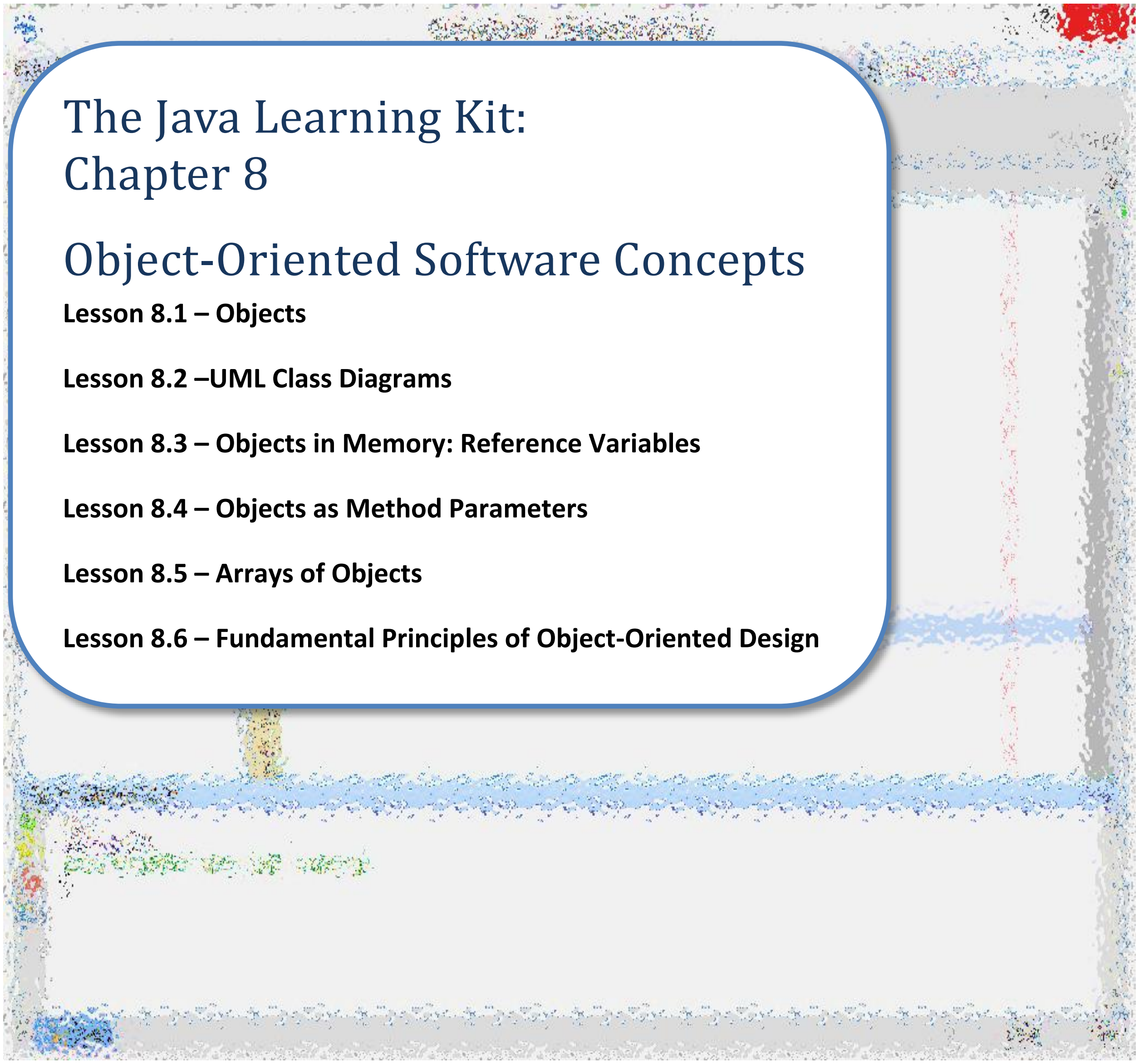


The Java Learning Kit: Chapter 8

Object-Oriented Software Concepts

Lesson 8.1 – Objects

Lesson 8.2 –UML Class Diagrams

Lesson 8.3 – Objects in Memory: Reference Variables

Lesson 8.4 – Objects as Method Parameters

Lesson 8.5 – Arrays of Objects

Lesson 8.6 – Fundamental Principles of Object-Oriented Design

The Java Learning Kit: Chapter – Oriented Software Concepts

Last edited January, 2015 by C. Herbert

This document is a chapter from a draft of the Java Learning Kit, written by Charles Herbert, with editorial input from Craig Nelson, Christopher Quinones, Matthew Staley, and Daphne Herbert. It is available free of charge for students in Computer Science courses at Community College of Philadelphia during the Spring 2015 semester.

Contents

The Java Learning Kit: Chapter 8

Object-Oriented Software Concepts

This is a short chapter introducing some fundamental concepts important for designing, creating and working with your own objects.

Chapter 8 Learning Outcomes

Upon completion of this chapter students should be able to:

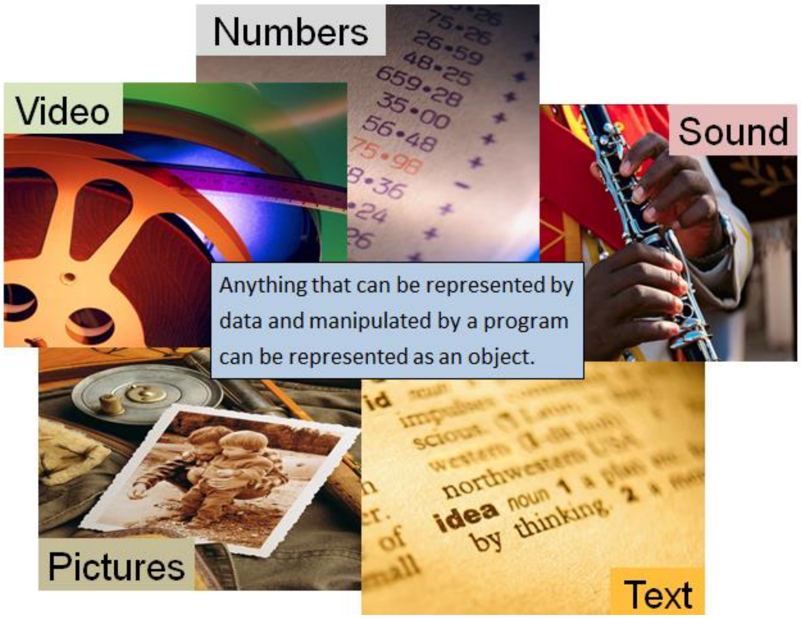
- Describe the nature of an object as a collection of properties and methods that manipulate the properties and the meaning of the terms object state and object behavior; Describe why an object declaration is sometimes called a blueprint for instances of an object.
- Describe how static members of a class differ from members associated with instances of a class.
- List and describe the different types of methods that can be remembered with the acronym CAMUS.
- Describe what a *visibility modifier* is and the different levels of visibility for properties, methods, and classes.
- Describe what Unified Modeling Language (UML) is and a brief history of how and why it appeared. Describe what is included in a UML class diagram.
- Design classes of objects for use in object-oriented programming, and create UML diagrams illustrating the class.
- Describe the concept of a reference variable and how objects are stored in memory using reference variables; describe how object reference variables affect parameters passing and how this works like two-way parameter passing; describe how using reference variables affects how objects are stored in an array.
- Describe the concept of encapsulation, the benefits of encapsulation, and how it is related to the notion of a problem domain and an implementation domain.
- Describe the concept of inheritance and how it makes programming more efficient.
- Describe the concept of polymorphism, including subtype polymorphism; the differences between ad hoc polymorphism and parametric polymorphism; and what is necessary to implement ad hoc polymorphism.

Lesson 8.1 Objects

So far, we have focused on a computer program as an implementation of an algorithm. Algorithms are most important in modern computer programming, but they are only part of the story. We need to understand object-oriented programming to truly understand modern computer software. We have seen objects, we have discussed objects, and we have even used objects, but we have not yet designed and built our own objects. This chapter introduces object-oriented design concepts, important for designing classes of objects and software based on the use of objects.

This chapter is paired with the next chapter, which is about users creating classes of objects in Java. This chapter introduces object concepts; the next introduces building classes of objects using Java. The two go together.

In true object-oriented software, all of the things that are manipulated by a computer are represented in the computer as objects. They may be things that exist in the physical world, such as an airplane manipulated by computerized controls, or they may be abstract concepts, such as a student's grade point average. Most modern software is made up of instances of objects from many different classes that interact with one another.



Anything that can be represented by data and manipulated by a program can be represented as an object.

The data that represents an object is organized into a set of properties. A **property** is a data element that describes an object in some way. For example, the weight of an airplane, its location, the direction in which it's facing, and so on, are each properties of an airplane. A computer manipulates an object by changing some of its properties.

The programs that manipulate the properties of an object are called the object's **methods**. In terms of object-oriented programming, as described back in chapter 1, an **object** is a collection of properties and the methods that are used to manipulate those properties.

The properties and methods in a class are also known as members of the class. The properties are known as **data members**, and the methods are called **member methods**.

Each individual copy of an object is known as an **instance** of the object. The class declaration acts as a blueprint for instances of an object. All of the instances will have the same properties and the same methods, but with different values stored in some of the properties.

Properties

The properties of an object – the object's data members – are also known as the object's *attributes*. Collectively, they describe the object. **Instance variables** are the variables that hold the properties for an instance of an object**.** The **state** of an instance of an object is the set of values stored in the instance variables for that instance at any one time. We can say that the state of an object is the collection of properties that define an object.

Each property has a data type, which is either a primitive data type or a data type defined by a class. For example, the cost of a diamond ring could be stored in a property with data type *double*, a primitive data type. The ring's style could be stored in a String, a data type that is itself a class of objects.

In addition to instance properties, a class can also have static properties. A **static property** is a class-level property associated with the class itself rather than with an instance of a class. For example, consider a class used to keep track of animals in a zoo. The weight of each animal could be stored in an instance variable named *weight*, while the total number of animals in the zoo could be stored in one static variable name *count*.

Instance variables are invoked using the name of the instance, while static variables are invoked using the name of the class. *JoeTheGiraffe.weight* is an example of an instance variable; *Animal.count* is an example of a static variable for the Animal class.

A **constant** is a property whose value cannot be changed. The value of a constant is normally defined as part of the class definition. Constants are commonly static class-level properties, such as the constants Math.PI (pi, 3.14159) and Math.E (Euler's number, 2.71828) included in Java's Math class.

Methods

Methods are programs that allow us to work with the properties of an object. The **behavior** of an object is the collection of actions performed by the object's methods.

To define an object we need to consider the collection of properties that will make up the state of an object and the collection of methods that will define the behavior of an object.

There are five different types of methods in Java and in most object-oriented programming, which can be remembered by the acronym CAMUS[1] –

- **Constructor** – a constructor is used to create an instance of an object. It has the same name as the class itself. For example, The method *Animal()* in the Animal class is a constructor for that class.
- **Accessor** – an accessor returns a value from the properties of an object. The value could be a property of the object, such as with a method named *getHours()* that returns an employee's *hoursWorked* property, or it could be a value that is derived from properties, such as with a *getGross()* method that returns (hoursWorked * hourlyRate). Often the name of an accessor starts with "*get*", as in the two examples just shown.
- **Mutator** – a mutator changes, or mutates, one or more properties of an object, thus changing the state of the object. Often the name of a mutator starts with "*set*", as in *setName().*

[1] remember this by remembering Albert Camus, 20th Century writer and philosopher, winner of the 1957 Nobel Prize for Literature.

- **Utility** – a utility method performs some function that does not access or change the properties of an object, such as the *Math.sqrt()* method in Java's Math class. Utility methods are often static methods.

- **Static** – a static method is a class level method, associated with a class and not with an instance of an object. It must be invoked using the class name, such as *Math.sqrt().* Static methods are often used as utility methods, providing functions for other software. They are also used to access static (class-level) variables.

  If a class has an executable *main()* method, then that method must be a static method, not associated with an instance of an object, but with the class itself. The programs we have created so far have all had a static *main()* method.

  Static methods may access other static methods, static variables and static constants directly, but they can only access instance methods, instance variables, or instance constants by invoking the name of an instance. Instance methods may access static methods, variables and constants directly.

Visibility Modifiers

Some of the properties and methods in a class may be accessed by methods from outside of the class and some properties and methods may only be accessed by member methods within the class. This level of access is known as a class member's **visibility**. There are three levels of visibility:

- **public** – methods from any class may access public properties and public methods.
- **private** – only methods within the same class may access private properties and private methods.
- **protected** – only members of the class and subclasses of the class may access protected properties and protected methods.

The *public* access modifier may also be used for the entire class. The class may be:

- **a public class** – the class is visible to any other classes if the access modifier *public* is used with the class name.
- **a package-private class** – if no access modifier is used with a class name, the class is only visible within its home package. The home package is the package in which the class is defined.

Lesson 8.2 UML Class Diagrams

Over the years, many different systems for describing computer software have been developed including many different ways to diagram software. In the 1990's three different systems for defining object-oriented software appeared:

- Object-Modeling Technique (OMT), developed by James Rumbaugh (1991)
- The Booch Method of Object-Oriented Design, developed by Grady Booch (1991)
- Use Case Object-Oriented Software Engineering, developed by Ivar Jacobson (1992)

Each of the three approaches had its own diagrams for modeling objects, in addition to many other software diagrams, such as flowcharts and state diagrams. All three of the developers listed above were hired by the Rational Corporation, which was later acquired by IBM. They worked together in the mid-1990's to develop **Unified Modeling Language (UML)**, a standardized set of diagrams for modeling object-oriented software systems and their development. UML was released in 1997 and adopted in 2000 by ISO as an international standard for the computer industry. The **Object Management Group (OMG)** is a nonprofit organization responsible for the UML standard.

A **UML class diagram** is a diagram showing the properties and methods in a class of objects. A UML class diagram is rectangle divided into three parts, stacked from top to bottom containing:

the Name of the Class
a list of the properties (data members) in the class
a list of the methods (member methods) in the class

The format is as follows:

ClassName
[visibility] [property name] : [data type] [visibility] [property name] : [data type] …
[visibility] [method name](parameter list) : [return data type] [visibility] [method name](parameter list) : [return data type] …

[visibility] is an access indicator, showing the level of access:

+ (a plus sign) for public

- (a minus sign)for private

(a number sign or hashtag) for protected

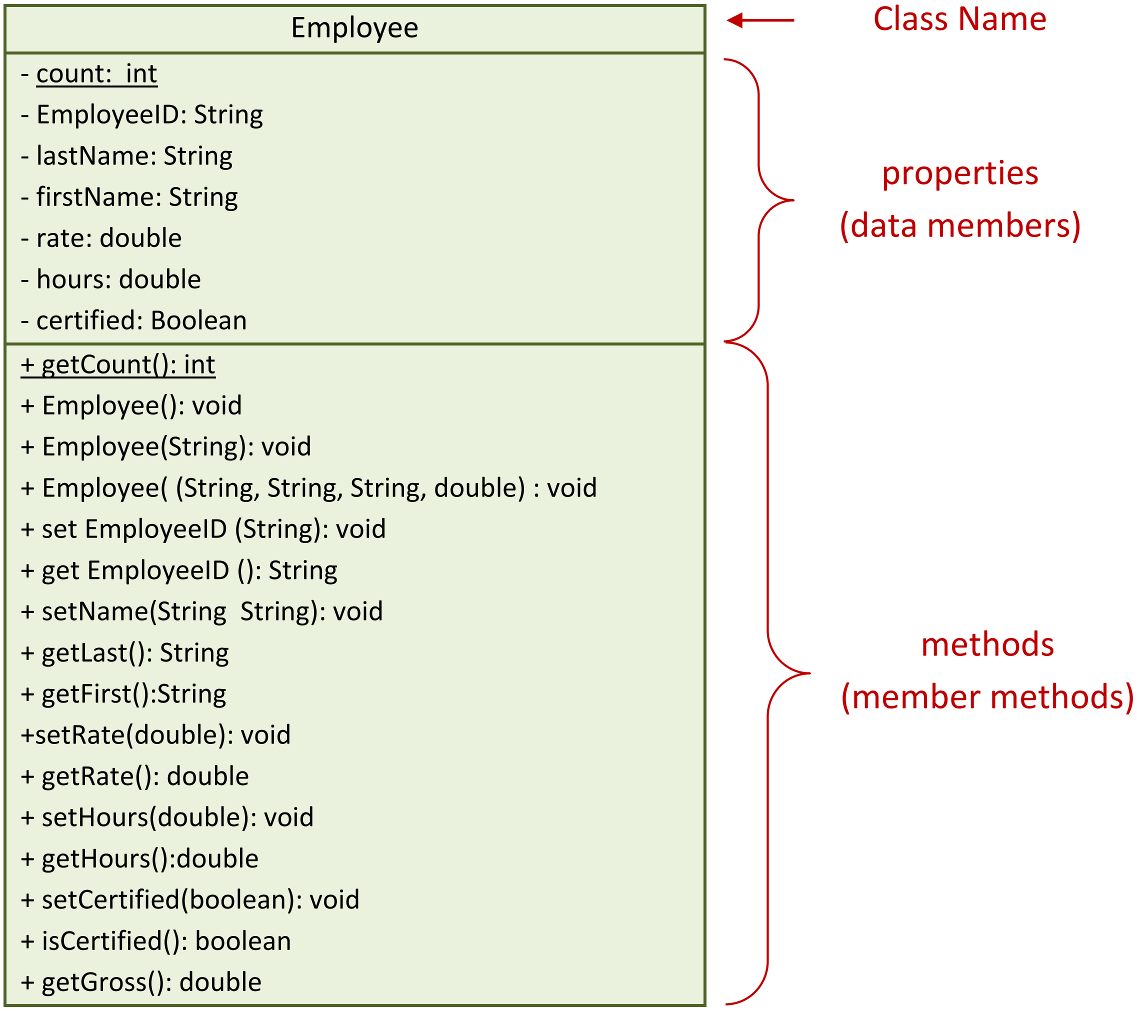
[property name] is the name of the property.
If the property is a static (class-level) property, it should be underlined.
If the property is a constant, it should be in ALL_CAPS.
The **[data type]** of the property is indicated following a colon after the property name.

[method name] is the name of a method.
If the method is a static (class-level) method, it should be underlined.
The **parameter list** lists the data types of the method's parameters (also known as the method's arguments) in parentheses immediately following the method name. A method without parameters is indicated by blank parentheses
The **[return data type]** of the value returned by the method is indicated following a colon after the method name. The word *void* is used if the method does not return a value.

Example - a UML Class Diagram

In this example we will have a class to keep track of employees in a payroll program. We need to decide what properties are needed in our class, and what methods will be needed to manipulate those properties. For this example, we need to keep track of the each employee's *first name, last name, pay rate, hours worked in the pay period*, and whether or not the person is *certified*. We need methods allowing us to work with these properties, and to return the *gross pay* for the week. We also wish to keep track of the number of employees.

Here is a UML class diagram based on the information above:



Employee

- count: int
- EmployeeID: String
- lastName: String
- firstName: String
- rate: double
- hours: double
- certified: Boolean

+ getCount(): int
+ Employee(): void
+ Employee(String): void
+ Employee((String, String, String, double) : void
+ set EmployeeID (String): void
+ get EmployeeID (): String
+ setName(String String): void
+ getLast(): String
+ getFirst():String
+setRate(double): void
+ getRate(): double
+ setHours(double): void
+ getHours():double
+ setCertified(boolean): void
+ isCertified(): boolean
+ getGross(): double

This UML diagram tells us a lot about the class, even though we can't see the details of the methods:

- There are three constructor methods, each with different parameters. This is an example of ad hoc polymorphism – multiple methods with the same name, but defined differently for different parameters.

- We can see that there is a static int variable named *count*, (because of the underlining) and a static method named *getCount()*, but no method to set the count. This is probably because the count is set when an instance is created using one of the three constructor methods.

- All of the properties of the objects are private, indicated by the minus sign in front of them. No software from outside of the class can access these methods directly; other software must use public methods to work with data inside the object. The plus signs indicate public methods.

- Many of the properties have accessors starting with *get* and mutators starting with *set*. The boolean property *certified* seems to match an accessor named *isCertified()*, which returns a boolean value. It could be used as the boolean condition in an *if* statement, such as:

```
if ( employee[i].isCertified() )
        system.out.println( getFirst.employee[i] + “ “ + getLast.employee[i] + “ is certified.”);
```

- There is no property named *gross*, but there is method named *getGross().* Such a method returns a derived property. A **derived property** is not a stored property of an object, but a value derived or calculated from stored properties. There is no method name *setGross()*, probably because there is no property named *gross*. It is possible that this object communicates with other objects which could have a stored property name *gross,* such as a *PayCheck object*.

- The *setName()* method takes two String parameters. It probably sets the *firstName* and *lastName* properties, which have no individual set methods.

Standard UML class diagrams are useful, but some people prefer annotated UML class diagram, which have comments providing more information. Here is an annotated UML diagram for the same class:

Employee	
- count: int	number of employees; updated by constructors
- EmployeeID: String	key field; unique for each employee
- lastName: String	
- firstName: String	
- rate: double	hourly pay rate
- hours: double	weekly hours
- certified: Boolean	
+ getCount(): int	
+ Employee(): void	default constructor
+ Employee(String): void	constructor with ID
+ Employee(String, String, String, double) : void	constructor with firstName, lastName, ID, rate
+ set EmployeeID (String): void	authorization required to use this method
+ get EmployeeID (): String	
+ setName(String String): void	sets firstName and lastName
+ getLast(): String	
+ getFirst():String	
+setRate(double): void	positive value <= 100.00
+ getRate(): double	
+ setHours(double): void	positive value <= 100.0
+ getHours():double	
+ setCertified(boolean): void	password required
+ isCertified(): boolean	
+ getGross(): double	includes overtime for hours > 40

Lesson 8.3 Objects in Memory: Reference Variables

Objects are stored in memory using reference variables, which is different from how primitive data types are stored in memory.

The value of a primitive variable is stored directly in the memory location associated with that variable. The diagram below shows how variables would be stored based on the code shown:

```
int x;
int y = 27;
double z;
x= 10;
z = 123.6;
```

variable	address	value
x	4000H	10
y	4004H	27
z	4008H	123.6

The value of *x* is 10, which is stored directly in the memory location associated with that variable; in this case it is 4000H. The value 27 is stored in the memory location for y, and the value 123.6 is stored in the memory location for z. The operating system manages this for us.

If we copy *x* to another variable, then the value in its associated memory location is copied:

```
y = x;
```

When this instruction is executed, the value of x is copied to y. *10* is copied to *4004H*.

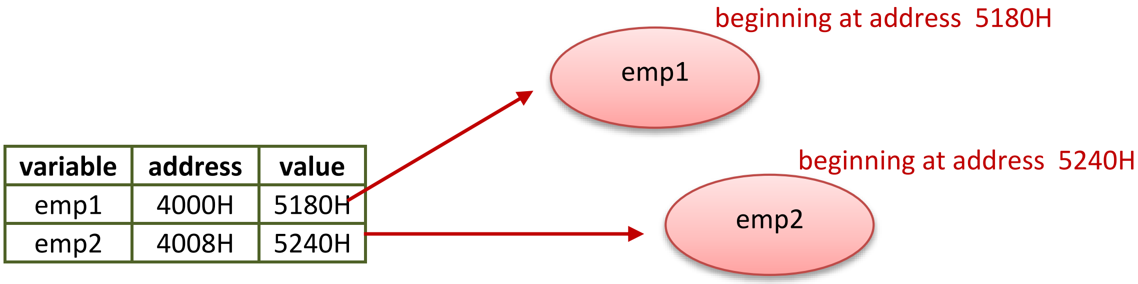
variable	address	value
x	4000H	10
y	4004H	10
z	4008H	123.6

The variables associated with objects are reference variables, which work differently. **Reference variables** associated with an object hold a reference to the location where the object is actually stored, as shown in the following diagram for variables of type String and Employee, which are objects:

```
Employee emp1 = new Employee();
Employee emp2 = new Employee();
String z;
```

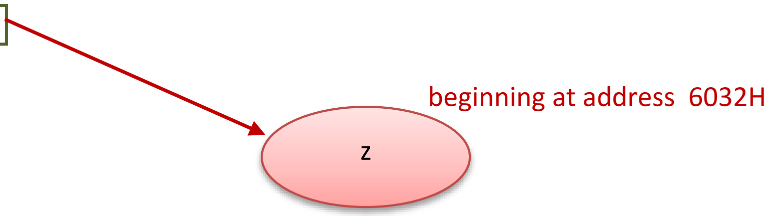
variable	address	value
emp1	4000H	5180H
emp2	4008H	5240H
z	4010H	6032H

In this case, the reference variables each hold the address in memory where an object is really stored, not the actual value of an object. In fact, they hold the base address for the object. The object's properties are stored in memory beginning at the reference location. *emp1* holds the value *5180H*. The actual values of the properties of *emp1* are stored in memory starting at location *5180H*. If we ask for something like *emp1.hours,* the operating system will use this information to find and return the actual values of *emp1.hours.*



variable	address	value
emp1	4000H	5180H
emp2	4008H	5240H

z	4010H	6032H



If we copy *emp1* to another variable, then the value in its associated memory location is copied, but in this case the value is a reference to where the object is actually stored, memory address *5180H.*

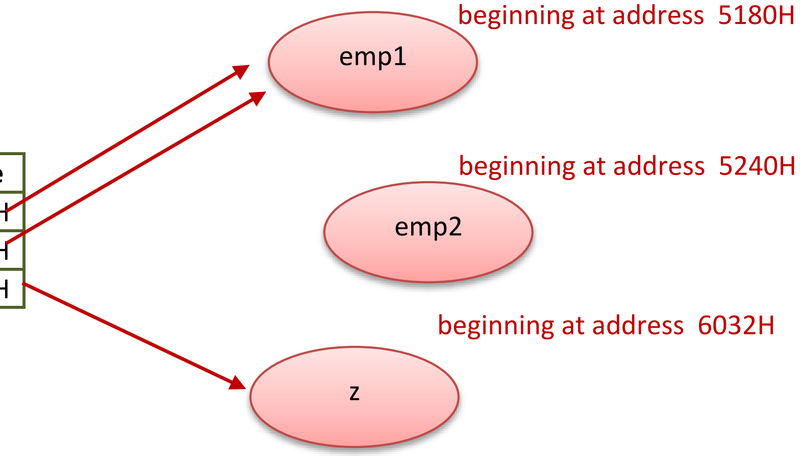
```
emp2 = emp1;
```

When this instruction is executed, the value of emp1 is copied to emp2. The reference address 5180H is copied.

variable	address	value
emp1	4000H	5180H
emp2	4008H	5180H
z	4010H	6032H

This is very different from what happens with primitive data types. We now end up with two different reference variables pointing to the same location in memory.

variable	address	value
emp1	4000H	5180H
emp2	4008H	5180H
z	4010H	6032H


beginning at address 5180H
emp1
beginning at address 5240H
emp2
beginning at address 6032H
z

emp2 will be now lost,
since nothing refers to it.

This difference is most important. It means, that whatever we do with *emp1* affects the values stored in memory beginning at address *5180H*, and whatever we do with *emp2* also affects the values stored in memory beginning at address *5180H*. *emp1* and *emp2* are now two different names for the same object. For example, if we change the value of *emp1.hours* to be *42*, then *emp2.hours* is *42* because they are the same object.

Lesson 8.4 Objects as Method Parameters

The use of reference variables affects parameter passing. If we use a reference variable as a parameter for a method, then the receiving variable has the same value as the sending variable, which is a reference to an object. This means that both variables – the one in the first method invoking the second method, and the one in the second method – refer to the same object. Whatever we do to the object in the second method happens to the object in the first method, because it is the same object. The affect is the same as if the method returned the entire object.

Here is an example:

```
public void methodA()
    {
    Employee emp1 = new Employee(); // emp1 now refers to an employee object somewhere in memory
    …                                                       // all of the code is not
shown
    emp1.hours = 36;
    method B(emp1);
    …
    System.out.println("Hours = " + emp1.hours);
    …
    } // end methodA
/***********************************************************/
public void methodB(Employee emp2)
    {
    …                                                      // all of the code is not
shown
    emp2.hours = 40;
    …
    } // end methodB
```

The code shown here will print:

Hours = 40

Both *emp1* and *emp2* refer to the same object. Remember how parameter passing works; the value of the actual parameter is passed to the variable listed in the formal parameter list. In this case, the value of *emp1* is passed to *emp2*, but the value of *emp1* is the reference address where the object is really stored in memory. This means *emp2* will have that same reference address, so *emp2* and *emp1* reference the same object. When *emp2.hours* is changed to *40*, *emp1.hours* is changed to *40* because *emp1* and *emp2* are two different references to the same object.

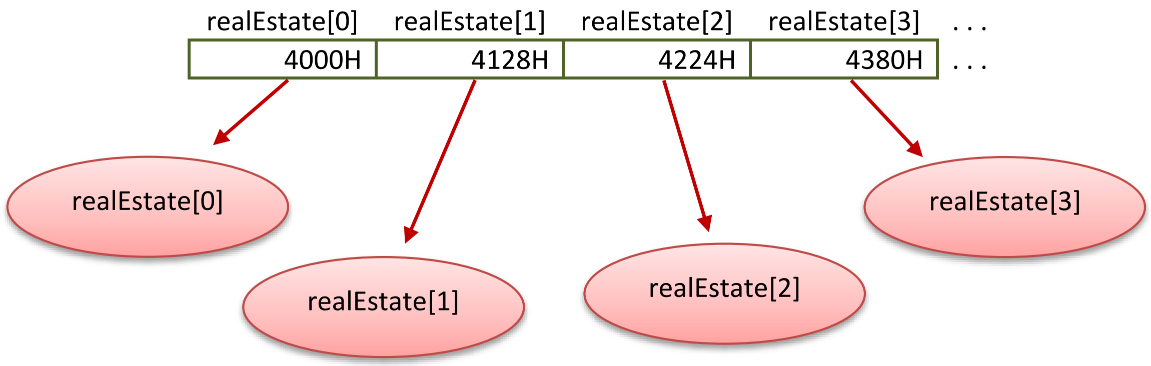
Lesson 8.5 Arrays of Objects

The use of reference variables affects how objects are stored in an array. An array of objects does not contain the actual objects, but references to where the objects are really stored in memory. Consider the following example:

We have a class of objects named *House*, with properties for things like the street address, the value, the number of bedrooms, and so on. We declare an array of houses.

```
// declares realEstate to be an array of type House
House[] realEstate = new House[100];
```

Each element in the array *realEstate* will now be a reference variable, holding the memory address where the corresponding object is actually stored in memory.



Each of the reference variables in the array will now work like any other reference variable.

Lesson 8.6 Fundamental Principles of Object-Oriented Design

Object-oriented design is an approach to software engineering that models a system as a group of interacting objects. Object-oriented design is intended to make it easier to design and build complex software systems. It emphasizes the development of reusable code. The preceding sections of this chapter discussed what objects are and how they work. Here we will look at three principles that are essential to true object-oriented design and object-oriented programming – encapsulation, inheritance, and polymorphism.

Encapsulation

To **encapsulate** an object is to hide the details of the object, as if it were in a capsule. From the outside, the only things that can be seen are the properties and methods that are publically accessible, with a *public* visibility modifier. Encapsulation is sometimes called *information hiding.*

The purpose of encapsulation is two-fold:

1. to protect the inner workings of an object from users of an object. We do this to stop users in the outside world from messing up the object by changing things they shouldn't change, and to make the objects more secure by limiting what details of the object are publicly available.
2. to protect users from needing to know and work with the complexities of an object. The public view of an object is often a simple one, not requiring users to know about or to work with all of the inner details of the object.

Encapsulation is a general design principle that often appears when a technology starts to become very complicated. In the industrial world it is related to the notion of streamlining machines, such as streamlined vehicles or household appliances.

Shown below are two railroad locomotives. The first is a 19th Century steam locomotive, *The Jupiter,* with most of its operating parts exposed – pistons, piston rods, boiler, steam tubes, oil lines, etc. The second is a modern Amtrak locomotive with its operating parts hidden. The streamlining affects the aerodynamics of the locomotive while it is moving, but it also encapsulates the locomotive so that its parts are protected from the public and the public is protected from its parts.

Most household appliances are encapsulated – washing machines, toasters, and so on. This is the same thing we do in object-oriented software; we encapsulate objects to protect their parts from users and to protect users from their parts.

Most often, all of the properties of an object are hidden from the public, available only through public accessor and mutator methods. This allows us to control how a user sees an object.

Encapsulation helps us to differentiate between the problem domain and the implementation domain when developing software solutions for problems or processes in the real world. The **problem domain** for a software engineering project is the real world external environment for which a software solution is being developed. The **implementation domain** is the internal world of computers and software development in which a solution is being developed. Encapsulation allows us to create a streamlined software solution with a look and feel that fits in the problem domain, using terminology, etc. on the outside that users who work in that domain will find familiar. The users are separated from the implementation domain – the internal world of computers.

For example, the problem domain for accounting software is the world of accounting with debits, credits, accounts payable, accounts receivable, etc. The implementation domain is the world of programming and computers, with classes, methods, data types, and so on. The user should see a GUI with terms like debit, credit, withdraw, deposit, and so on.

Microsoft Windows is an example of encapsulated software. We see folders, icons, buttons, etc., and do not see things such as system stack frames, priority queues, and software interrupts that implement the Windows operating system.

The Java Math class is also an example of encapsulation. We can use methods such as Math.sqrt() and Math.sin() in our software, but we don’t see the internal workings of these methods.

In all of these cases – accounting systems, Microsoft Windows, the Java Math class – the underlying system is protected from users who might mess things up, just as the user is protected from being overwhelmed by implementation details.

Inheritance

Inheritance was briefly discussed in the last chapter. **Inheritance** is the ability to define a **parent class** of objects with certain properties and methods, then declare a **child class** of the object, which automatically has the properties and methods of the parent class, along with any newly declared properties and methods of its own. The parent class is also called a superclass, and the child class is also called a subclass.

Many objects share the same properties and methods. Inheritance allows programmers to put these common members into one superclass, which can be extended to create subclasses that will automatically have those properties and methods. We can then add other properties and methods specific to a particular subclass.

Along with encapsulation and polymorphism, true object-oriented programing must allow for inheritance, which we explore in detail in a later chapter.

Polymorphism

Bjarne Stroustrup, who developed C++ as an object-oriented version of C, described **polymorphism** as "the provision of a single interface to entities of different types". For example, a print() method is polymorphic if can be used with different data types – if print(x) works regardless of the data type of x, then print() is a polymorphic method.

There are several different kinds of polymorphism.

Subtype polymorphism is related to inheritance; a method works for objects of the super class or for any of its subclasses, even though it might not perform exactly the same operation on each subtype.

Ad hoc polymorphism means that several different versions of a method have been defined, each with different formal parameters. This is most commonly seen with constructor methods such as in the Employee class used as an example in earlier in this chapter:

. . . Employee(): void Employee(String): void Employee(String, String, String, double) : void . . .	. . . default constructor constructor with ID constructor with firstName, lastName, ID, rate . . .

The three Employee constructor methods each take different parameters, and each has its own definition within the class.

The plus sign used in expressions in Java assignment statements is both polymorphic and encapsulated. The detailed process for adding two integers is different from the detailed process for adding two floating point numbers. The way to add integers 3 + 4 is different from the way to add the floating point values (3.640 x 10^3) + (1.456 x 10^5). To add the two floating point values we would need to modify them to have the same exponent, then add mantissas.

$$
\begin{aligned}
&(3.640 \text{ x } 10^3) + (1.456 \text{ x } 10^5) \\
&= (3.640 \text{ x } 10^3) + (145.6 \text{ x } 10^3) \\
&= 149.24 \text{ x } 10^3 \\
&= 1.4924 \text{ x } 10^5
\end{aligned}
$$

In Java, as in many other programming languages, `x = a + b;` works just as well whether the variables are of the int data type or the double data type. It even works with Strings, which would be concatenated to form one larger String. This is an example of both polymorphism and encapsulation, which often go hand-in-hand.

Ad hoc polymorphism is related to parametric polymorphism. **Parametric polymorphism** occurs when a method or an entire class of objects has been written so that it works independently of any data type. A parametrically polymorphic method is also called a generic method, because it will work with any data type.

Imagine an array that could hold any type of data, with functions to print the array, sort the array, etc. Such an array would be an example of parametric polymorphism. (In Math, the term parametric is related to the concept of deriving actual values indirectly from parameters. In computer programming, methods that are parametrically polymorphic involve such indirect derivation.)

Learning how to implement parametric polymorphism is beyond the scope of this course. It is related to ad hoc polymorphism in this way: parametric polymorphism works with any data type, while ad hoc polymorphism only works with certain data types, as defined in the declarations of the underlying methods. Constructor methods in most classes exhibit ad hoc polymorphism. We can implement ad hoc polymorphism simply by defining multiple methods with different names, but with different parameters.

True object-oriented programming languages are based on all three of these principles: encapsulation, inheritance, and polymorphism.

Key Terms in Chapter 8

After completing this chapter, You should be able to define each of the following key terms:

accessor, 5
ad hoc polymorphism, 15
behavior, 5
child class, 14
constant, 5
constructor, 5
derived property, 9
encapsulate, 13
implementation domain, 14
inheritance, 14
mutator, 5
Object Management Group (OMG), 7
object-oriented design, 13
package-private class, 6
parametric polymorphism, 16
parent class, 14
polymorphism, 15
private, 6
problem domain, 14
protected, 6
public, 6
public class, 6
reference variables, 10
static, 6
static property, 5
subtype polymorphism, 15
UML class diagram, 7
Unified Modeling Language (UML), 7
utility method, 6
visibility, 6

Chapter Questions

1. What does a constructor method do? What name should it have? What is the difference between accessor and mutator methods?
2. What is a static property? How are static properties invoked?
3. What is the difference between a public property and a private property? What is the default visibility for a class of objects?
4. What does a UML diagram look like? What does it show us? What is included in an annotated UML diagram?
5. What is actually held in the memory location for variables associated with objects? If we set one object variable equal to another – such as car2 = car1; – what is actually copied from one memory location to another?
6. What is passed from the actual parameter list to the variable in the formal parameter list if the parameter is an object? If we pass an object referred to by the variable *house1* in the sending method, to an object referred to by the variable *house2* in the receiving method, what happens to *house1.address* if we set *house2.address* to "*1600 Pennsylvania Avenue*"?
7. What is stored in each element in an array of objects? Where are the properties of the object stored?
8. Why do we encapsulate objects? How do we encapsulate objects? What can be seen from outside of a class if an object has been encapsulated?
9. What does inheritance allow a programmer to do?
10. When is a method polymorphic? What is the difference between ad hoc polymorphism and parametric polymorphism? What kind of polymorphism is related to inheritance, and how does it work?

Chapter Exercises

Exercise 1 – A UML diagram for a Bank Account Class

In this exercise you will create a UML diagram for a class of objects to keep track of bank accounts.

Design a class named *Account* that contains:

- A private int property named *id* for the account (default 0).
- A private double property named *balance* for the account (default 0).
- A private property of type *date* named *dateCreated* that stores the date when the account was originally created.
- A constructor without parameters that creates a default account.
- A constructor that creates an account with the specified id and initial balance.
- The accessor and mutator methods for id and balance.
- The accessor method for *dateCreated*.
- A method named *withdraw* that withdraws a specified amount from the account.
- A method named *deposit* that deposits a specified amount to the account.

Draw a UML diagram for the class.

Exercise 2 – designing Your Own Class

Design a class of objects to keep track of any one of the things listed below. For the object you choose, write a paragraph describing the data that needs to be stored for the object, and a paragraph listing what the methods for the object need to do. Create an annotated UML diagram that shows appropriate properties and methods for the object.

- houses for a real estate broker
- sales for a real estate broker
- insurance policies for a real estate insurance company
- animals in a zoo
- feedings for animals in a zoo
- cars for a car dealership
- sales for a car dealership
- the sales staff for a car dealership
- players in a monopoly game
- property in a monopoly game
- patients in a hospital
- doctors in a hospital
- procedures in a hospital

— End of Chapter 8 —