Chapter 2

The Essence of Objects

So, what exactly is Object Orientation? It is a problem solving technique used to develop software systems. Object orientation is the culmination of years of experience in finding an effective way to develop software, and is certainly the dominant method used to develop major software systems today. Object orientation is a technique of modeling a real-world system in software based on objects. The object is the core concept. An object is a software model of a real-world entity or concept.

Almost anything can be modeled as an object in software. For example, you could model a temperature sensor as an object. Or, in a more abstract system, you could model color as an object. Even something as basic as a number can be considered an object that has a value and a type. Typically, each object has an associated set of attributes such as value, state, or whatever else is needed to model the object. Sensor attributes might include a state such as active or inactive, an attribute indicating its current value, and information about its physical location. Objects usually provide the ability to modify their state as well. In addition to keeping track of the current temperature, for example, a temperature sensor object might provide a way to turn the sensor on or off.

The attributes, or data, used inside an object are really only a tiny part of what an object is and does. An object also has a set of responsibilities that it carries out by providing services to other objects. It is often more useful to think of an object in terms of its responsibilities rather than its attributes. For example, it is the responsibility of a sensor object to keep track of the state of the sensor. A sensor object might respond to requests from other objects that use sensors to check the status of a sensor, to turn a sensor on or off, or to report on the sensor's values. A sensor object could also maintain a history of its values as part of its responsibilities. The outside objects really don't care how a sensor object implements its attributes internally, but rather what services the sensor object can provide - its responsibilities.

While a program is running, individual objects usually don't stand alone. They belong to a collection of other similar objects that all are members of the same group, or class. A program will be made up of many different classes, each class made up of similar objects.

Classes and objects are closely related, but are not the same thing. A class is a description or definition of the characteristics of objects that belong to that class. An object is a single instance or member of a class that is created and exists while the program is running. A class may have just a single object or any number of member objects existing at any given time. All members of a class have similar behavior.

For example, consider a software system used to monitor various sensors in a factory. One obvious kind of object present is such a system is a sensor. A class called Sensor would be defined and used to model physical sensors. The class would define the basic characteristics of any Sensor, such as its location, value, and identification number, as well as a set of services used to carry out its responsibilities. Each individual physical sensor in the system would be represented as an object belonging to the class Sensor, and have specific values for the attributes described by the class definition.

The class description includes the means of accessing and changing the state of individual object members of that class. A common representation of color is called RGB, where the color is specified by the values of its red, green, and blue components. One possible design of a class called Color could provide the means of manipulating the color by both retrieving and setting the RGB values of a Color object.

In an object-oriented system, it is typical to describe one class based on a pre-existing class, either by extending the description of a higher level class, or by including the description of another class within the current class. For example, you could create one class that describes the general characteristics of all sensors, and then more specialized classes that describe specific sensors such as temperature or pressure sensors, each based on the original general Sensor class.

Placing attributes and responsibilities in a top level general class can provide many benefits. For example, if the responsibilities of the general Sensor class included keeping track of a history of readings from a sensor, programming that capability could get somewhat complex. By placing the history code in the top level class, that code need be defined only once, and won't be repeated in new classes. Each of the specialized sensors can then use the history capabilities of the top level Sensor class.

Objects and classes are really the heart of object orientation. OO software systems consist of objects of different classes that interact with each other using well-defined methods or services specified by the class definitions. When used properly and consistently, object-oriented software development leads to programs that are robust, and easy to debug, modify, and maintain.

To produce successful OO programs, it is important to always "think objects." Just because a program is written in Java or C++ does not mean it is an object-oriented program! If you have a programming background that is not OO based, or even if you've just learned Java, one of the great challenges is to switch the way you think about programming to use the object-oriented programming paradigm.

What Is an Object-Oriented System?

Just what is an object-oriented system? What makes an OO system different than other software systems? One way to define an object-oriented system is to use a list of properties that characterize object-oriented systems. A non-object-oriented system might share some...
properties such as using abstraction or encapsulation, but will not be built using objects or classes. It is also possible to use an object-oriented language to implement a system using classes or objects, but the system must have all the following properties to be considered a true object-oriented system.

Any object-oriented software system will have the following properties:

1. Abstraction with objects
2. Encapsulated classes
3. Communication via messages
4. Class hierarchies
5. Polymorphism
6. Object lifetime

The next section gives a brief overview of each of these properties, while the following sections give more detailed explanations of each.

Object orientation

A method of developing software that uses abstraction with objects, encapsulated classes, communication via messages, object lifetime, class hierarchies, and polymorphism.

Fundamental Properties of an Object-Oriented System

Abstraction with objects

An abstraction is a mechanism that allows a complex, real-world situation to be represented using a simplified model. Object orientation abstracts the real world based on objects and their interactions with other objects. For example, one possible abstraction of a color is the RGB model.

Encapsulated classes

The abstractions of related ideas are encapsulated into a single unit. The states and behaviors of the abstraction are incorporated into an encapsulated whole, called a class. The actual internal implementation of the states and behaviors is hidden from the rest of the system. While this is not a new programming technique, in OO the encapsulation is an inherent and integral part of the system and design. Earlier, we described a Color class that provided a way to change it red, green, or blue values. In fact, as long as the outside world continues to see and use a Color object in a consistent way, it wouldn't matter just how color is represented internally by the Color object. It could use either the HSV (hue, saturation, value) color model or the RGB model internally, and the outside world would be unaffected. The state and behavior of objects are controlled by well-defined and restricted interfaces to the object. Encapsulation ensures that the internal details of an object are hidden from the rest of the world, that each object maintains its own unique identity and state, and that the state can only be changed by well-defined messages.

Interaction via messages

In order to accomplish useful tasks, objects need to interact with other objects. The interaction can be between objects of the same class, or objects of different classes. This interaction is handled by sending messages (in Java, this is done by calling methods) to other objects to pass information or request action. For example, when a user selects a command button in a dialog box on the screen by clicking the mouse, a message is sent to the dialog object notifying it that the command button was pressed. Messages can be used to change the state of an object or to request an action by the object.

Object lifetime

All objects have a lifetime. They are created and initialized as they are needed during program execution, exist and carry out their functions, and are eventually destroyed. While objects exist, they maintain their own identity and state. Many objects that are instances of the same class can exist at any given time. Each object has a unique identity, and will have attributes that are different from other instances of objects in the same class.

Class hierarchies

In an OO design, classes of objects are arranged into hierarchies that model and describe relationships among the classes. The simplest relationship is an association. For example, there could be an employment association between a person and a company. These simple associations exist between different classes.

Hierarchies can also be used to build the definitions of individual classes. One way is to include other classes as part of one class. For example, consider a dialog graphical user interface class. Such a dialog would contain control objects such as buttons, lists, or value sliders. Thus, all the different control objects would be parts of the whole dialog class. This kind of hierarchy is called aggregation or composition.

A second way to use a hierarchy is to define more specialized classes based on a pre-existing generalized class. For example, a dialog class can be considered a specialized case of a more general window class. The more specialized class will automatically share the attributes of the more general class (e.g., size and screen position), and will probably add new attributes and behaviors to the generalized class (e.g., associated control objects). This kind of hierarchy is called inheritance.

Polymorphism

Polymorphism is the final fundamental characteristic of object-oriented systems. When inheritance is used to extend a generalized class to a more specialized class, it will usually include extending some of the behaviors of the generalized class. The specialized class will often implement a behavior that is somewhat different than the generalized class, but the name used to define the behavior will be the same. It is important that a given instance of an object use the correct behavior, and the property of polymorphism allows this to happen automatically and seamlessly. Polymorphism is actually easier to use than it is to explain. We will discuss polymorphism in more detail later.

If you read about OO in other sources, you will no doubt find slightly different terminology than we use here, but abstraction, encapsulation, messages, lifetime, hierarchies, and polymorphism are really the heart of the matter. The presence of all these properties is required for a software system to be considered object-oriented. If a system doesn't include abstraction, encapsulation, messages, lifetime, hierarchies, and polymorphism, then it isn't object-oriented, even if it is written using Java, C++, or some other OO language.

Abstraction with Objects

Abstraction with objects
Abstraction is one of the basic tools of all programming, not just OO programming. When trying to write a program to solve some real world problem, abstraction serves as a way model the real world problem. For example, if you were trying to write an address book program, you would use abstractions such as names, addresses, phone numbers, alphabetical order, and other concepts associated with an address book. You would also define operations for manipulating the attributes such as adding a new name or changing an address. Abstraction is modeling the real world in terms that can be implemented as a computer program.

Abstraction and OO fit together well. It is natural to model using objects. With an OO language such as Java, you can define objects with all the attributes and responsibilities needed to implement the model. The OO features of Java make it easy to map your abstractionstoo objects, once you know what your objects are. Designing with objects can be challenging, and it is not always easy to find the right objects for your model, but once you learn to think in objects, the process becomes almost second nature.

Almost anything you need to model in software can be considered an object - a temperature sensor in a control system, a person in a subscription system, a room of a building, a word in a sentence. Each of these objects has attributes and responsibilities. In the context of an abstraction, an object is a thing or concept. It can be a real-world thing or concept, or an abstraction of a thing or concept, expressed as a software representation.

**Encapsulated Classes**

Encapsulation is one of the most important aspects of OO. It is what allows each object to be independent. The exact implementation of attributes and of object behavior is hidden from the rest of the world through encapsulation.

The class definition is the main programming tool that is used to implement encapsulation. A class is a description of a collection of objects with common attributes, behavior, and responsibilities. The definition or specification of a class includes the definitions of the attributes comprising the state, the methods that carry out the responsibilities of the class by implementing the behavior, and how to set the initial attribute state of an object. A class is identified by a name.

A class should never allow direct access to state information by the outside world. Instead, it should change the state as part of its responsibilities, or sometimes provide methods for accessing and changing the state. As long as you maintain a well-defined interface to the rest of the world, you can easily modify your class definition without breaking the rest of the system.

| **attribute** | Used to hold state information of an object. An attribute might be as simple as an on/off boolean variable, or it might be a complex structure such as another object. A class definition describes the attributes and operations (methods) of a class. |
| **behavior** | The activity of an object that is visible to the outside world. Includes how an object responds to messages by changing its internal state or returning state information to other objects. |
| **method** | An operation or service performed upon an object, defined as part of the declaration of a class. Methods are used to implement object behavior. Synonyms for method include member function, operation, and service. |
| **state** | State reflects the current values of all the attributes of a given object, and is the result of the behavior of an object over time. |

Java programs are defined as collections of classes. Normally each Java class is defined in a separate file. The attributes of a class are defined by the declaration of variables of various types such as int or boolean. A Java class includes the definitions of the methods used to implement the behaviors of the class. The method definitions are integral parts of the class definition.

**Communication via messages**

Messages are how objects communicate with each other. Any object may send a message to other objects, and it may receive messages from other objects. In practical programming terms, sending a message is accomplished by invoking or calling some class method, and receiving a message is accomplished by having a class method called by a different object.

Usually, a message is sent by a method call as a normal part of the execution of the program logic of an object. However, messages may also originate from the operating system interface or language run-time system. Consider an object that implements a dialog interface with a user. When the user clicks on a button, a message is sent to the dialog object or button handler telling it that a specific button has been pressed (the implementation specifics aren't important). In this case, however, the user program usually doesn't monitor the mouse itself to determine which button was pressed. Instead, the underlying system monitors the mouse, and sends the message to the appropriate user program object. The Java run-time system and libraries provide many other support classes that can send and receive messages for user programs. This message to an object approach is easier to program than the technique usually known as callbacks used by non-OO programming languages. Instead of defining a separate callback procedure and then registering it with the system, a Java program will create an object based on a standard Java system library, and appropriate messages (such as a button press) will automatically be sent by the system to the appropriate object method.

There is another important aspect of the concept of messages. Messages drive program execution flow. The fact that messages can originate from the system as well as from the program itself means that OO programs will often not have the traditional linear program execution typical of non-OO programs (although they can, of course). Consider an interactive program with a graphical user interface (GUI). The parts of a GUI program required to execute in response to some command are controlled by the user interactively. Depending on which menu item the user selects, or what the user does with the mouse, different parts of the program will be executed. The messages corresponding to a menu pick or mouse gesture originate with the GUI system, and are sent to the appropriate program objects, which then have the responsibility to respond with some action. The order and timing of these messages is determined by the actions of the user, and not by the control flow of the program.

Object orientation is a natural for this kind of programming. In an OO program, what you often end up with is a set of objects that can respond to a set of messages originating from a variety of sources such as a mouse click, a sensor value change, or a database transaction. Individual encapsulated objects can respond to messages and send their own messages to other objects in response. Objects in the system interact via well-defined messages with other objects in the system.

**Object lifetime**

Objects are dynamic entities. They are created on an as-needed basis during program execution. When an object is created, it is instantiated, or bound to the class definition. An instantiated member of a class is called an object, or equivalently an instance. When a new object first comes into existence, a special method called the constructor is automatically invoked by the run-time system. The constructor is responsible for giving an instance its initial state. Once an object has been created, it can receive and send messages.

While it exists, an object has state and behavior. State is expressed by attributes, and behavior
is expressed by the methods associated with the object. State usually reflects changeable attributes of an object. Objects can also have nonstate attributes (e.g., a serial number). Individual objects have identity and are distinct things, and can be distinguished from other objects. In order to use any object requires the use of its identity. Java uses references to keep track of individual objects. Java references are variables declared using the class name or type of the object. It is possible to have more than one reference that refers to the same object. Messages are sent to an object by using its reference with the appropriate method name.

Once an object is no longer needed, it can be destroyed. Objects commonly go out of existence as a normal part of program execution. Perhaps the most common case of this is when temporary objects created by some method are no longer needed when the method is done and returns to its caller. Some programming languages (e.g., C++) allow for the explicit destruction of objects. However in Java, an object ceases to exist whenever it no longer has any references to it from other objects, at which point it may be garbage collected by the Java run-time system.

To get an idea of object lifetime, consider graphical user interface (GUI) classes such as the Swing library provided with Java. One type of object included in a GUI is a dialog interface. Upon some action by the user, say selecting a menu item, a given dialog object will be created. As part of the creation process, its constructor will be called. The constructor will set up the initial state of the dialog, which would likely include is size, the buttons and controls it has, and its position on the screen.

While the dialog object exists, it is able to respond to messages, and to send messages to other objects. For example, the dialog could respond to a message from the system that a particular button was clicked by sending another message to some other object in the program to take some action described by the button press.

When the user closes the dialog, the dialog object will no longer be needed. Once it no longer has any references to it, it can be garbage collected by the Java run-time.

### constructor
An operation that creates an object and defines its initial state. For complex objects, construction can be a significant activity, and cause the constructors of other objects to be invoked as well.

### garbage collection
The automatic detection and freeing of memory that is no longer in use. An object becomes available for garbage collection when it is no longer referred to by any other object. Java uses garbage collection rather than explicit destructors found in other OO languages such as C++.

### reference
A data element whose value is an address. In Java, all objects are accessed by reference. Any object variable will be a reference to an actual object, and not the object itself.

### identity
The characteristics or state of an object that allows it to be distinguished from other objects.

### instance
A specific object that is an instantiation of a class. An instance has specific attributes and behaviors, and a unique identity. Instance and object are often used interchangeably.

### instantiation
Creating an instance of an object of a given class. Instantiating an instance brings it into existence.

### object lifetime
The time an object exists - from its instantiation in its constructor until the object no longer exists and has been finalized by the Java garbage collector. The creation point of an object is under program control, but the exact moment when an object ceases to exist cannot be determined because of the way the Java garbage collector works.

---

### Basic UML Class Notation

The basic UML notation for a class is a rectangle with three horizontal parts. The top part is used to hold the name of the class. The middle part shows attributes, and the bottom is used to hold the class operations (methods). Depending on the level of detail needed, the middle attribute and bottom method parts may not be included.

Associations are shown by lines between classes, and are usually labeled with the name of the association.

Inheritance is shown by a line with a triangular arrow pointing to the more general class (the superclass).

Aggregation is shown by a line with a hollow diamond pointing to the whole class, while composition uses a solid diamond instead.

---

### Class Hierarchies

One of the most important aspects of creating object-oriented programs is the arrangement of classes into hierarchies. The simplest hierarchy is called an association. Two classes are associated by a named relationship. For example, consider a software system that tracks the books that readers check out from a library. Two classes present in this system could include a LibraryBook and a Reader. There is an association between LibraryBook and Reader that could be called either borrowing (readers borrow books from a library) or lending (a library lends books to readers).
Depending on what makes the association clearer, it can be labeled as a big-picture class-level association (e.g., borrowing as in Figure 2-1), or as a specific name for each class in the association (e.g., using borrowedBook and borrower by each class instead of borrowing). Figure 2-2 shows the alternate way to name the association.

**Figure 2-2. Alternate names for the association**

Classes in an association usually occupy equal places within a hierarchy. In our example, Readers and LibraryBooks are independent classes of equal standing. Associations are used to show the relationship between different, independent classes in the overall object-oriented design.

Associations also can have a multiplicity attribute. In the borrowing example, note the 0..* and 0..1 values right next to each class diagram. The 0..* by the LibraryBook diagram means that a Reader can borrow an unlimited number of books, from 0 up to an unspecified number. The 0..1 by the Reader diagram means that a given book will be borrowed by at most one reader. The multiplicity values can specify explicit values if needed (for example, 0..4 would mean a Reader could borrow at most 4 books). If a multiplicity is not specified, 1 is assumed.

Plain associations involve classes that are independent of each other. Hierarchies with classes that aren't independent are also an important part of OO systems. There are two ways commonly used to organize such class hierarchies.

The first is to include one class as a part of another. This is called a whole/part hierarchy, and is characterized by a has-a relationship. For example, a library is made up of a collection of books, which are themselves composed of pages, and so on. A library has some books, which have some pages. You can look at this as a part-of relationship. A page is a part of a book, which is part of a library.

The second kind of class relationship is the generalization/specialization hierarchy. Generalization/specialization (or gen/spec for short) is characterized by an is-a relationship. For example, if you were designing a class hierarchy to model animals, you might have a class for Dog, which is a specialization of the class Carnivore, which is a specialization of the class Mammal, and so on. Key to this concept is the fact that a Dog is a Carnivore, which is a Mammal, which is an Animal.3

The main mechanism for implementing a gen/spec hierarchy is called inheritance. With inheritance, a new subclass is derived from an existing parent superclass. The topmost class of the hierarchy is called the root class. Not only does the derived subclass inherit the properties of the superclass, but it can extend and modify its behaviors and attributes. Defining inheritance in code is not as simple as using a reference to an object, so all major OO programming languages, including Java, provide direct language support for generalization/specialization inheritance.

**Figure 2-1. A borrowing association**

Parts of a class can either be essential to its existence, or they can be parts that come and go. For example, a car has wheels, but would not be a car without the wheels. A library has books, but the books can come and go. A library without books is simply an empty library. A book without pages is not a book. Note that the borrowing association between a reader and a library book is independent of the whole/part relationship of a library and its books.

The common OO term for a whole/part hierarchy is aggregation. Objects that are in an aggregation association can come and go. If the object is an integral part of the whole, then the hierarchy is called composition. Most OO programming languages, including Java, haven’t defined special language support for whole/parts. Nevertheless, whole/part hierarchies are critical for most OO programs.

It is not difficult to define aggregation and composition in terms of existing programming language features. There are two ways to implement a whole/part relationship. In practice, the most common way to implement aggregation and composition is to include an instance of the aggregate object as an attribute of the containing class. For example, the definition of a book class could include a reference to the page class (or more likely, a list or vector of pages). Java allows also nested class declarations. Implementing aggregation and composition is discussed more fully in the Java chapter.

Plain associations involve classes that are independent of each other. Hierarchies with classes that aren’t independent are also an important part of OO systems. There are two ways commonly used to organize such class hierarchies.

**Figure 2-3. A Book Whole/Part Hierarchy**

**association** An association is a relationship between two or more classes. The association will indicate how objects of the different classes relate to each other.

**hierarchy** An ordering of classes. The most common OO hierarchies are inheritance and aggregation.

**multiplicity** An attribute that quantifies an association between objects. The multiplicity is specified as a range of the number of objects that can participate in the association, usually in the form n..m, where n is the lower limit and m the upper limit. A * means an unlimited number, while a single value can also be used. Common multiplicities include 1, 0..*, 1..*, and *.

Plain associations involve classes that are independent of each other. Hierarchies with classes that aren't independent are also an important part of OO systems. There are two ways commonly used to organize such class hierarchies.

The first is to include one class as a part of another. This is called a whole/part hierarchy, and is characterized by a has-a relationship. For example, a library is made up of a collection of books, which are themselves composed of pages, and so on. A library has some books, which have some pages. You can look at this as a part-of relationship. A page is a part of a book, which is part of a library.

The second kind of class relationship is the generalization/specialization hierarchy. Generalization/specialization (or gen/spec for short) is characterized by an is-a relationship. For example, if you were designing a class hierarchy to model animals, you might have a class for Dog, which is a specialization of the class Carnivore, which is a specialization of the class Mammal, and so on. Key to this concept is the fact that a Dog is a Carnivore, which is a Mammal, which is an Animal.

The main mechanism for implementing a gen/spec hierarchy is called inheritance. With inheritance, a new subclass is derived from an existing parent superclass. The topmost class of the hierarchy is called the root class. Not only does the derived subclass inherit the properties of the superclass, but it can extend and modify its behaviors and attributes. Defining inheritance in code is not as simple as using a reference to an object, so all major OO programming languages, including Java, provide direct language support for generalization/specialization inheritance.
Inheritance is an especially important and powerful concept. It means that an existing class can inherit from other existing classes. Consider the mammal hierarchy. All mammals share a number of common characteristics. These can be captured once in a generalized Mammal class. The general mammal characteristics are then available by inheritance to more specialized subclasses such as Carnivore or Rodent. The class Carnivore inherits all the general characteristics of a Mammal such as having hair and bearing live young which are fed milk, while extending the attributes to having certain kinds of teeth, and the behavior to eating meat. The Rodent subclass of Mammal would extend the Mammal superclass with different attributes than a Carnivore. The Mammal class itself could be derived from an even more general Animal class.

This really represents an economy of expression. We can describe the general characteristics in a superclass, while expressing the specializations in a subclass. We don’t need to repeat all the general characteristics for each instance of a mammal, just those specific to the subclass. And when the behaviors of different subclasses vary, such as the eating habits of specific mammals, these too can be specialized in a subclass.

In Java, a subclass can inherit from only one superclass. This is called single inheritance. Other OO programming languages, such as C++, allow classes to be defined that inherit from more than one superclass. This is called multiple inheritance. Compared to single inheritance, multiple inheritance is used infrequently, and can lead to some confusion in the program design. Java does not support multiple inheritance. Instead, Java supports what is called an interface, with an actual class definition supplied to implement the interface. This facility is often used in cases that would otherwise require multiple inheritance. In practical terms, implementing an interface allows a class to be used in well-defined ways by other classes.

Note that the is-a relationship is critical to proper design of a class. If a subclass cannot be defined with an is-a relationship to its superclass, then the design is likely faulty, and there is not an inheritance relationship. A Dog is a Mammal, but it is not a Rodent or a Color. You should always apply the is-a test when defining classes with inheritance hierarchies.

Many discussions of object orientation will give special emphasis to programming with inheritance, but in fact, both aggregation and inheritance are important parts of object-oriented programming. Both kinds of hierarchies are used to design programs that reflect the characteristics of the problem being modeled. Any class can be defined using a combination of both kinds of hierarchy.

- **default behaviors**: In an inheritance hierarchy, the class behaviors defined by superclasses that will be used by default unless they are overridden by some subclass.
- **derived**: In an inheritance hierarchy, a subclass is derived from a superclass. The derived subclass inherits the attributes and methods of the parent superclass.
- **generalization/specialization**: An inheritance hierarchy. Each subclass is a specialization of a more generalized superclass.
- **implements**: In Java, a specification that the class will implement the code required by an interface.
- **inheritance**: A mechanism that allows one class (subclass) to share the attributes and behaviors of another class (superclass). Inheritance defines an is-a relationship between classes. The subclass or derived class inherits the attributes and behaviors of the superclass, and will usually extend or modify those attributes and behaviors.
- **interface**: In Java, an interface is a specification of methods a class using the interface must implement. An interface is a specification, and does not define any code. It provides an alternative to multiple inheritance.
- **is-a**: A term used in inheritance hierarchies. In general, a subclass is-a specialized kind of a more general superclass.
- **is-a test**: A simple test to check for proper inheritance design. If you cannot say a subclass is-a kind of the superclass, then inheritance is probably not appropriate.
- **multiple inheritance**: When a subclass is derived from multiple superclasses, it is said to have single inheritance. Java does not allow multiple inheritance, but provides interfaces as an alternative.
- **overriding**: When a subclass specifies an alternative definition for an attribute or method of its superclass, it is overriding the definition in the superclass. Also called overloading. Java can only overload methods.
- **root class**: The top most or most generalized user class of an inheritance hierarchy. In Java, all classes are at least implicitly derived from the Java Object class, which make it the most

---

An animal Generalization/Specialization Hierarchy

Note that the UML uses an open arrowhead pointing to the more general class to show inheritance. Thus, in Figure 2-4 of the animal hierarchy, a Mammal inherits from Animal, and so on.

**UML**: You can show more or less detail in UML diagrams depending which information you want to concentrate on. For example, since Figure 2-4 is mainly focused on the inheritance relationships, and it doesn't show the attribute or operation boxes. We will use this same diagram in Figure 2-5, but show more detail, including some selected operations.

A superclass is extended without altering its definition or source code. A subclass can be selective about which properties of the superclass it inherits. The subclass can extend the superclass by adding new properties, and by selectively overwriting existing properties of the superclass.

Inheritance is an especially important and powerful concept. It means that an existing class can be used as-is by a new class, with its properties modified and extended through the inheritance mechanism. Classes can be designed to provide useful **default behaviors** and attributes that can be extended and modified only if the derived subclasses needs to.

Consider the mammal hierarchy. All mammals share a number of common characteristics. These can be captured once in a generalized Mammal class. The general mammal characteristics are then available by inheritance to more specialized subclasses such as Carnivore or Rodent. The class Carnivore inherits all the general characteristics of a Mammal such as having hair and bearing live young which are fed milk, while extending the attributes to having certain kinds of teeth, and the behavior to eating meat. The Rodent subclass of Mammal would extend the Mammal superclass with different attributes than a Carnivore. The Mammal class itself could be derived from an even more general Animal class.

This really represents an economy of expression. We can describe the general characteristics in a superclass, while expressing the specializations in a subclass. We don’t need to repeat all the general characteristics for each instance of a mammal, just those specific to the subclass. And when the behaviors of different subclasses vary, such as the eating habits of specific mammals,
Polymorphism

For a given class hierarchy, it is possible for different subclasses to be derived from a common superclass. Each of the subclasses can override and extend the default properties of the superclass differently. Polymorphism is a characteristic of inheritance that ensures that instances of such subclasses behave correctly.

When a subclass overrides a default method, it uses the same name defined in the superclass. If the behavior of the default method is adequate, a given subclass does not need to override the method, even if other subclasses do. The derived method can implement completely new behavior, or use the default method while extending it with additional behaviors.

Figure 2-5.

Polymorphism means a Cat uses its own eatMeal, a Dog uses the Carnivore eatMeal, and a Rodent the Mammal eatMeal.

The figure shows an Animal hierarchy. Since all mammals need to eat, there would likely be a general method defined by the Mammal class to handle eating, called eatMeal, for example. While all Mammal objects might share some eating behaviors that are defined in the general Mammal eatMeal method, some would require some specialized eating behaviors that are different than other Mammal objects. The eating habits of a Dog are different than a Cat, which are different than a Rodent, which are different than a Primate. Thus, each subclass definition can include an eatMeal method that implements the specialized eating for that subclass. Not all subclasses need implement a specialized method if the superclass method is satisfactory. In figure, the Dog uses the more general Carnivore eatMeal, while the Cat has its own specialized eatMeal. The Rodent uses the general Mammal eatMeal. Since all animals may not need an eatMeal, there is no eatMeal method defined by the Animal class. Note that all these methods have the same name, eatMeal, even though they implement different behaviors.

If the system were to process a mixed list of different Mammal objects, then it would need to use the appropriate eatMeal method for each Mammal. For example, if the Mammal instance were a Dog, then the eatMeal method defined for the class Carnivore would need to be used, and not the eatMeal for a Rodent.

Polymorphism is what allows the appropriate method for any given object to be used automatically. Polymorphism goes hand in hand with inheritance and classes derived from a common superclass. The mechanism that allows polymorphism to work is dynamic binding. The actual binding of a call to a specific method is delayed until runtime. Only then can the class a particular object instance belongs to be determined, and the correct method from that class then called.

Polymorphism almost seems like magic. It can be difficult to really believe that the proper eatMeal will be used for each object. Fortunately, polymorphism is easier to use than it is to understand completely, and you won’t have to think about it explicitly most of the time. Using it comes automatically, and seems a natural part of using objects with inheritance.

An Example - Putting it All Together

In this section, we will present a small example design that uses all the major OO relationships: association, aggregation, composition, and inheritance. This small example could be considered a starting point for designing a full application for a library system. This example will show the relationships between a library customer and the library. Figure 2-6 shows the UML for this design.
This example has two generalized superclasses, Book and Person. Each of these describes generic objects. Presumably, each would contain attributes common to all Books and all Persons, such as a name. Note that a Book is composed of Pages. The composition relationship shown indicates that a Book will have from one to any number of pages. Any methods a Book requires to manipulate Pages would be implemented in the Book class, and would be available for subclasses of Book. A LibraryBook is derived from Book, and is a specialized kind of Book. It might have additional attributes such as an acquisition date, borrowing status, and an identification number.

A Reader is a special case of a Person who uses the Library. A Reader object would contain specialized attributes such as a list of checked out books.

The Borrowing class in this example is used to implement the same borrowing association between a Reader and a Library book as we used in Figure 2-1. Now, however, a Borrowing is also a part of a library - a list of all books borrowed. We could use the same simple direct association as we used before, but then the library would still need some other object to track books that have been borrowed. In addition, the Borrowing class can now have added responsibilities of tracking just when a Reader borrows a LibraryBook, for example, and or could track overdue books. The dashed line from Borrowing is the UML notation to show this relationship. In our earlier example, the borrowing association could be implemented as a simple double link between a LibraryBook and a Reader. Using a class to implement this association means the links between the LibraryBook and the Reader will be managed by the Borrowing class. Note the 0..4 multiplicity for a Reader borrowing LibraryBooks. This would indicate that a reader could have from 0 to 4 books borrowed.

Figure 2-6 shows an aggregation relationship between a Library and LibraryBooks. One might think that a library is not a library without any books, and that the relationship should be a composition. Perhaps. But one of the differences between aggregation and composition is that when an object is destroyed, so are all of its components, while the parts of an aggregation remain. So, if you destroy a Book, the Pages are gone, too. On the other hand, if you close a Library, all the Books will still exist.

And finally, note that both LibraryBook and Reader pass the is-a test for inheritance. A LibraryBook is a Book, and a Reader is a Person.

Figure 2-6. Relationships between a Library and its customers

Other OO Concepts

The six basic principles we've just discussed: abstraction with objects, encapsulated classes, communication via messages, object lifetime, class hierarchies, and polymorphism represent the pure essence of object orientation. While these six principles are the core of object orientation, there are other important concepts that are essential parts of real object-oriented programming languages and designs. This section covers some of these other concepts.

AbstractClasses

When building a class hierarchy, it is common to design some classes that will never have any instances, and are intended to be used only by subclasses. For example, the Animal class in the animal hierarchy is such a class. There will never be an instance of an Animal. Instead, there would be more specialized subclasses of Animal such as Horse or Snake that would have instances. Classes that have no instances are called abstract classes. Abstract classes usually define a common interface for subclasses by specifying methods that all subclasses must override and define. For example, the Animal class might define a method called reproduce. Since all animals reproduce, each subclass would have to define a reproduce method. But the definition in reproduce for the Animal class would be empty since it is an abstract class, and serves as a guideline or specification for derived subclasses.

A concrete class, on the other hand, is one that can have actual objects or instances. Dog and Cat are examples of concrete classes because there will be instances of Dog or Cat. In Java, interface definitions can be considered as a type of abstract class that has only methods and no attributes.

A related concept is the root class we discussed earlier. In any hierarchy, the root class is at the top of the hierarchy, and does not have a superclass. The root class in the hierarchy we've been using is Animal, but the root class could even be more general (for example, Life) if necessary. A root class may or may not be an abstract class. It is more common that an abstract class will also be a root class, although it isn't required.

An OO system can have many root classes for different object hierarchies. This can be visualized (Figure 2-7) as a forest of class trees. In some OO programming languages, including Java, all classes are actually derived from a single system defined root class. There are various advantages as well as disadvantages to this requirement. The master root class in Java is the Object class. Although all Java classes are derived from the Object class, this is most commonly done implicitly; i.e., you don't have to explicitly include the Object class in your class definitions. The class forest in the figure does not include a topmost level root class.

Abstract class A class that has no instances. It is usually defined with the assumption that concrete subclasses will be derived from it, and extend its basic attributes and behavior. In Java, an abstract class is one that includes an abstract method.

Concrete class A class that is completely specified and can have instances. A Java class derived from and abstract class will define all the abstract methods from the abstract class.
Visibility of Methods

Recall that encapsulation is one of the most important characteristics of object-oriented programming. One aspect of encapsulation is the hiding of all the implementation details of an object inside the class definition, while presenting a well-defined interface to the outside world via the class’s methods. Object-oriented programming languages such as Java provide direct language support to control the visibility of a class’s attributes and methods.

visibility The ability of one class to see and use the resources of another class. Visibility is controlled by various programming language features such as Java’s public, protected, and private specifiers.

Object-oriented languages usually provide four levels of visibility for classes. These levels of visibility are usually directly supported by language keywords. Normally, the levels of visibility will apply to both the attributes and operations of the class. The four levels are:

1. **Public Visibility.** Public attributes and operations are visible to the whole world. Any other class can access the public items of a class.
2. **Private Visibility.** Private attributes and operations are visible only to members of the given class.
3. **Protected Visibility.** Protected attributes and operations are visible to the class and its subclasses.
4. **Friend Visibility.** Friend attributes and operations are visible to a specified set of other classes. In Java, the package is used to define friend visibility. In C++, the friend specifier is used.

Encapsulation is enforced by limiting the number of public items. Normally, attributes are never defined to be public. Instead, a class should provide only public operations, which can provide services, and allow other classes to interact with its attributes via a well-defined public interface.

Fig. 2-7. Some relationships shown by this “class forest” include: A and P are root classes. P is a superclass of Q, R, T, and U. A, C, and E are all subclasses of F and G. B is a subclass of A, and D is a subclass of B. B and C are both derived from the common superclass A, using single inheritance. U is derived from R and implements the interface S. D, F, G, T, U, and V will certainly be concrete classes since they are at the bottom of the hierarchy. Higher level classes could be either abstract or concrete. Note that in Java, A and P would be derived (probably implicitly) from the Java Object class.

When a class needs to provide information about its state, the general convention is to use what are known as setters and getters, also known as accessors and mutators. In general, however, it is best to minimize the number of setters and getters defined by a class. Classes that are more data oriented (such as a Color or Coordinate class) will be more likely to require setters and getters. Other classes should modify their internal attributes as a result of the operations they perform, and not by direct requests.

**setter** A method that allows the outside world to modify an attribute of a class. Setter methods are also known as mutators. Setter methods by convention have names such as setLimit or setWidth.

**getter** A method that returns the value of a class attribute. Getters are also known as accessors or selectors. By convention, getter methods have names such as getLimit or getWidth.

Class vs. Instance

When a class is defined, it can have two different kinds of attributes and methods. The difference between these two is whether they apply to the class as a whole, or if they relate to specific instances of the class. **Class attributes and class methods** apply to the class as a whole. For example, it might be useful to know how many instances of a given class have been created. In that case, a class attribute called instances could be defined to track this information for the class as a whole.

**Instance attributes** and **instance methods**, on the other hand, relate to specific instances of a class. Commonly, any instance of a class will need its own copies of attributes, and methods that use those instance-specific attributes.

**instance attribute** Attributes of a class that are shared by all instances of the class. There will be only one copy of each class attribute, and it is possible to access these class attributes without creating any instances of the class. These are sometimes called static attributes in Java.

**instance method** Methods defined by a class that operate on instance attributes. This is the most common type of method defined by a class, and an instance method will be used only with its associated instance of the class.

**class method** A method defined by a class that only operates on class attributes. Class methods can be used without creating any instances of the class. These are sometimes called static methods in Java.

**class attribute** Attributes of a class that are shared by all instances of the class. There will be only one copy of each class attribute, and it is possible to access these attributes without creating any instances of the class. These are sometimes called static attributes in Java.

Accessing Objects

Object-oriented languages provide the basic mechanisms needed to access the various parts of an object. But just as convention calls for setters and getters, there are other special cases for accessing the attributes and methods of an object that have their own terminology. The following definitions cover the basic terms used to describe different kinds of object access.

**container** A class whose instances are collections of other objects. These collections may be objects all of the same type, or of mixed types, although they usually have a common superclass. Containers include lists, stacks, queues, bags, and others. They usually provide a method to iterate over each object in the container.

**iterator** An iterator is a method (or methods) used to access or visit each part of an object. This allows the outside world controlled access to all important parts of an object without the need to know the internal implementation details of a specific object. Iterators are often used...
with container classes, and typically work by accessing the first item in a container, and then each subsequent object until all objects have been accessed.

**mix-in** A class (or usually an interface in Java) that is used to define a single behavior. Mix-ins are usually not standalone classes, but are used to provide a standard for implementing the designed behavior.

callback A method that is called when an event has taken place. Usually used in association with a listener. When a listener detects an event, it will invoke the callback of objects that need to know that the event has occurred.

**listener** A method that responds to events. These are usually system events such as mouse clicks or timer events. The listener will typically invoke callbacks of objects that need to respond to the event.

**link** A reference to another class. Used to build associations between classes.

**this** Also called self. A reference to the current object. Within a class definition, references to the attributes and methods of the class are implicit. The this reference can be used for clarity to make a reference explicit. Most commonly, however, this is used to pass a reference to the current instance to another object. It can also be used to set a class variable to refer to specific instance of the class.

### A Low-Level View of Objects

At some point while a programming is running, class instances must have some low-level implementation in the memory of a computer. Having an understanding of how objects can be implemented at this low-level can help you understand just what is going on.

The UML representation of an object with the separate attribute and operation fields is in fact a reasonable model of how objects are represented in memory. There are really two main components of any object: the data or attributes, and the code that implements the methods. We will discuss the code part first.

A class definition includes the code for all methods of a class. The compiler will translate the high level Java code into low-level Java Virtual Machine code. This is the code that runs on each computer. Because every instance of a class will use the same methods, there really needs to be only one copy of each method defined by a class. There does not need to be a separate copy for each object instance. This is true for both instance and class methods and attributes. So, for each class, there will be a single copy of all the code for the methods, as well as the storage for any class attributes.

On the other hand, the instance data associated with each object will be unique to each instance of a class. Each object exists as a separate entity with its own identity. Thus, when an object is first created with a new operator, Java will dynamically create the storage space required for the instance attributes. Because this dynamic storage must be initialized, most object-oriented languages, including Java, will invoke a special class method called the constructor whenever an object is first created.

Java will create new objects only with by an explicit use of the new operator. It never automatically creates new objects. Since only class methods (static methods) can be invoked without creating an instance of a class, your program's main entry point is the class method static void main.

Java will automatically create temporary storage for primitive items like ints as well as object references. This storage is created to hold the parameters passed when a method is called.

Most implementations of Java will use four different areas of memory to store these various items. First, there will be two areas of memory to hold the method code and the class or static data. This memory is allocated only once when a program or object is first loaded. The dynamic object data and parameter storage is usually implemented using two dynamic areas of memory. The first is called the stack, and is used to hold the temporary copies of parameters and copies of variables local to a method. When a method is called, enough stack is used to hold the parameters and local method variables needed for that call. This is done on a last in, first out basis, so that when a method returns to its caller, the temporary parameter values and method locals are simply removed, or popped, from the stack. The memory required for instance variables when an object is first created is allocated to a dynamic area of memory called the heap. Unlike parameters and locals, instance variables are needed as long as an object still exists. The heap usually consists of most of the free memory available.

When the memory for an object is created, it will consist of the space required to hold all the attributes of the object. It will also contain a reference to the methods defined by the class. It is this dynamic allocation of method references that allow polymorphism to be implemented efficiently. Depending on the actual type of the object, the proper method will be bound to the object at run time.

Just as objects have memory allocated for them when they are created, it is necessary to free that memory when object are no longer needed. Various object-oriented languages use different techniques to control the freeing of this unused storage. We will discuss Java's technique, garbage collection, more in the next chapter.

What does all this have to do with object-oriented programming? It is part of the nature of objects that they each have their own existence, their own identity, and their own lifetime. As a consequence, each object has a representation in memory. The difference between instance and class attributes and methods affects just how a particular object will work. The fact that an object must be created and initialized, and that objects can go out of existence, means you must be aware of these consequences when you program.

### Chapter Summary

- Object Orientation is a significantly different way of thinking about solving problems and developing software solutions.
- An Object-Oriented System is one that has been designed using abstraction with objects, encapsulated classes, communication via messages, object lifetime, class hierarchies, and polymorphism.
- An object represents an instance of a real or abstract thing.
- A class is the definition of the attributes and methods needed to model a related group of objects.
- Classes can be organized into hierarchies. Association is a relationship between classes.
- Aggregation and composition represent whole/part relationships. Inheritance represents generalization/specialization.
- Inheritance allows subclasses to selectively derive the properties of a superclass.
- Polymorphism goes hand in hand with inheritance, and means the right methods are used for individual objects in a derived class.
- An understanding of how objects are created and implemented can make it easier to write good OO programs.
- Just as any other specialized discipline, object-orientation has its own vocabulary.

### Resources

**Object-Orientation**
You can find many different definitions of what makes a system object-oriented, but you would have trouble finding two
OO developers who would agree to any one definition. We will use this definition of object-orientation.

Aggregation and composition are similar. We will cover the differences later.

These may not be exactly biologically correct, but they serve our purposes for this example.

Note the names Animal and Mammal are shown in italics in the UML diagram. UML uses italics to designate abstract
classes.

The root system class is not usually shown in UML diagrams.