Chapter 2
Object-Oriented Concepts

Software engineering trends before the mid-1970s followed one basic programming methodology, known as structured programming. The **structured paradigm**, also called the **classical paradigm**, offered a very straight-forward approach to software engineering which seemed sufficient for the time. This model provided a simple view of a software product: the product was either operation oriented or data oriented (Schach 2008). In the first case, primary attention was paid to the functions performed. In the second, the primary focus was on the information being handled. The picture painted by the classical paradigm seemed uncomplicated in the beginning, but as the field of software engineering grew, a fundamental flaw became apparent. Describing a software application based exclusively on either the operations it was to perform or the data it was to manage was just too simplistic.

As solutions to the problem of oversimplification were being investigated, brand new programming concepts began to emerge. The most influential of these, and ultimately the most successful, is known today as the **object-oriented paradigm**. In object-oriented programming, software is not broken down into either operations or data, but rather into abstract software artifacts, called objects. These objects are designed both to manage data and to carry out operations, thus combining the two fundamentally related halves of a software product, which had in the past been kept separate under the classical paradigm.

To better explain this concept, we turn to a real world example: a human being. In the classical model, specific actions and physical characteristics were kept separate from one and other. For example a person's name, eyes, birth date, hands, and taste in music would be separated from the actions he or she is able to perform like, holding a conversation, winking at a friend, having a birthday party, listening to a CD, or shaking an acquaintance's hand. If we apply the object-oriented paradigm however, all of these attributes and actions are combined to make up a single unified object: a human being. This object is able to use its attributes, which are stored internally, to perform actions, and thus can be thought of as a well-defined, independent entity rather than an over encumbering conglomeration of functions and data.

R. Y. Lee, *Software Engineering: A Hands-On Approach*,
DOI: 10.2991/978-94-6239-006-5_2, © Atlantis Press and the author 2013

The advent of the object-oriented paradigm had a profound change on the field of software engineering. The use of objects allows software engineers to create models of the real world in ways that had been previously thought impossible. After all, software seeks to facilitate real-world situations, many of which cannot rightly be broken down into a simple sequence of instructions to be carried out in line-by-line order. Consider a local bank that, like any bank, stores money for some number of clients. The classical model would have led to the creation of a program consisting of many lines of code that dictate the exact order of operations to be performed. Code created in this manner could never be very well organized. This is not due to a fault of the programmer, however, it is the classical paradigm that provided no system for breaking the program down into logical pieces. This created a need for the objected-oriented paradigm, which allows us to model the bank in a way that is logical for our system, capturing and focusing on the important details, while leaving out those that are less important. This concept, known as abstraction, allows us to create various objects that each represent various portions of the bank, such as clients, bank accounts, employees, and money transactions (Jia 2003). In the end, the object-oriented method allows us to design and implement a software system more intuitively, compartmentalized, and manageable than previously possible using the classical method. In this chapter, we will discuss the specifics that make this possible.

2.1 What is an Object?

In the real world, an object can be anything at all from a pencil to a monster truck. If it has a name, certain characteristics, and certain actions that can be done to it or be accomplished with it, then simply put it is an object. Objects characteristics can take on many forms, forms as simple as color to as complex as molecular structure. Likewise, an **object** in the world of computer programming is an entity with attributes that belong to and describe it. These attributes can be actions the object is capable of performing, an interface to access those attributes and actions, and most importantly, a unique identity. An object is a specific instance of a **class**, which can be thought of as a blueprint for that object (classes will be discussed further in the next section). Because a single class can be used to create many objects, an object's unique identity is crucial in distinguishing it from other objects. Think of the monster truck that we mentioned earlier. Just like our software object comes from some defining class, the `monster truck` object might be an instance of the class `truck`. We use the identifier "monster truck" to ensure that we can distinguish it from other instances of the `truck` class, such as `fire truck`, `pickup truck`, or `dump truck`. An object inherits most of its attributes from its class, but these attributes may differ amongst the object instances of that class. An object is described as being in a certain **state** at any given moment, which is defined by the value of its attributes. The state of an object plays a critical

role in the functionality of that object. Many of the actions that are performed by or to an object are based on the objects current state.

Programs are written so that a change in the state in an object will in some way affect the program as a whole. For example, the constant monitoring of certain object states is the driving force behind event-driven programming styles, which create programs that react to the occurrence of certain events (specified changes in object states). In this manifestation of object-oriented programming, a program might contain a user interface (UI) that listens for the user to perform a predefined action. For instance, the UI might contain a button that the user can click. When the user clicks said button, the state of some related object changes to reflect that the button has been clicked. This state change then triggers some associated reaction in the program. In this way, the operation of, and progression through, the program is dictated by the states of the objects in it.

We have said that objects are able to both change states and react to state changes. Of course, however, an object cannot simply 'know' how to change states, or how to react to another object's state change. Rather, objects are coded with **methods** that perform tasks when called upon to do so. Think again of the `monster truck` from before. `Monster truck` contains a method, which is derived from the `truck` class (a concept called inheritance, which will be discussed later on the chapter), that turns the engine on. This method might be called `start`, and it reacts to two specific events: the presence of a key in the ignition, and the turning of that key to the right position. Essentially then, the `monster truck` object reacts to certain state changes (placing the key in the ignition and turning it) by calling the method `start`, which turns the vehicle on. This method was implemented in the object by an engineer, much in the way that a software engineer implements methods in a software object.

The methods that are included in an object can vary greatly, as different objects are created for different purposes. There are two methods, however, that are common to most objects: constructors and destructors. Construction is the action of creating an object from its respective class, and initializing its attributes with either assigned values or those that are given to it by default. Object construction physically places that object into the computer's memory. An object's constructor is the actual method that handles the creation and initialization of the object. Likewise, destruction is the action of erasing an object, and thus freeing up the memory in which it resided. The destructor method of an object performs this action. Destruction of an object is used to provide room in the system memory, potentially for the creation of other objects. It also prevents faulty code from creating memory leaks. Constructors and destructors will be discussed in greater detail in Sect. 2.3.4.

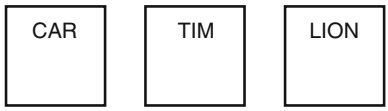

Fig. 2.1 Real world objects

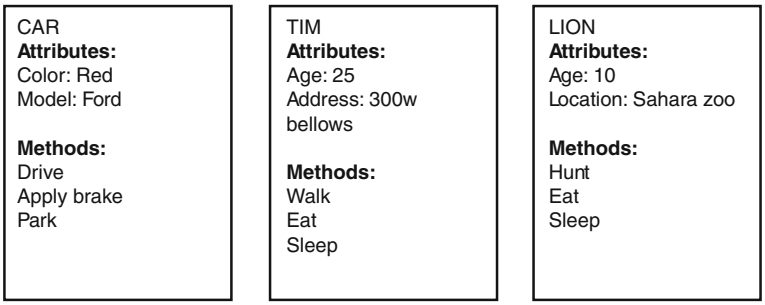

Fig. 2.2 Real world objects with attributes and methods

In the next section we will discuss classes in further detail; but first, Figs. 2.1 and 2.2 provide examples of real world objects and their associated attributes and methods.

2.2 Classes

The class-object relationship is essential to the object-oriented paradigm. The two are intrinsically linked, and, truly, one cannot be discussed without mention of the other. They are not, however, the same. This section defines classes and describes the relationship and differences between classes and objects. It also explains the distinct roles that the two concepts play in object-oriented programming.

2.2.1 Classes Versus Objects

Objects are instances of classes. It seems natural, then, to say that classes can be thought of as parents to objects. This logic, however, misses a fundamental point in the object-oriented paradigm. To say that classes are related to objects as parents are related to children suggests that classes and objects are the same type of entity, in the way that a parent and his or her child are both human beings. This is not accurate. Rather, using the same example, a class can be thought of as the DNA of an object. DNA is not itself a human being, but the description of a human being's attributes. Both parents and children, on the other hand, can be thought of as objects created according to their DNA. In short, a class is the concept behind an object, or the "essence" of an object, while an object itself is a tangible entity with a place in space and time (or system memory) (Booch 1994).

A second, related definition of a class is similar to the more common definition of the word in everyday speech. In standard use, a class is a "group, set, or kind sharing common attributes" (Merriam-Webster 2009). In object-oriented programming, the usage is the same, but more specifically describes a group of objects

with common attributes. This definition is a direct result of the previous definition: objects created using a class will be distinct from each other, but will share the characteristics given to them by that class, and will therefore constitute a group of similar objects. These characteristics include not only the objects' attributes, but also their methods and interfaces, all as defined by the instantiating class.

2.2.2 The Class-Object Hierarchy

We often describe classes and objects in terms of real-world concepts. Fortunately, though, some of the controversies of the real-world do not carry over into the field of software engineering. There is no chicken-egg controversy in object-oriented programming. The class comes first. As we have said, classes are essentially the blueprints or templates behind objects. In our definition from Sect. 2.1, the `monster truck` object could not exist without the `truck` class that it is derived from. An object is an **instance** of a certain class, and for that reason a class must be defined before the object can even be conceived. Once a class has been defined, an object of that type can be created from it, using the constructor method specified in that class. This action of creating an object from a class is **instantiation**, and the class used to create the object is referred to as the **instantiating** class. A single class may be used to instantiate any number of objects. All of these objects are then referred to as **members** of the instantiating class.

2.2.3 Why Use Objects and Classes?

We have said a good deal about what objects and classes are, how they related, and what they are capable of doing. However, object-oriented programming only represents one line of thought in the world of computer programming. Why should we use this paradigm? The next sections will describe the key features and advantages of the object-oriented paradigm as afforded by the use of objects and classes.

2.3 Modularity

The move from the classical structured programming model to the modern object-oriented paradigm represented a fundamental shift in the practice of software engineering. One portion of this shift dealt with the conceptualization, organization, and management of a software. If a large software engineering project is attacked as a single program to be written, the resulting code will undoubtedly be cumbersome, arduous to navigate, and extremely difficult to debug. As a solution

to this, the use of modularity, borrowed from other engineering fields, worked its way into software engineering. Modular programming focuses on the partitioning of a program into smaller modules in order to reduce its overall complexity. The resulting modules constitute definite and intuitive boundaries within the program, and facilitate the use of multiple software engineering teams, each of which can focus on an individual module. The layout of these modules constitutes the physical architecture of a software system.

Booch describes modularity as "the property of a system that has been decomposed into a set of cohesive and loosely coupled modules" (Booch 1994). This not only covers the advantage of workable units, but also touches on the goal of reducing dependencies among different portions of the program. Such dependencies can make the modification of a program a tremendously large task, as editing a single class will affect all portions of the program which were dependent on that class. Those changes will have to be accounted for throughout the entire system, often at a very high cost. Modularity offers a potential solution to this problem through the isolation of individual program portions from each other. The way in which the objects and classes of a module are accessed can then be easily defined through the implementation of an interface for interaction with other parts of the program.

The theories of encapsulation and information hiding will be discussed in further detail later in this chapter, but it is important to understand that, if implemented properly, they can ensure the integrity and dependability of data across a software system.

The most important benefit of modularization is the efficiency and workability afforded by the separation of different programming concerns into manageably sized, logical modules. The result is a program that is more flexible and much easier to understand, change, and debug. Additionally, because modules are constructed independently, they can be easily reused in other applications or stored for later reference and modification. Modularity also calls for the separate compilation of the modules, which facilitates an incremental development process, and allows for easier unit testing. The following sections will discuss in more detail a few of the specific benefits of modularity.

2.3.1 Reuse

The modularization of a software product results in the creation of any number of independent components, each with a distinct function. One advantage to this approach comes from the potential use of these components in future projects. This practice, known as **reuse**, can save developers from consuming resources in order to remake something that they have developed in the past. For this reason, reuse is a common practice in the field of software engineering, and one which needs not be limited only to software components. Methods of organization, planning

procedures and test data may all be reused, as can any other portion of a software engineering project that might serve some function in the development of a different project.

Schach draws the distinction between two types of reuse. The first, opportunistic, or accidental reuse, occurs when a component that was not originally developed with the mindset for future implementation, is discovered to be appropriate for use in a new project. Opportunistic reuse, however, runs an increased risk over the second type, deliberate reuse, which refers to making use of a component that was developed with reusability in mind. In the case of deliberate reuse, attention is paid to ensure that the component in question will have no unintended side effects or disabling dependencies which might have negative consequences for future projects. Opportunistic reuse, on the other hand, has a greater potential for such mistakes, as avoiding them was not a priority during development (Schach 2008). When a software component is developed with reusability in mind, rigorous testing is performed and thorough documentation is compiled to guide software engineers in future implementations of the component in question. This extra attention, however, results in an increased cost for the development of that component.

For the reasons described above, software engineering firms must weigh the potential benefits of reuse against the cost of development when considering the components involved in a software engineering project. A piece of software is developed for a distinct reason, and thus, usually neither the application as a whole nor all of its constituent parts are determined to be viable candidates for reuse. Even pieces of software developed for the most specific purposes will generally make use of very common routines. For this reason, extensive subroutine libraries have been developed for many programming languages. Modules containing common routines, such as math procedures, accounting functions or graphical interface components, are examples of widely used reusable components. Because the development of many such modules took place in the early development of these languages, they are often taken for granted; but in truth, they are the quintessential examples of reusable development, without which software engineers would have to develop each and every portion of a software engineering project from scratch. The myriad benefits of reuse have encouraged many companies to both consider reusability in the development of new software engineering components and to take into account the potential for making use of previously developed components when designing and implementing a new software engineering product.

2.3.2 Encapsulation and Information Hiding

As we have said, a modularized software product is essentially a collection of independent modules which serve distinct purposes and are configured to interact with each other. In addition to lending itself to logical project decomposition and

component reuse, this use of the object-oriented paradigm can also provide integrity for a software system. To explain this, let us consider again the `monster truck` and its `start` method that we described in the beginning of the chapter. The `start` method uses the ignition and a key in order to turn the vehicle on. In fact, the insertion and rotation of the key are the only actions needed to start this very complex mechanical system. In software engineering terms, this is referred to as the use of an **interface**. Essentially, an interface provides a defined way for using an object, and for accessing those properties of an object that are intended to be accessed. The processes involved in turning on a vehicle are extremely complex, and involve everything from the use of a proper fuel mixture to the completion of certain electrical circuits to the correctly timed firing of pistons. For the average operator of the vehicle, such in depth knowledge of the vehicle's operation is unnecessary. In fact, were the operator required to manually specify all of the settings and actions required to turn the vehicle on, we can safely assume that our `monster truck` would never leave the sales lot. Luckily for us, these intricacies are hidden from the user, and instead the `monster truck` is engineered to perform these actions on its own, in response to an appropriate interaction with a predetermined interface, in this case the vehicle's ignition.

This practice of hiding a system's inner workings is known as **information hiding**. Information hiding is a key concept in the larger process at work here: **encapsulation**. Booch describes encapsulation as the "process of compartmentalizing the elements of an abstraction that constitute its structure and behavior" (Booch 1994). By this, we mean that objects should be designed to separate their internal composition and function from their external appearance and purpose. As previously stated, this provides integrity in a software system. That integrity results from the closing off of an object's internal workings so that the object in question is only accessed in the desired manner, and unintended changes to the object cannot be made by the program. Furthermore, encapsulation ensures the independence of an object from the rest of a software system. The internalization of an object's attributes and methods means that if a change is made to one part of the system, the object maintains its integrity and functionality, and is still accessed and will still respond in the intended manner. This localization of data facilitates not only changes implemented during the original development process, but also throughout the ongoing maintenance that the software will undergo during its serviceable lifespan.

2.3.3 Access Levels

We have said that the use of modularity and encapsulation can provide integrity within a software system through the use of information hiding. One key practice that leads to this advantage is the designation of access levels within classes, and

thus within the objects derived from them. Levels of access are assigned in order to specify how classes, as well as their attributes and methods, can be used by other objects during execution. While cooperation among objects is desirable, these interactions must be controlled in order to ensure stability. The three standard access levels are described below:

Public: the class in question, and instances of it, can be accessed by any other member simply by a call to the class or derived object name.
Protected: can be accessed by the class itself, and by all subclasses that are derived from it.
Private: can only be accessed by the class itself. Thus, only methods that are part of the class are allowed access.

Access levels can be applied to individual attributes or methods within a class, and form the basis of an interface. Those characteristics of a class which we earlier described as the class's "inner workings" are designated as restricted. This might encompass nearly all of a class's attributes and methods. Those characteristics which are left open for access by other parts of the software system constitute the interface, and generally will communicate with the inner workings of the class in a predefined way in order to return information, perform an action, or modify the object in question.

2.3.4 Delegation

In the next section, we will discuss a practice in the object-oriented paradigm called inheritance, which aims to provide a class with the properties of a different class. Here we briefly discuss an alternative method to this practice, known as **delegation**. Delegation provides opportunity for code reuse not by directly inheriting the attributes of some class, but rather by simply using the methods of another class to accomplish the desired result (Bruegge and Dutoit 2004). While inheritance is generally considered a static method of implementing attributes, delegation allows the dynamic use of only those desired attributes at a specified time.

2.4 Inheritance

Object-oriented software engineering aims to efficiently produce reliable software by reducing redundancy and ensuring integrity within a software system. The class-object structure facilitates this end by providing an intuitive system of modularization which easily lends itself to reusability. Central to both of these

principles is a feature of object-oriented programming languages known as **inheritance**. Inheritance is a relationship between different classes in which one class shares attributes of one or more different classes. In this way, the class in question, the **subclass**, *inherits* the qualities of other classes that have already been created, the **superclasses**. There are two general cases of inheritance, which are defined by the number of superclasses, or **parent classes**, from which the subclass, or **child class**, directly inherits attributes: **single inheritance** describes a relationship in which a class has only one ancestor from which it directly inherits its attributes. **Multiple inheritance**, on the other hand, occurs when a class calls on more than one superclass for properties. The rest of this section will be devoted to discussing the various principles behind the use of inheritance.

For starters, inheritance offers an obvious solution for the elimination of redundancy through the implementation of reuse. The creation of a new class from some other class with a similar purpose and set of attributes saves the software engineer from rewriting code that is already in use elsewhere. This implementation of inheritance is often used to facilitate the creation of some number of differing classes that share common attributes and a common purpose. Consider, again, the `monster truck` example previously discussed. We described this as being an instance of the `truck` class, along with a few other potential versions of that class: `fire truck` and `dump truck`. In this example, each of our three subclasses of `truck` can be thought of as more specific versions of the superclass. The superclass, `truck`, can then be thought of as a generalization to be used in the formation of those subclasses. This type of class is called an **abstract class** and is not meant to ever be instantiated (that is, no object will be created from it). Rather, it exists only to pass on common characteristics to more specific versions of itself. In our example, these would be characteristics or important information common to all trucks, such as `numberOfWheels`, `allWheelDrive` or `groundClearance`. Each subclass of `truck` might then add more specific characteristics in their separate implementations. `Monster truck`, for instance, might include a `crushCar` method and a `paintJob` attribute. These are examples of single inheritance, which we can now describe as a relationship in which one class inherits a set of characteristics from a more general class. We often refer to this as an "is-a" relationship, a term derived from the situational semantics. That is, a `monster truck` is a `truck`, which is a `vehicle`, and so on from one class to another, more general class.

To understand this further, we can extend our example again to an additional superclass, `vehicle`, of which `truck` is a child. This new class is a step toward greater generalization; an even more abstract class that can be used to create other types of vehicles, such as `car` or `spaceship`. The vehicle class might contain a few very general methods like `move` and `stop`, and some basic attributes like `fuelType` and `manufacturer`, which will most likely be useful to all subclasses. Figure 2.3 illustrates this example.

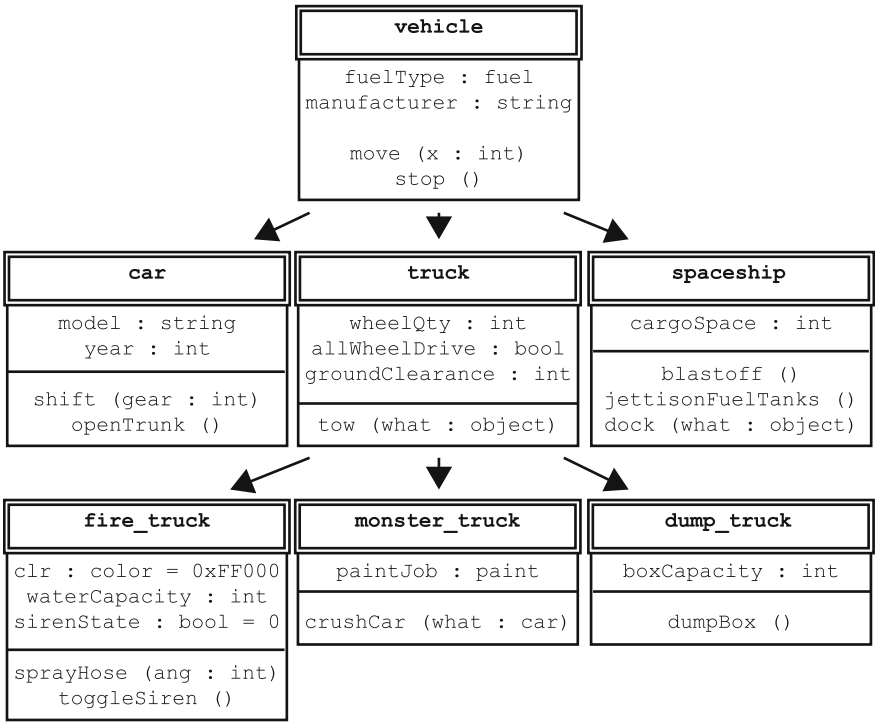

Fig. 2.3 Class hierarchy

2.4.1 Overloading

Overloading is an object-oriented programming practice by which, in certain circumstances, different methods of a class can share the same name. This is known as *overloading* the name with multiple implementations (Jia 2003). One of two criteria must be met in order for two methods to overload the same name:

1. The methods must accept a different number of arguments.
2. The arguments accepted by the methods must be of different data types.

Due to the potential for overloading, programming languages do not use only the name of a method, but rather the signature, which consists of the name in combination with the arguments passed, in order to determine which implementation of a method to call. The following class description provides an example of overloading.

```
public class ChatterBox {
 protected String fristName;

 public ChatterBox() {
    firstName = "Chatty";
 }

 public ChatterBox(String firstName) {
    this.firstName = firstName;
 }

 public String sayHello() {
    String s = "Hello! My name is " + firstName +
".";
    return s;
 }

 public String sayHello(String userName) {
    String s = "Hello, " + userName + "! My name is
" + firstName + ".";
    return s;
 }

 public String sayHello(int times) {
    String s = "";
    while (times > 0) {
          s += "Hello! ";
          times--;
    }
    s += "My name is " + firstName + ".";
    return s;
 }
}
```

This example, written in Java, uses an overloaded constructor, `ChatterBox`, to handle the potential need for a default value to be assigned to its only attribute, `firstName`. This common usage of overloading allows software engineers to apply a minimalist approach to object creation, and avoid writing unnecessary code. In the case of the `ChatterBox` class, if a string argument is passed with the constructor, that string will be assigned to the `firstName` attribute. On the other hand, if no string is passed, `firstName` is given a default value, "Chatty".

The class description above also demonstrates the second usage of overloading: convenience. The three implementations of the sayHello method all serve a slightly different, but related purpose. Overloading is used here to provide logical

Table 2.1 Overloading methods of the `ChatterBox` class

Method	Signature
`string sayHello()`	`sayHello()`
`string sayHello(String userName)`	`sayHello(String)`
`string sayHello(int times)`	`sayHello(int)`

access to the desired actions by differentiating the implementations by the arguments passed. Table 2.1 lists the signatures of each of these implementations.

The following code segment provides examples for possible calls to the `sayHello` method with different arguments.

```
        ChatterBox c = new ChatterBox('Bob');          //invoke
ChatterBox(String)

        c.sayHello();                                  //invoke
                                                       sayHello
                                                       ()
                                                       //return
                                                       "Hello!
                                                       My  name
                                                       is Bob."

        c.sayHello('John');                            //invoke
                                                       sayHello
                                                       (String)
                                                       //return
                                                       "Hello,
                                                       John! My
                                                       name  is
                                                       Bob."

        c.sayHello(3)                                  //invoke
                                                       sayHello
                                                       (int)
                                                       //return
                                                       "Hello!
                                                       Hello!
                                                       Hello!
                                                       My  name
                                                       is Bob."
```

2.4.2 Overriding

We have said that inheritance permits a class to inherit the characteristics of some other class. What happens, though if we need to modify some inherited method within a subclass? **Overriding** is the practice of replacing a method inherited from a superclass with a newly defined method in the sub class. Unlike overloading, which differentiates different implementations of methods with the same name by unique signatures, overriding requires that a method have exactly the same signature and return type as the method it is replacing. The following Java code segment illustrates the use of overriding on two classes related by inheritance.

```
  public class A {
   public String greetings(String userName) {
       String s = "Hello, " + userName;
       return s;
   }
}
public class B extends A {
   public String greetings(String userName) {
       String s = "Hola, " + userName;
       return s;
   }
}
```

In the example above, class `B`, which uses the Java `extends` keyword to inherit the structure of class `A`, requires a different implementation of the `greetings` method. Thus, the method is overridden by coding a new method with an identical signature as the inherited method.

Overriding can be extremely useful, but it carries with it significant risks. Overriding can potentially negate the integrity that is provided by the use of inheritance. This occurs when the dependencies within a class are disturbed. Consider the example in Fig. 2.4, which illustrates the subclass `employee` of the class `person`. Assume that the `setDateOfBirth` method defined in the `person` class accepts some month-day-year style argument as a date of birth. Overriding allows us to redefine this method in the `employee` class to accept a day-month-year style date of birth. However, the `calculateAge` method, which returns an age based on the current date and the `dateOfBirth` attribute, will now produce an error unless it too is overridden.

2.4.3 Polymorphism

Imagine for a moment that the only vehicle you had ever driven was a compact sedan. This is the vehicle that you used when learning to drive, and the only

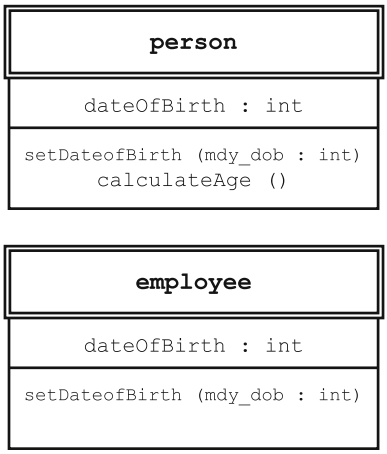

Fig. 2.4 Overriding

vehicle that you have used since. This is the only car that you are familiar with; the only car that you know how to operate. Now, imagine that one day, your compact sedan breaks down. In order to go to work, you are required to borrow a friend's car. This car, though, is not a compact sedan. It is a luxury SUV. How, then, will you be able to drive this vehicle without prior knowledge of its specific operation?

In reality, we know that the problem posed above probably will not be a problem at all. It seems reasonable enough, though, to assume that a compact sedan and a luxury SUV have different enough inner workings to require separate and distinct methods of operation. Why, then, would the knowledge of how to operate one allow us the ability to operate the other? The answer is that both the compact sedan and the luxury SUV share a common interface through which they can be accessed. Simply put, most, if not all, of the control methods for one also work for the other. This common method of access is of course based on some general idea of how an automobile should be used. Put another way, both `compact_sedand` and `luxury_SUV` have inherited a common interface from the superclass `automobile`, which allows them to perform a set of general functions based on the same methods of access.

Polymorphism is an engineering concept concerning the ability of separate classes and their derived instances to be accessed in the same way, assuming they are derived from the same superclass (Booch 1994). This method of access is provided for in a common interface that is defined in the superclass. This provides a level of encapsulation by hiding the inner workings of a class or object from the user, while allowing access in a familiar way. In the example above, the compact sedan and the luxury SUV are both started in the same manner, with the insertion of a key into the ignition, and the turning of that key. The internal actions that fire up the two vehicles, however, may be entirely different from one and other. So, to narrow our definition, polymorphism dictates that a common interface can be used to access the unique inner workings of separate classes that are related by a common superclass. Polymorphism is a powerful tool that permits the use of

unique but related objects with only a general understanding of how those types of objects are to be accessed rather than requiring the specific knowledge of the individual objects themselves.

2.5 Abstraction

Abstraction is a fundamental concept in object-oriented software engineering that allows for the efficient management of complexity through the use of generalization. It is the practice of separating those details of a situation that are significant to the current purpose from those that are not, resulting in an abstraction of the situation as a whole. Booch explains that "an abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provides crisply defined conceptual boundaries" of that object (Booch 1994). By this definition, an abstraction is a generalization of an object which includes only those details necessary to define it and to differentiate it from other objects. It describes the "outside view" of an object, and not the objects inner workings, or encapsulated information. An object's abstraction is a simplified form of its original self. It retains only those details required for accurate description and identification of the object, but excludes all other nonessential information (Jia 2003).

The concept of abstraction is closely related to those of encapsulation and information hiding. Essentially, the idea behind the use of an interface is to present a view of an object that includes only the relevant and necessary details; an abstraction of an object. Hiding the inner workings promotes economical use of that object and aids in securing its internal integrity.

2.5.1 Abstract Classes

One of the most fundamental forms of abstraction in object-oriented software engineering is the use of **abstract classes**. Earlier, we briefly described an abstract class as a class which is never meant to be instantiated, but rather exists to pass its characteristics on to more highly specified versions of itself: its subclasses. We can now identify an abstract class as an abstraction comprised of a set of highly generalized, often largely unspecified attributes and methods that are to be passed on to subclasses which are then able to tailor those characteristics to their specific purposes and inner workings. An abstract class is often the root of an inheritance hierarchy and provides the initial interface for the classes that will be derived from it. The abstract class, in this case a base class, expresses the functionality of all subclasses, but does so with such a high level of abstraction that it is impossible to explicitly define all of the implementations behind the interface (McGregor and Sykes 1992).

Though technically identical to other classes in that it consists of methods and attributes which can be used to create instantiations of itself, a class is only considered an abstract class when it is used only as a superclass for other classes,

and is not itself instantiated. An abstract class only specifies properties, that is to say, it is not used to create objects. Abstract classes provide the structure of all of the classes in a class hierarchy, while the subclasses are responsible for defining the specifics of the properties that they inherit. The following code segments illustrate the use of abstract classes.

```
  /*Set of classes for dice with different number of
sides*/

  /*This is an abstract class for Dice*/
  /*It is the base class for all other Dice
Subclasses*/
  class Die {
   /*Declares unspecified protected variable for
Number of sides*/
   protected int sides;

   /*Declares function to roll Die*/
   public int roll() {
      int i = (int)(Math.ceil(Math.random() *
this.sides));
      return i;
   }
}

/*Class for six-sided Die*/
class Die_Six extends Die {
   Die_Six() {
      this.sides = 6;
   }
}

/*Class for ten-sided Die*/
class Die_Ten extends Die {
   Die_Ten() {
      this.sides = 10;
   }
}

/*Class for Twenty-sided Die*/
class Die_Twenty extends Die {
   Die_Twenty() {
      this.sides = 20;
   }
}
```

In the code segment above, classes representing dice with different numbers of sides are created using an abstract base class `Die`. In the base class, one attribute and one method are defined. The attribute, `sides`, is an integer that denotes the number of sides on a die. Note that this attribute is only *declared* in our base class, and is not assigned a value. The method defined in the base class is `roll`, which takes advantage of Java's `Math.random` method to select a random integer between 1 and the number denoted by the `sides` attribute. We are able to define this method in our abstract class because its implementation will not change in any of our subclasses: all of our dice will produce a random number based on their number of sides when rolled. It is important to understand, though, that this method cannot be used in our abstract class, because there is no assigned value for the variable `sides`.

The subclasses which we create next are specific version of the abstract class `Die`. That is, they are subclasses of `Die` for which a number of sides is specified (six for `Die_Six`, ten for `Die_Ten`, and twenty for `Die_Twenty`). Each class contains just one method, a constructor which is used to create an instance of the class in the form of an object. In each constructor method, the number of sides for that specific die is assigned to the `sides` attribute, which is inherited from the superclass (or abstraction) `Die`. In addition, each subclass of `Die` also inherits the `roll` method. With this, we now have functioning die objects, which, when this method is called, return a random number between 1 and the value of their individual `sides` attribute.

2.5.2 Template

Closely related to the concept of an abstract class is the concept of **templates**. The use of templates provides for the declaration of the structure and function of subclasses without regard for the data type to be handled by them. In other words, a template is a sort of abstract class definition intended to be used on a data type that is yet to be defined; a *template* for a class. The data type in question is declared only upon the instantiation of an object from the template class, along with the actual class definition. Before this point, in the template itself, a placeholder is used to refer to the data type. Take a look at the following code segment for an example.

```
template class List for X {

   /*Data structure needed to be implemented with
 some sort of list that reacts to the following
 methods*/

 append(X element) { … };
 X.getFirst() { … };
 X.getNext() { … };
}
```

The above template class `List` looks similar to any other class definition, except that the first line denotes it as a template for use with the undefined type `X`. This identifier, `X`, is the placeholder that will be replaced when the template class is instantiated by some concrete data type that is to be acted on. The `append` method, for example, will then accept a single argument containing that data type and add it to the list. The data type of the element will be declared upon the creation of a list object, as in the following example.

```
  class Apple{

   /*Data structure relating to the qualities of an
apple*/
   Apple() { … };  /*Apple constructor*/

  }


  /*Create a List object and specify the data type to
be used*/
  List for Apple appleList;

  /*Make some apples from the Apple class*/
  Apple appleA;
  Apple appleB;

  /*Add them to the list*/
  appleList.append(appleA);
  appleList.append(appleB);
```

In the first bit of code above, we create a class `Apple`. Next, we instantiate the template class `List` into an object named `appleList`, to be used with the data type `Apple`. We go on to create two instances of the `Apple` class, `appleA` and `appleB`, and use the `append` method derived from the template class to add them to `appleList`. The statement `List for Apple appleList` substitutes every occurrence of the placeholder `X` from our template class with the data type `Apple` for `appleList`. In this way, templates provide for yet another level of abstraction by allowing for dynamic data type declarations with classes.

2.5.3 Generic Components

The principle of abstraction lies at the heart of component based software engineering. Abstraction allows for the creation of generalized components which can be modified and implemented for specific situations. This generalization enhances

the reusability of components, assuming that the component in question is generic enough to both be used in different contexts and to capture the common features of those contexts (D'Sourza and Wills 1999). For this reason, software engineers often look to enhance the generic quality of components. This can be performed in several ways, and may result in the modification of a class created for some specific purpose and circumstance, into a more general class which can then be implemented in various other contexts. The use of various forms of inheritance through the creation of templates and abstract classes work toward this end.

2.5.4 Interfaces

We have described the concept behind interfaces at various points throughout this chapter. Simply put, an interface is a system that allows two separate entities to interact with each other. It does this by closing off an object's outward appearance from its inner workings and by providing a set of methods for interaction. As we have said, the interface is not unique to software engineering, but is a common feature that can be found in countless forms: spoken language acts as an interface between people, a keyboard is an interface into a computer, a faucet handle is an interface for controlling water in a sink, and a mouth is an interface between an animal and the food that it consumes.

In software engineering, the use of an interface defines a manner for interacting with a class or an object. In some programming languages an interface is itself a data type, just like any other class. Like a class, an interface defines a set of methods, however, unlike a class the interface never implements those methods. Instead, an interface is used by a class, which implements those methods for its own use. A class can even make use of multiple interfaces to allow for different manners of interaction.

In the example that we have used throughout this chapter, the `monster_truck` class and its class hierarchy define what a `monster_truck` is; what it can and cannot do. A monster truck, however, can be used in other ways. For instance, a monster truck sitting at a car lot must be inventoried, inspected and categorized according to various characteristics, including price. The system responsible for managing the lot's inventory does not care one bit about interacting with a monster truck in the way that a driver would, nor does it care whether or not it is managing a monster truck at all. So, instead of accessing our `monster_truck` class as defined by its normal implementation, the inventory system might set up some other sort of communication protocol to communicate information such as price and model number. The system that it creates, of course, is just another interface through which we are now able to interact with the `monster_truck` class. For this system to work, both the `monster_truck` class and the inventory system must agree to this protocol by implementing their own forms of the interface, and thus all of the methods defined in that interface.

Interfaces are extremely useful tools that are vital to many object-oriented concepts, including information hiding, efficient reuse, abstraction, encapsulation and the proper use of inheritance.

2.6 Chapter Conclusion and Summary

This chapter focused on the object-oriented paradigm and its relationship to software engineering. We defined, first, the base components of object-oriented theory: objects and classes. Classes, as we have said, are not tangible entities, but rather they are blueprints for creating such entities. Objects, on the other hand, are created by classes, which represent the structure of a software system, and are made up of attributes, methods and some unique identifier.

Object-oriented theory is strongly rooted in the concept of modularity. A modular software system is comprised of independent components which are properly implemented in order to function together. The concept of modularity enhances the creation of reusable software components through the proper use of encapsulation and information hiding. Encapsulation allows us to separate an object's inner workings from its outward appearance, and thus lend the object internal integrity.

The creation of a modular software system relies on the use of abstraction to create a general view of the system's components. Through abstraction, a series of generalized components can both provide the boundary to be used in encapsulation and establish a logical class hierarchy for the specification of individual components and classes.

2.7 Exercises

1. Using examples, explain the difference between a class and an object.
2. Explain how the concepts of the object-oriented paradigm are used to reduce the complexity of a software system.
3. We have said that an object consists of attributes and methods. What are these? Describe one attribute and one method of a pencil?
4. Explain how inheritance might jeopardize encapsulation. Can you think of a solution for this?
5. Describe an inheritance hierarchy connecting a button down shit to its root class, clothing.

References

Booch G (1994) Object-oriented analysis and design with applications, 2nd edn. The Benjamin/Cummings Publishing Companys, Inc., New York

Bruegge B, Dutoit A (2004) Object-oriented software engineering: using UML, patterns, and java, 2nd edn. Pearson Education, Ltd., Upper Saddle River

D'Sourza D F, Wills A C (1999) Objects, components, and frameworks with UML: the catalysis approach. Addison Wesley Longman, Inc., Sydney

Jia X (2003) Object-oriented software development using java: principles, patterns, and frameworks, 2nd edn. Addison-Wesley, Boston

McGregor J D, Sykes D A (1992) Object-oiented software development: engineering software for reuse. Von Nostrand Reinhold Co., New York

Merriam-Webster (2009) Online Dictionary. http://www.merriam-webster.com/dictionary/class. Accessed 30 Jan 2009

Schach S (2008) Object-Oriented Software Engineering. McGraw-Hill Higher Education, Boston

http://www.springer.com/978-94-6239-005-8

Software Engineering: A Hands-On Approach

Lee, R.Y.

2013, XXIV, 288 p. 70 illus., 3 illus. in color., Hardcover
ISBN: 978-94-6239-005-8
A product of Atlantis Press