) .) Outside classes should only be allowed to make a request that an attribute be altered; then it is up to the class's methods to determine whether the request is appropriate. When using a door, you usually are unconcerned with the latch or hinge construction, and you don't have access to the interior workings of the knob. You care only about the functionality and the interface, the user-friendly boundary between the user and internal mechanisms of the device. When you turn a door's knob, you are interacting appropriately with the interface. Banging on the knob would be an inappropriate interaction, so the door would not respond. Similarly, the detailed workings of objects you create within object-oriented programs can be hidden from outside programs and modules if necessary, and the methods you write can control how the objects operate. When the details are hidden, programmers can focus on the functionality and the interface, as people do with real-life objects.

In summary, understanding object-oriented programming means that you must consider five of its integral components: classes, objects, polymorphism, inheritance, and

encapsulation.

Two Truths & A Lie

Principles of Object-Oriented Programming

1. Learning about object-oriented programming is difficult because it does not use the concepts you already know, such as declaring variables and using modules.

   T F

2. In object-oriented terminology, a class describes a group or collection of objects with common attributes; an instance of a class is an existing object of a class.

   T F

3. A program or class that instantiates objects of another prewritten class is a class client or class user.

   T F

Chapter 10: Object-Oriented Programming: 10-2 Defining Classes and Creating Class Diagrams
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

10-2 Defining Classes and Creating Class Diagrams

A class is a category of things; an object is a specific instance of a class. A **class definition** (A class definition is a set of program statements that list the characteristics of the class's objects and the methods each object can use.) is a set of program statements that list the characteristics of each object and the methods each object can use.

A class definition can contain three parts:

- Every class has a name.
- Most classes contain data, although this is not required.
- Most classes contain methods, although this is not required.

For example, you can create a class named `Employee`. Each `Employee` object will represent one employee who works for an organization. Data members, or attributes of the `Employee` class, include fields such as `lastName`, `hourlyWage`, and `weeklyPay`.

The methods of a class include all the actions you want to perform with the class. Appropriate methods for an `Employee` class might include `setHourlyWage()`, `getHourlyWage()`, and `calculateWeeklyPay()`. The job of `setHourlyWage()` is to provide values for an `Employee`'s wage data field, the purpose of `getHourlyWage()` is to retrieve the wage value, and the purpose of `calculateWeeklyPay()` is to multiply the `Employee`'s `hourlyWage` by the number of hours in a workweek to calculate a weekly salary. With object-oriented languages, you think of the class name, data, and methods as a single encapsulated unit.

Declaring a class does not create actual objects. A class is just an abstract description of what an object will be if any objects are actually instantiated. Just as you might understand all the characteristics of an item you intend to manufacture before the first item rolls off the assembly line, you can create a class with fields and methods long before you instantiate objects that are members of that class. After an object has been instantiated, its methods can be accessed using the object's identifier, a dot, and a method call. When you declare a simple variable that is a built-in data type, you write a statement such as one of the following:

```
num money
string name
```

When you write a program that declares an object that is a class data type, you write a statement such as the following:

```
Employee myAssistant
```

In some object-oriented programming languages, you need to add more to the declaration statement to actually create an `Employee` object. For example, in Java you would write:

```
Employee myAssistant = new Employee();
```

You will understand more about the format of this statement when you learn about constructors in Chapter 11.

When you declare the `myAssistant` object, it contains all the data fields and has access to all the methods contained within its class. In other words, a larger section of memory is set aside than when you declare a simple variable, because an `Employee` contains several fields. You can use any of an `Employee`'s methods with the `myAssistant` object. The usual syntax is to provide an object name, a dot (period), and a method name. For example, you can write a program that contains statements such as those shown in Figure 10-4.

Figure 10-4

Application that Declares and Uses an `employee` Object

```
start
   Declarations
      Employee myAssistant
   myAssistant.setLastName("Reynolds")
   myAssistant.setHourlyWage(16.75)
   output "My assistant makes ",
      myAssistant.getHourlyWage(), " per hour"
stop
```

The program segment in Figure 10-4 is very short. In a more useful real-life program, you might read employee data from a data file before assigning it to the object's fields, each `Employee` might contain dozens of fields, and your application might create hundreds or thousands of objects.

Besides referring to `Employee` as a class, many programmers would refer to it as a **user-defined type** (A type that is not built into a language but is created by the programmer.), but a more accurate term is **programmer-defined type** (A type that is not built into a language but is created by the programmer.). Object-oriented programmers typically refer to a class like `Employee` as an **abstract data type (ADT)** (A programmer-defined type, such as a class.); this term implies that the type's data can be accessed only through methods.

When you write a statement such as `myAssistant.setHourlyWage(16.75)`, you are making a call to a method that is contained within the `Employee` class. Because `myAssistant` is an `Employee` object, it is allowed to use the `setHourlyWage()` method that is part of its class. You can tell from the method call that `setHourlyWage()` must accept a numeric parameter.

When you write the application in Figure 10-4, you do not need to know what statements are written within the `Employee` class methods, although you could make an educated guess based on the method names. Before you could execute the application in Figure 10-4, someone would have to write appropriate statements within the `Employee` class methods. If you wrote the methods, of course you would know their contents, but if another programmer has already written the methods, you could use the application without knowing the details contained in the methods. To use the methods, you only need to know

their signatures—their names and parameter lists.

In Chapter 9, you learned that the ability to use methods as a black box without knowing their contents is a feature of encapsulation. The real world is full of many black-box devices. For example, you can use your television and microwave oven without knowing how they work internally—all you need to understand is the interface. Similarly, with well-written methods that belong to classes you use, you need not understand how they work internally to be able to use them; you need only understand the ultimate result when you use them.

In the client program segment in Figure 10-4, the focus is on the object—the `Employee` named `myAssistant`—and the methods you can use with that object. This is the essence of object-oriented programming.

In older object-oriented programming languages, simple numbers and characters are said to be **primitive data types** (In a programming language, simple number and character types that are not class types.) ; this distinguishes them from objects that are class types. In the newest programming languages, every item you name, even one that is a numeric or string type, is an object that is a member of a class with both data and methods.

When you instantiate objects, their data fields are stored at separate memory locations. However, all members of the same class share one copy of the class's methods. You will learn more about this concept later in this chapter.

10-2a Creating Class Diagrams

A **class diagram** (A tool for describing a class that consists of a rectangle divided into three sections that show the name, data, and methods of a class.) consists of a rectangle divided into three sections, as shown in Figure 10-5. The top section contains the name of the class, the middle section contains the names and data types of the attributes, and the bottom section contains the methods. This generic class diagram shows two attributes and three methods, but a given class might have any number of either, including none. Programmers

often use a class diagram to plan or illustrate class features. Class diagrams also are useful for describing a class to nonprogrammers.

Figure 10-5

Generic Class Diagram

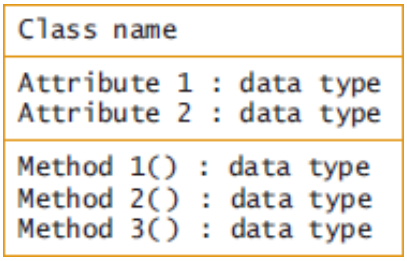
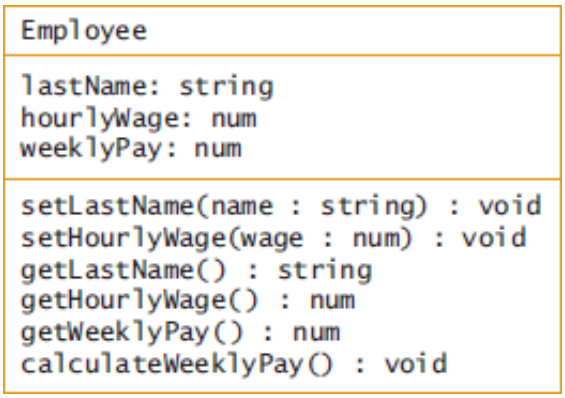


Figure 10-6 shows the class diagram for the `Employee` class. By convention, a class diagram lists the names of the data items first; each name is followed by a colon and the data type. Method names are listed next, and each is followed by its data type (return type). Listing the names first and the data types last emphasizes the purposes of the fields and methods.

Figure 10-6

`Employee` Class Diagram



Class diagrams are a type of Unified Modeling Language (UML) diagram.

Chapter 13 covers the UML.

Figures 10-5 and 10-6 both show that a class diagram is intended to be only an overview of class attributes and methods. A class diagram shows *what* data items and methods the class will use, not the details of the methods nor *when* they will be used. It is a design tool that helps you see the big picture in terms of class requirements. Figure 10-6 shows the `Employee` class containing three data fields that represent an employee's name, hourly pay rate, and weekly pay amount. Every `Employee` object created in a program that uses this class will contain these three data fields. In other words, when you declare an `Employee` object, the single declaration statement allocates enough memory to hold all three fields.

Figure 10-6 also shows that the `Employee` class contains six methods. For example, the first method is defined as follows:

```
setLastName(name : string) : void
```

This notation means that the method name is `setLastName()`, that it takes a single `string` parameter named `name`, and that it returns nothing.

Various books, Web sites, and organizations use class diagrams that describe methods in different ways. For example, some developers use the method name only, and others omit parameter lists. This book will take the approach of being as complete as possible, so the class diagrams you see here will contain each method's identifier, parameter list with types, and return type.

The `Employee` class diagram shows that two of the six methods take parameters (`setLastName()` and `setHourlyWage()`). The diagram also shows the return type for each method—three void methods, two numeric methods, and one string method. The class diagram does not indicate what takes place inside the method, although you might be able to make an educated guess. Later, when you write the code that actually creates the `Employee` class, you include method implementation details. For example, Figure 10-7 shows some pseudocode you can use to list the details for the methods in the `Employee` class.

Figure 10-7

Pseudocode for `employee` Class Described in the Class Diagram in Figure 10-6

In Figure 10-7, the `Employee` class attributes are identified with a data type and a field name. In addition to listing the required data fields, the figure shows the complete methods for the `Employee` class. The purposes of the methods can be divided into three categories:

- Two of the methods accept values from the outside world; these methods, by convention, have the prefix *set*. These methods are used to set the data fields in the class.
- Three of the methods send data to the outside world; these methods, by convention, have the prefix *get*. These methods return field values to a client program.
- One method performs work within the class; this method is named `calculateWeeklyPay()`. This method does not communicate with the outside; its purpose is to multiply `hourlyWage` by the number of hours in a week.

10-2b The Set Methods

In Figure 10-7, two methods begin with the word *set*; they are `setLastName()` and `setHourlyWage()`. The purpose of a **set method** (An instance method that sets the values of a data field within a class.) or **mutator method** (An instance method that can modify an object's attributes.) is to set the values of data fields within the class. There is no requirement that such methods start with *set*; the prefix is merely conventional and clarifies the intention of the methods. The method `setLastName()` is implemented as follows:

```
void setLastName(string name)
   lastName = name
return
```

In this method, a string `name` is passed in as a parameter and assigned to the field `lastName`. Because `lastName` is contained in the same class as this method, the method has access to the field and can alter it.

Similarly, the method `setHourlyWage()` accepts a numeric parameter and assigns it to the class field `hourlyWage`. This method also calls the `calculateWeeklyPay()` method, which sets `weeklyPay` based on `hourlyWage`. By writing the `setHourlyWage()` method to call the `calculateWeeklyPay()` method automatically, you guarantee that the `weeklyPay` field is updated any time `hourlyWage` changes.

When you create an `Employee` object with a statement such as `Employee mySecretary`, you can use statements such as the following:

```
mySecretary.setLastName("Johnson")
mySecretary.setHourlyWage(15.00)
```

Instead of literal constants, you could pass variables or named constants to the methods as long as they were the correct data type. For example, if you write a program in which you make the following declaration, then the assignment in the next statement is valid.

```
Declarations
   num PAY_RATE_TO_START = 8.00
mySecretary.setHourlyWage(PAY_RATE_TO_START)
```

In some languages—for example, Visual Basic and C#—you can create a **property** (A property provides methods that allow you to get and set a class field value using a simple syntax.) instead of creating a set method. Using a property provides a way to set a field value using a simpler syntax. By convention, if a class field is `hourlyWage`, its property would be `HourlyWage`, and in a program you could make a statement similar to `mySecretary.HourlyWage = PAY_RATE_TO_START`. The implementation of the property `HourlyWage` (with an uppercase initial letter) would be written in a format very similar to that of the `setHourlyWage()` method.

Like other methods, the methods that manipulate fields within a class can contain any statements you need. For example, a more complicated `setHourlyWage()` method might be written as in Figure 10-8. In this version, the wage passed to the method is tested against minimum and maximum values, and is assigned to the class field `hourlyWage` only if it falls within the prescribed limits. If the wage is too low, the `MINWAGE` value is substituted, and if the wage is too high, the `MAXWAGE` value is substituted.

Figure 10-8

More Complex `sethourlywage()` Method

Similarly, if the set methods in a class required them, the methods could contain output statements, loops, array declarations, or any other legal programming statements. However, if the main purpose of a method is not to set a field value, then for clarity the method should not be named with the *set* prefix.

10-2c The Get Methods

The purpose of a **get method** (An instance method that returns a value from a class.) or **accessor method** (A method that gets values from class fields.) is to return a value to the world outside the class. In the `Employee` class in Figure 10-7, the three get methods have the prefix *get*: `getLastName()`, `getHourlyWage()`, and `getWeeklyPay()`. The methods are implemented as follows:

Each of these methods simply returns the value in the field implied by the method name. Like set methods, any of these get methods could also contain more complex statements as needed. For example, in a more complicated class, you might return the hourly wage of an employee only if the user had also passed an appropriate access code to the method, or you might return the weekly pay value as a string with a dollar sign attached instead of as a numeric value. When you declare an `Employee` object such as `Employee mySecretary`, you can then make statements in a program similar to the following:

In other words, the value returned from a get method can be used as any other variable of its type would be used. You can assign the value to another variable, display it, perform arithmetic with it, or make any other statement that works correctly with the returned data type.

In some languages—for example, Visual Basic and C#—instead of creating a get method, you can add statements to the property to return a value using simpler syntax. For example, if you created an `HourlyWage` property, you could write a program that contains the statement `output mySecretary.HourlyWage`.

Chapter 10: Object-Oriented Programming: 10-2d The Work Methods
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

10-2d The Work Methods

The `Employee` class in Figure 10-7 contains one method that is neither a get nor a set method. This method, `calculateWeeklyPay()`, is a **work method** (A method that performs tasks within a class.) within the class. A work method is also known as a **help method** (A work method.) or **facilitator** (A work method.) . It contains a locally named constant that represents the hours in a standard workweek, and it computes the `weeklyPay` field value by multiplying `hourlyWage` by the named constant. The method is written as follows:

No values need to be passed into this method, and no value is returned from it because the method does not communicate with the outside world. Instead, this method is called only from another method in the same class (the `setHourlyWage()` method), and that method is called from the outside world. Each time a program uses the `setHourlyWage()` method to alter an `Employee`'s `hourlyWage` field, `calculateWeeklyPay()` is called to recalculate the `weeklyPay` field. No `setWeeklyPay()` method is included in this `Employee` class because the intention is that `weeklyPay` is set only inside the `calculateWeeklyPay()` method each time the `setHourlyWage()` method calls it. If you wanted programs to be able to set the `weeklyPay` field directly, you would have to write a method to allow it.

Programmers who are new to class creation often want to pass the `hourlyWage` value into the `calculateWeeklyPay()` method so that it can use the value in its calculation. Although this technique would work, it is not required. The `calculateWeeklyPay()` method has direct access to the `hourlyWage` field by virtue of being a member of the same class.

For example, Figure 10-9 shows a program that declares an `Employee` object and sets the hourly wage value. The program displays the `weeklyPay` value. Then a new value is assigned to `hourlyWage`, and `weeklyPay` is displayed again. As you can see from the output in Figure 10-10, the `weeklyPay` value has been recalculated even though it was never set directly by the client program.

Figure 10-9

Program that Sets and Displays `employee` Data Two Times

Figure 10-10

Execution of Program in Figure 10-9

Two Truths & A Lie

Defining Classes and Creating Class Diagrams

1. Every class has a name, data, and methods.

   T F

2. After an object has been instantiated, its methods can be accessed using the object's identifier, a dot, and a method call.

   T F

3. A class diagram consists of a rectangle divided into three sections; the top section contains the name of the class, the middle section contains the names and data types of the attributes, and the bottom section contains the methods.

   T F

10-3 Understanding Public and Private Access

When you buy a new product, one of the usual conditions of its warranty is that the manufacturer must perform all repair work. For example, if your computer has a warranty and something goes wrong with its operation, you cannot open the system unit yourself, remove and replace parts, and then expect to get your money back for a device that does not work properly. Instead, when something goes wrong, you must take the computer to an approved technician. The manufacturer guarantees that your machine will work properly only if the manufacturer can control how the computer's internal mechanisms are modified.

Similarly, in object-oriented design, you do not want outside programs or methods to alter your class's data fields unless you have control over the process. For example, you might design a class that performs complicated statistical analysis on some data, and you would not want others to be able to alter your carefully crafted result. Or, you might design a class from which others can create an innovative and useful GUI screen object. In this case you would not want anyone altering the dimensions of your artistic design. To prevent outsiders from changing your data fields in ways you do not endorse, you force other programs and methods to use a method that is part of your class to alter data. (Earlier in this chapter, you learned that the principle of keeping data private and inaccessible to outside classes is called *information hiding* or *data hiding*.)

To prevent unauthorized field modifications, object-oriented programmers usually specify that their data fields will have **private access** (A privilege of class members in which data or methods cannot be used by any method that is not part of the same class.) —the data cannot be accessed by any method that is not part of the class. The methods themselves, like `setHourlyWage()` in the `Employee` class, support public access. When methods have **public access** (A privilege of class members in which other programs and methods may use the specified data or methods within a class.) , other programs and methods may use the methods to get access to the private data.

Figure 10-11 shows a complete `Employee` class to which access specifiers have been added to describe each attribute and method. An **access specifier** (The adjective that defines the type of access outside classes will have to the attribute or method.) is the adjective that defines the type of access (`public` or `private`) outside classes will have to the attribute or method. In the figure, each access specifier is shaded.

Figure 10-11

`Employee` Class Including `public` and `private` Access Specifiers

In many object-oriented programming languages, if you do not declare an access specifier for a data field or method, then it is private by default. This book will follow the convention of explicitly specifying access for every class member.

In Figure 10-11, each of the data fields is private, which means each field is inaccessible to an object declared in a program. In other words, if a program declares an `Employee` object, such as `Employee myAssistant`, then the following statement is illegal:

Instead, `hourlyWage` can be assigned only through a public method as follows:

If you made `hourlyWage` public instead of private, then a direct assignment statement would work, but you would violate the important OOP principle of data hiding using encapsulation. Data fields should usually be private, and a client application should be able to access them only through the public interfaces—in other words, through the class's public methods. That way, if you have restrictions on the value of `hourlyWage`, those restrictions will be enforced by the public method that acts as an interface to the private data field. Similarly, a public get method might control how a private value is retrieved. Perhaps you do not want clients to have access to an `Employee`'s `hourlyWage` if it is more than a specific value, or maybe you want to return the wage to the client as a string with a dollar sign attached. Even when a field has no data value requirements or restrictions, making data private and providing public set and get methods establishes a framework that makes such modifications easier in the future.

In the `Employee` class in Figure 10-11, only one method is not public; the `calculateWeeklyPay()` method is private. That means if you write a program and declare an `Employee` object such as `Employee myAssistant`, then the following statement is not permitted:

Because it is private, the only way to call the `calculateWeeklyPay()` method is from another method that already belongs to the class. In this example, it is called from the `setHourlyWage()` method. This prevents a client program from setting `hourlyWage` to one value while setting `weeklyPay` to an incompatible value. By making the `calculateWeeklyPay()` method private, you ensure that the class retains full control over when and how it is used.

Classes usually contain private data and public methods, but as you have just seen, they can contain private methods. Classes can contain public data items as well. For example, an `Employee` class might contain a public constant data field named `MINIMUM_WAGE`; outside programs then would be able to access that value without using a method. Public data fields are not required to be named constants, but they frequently are.

In some object-oriented programming languages, such as C++, you can label a set of data fields or methods as public or private using the access specifier name just once, then following it with a list of the items in that category. In other languages, such as Java, you use the specifier *public* (+) or *private* (–) with each field or method. For clarity, this book will label each field and method as public or private.

Many programmers like to specify in class diagrams whether each component in a class is public or private. Figure 10-12 shows the conventions that are typically used. A minus sign (–) precedes the items that are private; a plus sign (+) precedes those that are public.

Figure 10-12

`Employee` Class Diagram with Public and Private Access Specifiers

When you learn more about inheritance in Chapter 11, you will learn about an additional access specifier—the protected access specifier. You use an octothorpe, also called a pound sign or number sign (#), to indicate protected access.

In object-oriented programming languages, the main program is most often written as a method named `main()` or `Main()`, and that method is virtually always defined as public.

Watch the video *Creating a Class*.

Two Truths & A Lie

Understanding Public and Private Access

1. Object-oriented programmers usually specify that their data fields will have private access.

   T F

2. Object-oriented programmers usually specify that their methods will have private access.

   T F

3. In a class diagram, a minus sign (–) precedes the items that are private; a plus sign (+) precedes those that are public.

   T F

10-4 Organizing Classes

The `Employee` class in Figure 10-12 contains just three data fields and six methods; most classes you create for professional applications will have many more. For example, in addition to a last name and pay information, real employees require an employee number, a first name, address, phone number, hire date, and so on, as well as methods to set and get those fields. As classes grow in complexity, deciding how to organize them becomes increasingly important.

Although it is not required, most programmers place data fields in some logical order at the beginning of a class. For example, an ID number is most likely used as a unique identifier for each employee, so it makes sense to list the employee ID number first in the class. An employee's last name and first name "go together," so it makes sense to store the two components adjacently. Despite these common-sense rules, you have considerable flexibility when positioning your data fields within a class. For example, depending on the class, you might choose to store the data fields alphabetically, or you might group together all the fields that are the same data type. Alternatively, you might choose to store all public data items first, followed by private ones, or vice versa.

In some languages, you can organize data fields and methods in any order within a class. For example, you could place all the methods first, followed by all the data fields, or you could organize the class so that data fields are followed by methods that use them. This

book will follow the convention of placing all data fields first so that you can see their names and data types before reading the methods that use them. This format also echoes the way data and methods appear in standard class diagrams.

For ease in locating a class's methods, some programmers store them in alphabetical order. Other programmers arrange them in pairs of get and set methods, in the same order as the data fields are defined. Another option is to list all accessor (get) methods together and all mutator (set) methods together. Depending on the class, you might decide to create other logically functional groupings. Of course, if your company distributes guidelines for organizing class components, you must follow those rules.

Two Truths & A Lie

Organizing Classes

1. As classes grow in complexity, deciding how to organize them becomes increasingly important.

   T F

2. You have a considerable amount of flexibility in how you position your data fields within a class.

   T F

3. In a class, methods must be stored in the order in which they are used.

   T F

10-5 Understanding Instance Methods

Classes contain data and methods, and every instance of a class possesses the same data and has access to the same methods. For example, Figure 10-13 shows a class diagram for a simple `Student` class that contains just one private data field for a student's grade point average. The class also contains get and set methods for the field. Figure 10-14 shows the pseudocode for the `Student` class. This class becomes the model for a new data type named `Student`; when `Student` objects are created, eventually each will have its own `gradePointAverage` field and have access to methods to get and set it.

Figure 10-13

Class Diagram for `student` Class

Figure 10-14

Pseudocode for the `student` Class

If you create multiple `Student` objects using the class in Figure 10-14, you need a separate storage location in computer memory to store each `Student`'s unique grade point average. For example, Figure 10-15 shows a program that creates three `Student` objects and assigns values to their `gradePointAverage` fields. It also shows how the `Student` objects look in memory after the values have been assigned.

Figure 10-15

`Studentdemo` Program and How `student` Objects Look in Memory

It makes sense for each `Student` object in Figure 10-15 to have its own `gradePointAverage` field, but it does not make sense for each `Student` to have its own copy of the methods that get and set `gradePointAverage`. Creating identical copies of a method for each instance would be inefficient. Instead, even though every `Student` has its own `gradePointAverage` field, only one copy of each of the methods `getGradePointAverage()` and `setGradePointAverage()` is stored in memory; however, each instantiated object of the class can use the single method copy. A method that works appropriately with different objects is an **instance method** (A method that operates correctly yet differently for each class object; an instance method is nonstatic and receives a `this` reference.) .

Although the `StudentDemo` class contains only one copy of the get and set methods, they work correctly for any number of instances. Therefore, methods like `getGradePointAverage()` and `setGradePointAverage()` are instance methods. Because only one copy of each instance method is stored, the computer needs a way to determine which `gradePointAverage` is being set or retrieved when one of the methods is called. The mechanism that handles this problem is illustrated in Figure 10-16. When a method call such as `oneSophomore.setGradePointAverage(2.6)` is made, the true method call, which is invisible and automatically constructed, includes the memory address of the `oneSophomore` object. (These method calls are represented by the three narrow boxes in the center of Figure 10-16.)

Figure 10-16

How `student` Addresses Are Passed from an Application to an Instance Method of the `student` Class

Within the `setGradePointAverage()` method in the `Student` class, an invisible and automatically created parameter is added to the list. (For illustration purposes, this parameter is named `aStudentAddress` and is shaded in the `Student` class definition in Figure 10-16. In fact, no parameter is created with that name.) This parameter accepts the address of a `Student` object because the instance method belongs to the `Student` class; if this method belonged to another class—`Employee`, for example—then the method would accept an address for that type of object. The shaded addresses are not written as code in any program—they are "secretly" sent and received behind the scenes. The address variable in Figure 10-16 is called a `this` reference. A **this reference** (An automatically created variable that holds the address of an object and passes it to an instance method whenever the method is called.) is an automatically created variable that holds the address of an object and passes it to an instance method whenever the method is called. It is called a `this` reference because it refers to "this particular object" that is using the method at the

moment. In other words, an instance method receives a `this` reference to a specific class instance. In the application in Figure 10-16, when `oneSophomore` uses the `setGradePointAverage()` method, the address of the `oneSophomore` object is contained in the `this` reference. Later in the program, when the `oneJunior` object uses the `setGradePointAverage()` method, the `this` reference will hold the address of that `Student` object.

Figure 10-16 shows each place the `this` reference is used in the `Student` class. It is implicitly passed as a parameter to each instance method. You never explicitly refer to the `this` reference when you write the method header for an instance method; Figure 10-16 just shows where it implicitly exists. Within each instance method, the `this` reference is implied any time you refer to one of the class data fields. For example, when you call `setGradePointAverage()` using a `oneSophomore` object, the `gradePointAverage` assigned within the method is the "*this* `gradePointAverage`", or the one that belongs to the `oneSophomore` object. The phrase "this `gradePointAverage`" usually is written as `this`, followed by a dot, followed by the field name—`this.gradePointAverage`.

The `this` reference exists throughout every instance method. You can explicitly use the `this` reference with data fields, but it is not required. Figure 10-17 shows two locations where the `this` reference can be used implicitly, or where you can (but do not have to) use it explicitly. Within an instance method, the following two identifiers mean exactly the same thing:

- any field name defined in the class
- `this`, followed by a dot, followed by the same field name

For example, within the `setGradePointAverage()` method, `gradePointAverage` and `this.gradePointAverage` refer to exactly the same memory location.

Figure 10-17

Explicitly Using `this` in the Student Class

The `this` reference can be used only with identifiers that are field names. For example, in Figure 10-17 you can refer to this.`gradePointAverage`, but you cannot refer to `this.gpa` because `gpa` is not a class field—it is only a local variable.

The syntax for using `this` differs among programming languages. For example, within a class in C++, you can refer to the `Student` class `gradePointAverage` value as `this->gradePointAverage` or `(*this).gradePointAverage`, but in Java you refer to it as `this.gradePointAverage`. In Visual Basic, the `this` reference is named `Me`, so the variable would be `Me.gradePointAverage`.

Usually you do not need to use the `this` reference explicitly within the methods you write, but the `this` reference is always there, working behind the scenes, accessing the data field for the correct object.

Your organization might prefer that you explicitly use the `this` reference for clarity even though it is not required to create a workable program. It is the programmer's responsibility to follow the conventions established at work or by clients.

As an example of when you might use the `this` reference explicitly, consider the following `setGradePointAverage()` method and compare it to the version in the `Student` class in Figure 10-17.

In this version of the method, the programmer has used the variable name `gradePointAverage` as the parameter to the method and as the instance field within the class. Therefore, `gradePointAverage` is the name of a local variable within the method whose value is received by passing; it also is the name of a class field. To differentiate the two, you explicitly use the `this` reference with the copy of `gradePointAverage` that is a member of the class. Omitting the `this` reference in this case would result in the local parameter `gradePointAverage` being assigned to itself, and the class's instance variable would not be set.

Any time a local variable in a method has the same identifier as a field, the field is hidden; you must use a `this` reference to distinguish the field from the local variable.

Watch the video *The* `this` *Reference*.

Two Truths & A Lie

Understanding Instance Methods

1. An instance method operates correctly yet differently for each separate instance of a class.

   T F

2. A `this` reference is a variable you must explicitly declare with each class you create.

   T F

3. When you write an instance method in a class, the following two identifiers within the method always mean exactly the same thing: any field name or `this` followed by a dot, followed by the same field name.

   T F

Chapter 10: Object-Oriented Programming: 10-6 Understanding Static Methods
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

10-6 Understanding Static Methods

Some methods do not require a `this` reference because it makes no sense for them either implicitly or explicitly. For example, the `displayStudentMotto()` method in Figure 10-18 does not use any data fields from the `Student` class, so it does not matter which `Student` object calls it. If you write a program in which you declare 100 `Student` objects, the `displayStudentMotto()` method executes in exactly the same way for each of them; it does not need to know whose motto is displayed and it does not need to access any specific object addresses. As a matter of fact, you might want to display the `Student` motto without instantiating any `Student` objects. Therefore, the `displayStudentMotto()` method can be written as a static method instead of an instance method.

Figure 10-18

`Student` Class `displaystudentmotto()` Method

When you write a class, you can indicate two types of methods:

- **Static methods** (Methods for which no object needs to exist; static methods are not instance methods and they do not receive a `this` reference.) , also called **class methods** (A static method; class methods are not instance methods and they do not receive a this reference.) , are those for which no object needs to exist, like the `displayStudentMotto()` method in Figure 10-18. Static methods do not receive a `this` reference as an implicit parameter. Typically, static methods include the word `static` in the method header, as shown shaded in Figure 10-18.

- **Nonstatic methods** (Methods that exist to be used with an object; they are instance methods and they receive a `this` reference.) are methods that exist to be used with an object. These instance methods receive a `this` reference to a specific object. In most programming languages, you use the word `static` when you want to declare a static class member, but you do not use a special word when you want a class member to be nonstatic. In other words, methods in a class are nonstatic instance methods by default.

In everyday language, the word `static` means “stationary”; it is the opposite of *dynamic*, which means “changing.” In other words, static methods are always the same for every instance of a class, whereas nonstatic methods act differently depending on the object used to call them.

In most programming languages, you use a static method with the class name, as in the following:

In other words, no object is necessary with a static method.

In some languages, notably C++, besides using a static method with the class name, you also can use a static method with any object of the class, as in `oneSophomore.displayStudentMotto()`.

Two Truths & A Lie

Understanding Static Methods

1. Class methods do not receive a `this` reference.

   T F

2. Static methods do not receive a `this` reference.

   T F

3. Nonstatic methods do not receive a `this` reference.

   T F

Chapter 10: Object-Oriented Programming: 10-7 Using Objects
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

10-7 Using Objects

A class is a complex data type defined by a programmer, but in many ways you can use its instances like you use items of simpler data types. For example, you can create an array of objects, pass an object to a method, or return an object from a method.

Figure 10-19

`InventoryItem` Class

Consider the `InventoryItem` class in Figure 10-19. The class represents items that a company manufactures and holds in inventory. Each item has a number, description, and price. The class contains a get and set method for each of the three fields.

Once you declare an `InventoryItem` object, you can use it in many of the ways you would use a simple numeric or string variable. For example, you could pass an `InventoryItem` object to a method or return one from a method. Figure 10-20 shows a program that declares an `InventoryItem` object and passes it to a method for display. The `InventoryItem` is declared in the main program and assigned values. Then the completed item is passed to a method, where it is displayed. Figure 10-21 shows the execution of the program.

Figure 10-20

Application that Declares and Uses an `InventoryItem` Object

Figure 10-21

Execution of Application in Figure 10-20

The `InventoryItem` declared in the main program in Figure 10-20 is passed to the `displayItem()` method in much the same way a numeric or string variable would be. The method receives a copy of the `InventoryItem` that is known locally by the identifier item. Within the method, the field values of the local item can be retrieved, displayed, and used in arithmetic statements in the same way they could have been in the main program where the `InventoryItem` was originally declared.

Figure 10-22 shows a more realistic application that uses `InventoryItem` objects. In the main program, an `InventoryItem` is declared and the user is prompted for a number. As long as the user does not enter the `QUIT` value, a loop is executed in which the entered inventory item number is passed to the `getItemValues()` method. Within that method, a local `InventoryItem` object is declared. This local object gathers and holds the user's input values. The user is prompted for a description and price, and then the passed item number and newly obtained description and price are assigned to the local `InventoryItem` object via its set methods. The completed object is returned to the program, where it is assigned to the `InventoryItem` object. That item is then passed to the `displayItem()` method. As in the previous example, the method calculates tax and displays results. Figure 10-23 shows a typical execution.

Figure 10-22

Application that Uses `InventoryItem` Objects

Figure 10-23

Typical Execution of Program in Figure 10-22

In Figure 10-22, notice that the return type for the `getItemValues()` method is `InventoryItem`. A method can return only a single value. Therefore, it is convenient that the `getItemValues()` method can encapsulate two strings and a number in a single `InventoryItem` object that it returns to the main program.

Two Truths & A Lie

Using Objects

1. You can pass an object to a method.

   T F

2. Because only one value can be returned from a method, you cannot return an object that holds more than one field.

   T F

3. You can declare an object locally within a method.

   T F

10-8 Chapter Review

10-8a Chapter Summary

- Classes are the basic building blocks of object-oriented programming. A class describes a collection of objects; each object is an instance of a class. A class's fields, or instance variables, hold its data, and every object that is an instance of a class has access to the same methods. A program or class that instantiates objects of another prewritten class is a class client or class user. In addition to classes and objects, three important features of object-oriented languages are polymorphism, inheritance, and encapsulation.

- A class definition is a set of program statements that list the characteristics of each object and the methods that each object can use. A class definition can contain a name, data, and methods. Programmers often use a class diagram to illustrate class features. The purposes of many methods contained in a class can be divided into three categories: set methods, get methods, and work methods.

- Object-oriented programmers usually specify that their data fields will have private access—that is, the data cannot be accessed by any method that is not part of the class. The methods frequently support public access, which means that other programs and methods may use the methods that control access to the private data. In a class diagram, a minus sign (–) precedes the items that are private; a plus sign (+) precedes those that are public.

- As classes grow in complexity, deciding how to organize them becomes increasingly important. Depending on the class, you might choose to store the data fields by listing a key field first. You also might list fields alphabetically, by data type, or by accessibility. Methods might be stored in alphabetical order or in pairs of get and set methods.

- An instance method operates correctly yet differently for every object instantiated from a class. When an instance method is called, a `this` reference that holds the object's memory address is automatically passed to the method.

- Some methods do not require a `this` reference. When you write a class, you can indicate two types of methods: static methods, which are also known as class methods and do not receive a `this` reference as an implicit parameter; and nonstatic methods, which are instance methods and do receive a `this` reference.

- You can use objects in many of the same ways you use simpler data types.

Chapter Review

10-8b Key Terms

Object-oriented programming (OOP) (A programming technique that focuses on objects, or "things," and describes their attributes and behaviors.)

class (A group or collection of objects with common attributes.)

object (One tangible example of a class; an instance of a class.)

instance (An existing object or tangible example of a class.)

instantiation (An instance of a class.)

instantiate (To create an object.)

Attributes (One field or column in a database table or a characteristic that defines an object as part of a class.)

is-a relationship (An is-a relationship exists between an object and its class.)

instance variables (The data components that belong to every instantiated object.)

fields (are object attributes or data.)

state (The set of all the values or contents of a class's instance variables.)

class client (A program or class that instantiates objects of another prewritten class.)

class user (A program or class that instantiates objects of another prewritten class.)

pure polymorphism (The situation in which one method implementation can be used with a variety of arguments in object-oriented programming.)

inheritance (The process of acquiring the traits of one's predecessors.)

Encapsulation (The act of containing a task's instructions and data in the same method.)

Information hiding (The concept that other classes should not alter an object's attributes—only the methods of an object's own class should have that privilege.)

data hiding (The concept that other classes should not alter an object's attributes—only the methods of an object's own class should have that privilege.)

class definition (A class definition is a set of program statements that list the characteristics of the class's objects and the methods each object can use.)

user-defined type (A type that is not built into a language but is created by the programmer.)

programmer-defined type (A type that is not built into a language but is created by the programmer.)

abstract data type (ADT) (A programmer-defined type, such as a class.)

primitive data types (In a programming language, simple number and character types that are not class types.)

class diagram (A tool for describing a class that consists of a rectangle divided into three sections that show the name, data, and methods of a class.)

set method (An instance method that sets the values of a data field within a class.)

mutator method (An instance method that can modify an object's attributes.)

property (A property provides methods that allow you to get and set a class field value using a simple syntax.)

get method (An instance method that returns a value from a class.)

accessor method (A method that gets values from class fields.)

work method (A method that performs tasks within a class.)

help method (A work method.)

facilitator (A work method.)

private access (A privilege of class members in which data or methods cannot be used by any method that is not part of the same class.)

public access (A privilege of class members in which other programs and methods may use the specified data or methods within a class.)

access specifier (The adjective that defines the type of access outside classes will have to the attribute or method.)

instance method (A method that operates correctly yet differently for each class object; an instance method is nonstatic and receives a `this` reference.)

this reference (An automatically created variable that holds the address of an object and passes it to an instance method whenever the method is called.)

Static methods (Methods for which no object needs to exist; static methods are not instance methods and they do not receive a `this` reference.)

class methods (A static method; class methods are not instance methods and they do not receive a this reference.)

Nonstatic methods (Methods that exist to be used with an object; they are instance methods and they receive a `this` reference.)

Chapter 10: Object-Oriented Programming: 10-8c Review Questions
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

Chapter Review

10-8c Review Questions

1. Which of the following means the same as *object*?
    a. class
    b. field
    c. instance
    d. category
2. Which of the following means the same as *instance variable*?
    a. field
    b. instance
    c. category
    d. class
3. A program that instantiates objects of another prewritten class is a(n) ____ .
    a. object
    b. client
    c. instance
    d. GUI
4. The relationship between an instance and a class is a(n) ____ relationship.
    a. has-a

b. is-a

c. polymorphic

d. hostile

5. Which of these does not belong with the others?

a. instance variable

b. attribute

c. object

d. field

6. The process of acquiring the traits of one's predecessors is ____ .

a. inheritance

b. encapsulation

c. polymorphism

d. orientation

7. When discussing classes and objects, *encapsulation* means that ____ .

a. all the fields belong to the same object

b. all the fields are private

c. all the fields and methods are grouped together

d. all the methods are public

8. Every class definition must contain ____ .

a. a name

b. data

c. methods

d. all of the above

9. Assume that a working program contains the following statement:

```
myDog.setName("Bowser")
```

Which of the following do you know?

a. `setName()` is a public method.

b. `setName()` accepts a string parameter.

c. both of the above

d. none of the above

10. Assume that a working program contains the following statement:

```
name = myDog.getName()
```

Which of the following do you know?

a. `getName()` returns a string.

b. `getName()` returns a value that is the same data type as `name`.

c. both of the above

d. none of the above

11. A class diagram ____ .

a. provides an overview of a class's data and methods

b. provides method implementation details

c. is never used by nonprogrammers because it is too technical

d. all of the above

12. Which of the following is the most likely scenario for a specific class?

a. Its data is private and its methods are public.

b. Its data is public and its methods are private.

c. Its data and methods are both public.

d. Its data and methods are both private.

13. An instance method ____ .

a. is static

b. receives a `this` reference

c. both of the above

d. none of the above

14. Assume that you have created a class named `Dog` that contains a data field named `weight` and an instance method named `setWeight()`. Further assume that the `setWeight()` method accepts a numeric parameter named `weight`. Which of the following statements correctly sets a `Dog`'s weight within the `setWeight()` method?

    a. `weight = weight`

    b. `this.weight = this.weight`

    c. `weight = this.weight`

    d. `this.weight = weight`

15. A static method is also known as a(n) ____ method.

    a. instance

    b. public

    c. private

    d. class

16. By default, methods contained in a class are ____ methods.

    a. static

    b. nonstatic

    c. class

    d. public

17. Assume that you have created a class named `MyClass`, and that a working program contains the following statement:

    ```
    output MyClass.number
    ```

    Which of the following do you know?

    a. `number` is a numeric field.

    b. `number` is a static field.

c. `number` is an instance variable.

d. all of the above

18. Assume that you have created an object named `myObject` and that a working program contains the following statement:

```
output myObject.getSize()
```

Which of the following do you know?

a. `getSize()` is a static method.

b. `getSize()` returns a number.

c. `getSize()` receives a `this` reference.

d. all of the above

19. Assume that you have created a class that contains a private field named `myField` and a nonstatic public method named `myMethod()`. Which of the following is true?

a. `myMethod()` has access to `myField` and can use it.

b. `myMethod()` does not have access to `myField` and cannot use it.

c. `myMethod()` can use `myField` but cannot pass it to other methods.

d. `myMethod()` can use `myField` only if it is passed to `myMethod()` as a parameter.

20. An object can be ____ .

a. stored in an array

b. passed to a method

c. returned from a method

d. all of the above

Chapter Review

10-8d Exercises

1. Identify three objects that might belong to each of the following classes:
    a. `Building`
    b. `Artist`
    c. `BankLoan`
2. Identify three different classes that might contain each of these objects:
    a. William Shakespeare
    b. My favorite red sweater
    c. Public School 23 in New York City
3. Design a class named `TermPaper` that holds an author's name, the subject of the paper, and an assigned letter grade. Include methods to set the values for each data field and display the values for each data field. Create the class diagram and write the pseudocode that defines the class.
4. Design a class named `Automobile` that holds the vehicle identification number, make, model, and color of an automobile. Include methods to set the values for each data field, and include a method that displays all the values for each field. Create the class diagram and write the pseudocode that defines the class.
5. Design a class named `CheckingAccount` that holds a checking account number, name of account holder, and balance. Include methods to set values for each data field and a method that displays all the account information. Create the class diagram and write the pseudocode that defines the class.
6. Complete the following tasks:
    a. Design a class named `StockTransaction` that holds a stock symbol (typically one to four characters), stock name, and price per share. Include methods to set and get the values for each data field. Create the class diagram and write the pseudocode that defines the class.
    b. Design an application that declares two `StockTransaction` objects and

sets and displays their values.

c. Design an application that declares an array of 10 StockTransactions. Prompt the user for data for each of the objects, and then display all the values.

7. Complete the following tasks:

   a. Design a class named `Pizza`. Data fields include a string field for toppings (such as pepperoni) and numeric fields for diameter in inches (such as 12) and price (such as 13.99). Include methods to get and set values for each of these fields. Create the class diagram and write the pseudocode that defines the class.

   b. Design an application that declares two `Pizza` objects and sets and displays their values.

   c. Design an application that declares an array of 10 `Pizza`s. Prompt the user for data for each of the Pizzas, then display all the values.

8. Complete the following tasks:

   a. Design a class named `MagazineSubscription` that has fields for a subscriber's name, the magazine name, and number of months remaining in the subscription. Include methods to set and get the values for each data field. Create the class diagram and write the pseudocode that defines the class.

   b. Design an application that declares two `MagazineSubscription` objects and sets and displays their values.

   c. Design an application that declares an array of six `MagazineSubscription`s. Prompt the user for data for each object, and then display all the values. Then subtract 1 from each "months remaining" field and display the objects again.

Find the Bugs

9. Your downloadable student files for Chapter 10 include DEBUG10-01.txt, DEBUG10-02.txt, and DEBUG10-03.txt. Each file starts with some comments that describe the problem. Comments are lines that begin with two slashes (//). Following the comments, each file contains pseudocode that has one or

more bugs you must find and correct.

Game Zone

10. a. Playing cards are used in many computer games, including versions of such classics as Solitaire, Hearts, and Poker. Design a `Card` class that contains a string data field to hold a suit (spades, hearts, diamonds, or clubs) and a numeric data field for a value from 1 to 13. Include get and set methods for each field. Write an application that randomly selects two playing cards and displays their values.

    b. Using two `Card` objects, design an application that plays a simple version of the card game War. Deal two `Card`s—one for the computer and one for the player. Determine the higher card, then display a message indicating whether the cards are equal, the computer won, or the player won. (Playing cards are considered equal when they have the same value, no matter what their suit is.) For this game, assume that the Ace (value 1) is low. Make sure that the two `Card`s dealt are not the same `Card`. For example, a deck cannot contain more than one Queen of Spades.

Up for Discussion

11. In this chapter, you learned that instance data and methods belong to objects (which are class members), but that static data and methods belong to a class as a whole. Consider the real-life class named `StateInTheUnitedStates`. Name some real-life attributes of this class that are static attributes and instance attributes. Create another example of a real-life class and discuss what its static and instance members might be.

12. Some programmers use a system called Hungarian notation when naming their variables and class fields. What is Hungarian notation, and why do many objectoriented programmers feel it is not a valuable style to use?