An Integrated Approach to Object-Oriented Software Design

Brahma Dathan Sarnath Ramnath

July 21, 2019

Contents

Chapter 2

Basics of Object-Oriented Programming

In the last chapter, we saw that the fundamental program structure in an object-oriented program is the object. We also outlined the concept of a class, which is used to define a new type, enabling a programmer to create objects of types that are not directly supported by the language.

In this chapter, we describe the basic process of object-oriented programming. We use the programming language Java (as we will do throughout the book). We begin with the basic processes associated with defining classes, and how these classes can be used in an application. We discuss interfaces, a concept that helps us specify program requirements, and present examples that use these. We then introduce the concept of a hierarchy of classes, related through the idea of inheritance. This is followed by concepts of polymorphic and dynamic binding, which enable the programmer to exploit the hierarchical class structure defined by inheritance.

The power of any language is enhanced through metaphors. Design patterns are a metaphor of object-oriented programming that document standard solutions to commonly arising problems. In this chapter we introduce this concept through the Iterator pattern.

Object-oriented languages usually provide support for genericity and run-time type identification. We conclude this chapter by introducing Java language features that enable us to specify generic classes and identify the type of objects at run-time.

2.1 The Basics

To understand the notion of objects and classes, we start with an analogy. When a car manufacturer decides to build a new car, it has to expend considerable effort before the first car can be rolled out of the assembly lines. For instance,

- the user community that will buy the car has to be identified and the user's needs have to be assessed. For this, the manufacturer may form a team.

- after assessing the requirements, the team may be expanded to include automobile engineers and other specialists who come up with a preliminary design.

- a variety of methods may be used to assess and refine the initial design: the team may have experience in building a similar vehicle; prototypes may be built; and simulations and mathematical analysis may be performed.

Perhaps after months of such activity, the design process is completed. Another step that needs to be performed is the building of the plant where the car will be produced. The assembly line has to be set up, and people have to be hired.

After such steps, the company is ready to produce cars. The design is now reused many times in the manufacturing process. Of course, the design may have to be fine-tuned during the process based on the company's observations and user feedback.

The development of software systems often follows a similar pattern. User needs have to be assessed, a design has to be completed, and then the product has to be built.

From the standpoint of object-oriented systems, a different aspect of the car manufacturing process is important. The design of a certain type of car will call for specific types of engine, transmission, brake system, and so on, and each of these parts in itself has its own design ("blue print"), production plants, etc. In other words, the company follows the same philosophy in the manufacture of the individual parts as it does in the production of the car. Of course, some parts may be bought from manufacturers, but they in turn follow the same approach. Since the design activity is costly, a manufacturer might reuse earlier designs, for the car as well as the parts.

The above approach can be compared with the design of object-oriented systems, which are composed of many objects that interact with each other. Often these objects represent real-life players and their interactions represent real-life interactions. Just like the design of a car, the design of an object-oriented system consists of designs of its constituent parts and the design of their interactions.

For instance, a banking system could have a set of objects that represent customers, another set of objects that stand for accounts, and a third set of objects that correspond to loans. An object is thus a program entity that often represents a real-life object.

Every object has a set of **properties** or **attributes** attached to it. The **attribute** or **property** of an object is a part of an object and it serves to partly describe the object and holds some value that is required in some interaction of the object with the rest of the system.

For example, a customer object may have properties or attributes such as name, address, phone number, etc. The values of all the properties of an object constitute the **state** of the object.

For every object, the system maintains a value for each of its attributes, thereby maintaining its state. For example, all customers may have the attributes name, phone number, and address. For a specific customer, the values associated with these attributes may be "Tom," "123-4567," and "1 Main Street." These three values together form the state of the object.

Also associated with an object is its **behavior**, which is the set of actions that can be performed on or by an object. For example, one of the behaviors exhibited by a bank customer may be a deposit she makes to one of her accounts. When a customer actually makes a deposit into her account in real life, the system acts on the corresponding account object to mimic the deposit in software. Another behavior may be that a customer takes out a loan; a new loan object is created and connected to the customer object. When a payment is made on the loan, the system acts on the corresponding loan object.

The system must provide a way for the programmer to describe the properties and behavior of all objects. The class mechanism provides this support. A class is a template definition of the methods and variables needed for a particular kind of object. An object is a particular instance of a class.

Thus from an object-oriented point of view, a banking system should have classes representing customers, loans, and accounts. When a new customer enters the system, we should be able to create a new customer object in software. This software entity, the customer object, should have all of the relevant features of the real-life customer. For example, it should be possible to associate the name and address of the customer with this object; however, customer's attributes that are not relevant to the bank will not be represented in software either. As an example, it is difficult to imagine a bank being interested in whether a customer is right-handed; therefore, the software entity will not have this attribute either.

A class is a design that can be reused any number of times to create objects. For example, consider an object-oriented system for a university. There are student objects, instructor objects, staff member objects, and so on. Before such objects are created, we create classes that serve as templates (or blue-prints) for students, instructors, staff members, and courses. The class for students is defined as follows:

```
public class Student {
  // code to implement a single student
}
```

Consider the first line of the above code, `public class Student {`. The first token, `public`, is a keyword that makes the corresponding class available throughout the file system. The second token, `class`, is a keyword that says that we are creating a class and that the following token is the name of the class. The third token, `Student` is the name of the class.

The left-curly bracket (`{`) signifies the beginning of the definition of the class. Following this are placed the details of the class, and the corresponding right-curly bracket (`}`) ends the definition.

Proceeding in this fashion, we can create classes for `Instructor`, `StaffMember` and `Course`.

```
public class Instructor {
  // code to implement a single instructor
}

public class StaffMember {
  // code to implement a single staff member
}

public class Course {
  // code to implement a single course
}
```

The above definitions are stubs that show how to create four classes, without giving any details. (To be of any use, the comments will need to be replaced by the details.)

New classes are defined in the above manner. The Java programming language has thousands of classes that are pre-defined, such as the `String` class. One way that a program can use a class, is by creating objects that belong to that class. The process of creating an object is also called **instantiation.** Each class introduces a new type name. Thus `Student`, `Instructor`, `StaffMember`, and `Course` are types that we have introduced.

The following code instantiates a new object of type `Student`.

```
new Student();
```

The `new` operator causes the system to allocate an object of type `Student` with enough storage for storing information about one student. The operator returns the address of the location that contains this object. This address is termed as **reference.**

The above statement may be executed when we have a new student admitted to the university.

Once we instantiate a new object, we must store its reference somewhere, so that we can use it later in some appropriate way. For this, we create a variable of type `Student`.

```
Student harry;
```

Notice that the above definition simply says that `harry` is a variable that can store references to objects of type `Student`. Thus, we can write

```
harry = new Student();
```

We cannot write

```
harry = new Instructor();
```

because `harry` is of type `Student`, which has no relationship (as far as the class declarations are concerned) to `Instructor`, which is the type of the object created on the right-hand side of the assignment.

Whenever we instantiate a new object, we must remember the reference to that object somewhere. When the program runs, it invokes various actions on the instantiated objects. The implementation of a class tell us what kinds of actions can be performed by objects of that class.

2.2 Implementing Classes

In this section, we give some of the basics of creating classes. Let us focus on the `Student` class that we initially coded as

```
public class Student {
  // code to implement a single student
}
```

We certainly would like the ability to store a name for the student: given a student object, we should be able to specify that the student's name is `"Tom"` or `"Jane"`, or, in general, some string. The name is thus an attribute of `student`, and this attribute can be accessed and modified through the variable `name`, which is of type `String`. The code is embellished as follows:

```
public class Student {
    private String name;  // field to remember the name
   // code for doing other things
}
```

A field is a variable defined directly within a class and corresponds to an attribute. Every instance of the object will have storage for the field.

Next, we would like to assign a name to a student. The action, such as assigning a name, is an example of a behavior of the object. The student object responds to the action by assigning the name to itself. i.e, storing the name in the field `name`.

```
public class Student {
  private String name;
  public void setName(String studentName) {
    name = studentName;
  }
}
```

The code that we added is called a **method**. The method's name is `setName`. A method is like a procedure or function in imperative programming in that it is a unit of code that is not activated until it is invoked. Again, as in the case of procedures and functions, methods accept parameters. Each parameter states the type of the parameter expected. A method may return nothing (as is the case here) or return an object or a value of a primitive type. Here we have put `void` in front of the method name meaning that the method returns nothing. The left and right curly brackets begin and end the code that defines the method.

Unlike functions and procedures, methods are usually invoked through objects. The `setName` method is defined within the class `Student` and is invoked on objects of type `Student`.

```
Student aStudent = new Student();
aStudent.setName("Ron");
```

The method `setName()` is invoked on that object referred to by `aStudent`. Intuitively, the code within that method must store the name somewhere. Remember that every object is allocated its own storage.

Let us examine the code within the method `setName`. It takes in one parameter, `studentName`, and assigns the value in that String object to the field `name`.

It is important to understand how Java uses the `name` field. Every object of type `Student` has a field called `name`. We invoked the method `setName()` on the object referred to by `aStudent`. Since `aStudent` has the field `name` and we invoked the method on `aStudent`, the reference to `name` within the method will act on the `name` field of `aStudent`.

The `setName` method is an example of a **modifier** or a **setter**, i.e., a method that modifies(or sets) the value of a field. If a program wants to access the data stored in a field, it would use an **accessor** or **getter** method. The `getName()` method accesses or gets the contents of the `name` field and returns it.

```
  public String getName() {
    return name;
  }
```

To illustrate this further, consider two objects of type `Student`.

```
Student student1 = new Student();
Student student2 = new Student();
student1.setName("John");
student2.setName("Mary");
System.out.println(student1.getName());
System.out.println(student2.getName());
```

Members (fields and methods for now) of a class can be accessed by writing

```
<object-reference>.<member-name>
```

The object referred to by `student1` has its `name` field set to "John," whereas the object referred to by `student2` has its `name` field set to "Mary." The field `name` in the code

```
    name = studentName;
```

refers to different objects in different instantiations and thus different instances of fields.

Let us write a complete program using the above code.

```
public class Student {
  // code
  private String name;
  public void setName(String studentName) {
    name = studentName;
  }
  public String getName() {
    return name;
  }
  public static void main(String[] s) {
    Student student1 = new Student();
    Student student2 = new Student();
    student1.setName("John");
    student2.setName("Mary");
    System.out.println(student1.getName());
    System.out.println(student2.getName());
  }
}
```

The keyword `public` in front of the method `setName()` makes the method available wherever the object is available. But what about the keyword `private` in front of the field `name`? It signifies that this variable can be accessed only from code within the class `Student`. Since the line

```
    name = studentName;
```

is within the class, the compiler allows it. However, if we write

```
    Student someStudent = new Student();
    someStudent.name = "Mary";
```

outside the class, the compiler will generate a syntax error.

As a general rule, fields are often defined with the `private` access specifier and methods are usually made public. The general idea is that fields denote the state of the object and that the state can be changed only by interacting through pre-defined methods, which denote the behavior of the object. Usually, this helps preserve data integrity.

In the current example though, it is hard to argue that data integrity consideration plays a role in making `name` private because all that the method `setName()` does is change the name field. However, if we wanted to do some checks before actually changing a student's name (which should not happen that often), this gives us a way to do it.

For a more justified use of `private`, consider the grade point average (GPA) of a student. Clearly, we need to keep track of the GPA and need a field for it. GPA is not something that is changed arbitrarily: It changes when a student gets a grade for a course. So making it public could lead to integrity problems because the field can be inadvertently changed by bad code written outside. Thus, we code as follows.

```
public class Student {
  // fields to store the classes the student has registered for.
  private String name;
  private double gpa;
  public void setName(String studentName) {
    name = studentName;
  }
  public void addCourse(Course newCourse) {
    // code to store a ref to newCourse in the Student object.
  }
  private void computeGPA() {
    // code to access the stored courses, compute and set the gpa
  }
  public double getGPA() {
    return gpa;
  }
  public void assignGrade(Course aCourse, char newGrade) {
    // code to assign newGrade to aCourse
    computeGPA();
  }
}
```

We now write code to utilize the above idea.

```
Student aStudent = new Student();
Course aCourse = new Course();
aStudent.addCourse(aCourse);
aStudent.assignGrade(aCourse, 'B');
System.out.println(aStudent.getGPA());
```

The above code creates a `Student` object and a `Course` object. It calls the `addCourse()` method on the student, to add the course to the collection of courses taken by the student, and then calls `assignGrade()`. Note the two parameters: `aCourse` and `'B'`. The implied meaning is that the student has completed the course (`aCourse`) with a grade of `'B'`. The code in the method should then compute the new GPA for the student using the information presumably in the course (such as number of credits) and the number of points for a grade of 'B'.

2.2.1 Constructors

In the above code, the operation `new Student()` created a new `Student` object with no information. We set the name of the student after creating the object, which is somewhat unnatural. Since every student has a name, when we create a student object, we probably know the student's name as well. It would be convenient to store the student's name in the object as we create the student object.

Java and other object-oriented languages allow the initialization of fields by using what are called **constructors.** A constructor is like a method in that it can have an access specifier (like public or private), a name, parameters, and executable code. However, constructors have the following differences or special features.

1. Constructors cannot have a return type: not even void.
2. Constructors have the same name as the class in which they are defined.
3. Constructors are called when the object is created.

For the class `Student` we can write the following constructor.

```
public Student(String studentName) {
  name = studentName;
}
```

The syntax is similar to that of methods, but there is no return type. The keyword `public` signifies that the constructor can be invoked from any other program that has access to the `Student` class. The code in the constructor stores the parameter `studentName` in the attribute `name`.

Let us rewrite the `Student` class with this constructor and a few other modifications.

```
public class Student {
  private String name;
  private String address;
  private double gpa;
  public Student(String studentName) {
     name = studentName;
  }
  public void setName(String studentName) {
    name = studentName;
  }
  public void setAddress(String studentAddress) {
    address = studentAddress;
  }
  public String getName() {
    return name;
  }
  public String getAddress() {
    return address;
  }
  public double getGpa() {
    return gpa;
  }
  public void computeGPA(Course newCourse, char grade) {
    // use the grade and course to update gpa
  }
}
```

We now maintain the address of the student and provide methods to set and get the name and the address.

With the above constructor, an object is created as below.

```
Student aStudent = new Student("John");
```

When the above statement is executed, the constructor is called with the given parameter, "John." This gets stored in the `name` field of the object.

In previous versions of the `Student` class, we did not have a constructor. In such cases, where we do not have an explicit constructor, the system inserts a constructor with no arguments. Once we insert our own constructor, the system removes this default, no-argument constructor.

As a result, it is important to note that the following is no longer legal because there is no constructor with no arguments.

```
Student aStudent = new Student();
```

A class can have any number of constructors. All constructors should all have different signatures, i.e, they should differ in the way they expect parameters. The following adds two more constructors to the Student class.

```
public class Student {
  private String name;
  private String address;
  private double gpa;
  public Student(String studentName) {
     name = studentName;
  }
  public Student(String studentName, String studentAddress) {
     name = studentName;
     address = studentAddress;
  }
  public Student() {
  }
 //other code deleted
}
```

Notice that all constructors have the same name, which is the name of the class. One of the new constructors accepts the name and address of the student and stores it in the appropriate fields of the object. The other constructor accepts no arguments and does nothing: as a result, the name and address fields of the object are `null`. All of the following are therefore valid ways of instantiating the `student` object `aStudent`:

```
Student aStudent = new Student();
Student aStudent = new Student("John");
Student aStudent = new Student("John", "201 Wall St., New York, NY")
```

2.2.2 Printing an Object

Suppose we want to print an object. We might try

```
System.out.println(student);
```

where `student` is a reference of type `Student`. Unfortunately, when we compile and run the program, we see something like

```
Student@dc8569
```

which is not very meaningful to someone expecting to see the name and address. By default, for all objects, Java prints the name of the class of which the object is an instance, followed by the `@` symbol and a value, which is the unsigned hexadecimal representation of the hash code of the object. It does not print the values stored for any of the fields.

To get meaningful information, we add a method called `toString()` in the class. This method contains code that tells Java how to convert the object to a `String`.

```
public String toString() {
  // return a string
}
```

Whenever an object is to be represented as a `String`, Java calls the `toString` method on the object. The method call `System.out.println()` uses the `toString()` method to convert its arguments to the string form .

We can complete the `toString` method for the Student class as below.

```
public String toString() {
  return "Name " + name + " Address " + address + " GPA " + gpa;
}
```

It is good practice to put the `toString` method in every class and return an appropriate string. The author of the class can decide which fields are to be included in the string representation. Sometimes, the method may get slightly more involved than the simple method we have above; for instance, we may wish to print the elements of an array that the object maintains, in which case a loop might be employed to concatenate the elements.

2.2.3 Sharing Data Across All Objects

Sometimes, we need fields that are common to all instances of a class, i.e., such a field must have the same value for all objects of the class. This is ensured by creating only one instance of the field and sharing this instance among all objects of the class. Such fields are called **static** fields. In contrast, fields maintained separately for each object are called **instance** fields.

Let us turn to an example. Most universities usually have the rule that students not maintaining a certain minimum GPA will be put on academic probation. This minimum standard is the same for all students, but it is conceivable that a university may raise or lower this standard.

We would like to introduce a field for keeping track of this minimum GPA. Since the value has to be the same for all students, it is unnecessary to maintain a separate field for each student object. In fact, it is risky to keep a separate field for each object: since every instance of the field has to be given the same value, special effort will have to be made to update all copies of the field whenever we decide to change its value. This can give rise to integrity problems and is also quite inefficient.

Suppose we decide to call this new field, `minimumGPA`, and make its type `double`. We define the variable as below.

```
private static double minimumGPA;
```

The specifier `static` means that there will be just one instance of the field `minimumGPA`; The field will be created when the class is initialized by the system. Note that there does not have to be any objects for this field to exist. This instance will be shared by all instances of the class.

Suppose we need to modify this field occasionally and that we also want a method that tells us what its value is. We typically write what are called **static methods** for doing the job.

```
public static void setMinimumGPA(double newMinimum) {
  minimumGPA = newMinimum;
}
public static double getMinimumGPA() {
  return minimumGPA;
}
```

The keyword `static` specifies that the method can be executed without using an object. The method is called as below.

```
<class_Name>.<method_name>
```

For example,

```
Student.setMinimumGPA(2.0);
System.out.println("Minimum GPA requirement is " + Student.getMinimumGPA());
```

Methods and fields with the keyword `static` in front of them are usually called **static methods** and **static fields** respectively. They are also referred to as **class members**.

It is instructive to see in the above case, why we want the two methods to be static. Suppose they were instance methods. Then they have to be called using an object as in the following example.

```
Student student1 = new Student("John");
student1.setMinimumGPA(2.0);
```

While this is technically correct, it has the following disadvantages:

1. It requires that we create an object and use that object to modify a static field. This goes against the spirit of static members; they should be accessible even if there are no objects.
2. Someone reading the above fragment may be lead to believe that `setMinimumGPA()` is used to modify an instance field.

On the other hand, a static method cannot access any instance fields or methods. This is because a static method may be accessed without using any objects at all; hence there may not be any objects created yet when the static method is in use.

2.3 Creating and Working with Related Classes

Even the simplest object-oriented application will have classes that are related. The simplest relationship is when every object of a given class holds a reference to an object of another given class. Consider the university system we introduced earlier in this chapter, where we identified and wrote the skeletons of four classes: `Student`, `Instructor`, `StaffMember`, and `Course`. To understand the nature of their relationships, we take a close look at the structure of each class.

Objects of the `Course` class represent courses that are taught at the university. A course exists in the school catalog, with a name, course id, brief description, and number of credits. Here is a possible definition of the class.

```
public class Course {
  private String id;
  private String name;
  private int numberofCredits;
  private String description;
  public Course(String courseId, courseName) {
    id = courseId;
    name = courseName;
  }
  public void setNumberOfCredits(int credits) {
    numberOfCredits = credits;
  }
  public void setDescription(String courseDescription) {
    description = courseDescription;
  }
  public String getId() {
    return id;
  }
  public String getName() {
    return name;
  }
  public int getNumberOfCredits() {
    return numberOfCredits;
  }
  public String getDescription() {
    return description;
  }
}
```

A department selects from the catalog a number of courses to offer every semester. A section is a course offered in a certain semester, held in a certain place on certain days at certain times. (We will not worry about the instructor for the class, capacity, etc.) Let us create a class for this.

We will use `String` objects for storing the place, days, time, and semester. Thus, we have three fields named `place`, `daysAndTimes`, and `semester` with the obvious semantics.

Clearly, this seems inadequate: this class does not hold the name and other important details of the course. On the other hand, it is redundant to have fields for these because the information is available in the corresponding `Course` object. What we do therefore, is to provide a field that remembers the corresponding course. We can do this by having the following field declaration.

```
private Course course;
```

When the `Section` instance is created, this field can be initialized.

```
public class Section {
  private String semester;
  private String place;
  private String daysAndTimes;
  private Course course;
  public Section(Course theCourse, String theSemester,
                 String thePlace, String theDaysAndTimes) {
    course = theCourse;
    place = thePlace;
    daysAndTimes = theDaysAndTimes;
    semester = theSemester;
  }
  public String getPlace() {
    return place;
  }
  public String getDaysAndTimes() {
    return daysAndTimes;
  }
  public String getSemester() {
    return semester;
  }
  public Course getCourse() {
    return course;
  }
  public void setPlace(String newPlace) {
    place = newPlace;
```

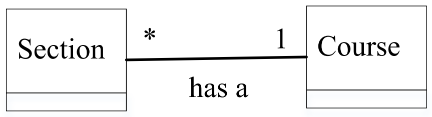


Figure 2.1: Figure showing a many-to-one relationship between section and course. Several sections can be associated with a single course.

```
  }
  public void setDaysAndTimes(String newDaysAndTimes) {
    daysAndTimes = newDaysAndTimes;
  }
}
```

By storing a reference to a `Course` object within each `Section` object, we have created an **association** between the two classes. This association captures the relationship "every section **has a** course." This association is depicted by having a rectangle each to represent the `Course` and `Section` objects, and then connecting the two rectangles by a line. This is shown in Figure 2.1.

In a typical university, each section would correspond to exactly one course, but a course would have a number of sections associated with it. It is also possible that a course has been defined, but has never been offered, i.e., there are zero sections corresponding to a course. This aspect of the association is reflected in the figure by placing an asterisk(`*`) next to `Section` and the numeral `1` next to `Course`. We implement this association by storing a reference to a `Course` object within each `Section` object.

Consider the piece of code that has to create courses and sections. A course would be created by invoking the constructor. We have a couple of options on how `Section` objects are created. One approach would be to invoke the constructor for `Section` directly.

```
Course cs350 = new Course("CS 350", "Data Structures");
Section cs350Section1 = new Section(cs350, "Fall 2004",
                                    "Lecture Hall 12", "T H 1-2:15");
Section cs350Section2 = new Section(cs350,"Fall 2004",
                                    "Lecture Hall 25", "‘M W F 10-10:50");
```

A drawback of this approach is we are not verifying that the `Course` object exists. For instance, it would be syntactically correct to write:

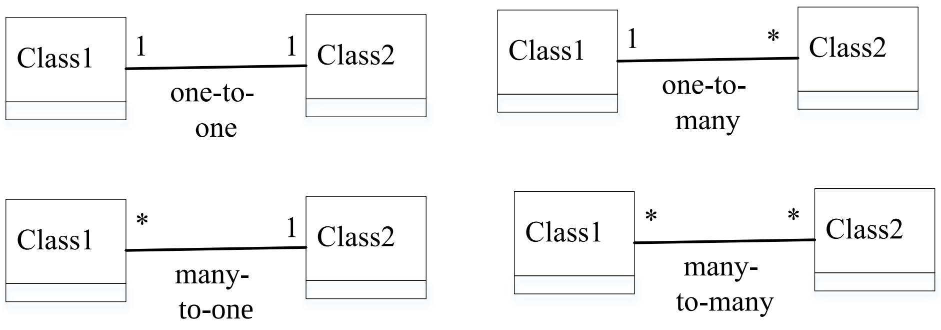
```
Course c1 = null;
Section cs350Section1 = new Section(c1, "Fall 2004",
                                    "Lecture Hall 12", "T H 1-2:15");
Section cs350Section2 = new Section(c1,"Fall 2004",
                                    "Lecture Hall 25", "‘M W F 10-10:50");
```

At some later stage in the program, one might try to access the `Course` object associated with `cs350Section1`, resulting in an error. An alternate, safer, approach is to create the `Section` in `Course`. This would require that we add a new method named `createSection` in `Course`, which accepts the semester, the place, days, and time as parameters and returns an instance of a new `Section` object for the course. Code using this method is as follows.

```
Course cs350 = new Course("CS 350", "Data Structures");
Section cs350Section1 = cs350.createSection("Fall 2004",
                                    "Lecture Hall 12", "T H 1-2:15");
Section cs350Section2 = cs350.createSection("Fall 2004",
                                    "Lecture Hall 25", "‘M W F 10-10:50");
```

Associations between classes

Associations between classes can be of different kinds. As depicted in the figure below, we could have one-to-one, one-to-many, many-to-one or many-to-many relationships.



A one-to-one relationship is the simplest situation. An example of this is the `name` field of `Student`. There has to be a unique `String` object that is related to each `Student` object. A student cannot have more than one name. Even if two students have the exact same name, the name fields cannot be shared by the two `Student` objects.
One-to-many relationship occurs when an object of one class is related to several objects of another class. For instance, a `Section` object can be connected to (containing references to) multiple `Student` objects. The asterisk (`*`) stands for any value greater than or equal to zero.This is significant since we may be dealing with a section that has no students. The many-to-one relationship represents the converse situation where several objects of one class could be related to the same object of another class. An example of this is where multiple (possibly none) `Section` objects store refernces to the same `Course` object. It is reasonable to ask how these two kinds of relationships are different. In the one-to-many example, note that each `Section` object is expected to track all the `Student` objects in some way. No such expectation is there in the many-to-one situation; all the `Section` objects maintain a reference to the object for the appropriate `Course`.
A many-to-many relationship typically occurs in the context of a larger system. For instance, if we have a university system with several courses, multiple sections of each course and several students, we expect that a student will be enrolled in multiple sections. This will imply that a `Student` object must keep references to multiple `Section` objects; combining this with the relationship discussed in the previous paragraph yields a many-to-many connection. Larger systems like this will be discussed in later chapters.

Let us get to the task of coding the createSection method. It looks like the following:

```
public Section createSection(String semester, String place, String time) {
  return new Section(this, semester, place, time);
}
```

To invoke the constructor of Section from the createSection we need a reference to the Course object, and references to the semester, place, and days and times available. References to the semester, place, and days and times available were passed as parameters to createSection, but we did not pass a reference to the Course object. Although this is not an explicit parameter to the method, the Course object on which the createSection method is invoked is itself the reference we need. Here the language comes to our aid. In the createSection method, the reference to the object on which the method was invoked is available via a special keyword called this.

We can code the createSection method as below.

```
public Section createSection(String semester, String place, String time) {
  return new Section(this, semester, place, time);
}
```

The keyword this obtains the reference to the Course object and is passed to the constructor of Section.

In addition to passing a reference to itself to methods, we can use this to obtain the fields of the object, which come in handy for resolving conflicts. For example,

```
public class Section {
    private String place;
    public void setPlace(String place) {
        this.place = place;
    }
}
```

The identifier place on right hand side of the assignment refers to the formal parameter; on the left hand side it is prefixed by this, which makes it a reference to the private field.

2.4 Defining and Working with Collections

For any given section, a faculty member would like to see the list of students in the section. It makes sense, therefore, to store a list of the registered students within each Section object. One approach for implementing a collection is to define an array within the class. If we used such an approach within our Section class, we would add a field to reference the array as follows:

```
public class Section {
    private Student studentList[];
    private int numStudents;
    //  other code not shown
}
```

An additional field, `numStudents` could be used to keep track of the number of registered students. Within the constructor, we would have to instantiate the array object, as shown.

```
public class Section {
  public Section(Course theCourse, String theSemester,
                  String thePlace, String theDaysAndTimes) {
    course = theCourse;
    place = thePlace;
    daysAndTimes = theDaysAndTimes;
    semester = theSemester;
    studentList = new Student[30];
    numStudents = 0;
  }
  //  other code not shown
}
```

Say we want a method that prints out all the students in the section. Our code would be something like this.

```
  public void printStudents() {
    for (int count = 0; count < numStudents; count++) {
        studentList[count].toString();
    }
  }
```

A method to add a student to a section would be as follows:

```
  public void addStudent(Student student) {
    if (numStudents < 30) {
        studentList[numStudents] = student;
        numStudents++;
    }
  }
```

In the above code, the class `Section` uses the class `Student`, but defines its own collection of students, and defines the methods needed for managing the collection. Such an approach is not very desirable for the following reasons:

- *Lack of cohesion.* While implementing a type (in this case `Section`), we should not be distracted by implementation considerations for a different type, which is in this case a collection for the type `Student`. Such code is non-cohesive. The author of the type (`Section`) would need to be cognizant of the nuances of various collection implementation strategies.
- *Unnecessary Coupling.* Sometimes, we may want to change the kind of collection being used, say from an array to a linked list. If we employed a solution like the one above, changing the kind of collection would involve making complex changes to various parts of the application.

- *Lack of reuse.* In complex systems, we may have multiple classes accessing the same collection. For instance, in our university system both the billing sub-system and the registration sub-system may have to access the same list of students. Implementing a list separately in each sub-system is both wasteful and error-prone. Instead, a separate collection class could be implemented and utilized at different places in the application. This also provides opportunity for employing more sophisticated programming techniques such as generics.[1]

Object-oriented systems, therefore, use collection classes to manage collections.

2.4.1 Creating a collection class.

We create a simple collection class for managing a list of students, which provides methods for adding a student, deleting a student and printing the list of students.

```
public class StudentLinkedList {
  // fields for maintaining a linked list
  public void add(Student student) {
    // code for adding a student to the list
  }
  public void delete(String name) {
    // code for deleting a student from the list
  }
  public void print() {
    // code for printing the list
  }
  // other methods
}
```

The class `Section` can now be written to use this collection class as follows:

```
public class Section {
  private  StudentLinkedList studentList;
  private String semester;
  private String place;
  private String daysAndTimes;
  private Course course;
  public Section(Course theCourse, String theSemester,
                String thePlace, String theDaysAndTimes) {
    course = theCourse;
    place = thePlace;
    daysAndTimes = theDaysAndTimes;
    semester = theSemester;
    studentList = new StudentLinkedList();
  }
  public void addStudent(Student student) {
    studentList.add(student);
```

[1]Generics are discussed later in the book.

```
  }
  public void deleteStudent(String name) {
    studentList.delete(name);
  }
  public void printStudents() {
    studentList.print();
  }
}
```

Note that methods for adding, deleting and printing are very much simplified, when we leave the management of the list to the collection class. It is instructive to complete the code for `StudentLinkedList`.

2.4.2 Implementation of StudentLinkedList

A linked list consists of nodes each of which stores the address of the next. We thus write the following class.

```
public class StudentNode {
  private Student data;
  private StudentNode next;
  public StudentNode(Student student, StudentNode initialLink) {
    this.data = student;
    next = initialLink;
  }
  public Student getData() {
    return data;
  }
  public void setData(Student student) {
    this.data = student;
  }
  public StudentNode getNext() {
    return next;
  }
  public void setNext(StudentNode node) {
    next = node;
  }
}
```

How do objects refer to themselves?

In general, assume that we have a class `C` with a method `m` in it as shown below. Also shown is another class `C2`, which has a method named `m2` that requires an object of type `C` as its only parameter.

```
public class C {
  public void m() {
    // this refers to the object on whom m is being invoked
  }
}

public class C2 {
  public void m2(C aC) {
    // code
  }
}
```

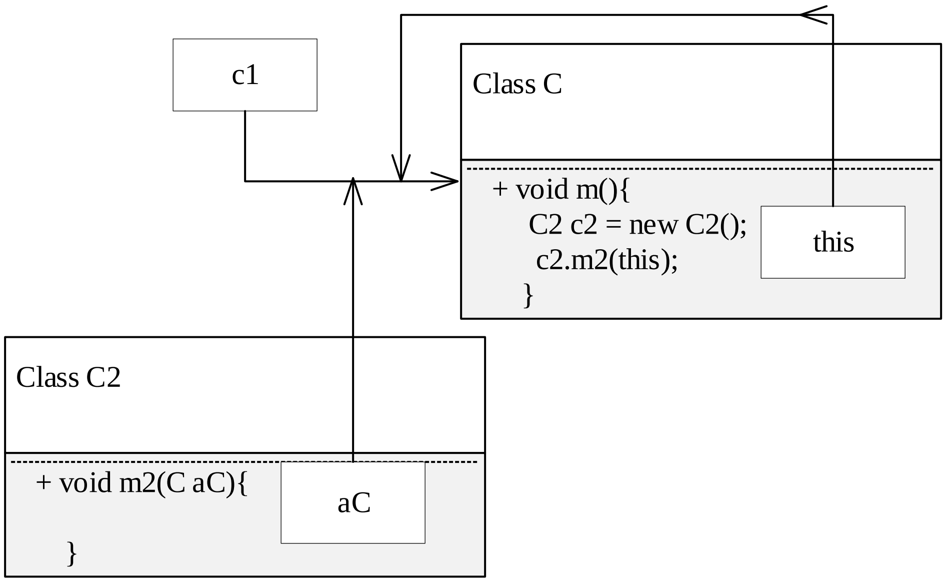
Suppose that we create an instance of `C` from the outside and invoke `m` as below.

```
C c1 = new C(); c1.m();
```

The reference `c1` points to an instance of `C`. Suppose the method `m` contained the following code:

```
public void m(){
   C2 c2 = new C2(); c2.m2(this);
}
```

In the above, `this` is a reference that points to the same object as `c1`. In summary, an object can refer to itself by using the keyword `this`.



This class will be needed in `StudentLinkedList` only. Therefore, we can use what are called **inner classes** in Java. An inner class is a class enclosed within another class. Thus, we write

```
public class StudentLinkedList implements StudentList {
  private StudentNode head;
  private class StudentNode {
    private Student data;
    private StudentNode next;
    public StudentNode(Student student, StudentNode initialLink) {
      this.data = student;
      next = initialLink;
    }
    public Student getData() {
      return data;
    }
    public void setData(Student student) {
      this.data = student;
    }
    public StudentNode getNext() {
      return next;
    }
    public void setNext(StudentNode node) {
      next = node;
    }
  }
  public void add(Student student) {
    // code for adding a student to the list
  }
  public void delete(String name) {
    // code for deleting a student from the list
  }
  public void print() {
    // code for printing the list
  }
}
```

The inner class `StudentNode` is now declared as `private`, so that it cannot be used from code outside of the class.

Let us code the `add` method.

```
  public void add(Student student) {
    head = new StudentNode(student, head);
  }
```

The code creates a new `StudentNode` and puts it at the front of the list.

Next, we code the `print()` method.

```
  public void print() {
    System.out.print("List: ");
    for (StudentNode temp = head; temp != null; temp = temp.getNext()) {
      System.out.print(temp.getData() + " ");
    }
    System.out.println();
  }
```

The code starts at the front of the list, extracts the data in the corresponding node and prints that data. Printing ends when the node it points to is `null`; that is, it doesn't exist. Assuming that the `Student` class has an appropriate `toString()` method, we will get the name, address, and GPA of each student printed.

Finally, we code the method to delete a student. We will need to look at each `Student` object and see if the `name` field matches the given name. How do we do this comparison? Suppose `temp` is a variable that refers to a `StudentNode` object within `studentList` . The call `temp.getData()` retrieves the `Student` object, and `temp.getData().getName()` gets the name of the student. To check if this name is equal to the given name, we can use the `equals()` method defined in `String`. The code for the `delete()` method is given below.

```
public void delete(String studentName) {
  if (head == null) {
     return;
  }
  if (head.getData().getName().equals(studentName)) {
     head = head.getNext();
  } else {
     for (StudentNode temp = head.getNext(), previous = head;
                            temp != null; temp = temp.getNext()) {
       if (temp.getData().getName().equals(studentName)) {
          previous.setNext(temp.getNext());
          return;
       }
     }
  }
}
```

The code first checks if the list is empty; if so, there is nothing to do. With a non-empty list, it checks if the name of the student at the front of the list is the same as the name supplied in the parameter. If they match, the `Student` object at the front of the list is deleted from the list by moving the head to the next object (which may not exist, in which case we have a null). If the element at the front of the list is not what we want, execution proceeds to a loop that examines all elements starting at the second position until the end of the list is reached or the student with the given name is located. The variable `previous` always refers to the object preceding the object referred to by `temp`. Once it is located, the object can be deleted using `previous`.

2.5 Interfaces

2.5.1 Why Use Interfaces?

In the previous section, we successfully separated the collection class (`StudentLinkedList`) from the client class that used the collection (`Section`). This level of separation still leaves some unwanted coupling between the client class and the collection class. For instance, the method `addStudent` in `Section` invokes the `add` method of `StudentLinkedList`. The code in the client class is, therefore, coupled with specific methods in the collection class. Now consider a situation where we later find that an array implementation of the collection is more desirable. Say we have a readymade implementation for the class `StudentArrayList`. We can switch to the new collection by replacing

```
 studentList = new StudentLinkedList();
```

with

```
 studentList = new StudentArrayList();
```

Next, we must make sure that the method names for the new class are being used. For instance, it is possible that the method for adding a student in `StudentAarrayList` is `addStudent`, instead of `add`. This requires us to go through all the code in `Section` and make the necessary changes. This is not an efficient way of reusing code.

2.5.2 Improving Reuse with an Interface

Reuse can be improved through the use of an **interface**, which is one way of partially specifying the requirements for a class. When we create a list of students, we should be able to add a student, remove a student, and print all students in the list. We can specify the syntax for the methods by creating an interface as given below.

```
public interface StudentList {
  public void add(Student student);
  public void delete(String name);
  public void print();
}
```

Notice that the syntax of the first line resembles the syntax for a class with the keyword `class` replaced by the keyword `interface`. We have *specified* three methods: `add` with a single parameter of type `Student`; `delete` with the name of the student as a parameter, and `print` with no parameters. Notice that we haven't given a body for the methods; there is a semi-colon immediately after the right parenthesis that ends the parameters.

Let us see how to utilize the above entity. We can now create a class that implements the above three operations as below.

```
public class StudentLinkedList implements StudentList {
  // fields for maintaining a linked list
  public void add(Student student) {
    // code for adding a student to the list
  }
  public void delete(String name) {
    // code for deleting a student from the list
  }
  public void print() {
    // code for printing the list
  }
  // other methods
}
```

The first line states that we are creating a new class named `StudentLinkedList`. The words `implements StudentList` mean that this class will have all of the methods of the interface `StudentList`. It is a syntax error if the class did not implement the three methods because it has claimed that it implements them. A `StudentArrayList` would be similarly created:

```
public class StudentArrayList implements StudentList {
  // fields for maintaining an array-based list
  public void add(Student student) {
    // code for adding a student to the list
  }
  public void delete(String name) {
    // code for deleting a student from the list
  }
  public void print() {
    // code for printing the list
  }
}
```

Just as a class introduces a new type, an interface also creates a new type. In the above example, `StudentList`, `StudentLinkedList` are both types. All instances of the `StudentLinkedList` and `StudentArrayList` classes are also of type `StudentList`.

We can thus write

```
StudentList students;
students = new StudentLinkedList();
// example of code that uses StudentList;
Student s1 = new Student(/* parameters */);
students.add(s1);
s1 = new Student(/* parameters */);
students.add(s1);
students.print();
```

We created an instance of the `StudentLinkedList` class and stored a reference to it in `students`, which is of type `StudentList`. We can invoke the three methods of the interface (and of the class) via this variable.

To a beginner, defining the interface `StudentList` and then defining `StudentLinkedList` may appear to be unnecessary. To understand our motivation for doing this, consider the following facts:

1. The class `StudentLinkedList` implements the interface `StudentList`, so variables of type `StudentLinkedList` are also of type `StudentList`.

2. We declared `students` as of type `StudentList` and *not* `StudentLinkedList`.

3. We restricted ourselves to using the methods of the interface `StudentList`.

This means that although the variable `students` holds a reference to an object of `StudentLinkedList`, all our code is written such that `students` is an object of type `StudentList`. When we wish to replace the class `StudentLinkedList` with the class `StudentArrayList`, we can rewrite the code that manipulates StudentList as below.

```
StudentList students;
students = new StudentArrayList();
// code that uses StudentList;
```

The only change that we need to make in our code for using the list is the one that creates the `StudentList` object. Since we restricted ourselves to using the methods of `StudentList` in the rest of the code (as opposed to using methods or fields unique to the class `StudentLinkedList`), we do not need to change anything else. This makes maintenance easier.

We saw the implementation for `StudentLinkedList` earlier. It is instructive to complete the code for `StudentArrayList` as well.

2.5.3 Array Implementation of Lists

We need to set up an array of `Student` objects. This is done as follows:

1. Declare a field in the class `StudentArrayList`, which is an array of type `Student`.
2. Allocate an array of the required size. We will allocate storage for as many students as the user wishes; if the user does not specify a number, we will allocate space for a small number, say, 10, of objects. In any case, when this array fills up, we will allocate more.

Therefore, we need two constructors: one that accepts the initial capacity and the other that accepts nothing. The code for the array field and the constructors is given below.

```
public class StudentArrayList implements StudentList {
  private Student[] students;
  private int initialCapacity;
  public StudentArrayList() {
    students = new Student[10];
    initialCapacity = 10;
  }
  public StudentArrayList(int capacity) {
    students = new Student[capacity];
    initialCapacity = capacity;
  }
  // other methods
}
```

Note that the code for the first constructor is a special case of the second constructor. Such a repetition is undesirable, since the code in the second constructor is general enough for handling both cases. To avoid the repetition, we need to do the following: when the user does not supply an initial capacity, we should somehow invoke the second constructor with a value of 10. This can be achieved by rewriting the first constructor as follows:

```
  public StudentArrayList() {
    this(10);
  }
```

In this case, `this(10)` invokes the other constructor of the class, with the appropriate parameter. The net effect would be the same as that of the user writing `new StudentArrayList(10)`.

The following approach is employed to manage the list. We will have two variables, `first` that gives the index of the first occupied cell, and `count` the number of objects in the list. When the list is empty, both are 0. When we add an object to the list, we will insert it at `(first + count)` % `array size` and increment `count`.

```
public class StudentArrayList implements StudentList {
  private Student[] students;
  private int first;
  private int count;
  private int initialCapacity;
  public StudentArrayList() {
    this(10);
  }
  public StudentArrayList(int capacity) {
    students = new Student[capacity];
    initialCapacity = capacity;
  }
  public void add(Student student) {
    if (count == students.length) {
      reallocate(count * 2);
    }
    int last = (first + count) % students.length;
    students[last] = student;
    count++;
  }
  public void delete(String name) {
    for (int index = first, counter = 0; counter < count;
              counter++, index = (index + 1) % students.length) {
      if (students[index].getName().equals(name)) {
        students[index] = students[(first + count - 1) % students.length];
        students[(first + count - 1) % students.length] = null;
        count--;
        return;
      }
    }
  }
  public Student get(int index) {
    if (index >= 0 && index < count) {
      return students[index];
    }
    return null;
  }
  public int size() {
    return count;
  }
  public void print() {
    for (int index = first, counter = 0; counter < count;
                             counter++, index = (index + 1) % students.length) {
      System.out.println(students[index]);
    }

  }
  public void reallocate(int size) {
    Student[] temp = new Student[size];
    if (first + count >= students.length) {
      int count1 = students.length - first;
      int count2 = count - count1;
      System.arraycopy(students, first, temp, 0, count1);
      System.arraycopy(students, first + count1, temp, count1, count2);
    } else {
      System.arraycopy(students, first, temp, 0, count);
    }
    students = temp;
    first = 0;
  }
}
```

Using `this` in a constructor vs using `this` in an instance method

The use of `this` in the constructor should not be confused with the use of `this` when an object refers to itself in instance methods. The code we wrote for `createSection` used `this` in the second context.

```
public Section createSection(String semester, String place, String time) {
  return new Section(this, semester, place, time);
}
```

Since this code is executed by the `Course` object, `this` refers to the particular instance of `Course`.
On the other hand, when we write:

```
  public StudentArrayList() {
    this(10);
  }
```

we are in a constructor (i.e., no instance has been defined) , and `this(10)` invokes the other constructor for the class.
Also, note the following aspects:

1. There can be no code before the statement `this()`. In other words, this call should be the very first statement in the constructor.
2. You can have code in the constructor after the call to another constructor.
3. You can call at most one other constructor from a constructor.

2.6 Abstract Classes

In a way, classes and interfaces represent the two ends of the spectrum of type definitions. When we write a class, we code every field and method; in other words, a complete implementation of the type. In an interface, we need to specify only the method signatures, i.e, a minimal definition of the type.

Sometimes, we might know the specifications for a class, but might not have the information needed to implement the class completely. For example, consider the set of possible shapes that can be drawn on a computer screen. While the set is infinite, let us consider only three possibilities: triangles, rectangles, and circles. We know that the set of fields needed to represent each object is different, but there are some commonalities as well: for example, all shapes have an area.

In such cases, we can implement a class partially using what are called **abstract** classes. In the case of a shape, we may code

```
public abstract class Shape {
  private double area;
  public abstract void computeArea();
  public double getArea() {
    return area;
  }
  // more fields and methods
}
```

The class is declared as abstract (using the keyword `abstract` prior to the keyword `class`), which means that the class is incomplete. Since we know that every shape has an area, we have defined the `double` field `area` and the method `getArea()` to return the area of the shape. We require that there be a method to compute the area of a shape, so we have written the method `getArea()`. But since the formula to compute the area is different for the three possible shapes, we have left out the implementation and declared the method itself as abstract.

Any class that contains an abstract method must be declared abstract. We cannot create an instance of an abstract class. The utility of an abstract class comes from the fact that it provides a basic implementation that other classes can "extend." This is done using the technique of inheritance, which is explained later in this chapter.

2.7 Dealing with Run-time Errors

In most object- oriented languages, run-time errors are manifested as **exceptions**. Consider the following lines of code:

```
  public class Except0 {
    public static void main(String[] s) {
      Student s1 = null;
      s1.toString();
    }
  }
```

We have declared the reference `s1` to be of type `Student`, but it does not contain a reference to any object. Such a reference is referred to as a `null` reference. If we place this code in the main method of a Java program it would compile correctly. When the program containing this piece of code is executed, we get something like the following error message:

```
 Exception in thread "main" java.lang.NullPointerException
        at Except0.main(Except0.java:4)
```

We say then that the program has thrown an **exception.** The programmer did not anticipate the possibility of the exception, and any code after the invocation of `toString()` will not be executed. In an object-oriented language, there is an object corresponding to each exception. We can access this object through a `try-catch` block, and **handle** the exception as shown below:

```
public class Except1 {
  public static void main(String[] s) {
    Student s1 = null;
    try {
      s1.toString();
    } catch (NullPointerException ne) {
      System.out.println(" Null Pointer Exception on s1. \n Continue(yes/no)?");
      Scanner myScanner = new Scanner(System.in);
      String answer = myScanner.next();
      if (answer.equals("no")) {
        System.exit(0);
      }
      myScanner.close();
    }
    System.out.println("continuing \n");
  }
}
```

The piece of code where an exception might be thrown is placed in braces after the keyword `try` (called the `try` block). Following the try block is the `catch` block, which names the exception that it is supposed to catch, and contains the code that should be executed before proceeding with the rest of the program. In our program, we handle the error by listing the problem (since we know what is happening) and ask if the user wishes to continue. If the user answers “no”, the system exits; otherwise the program continues execution with the statement that follows the `catch` block.

Exceptions can be more complicated, and can propagate back up the calling chain, until they are caught and handled. If they are not caught, the program will terminate. Consider the following example, with the simplified classes `Course` and `Section`.

```
class Course {
  private String name;

  public String getName() {
    return name;
  }
}

class Section {
  private Course course;
  private String id;

  public Section(Course course) {
    this.course = course;
  }

  public Section() {
  }

  public String toString() {
    return (course.getName() + id);          // line 21
  }
}

public class Except2 {
  public static void main(String[] args) {
    Section section1 = new Section();
    System.out.println(section1.toString()); // line 28
  }
}
```

Note that `Section` has two constructors, one which assigns a course and one which does not. In the `main()` method of `Except2`, we are creating a `Section` object where the `Course` reference is `null`. When this code is executed, an exception is thrown when `getName()` is invoked. The resulting error message shows all the methods and classes in the calling chain:

```
Exception in thread "main" java.lang.NullPointerException
        at Section.toString(Except2.java:21)
        at Except2.main(Except2.java:28)
```

The code at line 28 made a call, which resulted in a crash in line 28. The exception can be handled at any point along the calling chain; in that case, no error message appears.

```
public class Except3 {
  public static void main(String[] argss) {
    Section section1 = new Section();
    try {
      System.out.println(section1.toString());
    } catch (NullPointerException ne) {
      System.out.println("Continue(yes/no)?");
      Scanner myScanner = new Scanner(System.in);
      String answer = myScanner.next();
      if (answer.equals("no")) {
        System.exit(0);
      }
      myScanner.close();
    }
    System.out.println("continuing \n");
  }
}
```

The Java language defines many exceptions. In addition, a programmer can define custom exceptions for specific errors that are thrown by methods in the user-defined classes.

2.8 Inheritance

As we have noted earlier, classes are user-defined types created in the course of constructing an object-oriented program. With larger systems, it is common to encounter situations when

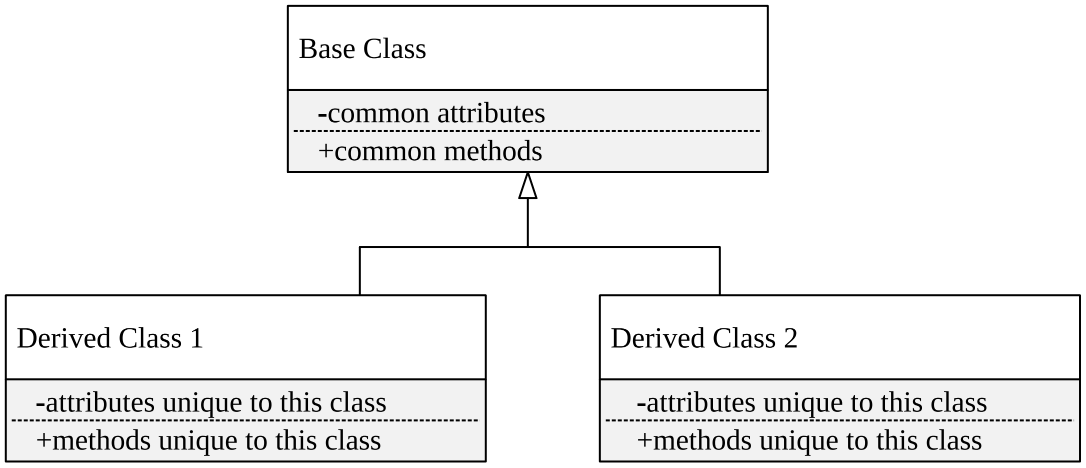

Base Class
-common attributes
+common methods
Derived Class 1
-attributes unique to this class
+methods unique to this class
Derived Class 2
-attributes unique to this class
+methods unique to this class

Figure 2.2: Basic Idea of Inheritance. We have a single base class that is common to multiple derived classes. The base class represents the commonality of all classes, and the derived class(es) represent the uniqueness.

there are several variants of a user-defined type. Trying to capture all the variants in a single class can lead to very complex code inside the class. At the same time, creating multiple independent classes will result in complicated code in the client class. In such situations it is beneficial to create a set of classes that are related by an **inheritance hierarchy**. Any object-oriented programming language, therefore, has to provide a mechanism to construct such a hierarchy.

In any object-oriented programming language, inheritance is a relationship characterized by a **superclass** and a **subclass**, which are also respectively termed an **Ancestor** or **Base** class and a **Descendant** or **Derived** class, represented using UML notation as in Figure 2.2. We draw one box per class and draw an arrow from each derived class to the baseclass.

2.8.1 An Example of a Hierarchy

Consider a company that sells various products such as television sets and books. Obvious differences between the products imply that they have different attributes to be tracked, and thus we need two classes, `Television` and `Book`. One way to accomplish this task is to create a class for television sets, say, `Television`, and a second class, for books, say, `Book`. However, in many situations, the company would like to think of books and televisions as simply products. For instance, the company needs to keep track of sales, profits (or losses), etc. for all products. Now, add to the above scenario, more products, say, CDs, DVDs, cassette players, pens, etc. Each may warrant a separate class, but, as just discussed, they all have common properties and behaviors; and to the company, they are all products.

What we see is an example of a situation where two classes have a great deal of similarities, but also substantial differences. The need to view different entities such as televisions and books as products suggests that we may benefit by having a new type `Product` introduced into the system. Since there is a fair amount of common functionality between the two products, we would like `Product` to be a class that implements the commonality found in `Television` and `Book`.

In Java, we do this as follows. We start off with a class that captures the essential properties and methods common to all products.

```
public class Product {
  // functionality for a product
}
```

The above class may have attributes such as the number of units sold and unit price. It also will have constructors and methods for recording sales, computing profits, and so on.

We are now ready to create a class that represents a single kind of TV set. For this, we note that a television is a product, and that we would like to utilize the functionality that we just implemented for products. In Java, we do this as below:

```
public class Television extends Product {
  // functionality that is unique for televisions
  // modifications
}
```

Informally speaking, the `Television` class inherits all of the properties and methods from the class `Product`. All we have done is add properties and methods unique to televisions, which will not, for obvious reasons, be implemented in `Product`.

In a similar manner, we implement the class `Book`.

```
public class Book extends Product {
  // functionality that is unique for books
  // modifications
}
```

The relationship between the three classes is depicted in Figure 2.3.

Now, notice the similarities and differences between the two classes: both classes, since they represent products, carry fields, `quantitySold` and `price` with their obvious meanings. The method `sale()` in both classes is invoked whenever one unit (a book or a TV set) is sold. The meaning of the `setPrice()` method should be obvious.

The two classes are somewhat different in other respects: `Book` has attributes `title` and `author` where as the `Television` class has the attribute `brand`. The `manufacturer` attribute is named differently from, but is not dissimilar to, `publisher`.

Here is where the power of the object-oriented paradigm comes into play. It allows the development of a baseclass or superclass that reflects the commonalities of the two classes and then extend or subclass this baseclass to arrive at the functionalities we discussed before. The class `Product` keeps track of the common attributes of `Book` and `Television` and implements the methods necessary to act on these attributes. `Television` and `Book` are now constructed as subclasses of `Product`; they will both inherit the functionalities of `Product`, so that they are now capable of keeping track of sales of these two products.

The code for `Product`, given below, is fairly simple. The variable `company` stores the manufacturer of the product. Otherwise, there are no special features to be discussed.

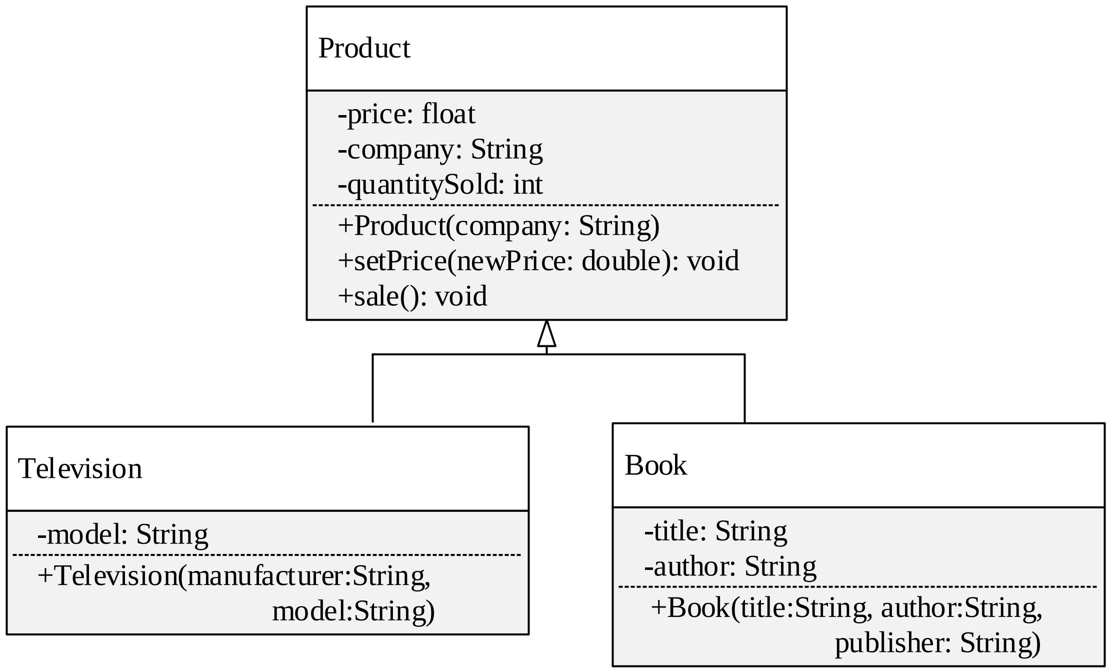

Product
-price: float
-company: String
-quantitySold: int
+Product(company: String)
+setPrice(newPrice: double): void
+sale(): void
Television
-model: String
+Television(manufacturer:String, model:String)
Book
-title: String
-author: String
+Book(title:String, author:String, publisher: String)

Figure 2.3: The derived classes, Television and Book, inherit from the base class, Product. The attributes common to Book and Television are available in Product.

```
public class Product {
  private String company;
  private double price;
  private int quantitySold;

  public Product(String company, double price) {
    this.company = company;
    this.price = price;
  }
  public void sell() {
    quantitySold++;
  }
  public void setPrice(double newPrice) {
    price = newPrice;
  }
  public String toString() {
    return "Company: " + company +  " price: " +
      price + " quantity sold " + quantitySold;
  }
}
```

Let us now construct `Television`, which extends `Product`. Any object of type `Television`, the subclass, can be thought of as having two parts: one that is formed from `Television` itself and the other from `Product`, the superclass. Thus, this object has four fields in it, `model`, `quantitySold`, `price`, and `company`. Typically, the code within the subclass is responsible for managing the fields within it, and the code in the superclass deals with the fields in the superclass.

Recall that objects are initialized via code in the constructor. When inheritance is involved, the part of the object represented by the superclass must be initialized *before* the fields of the subclass are given values; this is so because, the subclass is built from the superclass and thus the former may have fields that depend on the fact that superclass's attributes are initialized. An analogy may help: When a house is built, the roof is put only after the walls have been erected, which happens only after the foundation has been laid. Here, the roof depends on the walls, and therefore the latter must be built first.

To create a `Television` object, we to invoke a constructor of that class as below, where we pass the brand name, manufacturer name, and price:

```
Television set = new Television("RX3032", "Modern Electronics", 230.0);
```

Thus the constructor of `Television` must be defined as below.

```
public Television(String model, String manufacturer, double price) {
  // code to initialize the Product part of a television
  // code to initialize the television part of the object
}
```

The constructor of `Product` initializes the fields of a `Product` object. We therefore call that constructor, by the statement

```
super(/* parameters for superclass constructor go here*/)
```

Here `super` is a Java keyword. The call `super` with proper parameters always invokes the superclass's constructor. The superclass's constructor call can be invoked only as the very first statement from code within a constructor of a subclass; it cannot be placed after some code or placed in methods.

In this example, the parameters to be passed would be the manufacturer's name and price. The code for the constructor is then

```
public Television(String model, String manufacturer, double price) {
  super(manufacturer, price);
  // store the model name
}
```

The word `super` is a keyword in Java, which denotes the superclass. Invocation of the superclass's constructor is done using this keyword followed by the required parameters in parentheses.

The fields of the superclass are initialized before fields in the subclass. What this means in the context of object creation is that the constructor of `Television` can begin its work only after the constructor of the superclass, `Product`, has completed execution. Of course, when you wish to create a `Television` object you need to invoke that class's constructor, but the first thing the constructor `Television` does (and must do) is invoke the constructor of `Product` with the appropriate parameters: the name of the company that manufactured the set and the price.

The result of `super(manufacturer, price)` is, therefore, to invoke `Product`'s constructor, which initializes `company` and `price` and then returns. The `Television` class then gives a value to the `model` field and returns to the invoker.

As is to be expected, the class `Television` needs a field for storing the model name. We thus have a more complete piece of code for this class as given below.

```
public class Television extends Product {
  private String model;
  public Television(String model, String manufacturer, double price) {
    super(manufacturer, price);
    this.model = model;
  }
  public String toString() {
    return super.toString() + " model: " + model;
  }
}
```

The `toString()` method of `Television` works by first calling the `toString()` method of `Product`, which returns a string representation of `Product` and concatenates to it, the model name.

The ideas explained above are formalized as follows:

Suppose that C_1 and C_2 are two independent but related classes that we would like to connect through a hierarchy. We then extract the common aspects of C_1 and C_2 and create a class, say, B, to implement that functionality. The classes C_1 and C_2 could then be smaller, containing only properties and methods that are unique to them. This idea is called inheritance: C_1 and C_2 are said to **inherit** from B. B is said to be the baseclass or superclass, and C_1 and C_2 are termed derived classes or subclasses. The superclasses are generalizations or abstractions - we move toward a more general type, an “upward” movement - and subclasses denote specializations - toward a more specific class, a “downward” movement. The class structure then provides a hierarchy.

2.8.2 Inheriting from an Interface

Sometimes we have a group of classes, say, C_1, C_2, and C_3, all of which implement a set of common methods, say $m_1()$ and $m_2()$. Programmers may instantiate objects of type C_1 or C_2 or C_3 and invoke methods $m_1()$ and $m_2()$ on those instances. In such a situation it is useful to place the set of common methods (in this case $m_1()$ and $m_2()$) in an interface.

Earlier on, we defined an interface as a collection of methods that can be implemented by a class. Since an interface specifies the methods, and the implementing class implements them, an interface has been likened to a contract signed by the class implementing the interface. Implementing the interface is, therefore, viewed as a form of inheritance; the implementing class inherits an abstract set of properties from the interface. All the classes implementing the interface can be viewed as subclasses of the same superclass.

Java recognizes an interface as a type (as do several other object-oriented languages), which means that objects that belong to classes that implement a given interface also belong to the type represented by the interface. Likewise, we can declare a variable as belonging to the

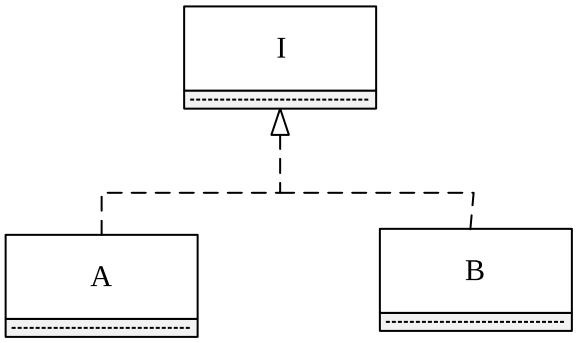


Figure 2.4: Interfaces. I is an interface that is implemented by both A and B.

type of the interface, and we can then use it to access the objects of any class that implements the interface.

```
public interface I {
  // details of I
}

public class  A implements I {
  //code for A
}

public class  B implements I {
  //code for B
}

I i1 = new A(); // i1 holds a reference to an object of type A

I i2 =  new B(); // i2 holds a reference to an object of type B
```

In the UML notation, this kind of a relationship between the interface and the implementing class is termed **realization** and is represented by a dotted line with a large open arrowhead that points to the interface as shown in Figure 2.4.

2.8.3 Polymorphism and Dynamic Binding

Consider a university application that contains, among others, three classes that form a hierarchy as shown in Figure 2.5. A student can be either an undergraduate student or a graduate student. Just as in real life, where we would think of an undergraduate or a graduate student as a student, in the object-oriented paradigm as well, we consider an `UndergraduateStudent` object or a `GraduateStudent` object to be of type `Student`. Therefore, we can write

```
Student student1 = new UndergraduateStudent();
Student student2 = new GraduateStudent();
```

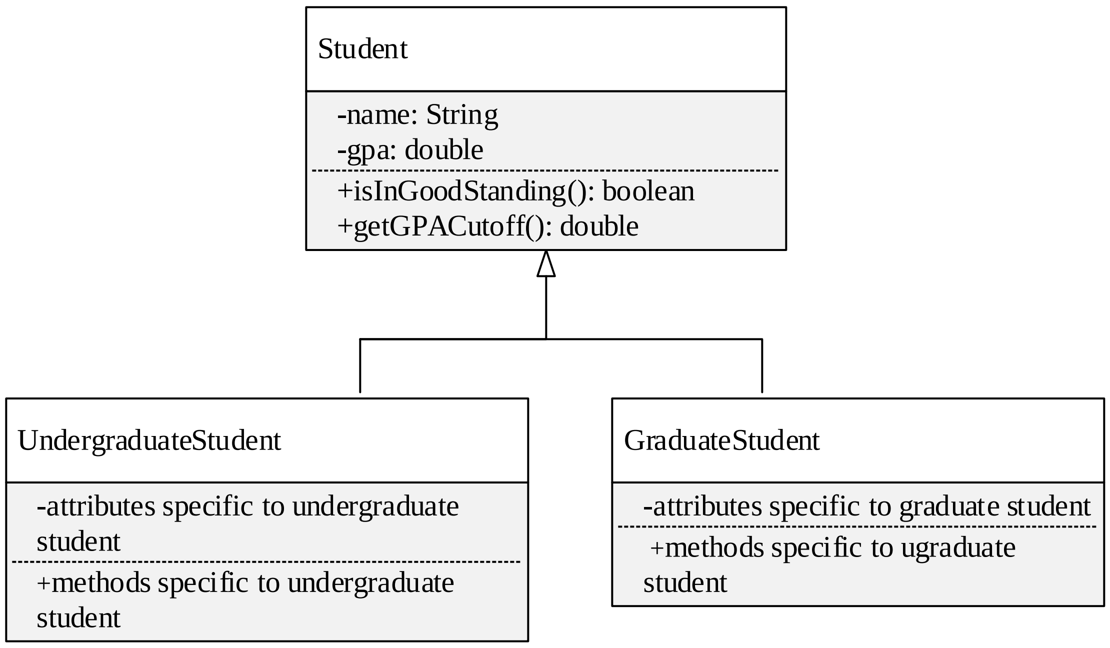

Student
-name: String
-gpa: double
+isInGoodStanding(): boolean
+getGPACutoff(): double
UndergraduateStudent
-attributes specific to undergraduate student
+methods specific to undergraduate student
GraduateStudent
-attributes specific to graduate student
+methods specific to ugraduate student

Figure 2.5: Student Hierarchy. The class Student holds the features common to all students, while GraduateStudent and UndergraduateStudent hold the features unique to each.

This is a powerful idea. We can now write methods that accept a `Student` and pass either an `UndergraduateStudent` or a `GraduateStudent` object to it as below.

```
public void storeStudent(Student student) {
  // code to store the Student object
}
```

We can then create `UndergraduateStudent` and `GraduateStudent` objects and pass them to the above method.

```
storeStudent(new UndergraduateStudent());
storeStudent(new GraduateStudent());
```

The parameter `student` in `storeStudent()` is a **polymorphic** reference. The adjective polymorphic is applied to anything that can take many forms. In this case, we see that the parameter `student` can take the form of `UndergraduateStudent` or a `GraduateStudent.` We can also write the following code:

```
Student student1 = new GraduateStudent();
Student student2 = new UndergraduateStudent();
```

Here we have two **polymorphic assignments**; objects of type `GraduateStudent` and `UndergraduateStudent` are stored in `student1` and `student2` respectively.

In real life, we usually don't think of a graduate student as an undergraduate student or vice-versa. In the same way, we cannot write the following code in Java.

```
UndergraduateStudent student1 = new GraduateStudent(); // wrong
GraduateStudent student2 = new UndergraduateStudent(); // wrong
```

Since we allow `Student` references to point to both `UndergraduateStudent` and `GraduateStudent` objects, we can see that some, but not all, `Student` references may point to objects of type `UndergraduateStudent`; similarly, some `Student` references may refer to objects of type `GraduateStudent`. Thus, we cannot write,

```
Student student1;
student1 = new UndergraduateStudent();
GraduateStudent student2 = student1; // wrong!
```

The compiler will flag that the code is incorrect.

But, the following code is intuitively correct, but the compiler will flag it as incorrect.

```
Student student1;
student1 = new GraduateStudent();
GraduateStudent student2 = student1; // compiler generates a syntax error.
```

The reason for this error is that the compiler *does not execute the code* to realize that `student1` is actually referring to an object of type `GraduateStudent`. It is trying to protect the programmer from the absurd situation that could occur if `student1` held a reference to an `UndergraduateStudent` object. It is the responsibility of the programmer to tell the compiler that `student1` actually points to a `GraduateStudent` object. This is done through a cast as shown below.

```
Student student1;
student1 = new GraduateStudent();
GraduateStudent student2 = (GraduateStudent) student1; //  O.K. Code works.
```

To reiterate, while casting a reference to a specialized type, the programmer must ensure that the cast will work correctly; the compiler will happily allow the code to pass syntax check, but carelessly-written code will crash when executed. See the following code.

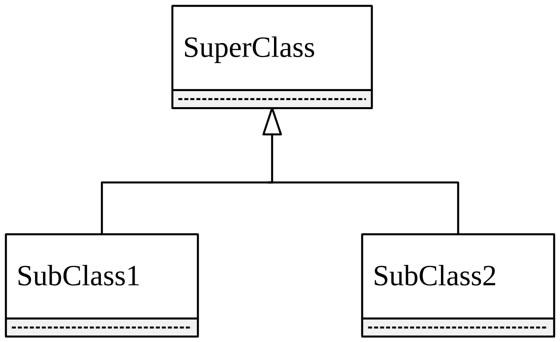
```
Student student1;
student1 = new UndergraduateStudent();
GraduateStudent student2 = (GraduateStudent) student1; // crashes
```

`Student1` does not point to a `GraduateStudent` object in the last line, so the system's attempt to cast the instance of `UndergraduateStudent` as an instance of `GraduateStudent` fails and a `ClassCastException` is thrown by the system. (`ClassCastException` is a kind of `RuntimeException` thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance.)

We have so far seen examples of polymorphic assignments. In one of these, we store a reference to an object of the class `GraduateStudent` in an entity whose declared type is `Student`. This equivalent to taking a bunch of bananas and storing them in a box labeled "fruit." The declared contents of the box (as given by the label) is fruit, just as the declared type of entity `student1` in the left-hand Side of the assignment is `Student`. By doing this, we have lost some information, since we can no longer find out what kind of fruit we have in the box without examining the contents. Likewise, when we operate on the entity `student1`, all we can assume is that it contains a reference to a `Student` object. *Thus, there is a loss of information in this kind of assignment.*

When can we store an object in a reference of another type?

Consider the simple hierarchy shown below:



The general rules are as follows.

1. Any object of type `SubClass1` or `SubClass2` can be stored in a reference of type `SuperClass`.
2. No object of type `SubClass1` (`SubClass2`) can be stored in a reference of type `SubClass2` (`SubClass1`).
3. A reference of type `SuperClass` can be cast as a reference of type `SubClass1` or `SubClass2`. If the cast fails, a `ClassCastException` is thrown by the system.

A reference is able to point to objects of different types as long as the actual types of these objects *conform* to the type of the reference. The above rules informally give the notion of **conformance.**
It is instructive to compare assignments and casts given above with the rules for assignments and casts of variables of primitive types. Some type conversions, for example, from `int` to `float`, don't need any casting: `float` variables have a wider range than `int` variables. This is analogous to the situation we have with polymorphism, where the set of all superclass objects is a superset of the objects in any subclass. Some others, `double` to `int` being an instance, are fine with casting; however, the programmer must be prepared for loss of precision. This is similar to the loss of information that happens when a subclass object is stored in a superclass reference. And the rest - any casts from (to) `boolean` to (from) any other type - are disallowed always.

The second kind of polymorphic assignment is the one where we moved a reference from an entity whose declared type is `Student` to an entity whose declared type is `GraduateStudent`.

(This would amount to taking the fruit out of the box labeled "fruit" and putting them in the box labeled "bananas;" we do this only if we know that the fruit are bananas.) As we saw with our cast and exception, this can only be done after ensuring that the entity being used is of type `GraduateStudent`. This is therefore an operation that "recovers" information lost in assignments of the previous kind.

What we conclude from this is that using polymorphism does result in a loss of information at run time. Why, then, do we use this? The answer lies in **dynamic binding.** This ability allows us to invoke methods on members of a class hierarchy without knowing what kind of specific object we are dealing with. To make a rough analogy with the real world, this would be like a manager in a supermarket asking an assistant to put the fruits on display (this is analogous to applying the "display" method to the "fruit" object). The assistant looks at the fruit and applies the correct display technique. Here the manager is like a client class invoking the "display" method and the assistant plays the role of the system and applies dynamic binding.

To get a concrete understanding of how dynamic binding works, let us revisit the example of the `Student` hierarchy. The code for `Student` - may be, once again, in a simplistic manner - is written as below.

```
public abstract class Student {
  private String name;
  private double gpa;
  // more fields
  public Student(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
  public boolean isInGoodStanding() {
    return (gpa >= getGPACutoff());
  }
  public abstract double getGPACutoff();
  // more methods
}
```

We assume that periodically, perhaps at the end of each semester or quarter, the university will check students to see if they are in "good standing." Typically, it would mean ensuring that the student is progressing in a satisfactory manner. We assume that for a student, good standing means that the student's GPA meets a certain minimum requirement. The minimum GPA expected of students may change depending on whether the student is an undergraduate or a graduate student. The method `getGPACutoff()` returns the minimum GPA a student must have to be in good standing. We will assume that this value is 2.0 and 3.0 for undergraduate and graduate students respectively. Note that the method is declared abstract in the `Student` class.

Let us now focus on the code for `UndergraduateStudent`, which is given below.

```
public class UndergraduateStudent extends Student {
  public UndergraduateStudent(String name) {
    super(name);
  }
  public double getGPACutoff() {
    return 2.0;
  }
}
```

The constructor gets the name of the student as its only parameter and calls the superclass's constructor to store it. Since this is a non-abstract class, the `getGPACutoff` method, which returns the minimum GPA is implemented.

All of the public and protected[2] methods of a superclass are inherited in the two subclasses. So the method `isInGoodStanding()` can be instantiated on an instance of `UndergraduateStudent` as well. Thus the following code is valid.

```
UndergraduateStudent student = new UndergraduateStudent("Tom");
// code to manipulate student
if (student.isInGoodStanding()) {
   // code
} else {
   // code
}
```

When the method is called, the `isInGoodStanding()` method in the superclass `Student` will be invoked.

Finally, we have the code for the class for graduate students. The constructor for the class is quite similar to the one for the `UndergraduateStudent` class. To make the class non-abstract, this class, too, should have an implementation of `getGPACutoff()`. In addition, we assume that to be in good standing, graduate students must meet the requirements imposed on all students and they cannot have more than a certain number of courses in which they get a grade below B.

What we would like is a redefinition or **overriding** of the method `isInGoodStanding()`. Overriding is done by defining a method in a subclass with the same name, return type, and parameters as a method in the superclass, so the subclass's definition takes precedence over the superclass's method. The modified code for the `isInGoodStanding()` method is shown below.

```
public class GraduateStudent extends Student {
  public GraduateStudent(String name) {
    super(name);
  }
  public double getGPACutoff() {
    return 3.0;
  }
  public boolean isInGoodStanding() {
    return super.isInGoodStanding() && checkOutCourses();
  }
```

[2]Protected access will be explained shortly.

```
  public boolean checkOutCourses() {
    // implementation not shown
    // checks to ensure that the student has
    // a grade below B in  in at most 3 courses
  }
}
```

Now, suppose we have the following code.

```
GraduateStudent student = new GraduateStudent("Dick");
// code to manipulate student
if (student.isInGoodStanding()) {
   // code
} else {
   // code
}
```

In this case, the call to `isInGoodStanding()` results in a call to the code defined in the `GraduateStudent` class. This in turn invokes the code in the `Student` class and makes further checks using the locally declared method `checkOutCourses` to arrive at a decision.

Recall the `StudentArrayList` class we defined in Section 2.5, which stores `Student` objects. The method to add a `Student` in this class looked as follows:

```
  public void add(Student student) {
    // code
  }
```

Since a `Student` reference may point to a `UndergraduateStudent` or a `GraduateStudent` object, we can pass objects of either type to the `add()` method and have them stored in the list. For example, the code

```
  StudentArrayList students = new StudentArrayList();
  UndergraduateStudent student1 = new UndergraduateStudent("Tom");
  GraduateStudent student2 = new GraduateStudent("Dick");
  students.add(student1);
  students.add(student2);
```

stores both objects in the list `students`.

Suppose the class also had a method to get a `Student` object stored at a certain index as below.

```
  public Student getStudentAt(int index) {
    // Return the Student object at position index.
    // If index is invalid, return null.
  }
```

Let us focus on the following code that traverses the list and checks whether the students are in good standing.

```
for (int index = 0; index < students.size(); index++) {
  if (students.getStudentAt(index).isInGoodStanding()) {
    System.out.println(students.get(index).getName()
                          + " is in good standing");
  } else {
    System.out.println(students.getStudentAt(index).getName()
                          + " is not in good standing");
  }
}
```

We assume that students Tom, an undergraduate student, and Dick, a graduate student, are in the list as per the code given a little earlier. The loop will iterate twice, first accessing the object corresponding to Tom and then getting the object for Dick. In both cases, the `isInGoodStanding()` method will be called.

What is interesting about the execution is that the system will determine at run time the method to call, and this decision is based on the actual type of the object. In the case of the first object, we have an instance of `UndergraduateStudent`, and since there is no definition of the `isInGoodStanding()` method in that class, the system will search for the method in the superclass, `Student`, and execute that. But when the loop iterates next, the system gets an instance of `GraduateStudent`, and since there is a definition of the `isInGoodStanding()` method in that class, the overriding definition will be called.

This is a general rule: Whenever a method call is encountered, the system will find out the actual type of the object referred to by the reference and see if there is a definition for the method in the corresponding class. If so, it will call that method. Otherwise, the search proceeds to the superclass and the process gets repeated. The actual code to be executed is bound dynamically, hence this process is called dynamic binding.

The above code shows the power of dynamic binding. The reader should understand that the code indeed performed correctly. In our calls to `isInGoodStanding()`, we were unaware of the type of objects. Simply by examining the code that calls the method, we cannot tell which definition of the `isInGoodStanding()` method will be invoked, i.e., *dynamic binding gives us the ability to hide this detail in the inheritance hierarchy.*

2.8.4 Protected Fields and Methods

Consider the hierarchy as shown in Figure 2.6. `ClosedFigure` has an attribute `area`, which stores the area of a `ClosedFigure` object. Since the classes `Polygon` and `ClosedCurve` are of the type `ClosedFigure`, we would like to make this attribute available to them. This implies that the attribute cannot be private; on the other hand making it public could lead to inappropriate usage by other clients. The solution to this is found in the `protected` access specifier. Loosely speaking, what this means is that this field can be accessed by `ClosedFigure` and its descendants as shown below.

```
public class ClosedFigure extends Figure {
  protected  double area;
  //other fields and methods
}

public class Polygon extends ClosedFigure {
```

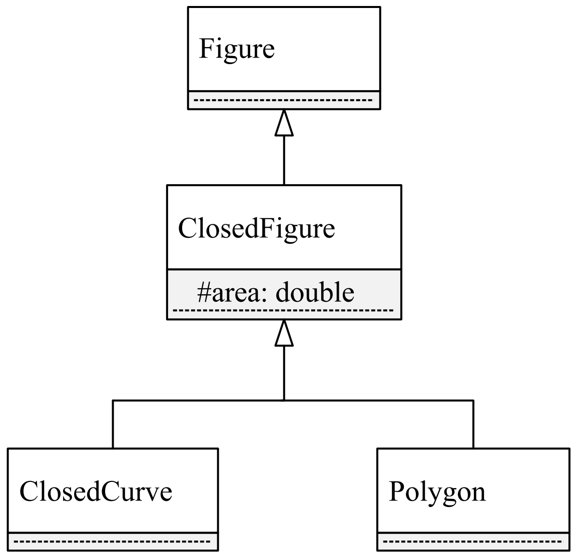


Figure 2.6: Figure Hierarchy. The class ClosedFigure has a protected attribute area, which is available to ClosedCurve and Polygon.

```
    public void insertVertex(Point point, int index) {
      // code to insert vertex point at position index
      area = computeArea();
    }
    private double computeArea() {
      //code to compute the area
    }
  }
```

Declaring it protected ensures that the field is available to the descendants, but cannot be accessed by code that resides outside the hierarchy rooted at `ClosedFigure`.

The above example is a simple one, since the class `Polygon` is modifying the field of a `Polygon` object. Consider the following situation.

```
 public class ClosedCurve {
   // other fields and methods
   public void areaManipulator(Polygon polygon) {
     polygon.area = 0.0;
   }
}
```

Here the class `ClosedCurve` is modifying the area of a polygon. Our loose definition says that `area` is visible to `ClosedCurve` which would make this valid. However, `ClosedCurve` is a sibling of `Polygon` and is therefore not a party to the design constraints of `Polygon`, and providing such access could compromise the integrity of our code. In fact, an unscrupulous client could easily do the following:

```
public class BackDoor extends ClosedFigure {
  public void setArea(double area, ClosedFigure someClosedFigure) {
    someClosedFigure.area = area;
  }
}
```

We therefore need the following stricter definition of protected access.

The code residing in a class A may access a protected attribute of an object of class B only if B is at least of type A, i.e., B belongs to the hierarchy rooted at A.

With this definition, methods such as `setArea` in `BackDoor` would violate the protected access (since `ClosedFigure` is not a subclass of `BackDoor`) and the violation can be caught at compile time. The compiler will not raise an objection if `someClosedFigure` is cast as `BackDoor` as shown below:

```
((BackDoor) someClosedFigure).area = area;
```

If `someClosedFigure` contained a reference to a `Polygon` object, the cast would fail at run time, preventing the access to the protected field.

2.9 Run-Time Type Identification

An inheritance hierarchy can involve a large number of classes. In addition, more classes can be added to the hierarchy over a period of time to allow for new functionality. Although dynamic binding is a powerful tool that can take care of a lot of situations, there are situations when it is not sufficient. Such situations typically happen when the original designer did not anticipate all possible changes. As a simple example, consider a `Shape` class with several subclasses like `Square` and `Circle`, and a `ShapeList` class, which holds a collection of `Shape` objects. If we access an item from this collection, we know that it will be of type `Shape`, but we do not know whether it will be a `Square` or a `Circle`. In a typical application, one would expect that new kinds of shapes would be added over time.

Say we have a new client application that needs to know the number of `Circle` objects in a `ShapeList` collection. We cannot modify `ShapeList`, since other applications are using it. The problem can be solved if the client iterates over all the `Shape` objects and checks which ones are circles. In such a situation, one approach would be to perform dynamic type checking, or **Run-time Type Identification (RTTI)**. Java provides the `instanceof` operator, which could be used as follows:

```
public class ShapeClient {
  private ShapeList shapeList = new ShapeList();
  public void countCircles() {
    int count =0;
    Iterator allShapes = shapeList.getShapes();
    while (allShapes.hasNext()) {
      if (allShapes.next() instanceof Circle)
        count++;
    }
    System.out.println("count " + count);
  }
}
```

The `instanceof` operator returns `true` if the object returned by `allShapes.next()` is an instance of the class `Circle`. An alternate approach would be to downcast the object as a `Circle` and increment the counter in a `try` block and then catch the `ClassCastException`, as shown below.

```
public void countCircles() {
  int count = 0;
  Circle circle;
  Iterator allShapes = shapeList.getShapes();
  while (allShapes.hasNext()) {
    try {
      circle = (Circle) (allShapes.next());
      count++;
    } catch(ClassCastException cce){
    }
  }
  System.out.println("count " + count);
}
```

As is obvious, the first approach is more elegant. A third approach is possible in Java through the reflection mechanism which we shall see in the next section. Use of RTTI can be minimized by employing good design principles. (In the above example, if the designer had anticipated the need for adding new functionality, the `Shape` class could have been designed to *accept visitors*, which would have avoided the need for RTTI; we shall learn about visitors in a later chapter.) However, it is not possible to anticipate all situations, and when the need arises, RTTI provides a quick (and somewhat dirty) way of dealing with them.

2.10 The `Object` Class

Java has a special class called `Object` from which every class inherits. In other words, `Object` is a superclass of every class in Java, and it is at the root of the class hierarchy. Thus we can declare a reference to be of type `Object`, and store a reference to any object, without regard to the class to which the object belongs. The `Object` class serves two important functions: providing utility methods, and system support. Utility methods are for general purpose operations that have been found to be useful for all classes. By defining these in `Object`, Java provides a shared understanding of their syntax and semantics. Java runs on a virtual machine(JVM), which is connected to the operating system. The `Object` class provides some methods that the JVM can invoke, without any knowledge of the class to which the object belongs.

2.10.1 Using `Object` as a General Reference

When we declare a class in Java that does not extend any other class, it automatically becomes a direct descendant of the `Object` class. Therefore every class in Java is either a direct or indirect descendant of `Object`. From our knowledge of polymorphic assignments, we can see that a variable of type `Object` can store the reference to an object of any other type. The following code is thus legal.

```
Object anyObject;
anyObject = new Student();
anyObject = new Integer(4);
anyObject = "Some string";
```

In the above, the variable `anyObject` first stores a `Student` object, then an `Integer` object, and finally a `String` object.

We can use an `Object` reference when we have to refer to objects that belong to any one of several unrelated classes. An example of where this ability is needed is the `equals` method (discussed in the next subsection) which checks if two object references are referring to the same object. Since this can be applied to any two objects, the parameter to the method is of type `Object.`

Having an identifier that can reference any object is very unusual in well-designed applications; in fact, such a situation can often indicate that some principle of good design is being violated. In earlier versions of Java (upto Java SE 1.4) there was no generics. In those versions, the `Object` class was used to create library classes that could be used for a variety of applications. For instance, in the `LinkedList` class in JDK 1.4, we see the method

```
void add(int index, Object element)
```

This method inserts the specified element at the specified position in this list. The parameter `element` is declared as belonging to the class `Object`. A programmer could employ an instance of this class, and add items belonging to any class. Such an approach was replaced by generics in Java SE 1.5, but may be encountered in legacy code.

2.10.2 Utility Methods Provided by `Object`

Java designers have decided that some methods are useful for all classes and should be invoked in a uniform way. That is accomplished defining these methods as a part of the `Object` class.

The `equals()` and `hashCode()` Methods

The `equals()` method provides a mechanism for checking if two *objects* are equal in value. Given any two variables of the same *primitive type*, it is easy for Java to decide whether they are equal: the variables are equal if they have the same value, which can be checked using the `==` operator. However, consider a class such as `Student`, which is a user defined class. How should we decide that two `Student` objects are equal? Here are some interpretations of equality:

1. *The two objects are equal if they occupy the same physical storage.* In this case the language would have to provide some mechanism to determine if this is the case.

2. *The two objects are equal if all the corresponding fields of the objects are equal.* This is a recursive definition. For example, in the `Student` class, the fields are `name`, `address`, and `gpa`. To decide if the `name` fields of two objects are equal, we have to know when two `String` objects are equal. Since `gpa` is a `double`, that field presents no problems.

3. *The two objects are equal if some subset of the corresponding fields of the objects are equal.* This would only check some key field, say, the `id` of a person.

The Java language supports the first interpretation through the `==` operator. To use one of the other interpretations, Java provides an `equals()` method which can be defined by the author of the class; if this is not defined, `equals()` method defaults to the `==` operator. The `equals()` method has the following template:

```
public boolean equals(Object someObject) {
  // implement the policy for comparison in this method.
  // return true if and only if this object is equal to someObject
}
```

We are given two objects: `this`, the one on which we invoke `equals()`, and `anObject`, which is an arbitrary object, and can be of any type. The author of the class is free to choose a mechanism that decides whether `anObject` is equal to `this`.

For example, let us say that a `Student` object is equal to another object only if that object is a `Student` object, and the names are equal, and they have the same address.

```
public boolean equals(Object anObject) {
  Student student = (Student) anObject;
  return student.name.equals(name) && student.address.equals(address);
}
```

The first thing that the `equals()` method does is to cast the incoming object as a `Student` object; the resulting reference is used to access all of the members of the corresponding `Student` object, and, in particular, the `name` and `address` fields.

After the cast, we check if the `name` field of the cast object is equal to the name field of `this`, which in our example is `student1`; note that we are doing this by invoking the `equals()` method on the object `student.name`, which is a `String`; thus, we are invoking the `equals()` method of the `String` class. It turns out that the `equals` method of the `String` class is defined to return `true` if and only if every character in one string is equal to the corresponding character of the other string.

The address fields are compared in a similar way. The method returns true if and only if the two fields match. The method is placed inside the `Student` class and is invoked as below.

```
Student student1 = new Student("Tom");
student1.setAddress("1 Main Street");
// some other code
Student student2 = new Student("Tom");
student2.setAddress("1 Main Street");
// more code
if (student1.equals(student2)) {
  System.out.println("student1 is the same as student2");
} else {
  System.out.println("student1 is not the same as student2");
}
```

After creating the two `Student` objects with the same name and address, we invoked the `equals` method on `student1` with `student2` as the actual parameter.

There is more to the issue of making comparisons reliable. When a class redefines the `equals()` method, it is expected that the redefinition satisfies the following conditions:

1. *Implements an equivalence relation.* For any two objects belonging to `C` the boolean relation defined by `equals()` should be *symmetric, reflexive* and *transitive.*

2. *Is consistent.* If two objects belonging to `C` are compared, we will get the same result as long as the objects have not been modified.

3. *Distinguishes the* `null` *reference.* For any reference, `c` of type `C`, `c.equals(null)` should return false as long as `c` is not `null`.

These requirements mean that the `equals()` method should return `true` only if the objects being compared belong to the exact same class. Consider a different implementation of `Student`, in which we use the ID as a unique key. We may then use the ID to search a `StudentList` for a particular student. It is tempting to use the `equals()` to compare a given ID with the ID of each `Student` in the list. The following code in the class `Student` would compile correctly:

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

  // other code deleted

  public boolean equals(Object anObject) {
    return ((String)obj).equals(id);
  }

}
```

The `equals()` method returns true if the current object's `id` field matches the given `String`. The method can be invoked as:

```
Student s = new Student(``John'', ``X1'');
String ids = new String(``X1'');
if (s.equals(ids)) {
// some code
}
```

The problem in the above code is that `equals()` compares `s`, a `Student`, with `ids` which is a `String`, and finds them to be equal. Such a relationship between objects is *not symmetric*. Symmetry implies that if we swap the two entities being compared for equality, the result should be the same. In the above code, we invoked `equals()` on `s` and passed `ids` as a parameter. If we invoke `equals()` on `ids` and pass `s` as a parameter, we get the following expression:

```
ids.equals(s)
```

The above expression will use the `equals()` method of `String` with a `Student` as parameter and therefore find them to be unequal. The `equals()` method in `Student` therefore violates the condition that `equals()` should implement an equivalence relation. Clearly a pair of objects cannot be included in an equivalence relation if they belong to different classes. Our first of order of business in the `equals()` method should therefore be to verify that `anObject`, is of type `Student`, which can be done using the `instanceof` operator. This guard should be placed at the start of the template for `equals()`; our earlier code for checking equality of `Student` objects by comparing `name` and `address` should therefore be rewritten as:

```
  public boolean equals(Object anObject) {
    if (!(anObject instanceof Student)) {
         return false;
    }
    Student student = (Student) anObject;
    return student.name.equals(name) && student.address.equals(address);
  }
```

The definition of `equals()` should also be in agreement with the way in which objects of the class are placed in hash-based data structures. If two objects considered equal can hash to different locations, our search operations will not be correct. To address this, the `Object` class includes a `hashCode()` method which must be defined to comply with the definition of `equals()`. Every object in Java has a **hashcode**, which is a 32-bit signed integer, which is used when the object has to be managed by a hash-based data structure. When we invoke the `hashCode()` method on an object, it returns an `int`, which is the hashcode for the object. The requirements on this method are:

1. the hashcode of an object should not change during the execution of a program as long as no information used in the computation of the `equals()` method has been modified.
2. if we have objects, `o1` and `o2`, such that `o1.equals(o2)` returns `true`, then they must have the same hashcode. However, if `o1.equals(o2)` returns `false`, `o1` and `o2` are not required to have different hashcodes.

Enforcing this relationship between `equals()` and `hashcode()` is not trivial. Consider again our class `Student`, in which two `Student` objects to be considered equal if their IDs are equal. To comply with the above requirements, we have to ensure that having equal IDs would ensure equal hashcodes. One way to do that would be to use the `hashcode()` method for `String`.

```
public class Student {
  // other code deleted

  public int hashCode()  {
    return  id.hashcode();
  }

}
```

The `toString()` Method

The `toString()` method, as we have seen earlier, gives us a textual representation of the object. By including this in the `Object` class, operations like `System.out.println()` can be completed without worrying about the specific object that is being printed to the output stream.

The `getClass()` and `clone()` Methods

In the field of computing, **reflection** is defined as the ability of a process to get knowledge of, and examine its own structure and behavior. Reflection should be used only by developers who have a strong grasp of the language and of the software development process. It is a very powerful technique that allows us to to perform operations which would otherwise be impossible.

Central to Java's support for reflection is the `Class` object. For each class, Java defines a `Class` object that represents the runtime class, and contains all the information specific to the class. Say we have a class `C` and an object `c1` belonging to `C`. Any application program can access the `Class` object for `C` through the `getClass()` method.

```
C c1  = new C();
Class  c = c1.getClass();
```

The object `c` is declared to be a reference to a `Class` object; after the second statement, it holds a ref to an object that contains all the information about the class `C`. The `Class` object provides us with several methods through which can get information about the class.

A simple application of reflection is in run time type identification using the `getName()` method in `Class`. This method returns a `String` which represents the name of the class. In the context of the above code, `c.getName()` returns the `String` "`C`", which is the name of `C`. We can use this method as an alternative to the `instanceof` operator. As an example, we had a piece of code in the previous section with the conditional expression

`(allShapes.next() instanceof Circle)`

This expression can be replaced by

`(allShapes.next().getClass().getName() == "Circle")`

Here we are explicitly getting the name of the class to which the `Shape` object belongs, and checking it against the string "Circle". One drawback of using reflection for RTTI is that the code with typographical errors can compile and return an incorrect result. For instance the conditional

`(allShapes.next().getClass().getName() == "Circule")` will compile, but will fail to identify any object as belonging to `Circle`. Such an error will be detected by the compiler when we employ the `instanceof` operator, unless we have also defined a class named `Circule`.

Cloning in the context of object oriented programming, refers to the making of an exact copy of an object. By providing the `clone()` method as part of `Object` Java provides a standard way of invoking this feature. It is not required that all classes provide a `clone()` method. Hence the method is declared to be `protected`, allowing all user-defined classes to deal with clonability as they see fit. However, it is expected by convention that any subclass of `Object` that makes itself clonable will invoke `super.clone()`. If this convention is followed, we can ensure that the `getClass()` method will work correctly for the cloned copy of the object. Since an object can contain references to objects of other classes, making a clone is complicated. We shall look at this in greater detail in a later chapter.

2.10.3 Methods for System Support

Since the JVM is like a virtual computer, it performs operating system operations, and also communicates with the underlying physical computer system. As a result there are several operations that the JVM needs to perform on individual objects. These operations must necessarily be invoked without knowledge of the class to which the object belongs, and are therefore placed in the `Object` class. By overriding these methods, a user-defined class can send specific instructions to the JVM and the operation system.

Memory Management

Computer systems have two kinds of memory: statically allocated memory on the *stack*, and dynamically allocated memory on the *heap.* Static allocation is done when the program starts execution, and the amount of memory allocated does not change during execution. Dynamic allocation happens as the program executes, and the amount of memory allocated varies depending on the specific execution path. Memory management in Java involves dynamically allocating space on the heap for new objects and removing unused objects to make space for new object allocations.

Garbage collection is the process of freeing unused space for new objects. An object is considered to be **garbage** if it cannot be reached, directly or indirectly through any reference. Consider the following code that creates a `Student` object with name “John” and ID “X1” and adds the object to a list:

```
Student s1 = new Student(‘‘John’’, ‘‘X1’’);
StudentList slist = new StudentList();
slist.add(s1);
```

At this point the `Student` object (with name “John” and ID “X1”) exists on the heap, and the reference to this object is stored in two places: in the variable `s1`, and inside the `StudentList` object referenced by `slist`. This means that both `s1` and `slist` store the address of the memory word(s) in which this `Student` object is stored. Later in the execution, we may no longer need to track this student, and may execute the code

```
slist.remove(‘‘X1’’)
```

which removes the reference to the `Student` object with ID "X1" from `slist`. This object is no longer needed, but `s1` still holds a reference to the object. If we free the memory word(s) in which the `Student` object was stored, those words may be allocated to some other object. However, `s1` stores the address of those words, and the statement `System.out.println(s1.getName());` will access those word(s) and print out whatever data is stored in there. In such a situation, `s1` is termed a **dangling reference.**

To avoid creating dangling references, JVM employs a sophisticated **garbage collection** process to detect the garbage object objects. It may however be possible that a useful object is identified as garbage, or that a object requires some specific cleanup operations to be performed before it is discarded. To take of these possibilities, the JVM invokes the `finalize()` method on the object, when it is identified as garbage for the first time. If the object is identified as garbage a second time, it is discarded. The finalize method is never invoked more than once by a Java virtual machine for any given object.

Thread Support

Most modern operating systems use the concept of **threads**, which serve as a mechanism for dividing a program into two or more simultaneously (or pseudo-simultaneously) running tasks. The JVM allows an application to have multiple threads of execution running concurrently. Threads in Java are associated with objects, and it follows that the `Object` class should provide methods for supporting thread-based execution.

The `notify()` method is used when we have situations where multiple threads want to access an object, but only one thread can access the object at a given time. The threads that do not have the access to the object are said to be **waiting**. The `notify()` method is used to wake up (reactivate) one of the waiting threads. The `notifyAll()` method is is used to wake up all the waiting threads. Whereas these two methods are used to reactivate waiting threads, the `wait()` method is invoked when a thread that is currently accessing the object has to be asked to wait. Threads are discussed in greater detail in Chapter 8.

2.11 Iterating over the Items in a List

In many applications, we need to maintain collections, which are objects that store other objects. For example, a telephone company system could have a collection object that stores an object for each of its customers; an airline system will likely maintain information about each of its flights and references to them may be stored in a collection object. In all such situations there is often a need to go through each item in the collection and perform a common task. For instance, a telephone company may want to generate a monthly billing statement for all its customers; this would require the billing method to iterate over the collection of customers.

Depending on the type of application, the actual data structure employed may differ. Popular data structures that implement collections include linked lists, queues, stacks, double-ended queues, binary search trees, B-Trees, and hash tables. One way to iterate over the collection is to access the underlying data structure and work through it. This requires the client class (the class holding a reference to a collection class and wanting to do the iteration) to access the data structure used by the collection. Such an approach makes things difficult for the client. It is therefore considered good practice for the collection class to provide a mechanism for iterating over all the items.

2.11.1 Need for a Common Mechanism

Consider the class `StudentList` that stores a list of `Student` objects. Say that some university process requires going through all the objects in the list and performing some action on each. This process would use an algorithm that does the following:

```
get the first student object from the list and process it
While (the list has more students)
     get the next student object from the list and process it
```

If the `StudentList` object was a `StudentArrayList`, this code would look something like this:

```
StudentArrayList slist;
Student s;
// other code deleted

int i = 0;
While (i < slist.size();) { //check if list has more
    s = slist.getStudent(i); //get next student
    s.process();
    i++;  // move forward
 }
```

If the `StudentList` object was a `StudentLinkedList`, the code would look something quite different:

```
StudentLinkedList slist;
Student s;
StudentNode sn;
// other code deleted

sn = slist.getHead();
While (sn != null) { //check if list has more
    s = sn.getData(); //get next student
    s.process();
    sn = sn.getNext(); // move forward
 }
```

Such an approach suffers from two problems:

1. *The code for processing has to be aware of how* `StudentList` *is implemented.* We can see that the methods being invoked in the code are different for `StudentArrayList` and `StudentLinkedList`.

2. *The code for traversing the list has to be modified if we change the implementation for* `StudentList`*.* If a new data structure is available, it is likely to have different methods.

One possible solution is that the `StudentList` interface specifies methods for going through the list. These methods could be as follows:

1. *A method* `void startIteration()` *that performs necessary housekeeping so to begin the iteration.* For the `StudentArrayList` this would set an index to zero; for the `StudentLinkedList` it would set a pointer to the head of the list.

2. *A method* `Student getNext()` *that returns the next Student from the list.* For the `StudentArrayList` this would return the `Student` object from the array location specified by the index; for the `StudentLinkedList` it would return the `Student` object in the `StudentNode` that the pointer is pointing to.

3. *A method* `boolean hasMore()` *that returns* `false` *if there are no more elements,* `true` *otherwise.* For the `StudentArrayList` this would return `true` only when the index is less than the size; for the `StudentLinkedList` it would return `true` only when the pointer is not null.

With these methods, our code for processing a list of students would look the same for both `StudentArrayList` and `StudentLinkedList`. However, this is not a satisfactory solution for the following reasons:

- *The traversal process is continuously modifying the collection object.* Although the request for traversing the list originates outside outside the collection object, there are variables stored inside the collection that are actually controlling the process. Thus the collection object is being modified by this process, which is not desirable.

- *We may have multiple traversals at the same time.* There is only one variable that keeps track of where the traversal has reached. If two or more simultaneous traversals are needed, things will get complicated.

- *There are several kinds of iterators.* Some collections may provide both forward and backward iterators (see exercises). If a collection has a binary tree-like structure we have can have three kinds of iterators for each of the three traversals. This interface cannot accommodate that.

Clearly this a very common problem, and we need to look for a solution that will work in all cases. We therefore try to identify standard patterns that can be used in various situations. A literate programmer or software developer would have knowledge of the relevant standard patterns and would employ the same when building the software. When working with classes and objects, these standard patterns are commonly referred to as **Design Patterns.**

2.11.2 What are Design Patterns?

As one may expect, a software engineer who has had experience developing a number of application systems is able to utilize the expertise so gained in future projects. Although two applications may not be alike and exhibit relatively little similarity at the outset, delving deeper into design may exhibit a number of similar issues. Working on a variety of projects, a software engineer gets exposure to problems that are common to multiple scenarios, which hones his/her ability to identify repeated instances of problems and spell out solutions for them fairly quickly. From an object-oriented perspective, what it means is that two different applications may provide design issues that are alike; the solutions may involve the development of a set of classes with similar functionalities and relationships. Thus the class structures for the two sub-problems may end up being the same although there may be differences in the details.

An example from the imperative paradigm may help the reader better understand the above discussion. Consider two applications, one a university course registration system and the other a Human Resource (HR) system for some organization. In the first example, we may wish to provide screens that allow students to register for classes that can be selected from a list. Let us say that we will list courses sorted according to the departments in the university and that within the department, the courses will be listed in ascending order of course identifiers; the information is to be retrieved from disk before it can be displayed. In the second application, let us assume that we want to retrieve employee related information from disk and print the information in the sorted order of departments, and within each department in the ascending order of employee names. Although the applications are quite different, the scenarios have similarity: both involve reading information, which is data related to some application, from disk and then sorting the data based on some fields in it. An efficient sorting mechanism should be used in both cases. We could envisage similar processing in many other applications as well. A professional who has some experience in application design and is conversant with such scenarios should be able to identify the proper approach to be taken for solving the problem and employ it effectively.

In object-oriented systems, we break up the system into objects and develop classes that serve as blueprints for creating objects. Therefore, unlike the imperative world where we need to recognize the appropriate algorithms for solving a problem, the task in object-oriented systems is to recognize the necessary classes, interfaces, and relationships between them for solving a specific design problem. Such an approach, which can then be tailored to solve similar design problems that recur in a multitude of applications, is called a **design pattern**.

Here are some quotes from the literature:

> "Design patterns are partial solutions to common problems, such as separating an interface from a number of alternate implementations, wrapping around a set of legacy classes, protecting a caller from changes associated with specific problems. A design pattern is composed of a small number of classes that, through delegation and inheritance, provide a robust and modifiable solution. These classes can be adapted and refined for the specific system under construction."[1]
>
> "A pattern is a way of doing something, or a way of pursuing an intent."[7]
>
> " . . . a software design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system. (Wikipedia)"

A number of design patterns are known, and as one may expect, they vary in the difficulty to comprehend and employ. We will introduce several design patterns throughout this book, and most of them are presented in a context that explains their significance.

2.11.3 The Iterator Pattern

Going back to our problem, what common mechanism should each collection provide to facilitate traversal? The traversal is a process, and we would like to keep this process separate from the data on which it is acting, i.e., the collection. We therefore *encapsulate the traversal process as an object*, and store the necessary data within that object. This object has a standard interface that the client can access. At the same time, it has knowledge of and access to the implementation details of the collection over which the iteration is taking place. Changes are inevitable in almost all applications, so we must ensure that these changes do not have widespread ramifications. If every collection class implements the `iterator()` method that returns an object of type `Iterator`, clients can use the iterator object to traverse the collection making the process independent of the collection implementation. This insulates the client code from changes in the collection class.

For example, if `myCollection` refers to an object of type `Collection`, the expression

```
myCollection.iterator()
```

returns an `Iterator` object.

The iterator supports a method called `next()`, which returns an element from the collection each time it is called. No element is returned more than once and if enough calls are made, all elements will be returned. The caller may check whether all elements have been returned by using the method `hasNext()`, which returns `true` if not all elements have been returned.

Thus, in our scheme, we have the following classes and interfaces:

1. `Collection`, an interface that allows the usual operations to add and delete objects, plus the method `iterator()` that returns an iterator object.

2. `Iterator`, an interface that supports the operations `hasNext()` and `next()` described earlier.

3. Implementation of the `Collection` interface. Obviously, every implementation must implement the `iterator()` method by creating an `Iterator` object and returning it.

4. Implementation of the `Iterator` interface. This class must cooperate with the code in (3) above to properly access and return the elements of the collection.

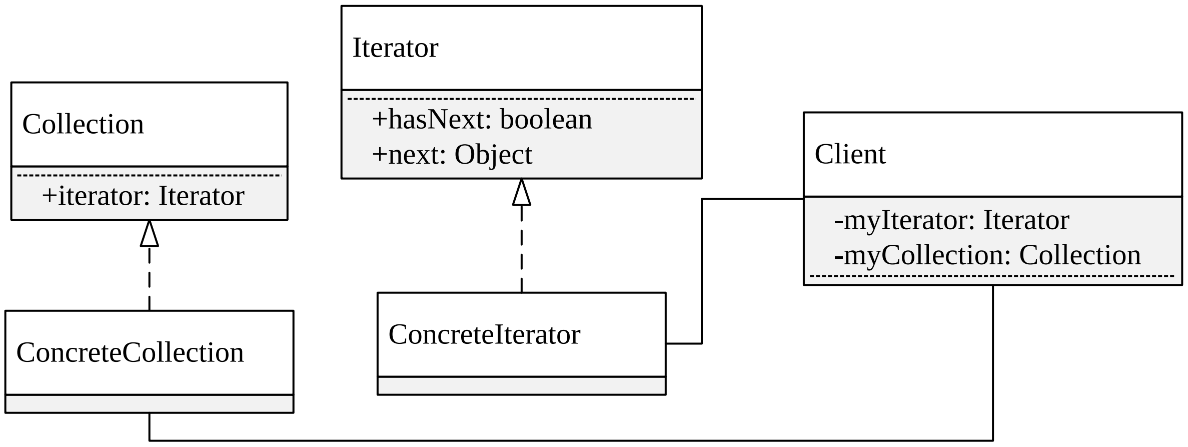
5. Client code that uses the collection.

Let us look at the class `LinkedList` in Java, which implements `Collection` and supports the `iterator()` method.

```
Collection collection = new LinkedList();
collection.add("Element 1");
collection.add(new Integer(2));
for (Iterator iterator = collection.iterator(); iterator.hasNext(); ) {
  System.out.println(iterator.next());
}
```

The Iterator Pattern

The **Iterator** pattern allows an external object to traverse the items in a collection object, and access these items without any knowledge of the data structures used by the collection. It is a **Behavioral** pattern, i,e., it identifies a common communication pattern between objects and realizes this communication pattern. Here, the communication pattern is that a client object would like to go over all the items of a collection in sequence, but does not want to be involved in details of the data structure used to store the collection. This communication is realized by having the collection provide the client with an iterator object with a standard interface. The iterator can traverse the collection, and can access the items; through the standard interface, the iterator provides the same functionality to the client. The consequence of this pattern is that it decouples a client class from a collection class.



The figure above shows the relationship between the participants. The interface `Collection` includes the method `iterator` that provides an iterator object with the standard `Iterator` interface. The class `ConcreteCollection` implements this interface, and has the ability to generate a `ConcreteIterator` object that implements the `Iterator` interface. The client can then traverse the collection and access its items through the following code:

```
Collection myCollection = new ConcreteCollection();
// code to perform various functions
...
Iterator myIterator = myCollection.iterator(); // start an iteration
Object object;
while (myIterator.hasNext()) { //traverse the collection
  object = myIterator.next();
  // other code to work with object
}
```

The first line creates a `LinkedList` object, whose reference is stored in the variable `collection`. We add two elements to the collection, a `String` object and an `Integer` object. Every object of type Collection supports the iterator() method, and this method is invoked in the initialization of the for loop. The returned object iterator is of type Iterator. Before entering the loop the first time or in any succeeding iteration, we make sure that we haven't processed all the elements. The method call iterator.hasNext() returns true if there is at least one element in the collection not yet retrieved since iterator was created. Such a collection element is retrieved in the body of the loop by the call iterator.next(). In this code, we simply print the elements. Thus we will end up printing Element 1 and 2 in successive lines.

2.11.4 Iterator Implementation

In this section, we describe how to implement an iterator in Java. Suppose we have the interface Queue, which allows adding and removing objects using the queue discipline (FIFO).

```
public interface Queue {
  public boolean add(Object value);
  public Object remove();
}
```

We implement the above interface in LinkedQueue. The inner class Node stores an object and the reference to the next element in the linked list. The head and tail of the queue are stored in the variables head and tail respectively.

```
import java.util.*;
public class LinkedQueue implements Queue {
  private Node head;
  private Node tail;
  private int numberOfElements;
  private class Node {
    private Object data;
    private Node next;
    private Node(Object object, Node next) {
      this.data = object;
      this.next = next;
    }
    public Object getData() {
      return data;
    }
    public void setNext(Node next) {
      this.next = next;
    }
    public Node getNext() {
      return next;
    }
  }
  // Queue methods
}
```

The add method creates an instance of Node and inserts it at the tail of the list. The code is straightforward.

```
  public boolean add(Object value) {
    Node node = new Node(value, null);
    if (tail == null) {
      tail = head = node;
    }
    else {
      tail.setNext(node);
      tail = node;
    }
    numberOfElements++;
    return true;
  }
```

The `remove` method also employs the standard approach to deleting from a queue. Before changing the value of `head`, we retrieve the contents of the first node in the queue, so we can return the deleted element.

```
  public Object remove() {
    if (head == null) {
      return null;
    }
    Object value = head.getData();
    head = head.getNext();
    if (head == null) {
      tail = null;
    }
    numberOfElements--;
    return value;
  }
  // The iterator method returns a new Iterator.
  public Iterator iterator() {
    return new QueueIterator();
  }
```

The iterator is implemented as an inner class. In the interface `java.util.Iterator`, there are three methods: `hasNext()`, `next()`, and `remove()`, the last operation being optional.

The `Iterator` object must maintain the list of elements in the queue that are not yet returned to the client. For this we take advantage of the fact that the `LinkedQueue` class itself has a linked list and that list is accessible from code within `QueueIterator`. However, the iterator class must not modify the field `head` in `LinkedQueue`; for this, we maintain a field called `cursor` within `QueueIterator`. This field is initialized to `head` when the iterator object is created.

```
  private class QueueIterator implements Iterator {
    private Node cursor;
    public QueueIterator() {
      cursor = head;
    }
    // hasNext, next, and remove
  }
```

Our plan is to return the elements as they appear in the queue. Therefore, the code for `hasNext` is quite simple: we just need to make sure that `cursor` is not `null`. Hence we have

```
    public boolean hasNext() {
      return cursor != null;
    }
```

To retrieve the next element, we must first make sure that there is at least one element not supplied to the client. That is, `hasNext()` does not return a `null` value. Then, we just move one element forward by setting `cursor` to `cursor.getNext()`.

```
  public Object next() {
    if (!hasNext()) {
      return null;
    }
    Object object = cursor.getData();
    cursor = cursor.getNext();
    return object;
  }
```

The implementation of the `remove()` method is trivial because we decided not to support this functionality.

```
  public void remove() {
  }
```

The above implementation shows the clean separation between the collection and the iterator. Another advantage of this approach is that we incur no additional complexity if there multiple iterators being employed simultaneously. The following code illustrates this.

```
LinkedQueue collection = new LinkedQueue();
collection.add(new Integer(1));
collection.add(new Integer(2));
for (Iterator iterator1 = collection.iterator(); iterator1.hasNext(); ) {
  Integer int1 = (Integer) iterator1.next();
  int count = 0;
  for (Iterator iterator2 = collection.iterator(); iterator2.hasNext(); ) {
    Integer int2 = (Integer) iterator2.next();
    if (int1.equals(int2)) {
      count++;
    }
  }
  System.out.println(int1 + count);
}
```

2.12 An Introduction to Generics in Java

Genericity is a mechanism for creating entities that vary only in the types of their parameters, and this notion can can be associated with any entity (class or method) that requires parameters of some specific types. As we have seen before, in the definition of any entity, the types of the involved parameters are specified. In case of a method, we specify the types of the arguments and the return type. In case of a class, the types of the arguments to the constructor(s), the return types and argument types of the methods are all specified. In any instance of the entity, the actual types of all these parameters must conform to the corresponding types specified in the definition. When we specify a generic entity, the types of the parameters are replaced by placeholders, which are called *generic parameters.* The entity is therefore *not fully specified* and cannot be used as such to instantiate any concrete objects. At the time of creating artifacts (objects, if our generic entity was a class) these placeholders must be replaced by actual types.

To understand the usefulness of genericity, consider the following implementation of a stack:

```
public class Stack {
  private class StackNode {
    Object data;
    StackNode next;
    // rest of the class not shown
  }
  public void push(Object data) {
    // implementation not shown
  }
  public Object pop() {
    // implementation not shown
  }
  // rest of the class not shown
}
```

Elements of the stack are stored in the `data` field of `StackNode`. Notice that `data` is of type `Object`, which means that any type of data can be stored in it.

We create a stack and store an `Integer` object in it.

```
Stack myIntStack = new Stack(); // line 1
myIntStack.push(new Integer(5)); // line 2
Integer x = (Integer) myIntStack.pop(); //line 3
```

This implementation has some drawbacks. In `line 2`, there is nothing that prevents us from pushing arbitrary objects into the stack. The following code, for instance, is perfectly valid.

```
Stack myIntStack = new Stack();
myIntStack.push("A string");
```

The reason for this is that the `Stack` class creates a stack of `Object` and will, therefore, accept any object as an argument for `push`. The second drawback follows from the same cause; the following code will generate an error.

```
Stack myIntStack = new Stack();
myIntStack.push("A string");
Integer x = (Integer) myIntStack.pop(); // erroneous cast
```

We could write extra code that handles the errors due to the erroneous cast, but it does not make for readable code. On the other hand, we could write a separate `Stack` class for every kind of stack that we need, but then we are unable to reuse our code.

Generics provides us with a way out of this dilemma. A generic `Stack` class would be defined something like this:

```
public class Stack<E> {
  //code for fields and constructors
   public void push(E item) {
     // code to push item into stack
   }
   public E pop() {
     // code to push item into stack
   }
}
```

A `Stack` that stores only `Integer` objects can now be defined as

`Stack<Integer> myIntStack = new Stack<Integer>();`

The statement

`myIntStack.push("A string");`

will trigger an error message from the compiler, which expects that the parameter to the `push` method of `myIntStack` will be a subtype of `Integer.`

2.13 Discussion and Further Reading

The concept of a class is fundamental to the object-oriented paradigm. As we have discussed, it is based on the notion of an abstract data type and one can trace its origins to the Simula programming language. This chapter also discussed some of the UML notation used for describing classes. In the next chapter we look at how classes interconnect to form a system, and the use of UML to denote these relationships.

The Java syntax and concepts that we have described in this chapter are quite similar to the ones in C++, so the reader should have little difficulty getting introduced to that language. A fundamental difference between Java and C++ is in the availability of pointers in C++, which can be manipulated using pointer arithmetic in ways that add considerable flexibility and power to the language. However, pointer arithmetic and other features in the language also make C++ more challenging to someone new to this paradigm.

Since our intention is to cover just enough language features to complete the implementations, some readers may wish to explore other features of the language. For those who want an exposure to the numerous features of the language, we suggest Core Java by Cornell and Horstmann [5]. A more gentle and slow exposure to programming in Java can be found in Liang [6]. If syntax and semantics of Java come fairly easy to you, but you wish to get more insights into Java usage, you may wish to take a look at Eckel [4].

It is important to realize that the concepts of object oriented programming we have discussed are based on the Java language. The ideas are somewhat different in languages such as Ruby, which abandons static type checking and allows much more dynamic changes to class structure during execution time. For an introduction to Ruby, see [2].

A thorough knowledge of inheritance is vital to anyone engaging in OOAD. While the notion of a class helps us implement abstract data types, it is inheritance that makes the object-oriented paradigm so powerful. Inheriting from a superclass makes it possible not only to reuse existing code in the superclass, but also to view instances of all subclasses as members of the superclass type. Polymorphic assignments combined with dynamic binding of methods makes it possible to allow uniform processing of objects without having to worry about their exact types.

Dynamic binding is implemented using a table of method pointers that give the address of the methods in the class. When a method is overridden, the table in the extending class points to the new definition of the method. For an easily understandable treatment of this approach, the reader may consult Eckel[3].

There is some overhead associated with dynamic binding. In C++, the programmer can specify that a method is *virtual*, which means that dynamic binding will be used during method invocation. Methods not defined as virtual will be called using the declared type of the reference used in the call. This helps the programmer avoid the overhead associated with dynamic binding in method calls that do not really need the power of dynamic binding. In C++ parlance, all Java methods are virtual.

2.14 Projects

1. A consumer group tests products. Create a class named Product with the following fields:

   (a) Model name,

   (b) Manufacturer's name,

   (c) Retail price,

   (d) An overall rating ('A', 'B', 'C', 'D', 'F'),

   (e) A reliability rating (based on consumer survey) that is a double number between 0 and 5,

   (f) The number of customers who contributed to the survey on the reliability rating.

   Remember that names must hold a sequence of characters and the retail price may have a fractional part.

   The class must have two constructors:

   (a) The first constructor accepts the model name, the manufacturer name, and the retail price in that order.

   (b) The second constructor accepts the model name and the manufacturer name in that order, and this constructor must effectively use the first constructor.

Have methods to get every field. Have methods to set the retail price and the overall rating.

Reliability rating is the average of the reliability ratings by all customers who rated this product. A method called `rateReliability` should be written to input the reliability rating of a customer. This method has a single parameter that takes in the reliability of the product as viewed by a customer. The method must then increment the number of customers who rated the product and update the reliability rating using the following formula

New value of Reliability rating = (old value of reliability rating * old value of number of customers + reliability rating by this customer) / new value of number of customers.

For example, suppose that the old value of reliability was 4.5 based on the input from 100 customers. If a new customer gives a reliability rating of 1.0, then the new value of reliability would be

```
(4.5 * 100 + 1.0) / 101
```

which is 4.465347

Override the `toString()` method appropriately.

2. Write a Java class called `LongInteger` as per the following specifications.

   Objects of this class store integers that can be as long as 50 digits. The class must have the following constructors and methods:

   (a) `public LongInteger():` Sets the integer to 0.

   (b) `public LongInteger(int[] otherDigits)`: Sets the integer to the given integer represented by the parameter. A copy of otherDigits must be made to prevent accidental changes.

   (c) `public LongInteger(int number)` Sets the integer to the value given in the parameter.

   (d) `public void readInt()`: reads in the integer from the keyboard. You can assume that only digits will be entered.

   (e) `public LongInteger add(int number)` Adds `number` to the integer represented by this object and returns the result.

   (f) `public LongInteger add(LongInteger number)` Adds `number` to the integer represented by this object and returns the result.

   (g) `public String toString()` returns a `String` representation of the integer.

   Use an array of 50 `ints` to store the digits of the number.

3. Study the interface `Extendable` given below.

```
public interface Extendable {
  public boolean append(char c);
  public boolean append(char[] sequence);
}
```

A class C that implements this interface maintains a sequence of characters. The method `append(char c)` appends a character to the character sequence. The second version of the method appends all characters in the array to this sequence. If for some reason the character(s) cannot be appended, the methods return `false`; otherwise they return `true`.

Write code for the class `SimpleBuffer` that implements the above interface and has a constructor of the following signature.

```
public SimpleBuffer(int size) {
```

The initial size of the array is passed as a parameter.

The class must have two fields: one stores the `char` array and the other stores the number of elements actually filled in the array.

This class must also implement the `toString()` method to correctly bring back a `String` representation of the `char` array. It should also implement the `equals()` method such that two objects are equal if and only if they are both `SimpleBuffer` objects containing the same sequence of characters.

4. Implement the interface `Extendable` in Programming Project 3 with a class named `AbstractBuffer`. This class stores an array of `char`s whose initial capacity is passed via a constructor.

   The class must have two fields, both `protected`; one stores the `char` array and the other stores the number of elements actually filled in the array.

   Do not implement either of the interface methods. So the class is declared `abstract`.

   This class must also implement the `toString()` method to correctly bring back a `String` representation of the `char` array.

   Next, implement `SimpleBuffer` so that it extends `AbstractBuffer` and actually implements the interface methods correctly. As before, it has a constructor that accepts the size of the array.

5. Consider the interface `Shape` given below:

```
public interface Shape {
  public double getArea();
  public double getPerimeter();
  public void draw();
}
```

   Design and code two classes `Rectangle` and `Circle` that implement `Shape`. Put as many common attributes and methods as possible in an abstract class from which `Rectangle` and `Circle` inherit. Ensure that your code is modular. For drawing a shape, simply print the shape type and other information associated with the object.

   Next, implement the following interface using any strategy you like. The interface maintains a collection of shapes. The draw method draws every shape in the collection.

```
public interface Shapes {
  public void add(Shape shape);
  public void draw();
}
```

Then, test your implementation by writing a driver that creates some `Shape` objects, puts them in the collection, and draws them.

Finally, draw the UML diagram for the classes and interfaces you developed for this exercise.

6. The following interface specifies a data source, which consists of a number of x-values and a corresponding set of y-values. The method `getNumberOfPoints` returns the number of x-values for which there is a corresponding y-value. `getX` (`getY`) returns the x-value (y-value) for a specific index ($0 \leq$ `index` $<$ `getNumberOfPoints`).

```
public interface DataSource {
  public int getNumberOfPoints();
  public int getX(int index);
  public int getY(int index);
}
```

The next interface is for a chart that can be used to display a specific data source.

```
public interface Chart {
  public void setDataSource(DataSource source);
  public void display();
}
```

A user will create a `DataSource` object, put some values in it, create a `Chart` object, use the former as the data source for the latter, and then call `display()` to display the data.

Here is a possible use. Note that `MyDataSource` and `LineChart` are implementations of `DataSource` and `Chart` respectively.

```
  DataSource source = new MyDataSource();
  Char chart = new LineChart();
  chart.setDataSource(source);
  chart.display();
```

Implement the interface `DataSource` in a class `MyDataSource`. Have methods in it to store x and y values.

Provide two implementations of `Chart`: `LineChart` and `BarChart`. For displaying the chart, simply print out the x and y values and the type of chart being printed. If needed, put the common functionality in an abstract superclass.

Draw the UML diagram for your design.

7. Implement three classes: `BinaryTreeNode`, `BinaryTree`, and `BinarySearchTree`. The first class implements the functionality of a node in a binary tree, the second is an abstract class that has methods for visiting the tree, computing its height, etc., and the third class extends the second to implement the functionality of a binary search tree.

8. Create a Java program to have the following classes and meet the given specifications.

    - An abstract class named `Position`, which stores the degree and minute of either a longitude or a latitude.
    - A subclass of `Position` named `Latitude`, which represents the latitude of a position. It should have the extra attribute (of type `String`) to store either “N” or “S.”
    - A subclass of `Position` named `Longitude`, which represents the longitude of a position. It should have the extra attribute (of type `String`) to store either “E” or “W.”
    - A class named `WeatherInformation` to store the maximum and minimum temperature of a point on the earth. It should have attributes to store the latitude, longitude, minimum temperature ever recorded, and the maximum temperature ever recorded at that position.
    - The class `WeatherInformation` must implement the following interface.

    ```
    public interface WeatherRecord {
      /**
        * Sets the maximum temperature
    * @param maxTemperature the new maximum temperature
      */
      public void setMaxTemperature(double maxTemperature);
      /**
        * Sets the minimum temperature
        * @param minTemperature the new minimum temperature
        */
        public void setMinTemperature(double minTemperature);
    }
    ```

    - A driver that does the following:
        - It creates 10 different Latitude objects and 10 different Longitude objects. Half of the latitudes must be north of the equator and the other half must be south of the equator. Half of the longitudes must be west of the Prime Meridian and the other half must be east of the prime meridian.
        - It creates 10 weather records for 10 locations on earth. All of the latitudes and longitudes created above must be used in one of the records. The maximum temperature must be more than the minimum temperature at every location. Approximately half of the minimum temperatures must be negative.
        - It prints the string representation of all weather records.
        - It changes the maximum temperature and minimum temperature of one of the locations and prints the new information.
        - Every class must override the `toString()` method properly.

– The constructors must conform to the following syntax.

```
public Latitude(int degree, int minute, String northOrSouth)
public Longitude(int degree, int minute, String eastOrWest)
public WeatherInformation(Latitude latitude, Longitude longitude,
                          double minTemperature, double maxTemperature)
```

2.15 Exercises

1. Given the following class, write a constructor that has no parameters, but uses the given constructor, so that x and y are initialized at construction time to 1 and 2 respectively.

```
public class SomeClass {
  private int x;
  private int y;
  public SomeClass(int a, int b) {
    x = a;
    y = b;
  }
  // write a no-argument (no parameters)
  // constructor here, so that x and y are
  // initialized to 1 and 2 respectively.
  // You MUST Utilize the given constructor.
}
```

2. In Section 2.3, we had a class called `Course`, which had a method that creates `Section` objects. Modify the two classes so that:
   (a) `Course` class maintains the list of all sections.
   (b) `Section` stores the capacity and the number of students enrolled in the section.
   (c) `Course` has a search facility that returns a list of sections that are not full.

3. Trace the following code and write that the program prints.

```
public class A {
  protected int i;
  public void modify(int x) {
    i = x + 8;
    System.out.println("A: i is " + i);
  }
  public int getI() {
    System.out.println("A: i is " + i);
    return i;
  }
}
public class B extends A {
  protected int j;
  public void modify(int x) {
    System.out.println("B: x is " + x);
```

```
      super.modify(x);
      j = x + 2;
      System.out.println("B: j is " + j);
    }
    public int getI() {
      System.out.println("B: j is " + j);
      return super.getI() + j;
    }
  }
  public class UseB {
    public static void main(String[] s) {
      A a1 = new A();
      a1.modify(4);
      System.out.println(a1.getI());
      B b1 = new B();
      b1.modify(5);
      System.out.println(b1.getI());
      a1 = b1;
      a1.modify(6);
      System.out.println(a1.getI());
    }
  }
```

4. Consider the class `Rectangle` in Programming Exercise 5. Extend it to implement a square.

5. A manager at a small zoo instructs the zoo-keeper to "feed the animals." Explain how a proper completion of this task by the zoo-keeper implies that the zoo operations are implicitly employing the concepts of inheritance, polymorphism and dynamic binding. (Hint: defining a class `Animal` with method `feed` could prove helpful.)

6. Write the `equals()` method in a situation where two `Student` objects are considered equal if their IDs are equal. There should be no exceptions thrown by the code.

7. Suggest an implementation for `hashcode()` when two `Student` objects are considered equal if their names and addresses are identical.

Bibliography

[1] B. Bruegge and A. H. Dutoit. *Object-Oriented Software Engineering.* Prentice Hall, 2000.

[2] P. Cooper. *Beginning Ruby: From Novice to Professional (Beginning from Novice to Professional).* apress, 2007.

[3] B. Eckel. *Thinking in C++ Volume 1 (2nd Edition).* Prentice Hall, 2000.

[4] B. Eckel. *Thinking in Java (4th Edition).* Prentice Hall, 2006.

[5] C. S. Horstmann and G. Cornell. *Core Java(TM), Volume I–Fundamentals (8th Edition).* Sun Microsystems, 2007.

[6] Y. D. Liang. *Introduction to Java Programming, Comprehensive Version.* Pearson Prentice Hall, 2007.

[7] S. J. Metsker. *Design Patterns Java Workbook.* Addison-Wesley, 2002.

Index