21 Java, Representation, and Object-Oriented Programming

Chapter Objectives

The primary representational constructs for Java are introduced including:
- Objects and classes

Polymorphism
Encapsulation
Inheritance constructs presented
- Single inheritance
- Interfaces
- Scoping and access

Java Standard Libraries
The Java idiom

Chapter Contents

21.1 Introduction to O-O Representation and Design

Java has its roots in several languages and their underlying ways of thinking about computer programming. Its syntax resembles C++, its semantics reflects Objective-C, and its philosophy of development owes a fundamental debt to Smalltalk. However, over the years, Java has matured into its own language. This chapter traces the roots of Java to the ideas of object-oriented programming, a way of thinking about programming, program structure, and the process of development that it shares with these earlier languages.

The origins of object-oriented programming are within the artificial intelligence research community. The first implementation of this programming paradigm was built at Xerox's Palo Alto Research Center with the creation of Smalltalk. The first release was Smalltalk-72 (in 1972). Object-orientation was a critical component of the early AI research community's search for representational techniques that supported intelligent activity both in humans and machines. The 1960s and 1970s saw the AI community develop semantic nets, frames, flavors, as well as other techniques, all of which had a role in the eventual development of object systems (see Luger 2009, Section 7.1).

This chapter will quickly cover the fundamentals of Java, but its purpose is not to teach the basics of Java programming, nor to provide a comprehensive guide of Java features. This chapter builds a conceptual framework with which to understand and reason about Java-style problem solving, as well as to set the stage for subsequent chapters.

21.2 Object-Orientation

In previous sections of this book, we discussed functional and declarative (or logic-based) programming as implemented in Lisp and Prolog respectively. Each of these programming models has its strengths and clear advantages for certain tasks. The underlying focus of OOP and Java is the desire to model a problem as a composition of pieces that interact with one another in meaningful ways. Among the goals of this approach is helping us to think about programs in terms of the semantics of the problem domain, rather than of the underlying computer; supporting incremental prototyping by letting us build and test programs one object at a time; and the ability to create software by assembling prefabricated parts, much like manufacturers create goods from raw inputs. Indeed, OOP grew out of a need for a different software representation and development process altogether.

The key to OOP is its representational flexibility supporting the decomposition of a problem into components for which one can define behaviors and characteristics: A typical car has an engine that transfers power to the wheels, both of which are attached to a frame and enclosed in a body. Each of these components can be further broken down: an engine consists of valves, pistons, and so forth. We can proceed down to some atomic level, at which components are no longer composed of some fundamentally simpler part. This introduces the idea of abstraction, whereby we regard some composition of pieces as a single thing; for example, a car. In Java, these pieces are called objects. One of the central ideas of object-orientation is that these objects are complete definitions of their "real-world" correlates. They define both the data and behaviors needed to model those objects.

A major consequence of OOP is the goal of creating general, reusable objects that can be used to build different programs, much like how a 9-volt battery from any vendor can be plugged into thousands of entirely different devices. This requires separating the definition of what an object does from the implementation of how it does it, just as the battery's specification of its voltage frees its users from worrying about the number of cells or the chemical reactions inside it. This idea has been very important to the growth of Java.

Originally, the Java language started out small, with only a few constructs more than languages like C. However, as it has found use, Java has grown through the efforts of developers to create packages of reusable objects to manage everything from user interface development, the creation of data structures, to network communication. In fact, if we look closely at Java, it retains this flavor of a small kernal that is enhanced by numerous reusable packages the programmer can draw upon in their work. To support this

growth, Java has provided rich, standardized mechanisms for creating and packaging reusable software components that not only support hiding implementation details behind object interfaces, but also give us a rich language of types, variable scoping, and inheritance to manage these interface definitions.

21.3 Classes and Encapsulation

The first step in building reusable composite parts is the ability to describe their composition and behaviors. Java uses the notion of classes to describe objects. A class is simply a blueprint for an object. It describes an object's composition by defining state variables, and the object's behaviors by defining methods. Collectively, state variables and methods are called the *fields* of an object.

In general, a program can create multiple objects of a given class; these individual objects are also called instances of a class. Typically, the state variables of each instance have distinct values. Although all members of a class have the same structure and types of behaviors, the values stored in state variables are unique to the instance. Methods, on the other hand, are simply lists of operations to perform. All instances of a class share the same methods.

Let's say we are designing a class to model microwave ovens. Such a class would be composed of a magnetron and a timer (among other things), and would provide methods for getting the object to do something, such as cooking the food. We next show how this class may look in Java. `Magnetron` and `Timer` are classes defined elsewhere in the program, and `mag` and `t` are state variables of those types. `setTimer` and `cookMyFood` are the methods. State variables establish what is called an assembly, or "has a," relationship between classes. In this example, we would say that a `Microwave` "has a" `Magnetron` and "has a" `Timer`. This approach is analogous to the early AI notion of inheritance reflected in semantic networks (Luger 2009, Section 7.1).

```
public class Microwave {
  private Magnetron mag;
  private Timer t;
  public void setTimer (Time howLongToCook) {...}
  public Food cookMyFood (Food coldFood) {...}
}
```

Objects encapsulate their internal workings. A well-designed object does not allow other objects to access its internal structure directly. This further reinforces the separation of what an object does from how it does it. This pays benefits in the maintenance of software: a programmer can change the internals of an object, perhaps to improve performance or fix some bug, but as long as they do not change the syntax or semantics of the object interface, they should not affect its ability to fit into the larger program. Think again of the `Microwave` class (or the oven itself). You don't touch the magnetron. You don't even have to know what a magnetron is. You control the behavior that matters to you by adjusting relevant settings via

the interface, and leave the rest to the microwave alone. Java provides the access modifiers to control what it exposes to the rest of the program. These modifiers can be *private*, *public*, and *protected*, and are described in further detail in Section 21.6.

21.4 Polymorphism

Polymorphism has its roots in the Greek words "polos" meaning many, and "morphos" meaning form, and refers to a particular behavior that can be defined for different classes of objects. Whether you drive a car or a truck, you start it by inserting a key and turning it. Even if you only know how to start a car, you can easily start a truck because the interface to both is the same. The mechanical and electrical events that occur, when we start a truck's diesel engine say, are different from those of starting a car, but at a level of abstraction, the actions are the same. If we were defining cars and trucks in an object-oriented language, we would say that the start method is polymorphic across both types of vehicles.

Java supports polymorphism by allowing different objects to respond to different methods with the same name. In other words, two different classes can provide method implementations of the same name. Let's say we have two classes, one modeling microwave ovens and the other modeling traditional stoves. Both can implement their own `startCooking` method, even if they do so in fundamentally different ways, using completely different method codes to do it.

Polymorphism is one benefit of OOP's separation of an object's interface from its implementation, and it provides several benefits. It simplifies the management of different classes of related types, and lets us write code that will work with arbitrary classes at some point in the future. For example, we can write a program for controlling a car object, leaving it up to some other developer to actually provide the car object itself. Let's say your code makes calls to the car methods `turnOn` and `shiftGear`. With such a framework, a developer might decide to plug in a truck object instead of a car. If the appropriate measures are taken, it can actually work – even though we never expected our program to work with trucks.

The benefits of polymorphism are in simplifying the structure of the software by exploiting similarities between classes, and in simplifying the addition of new objects to existing code. As a more practical example, suppose we want to add a new type of text field to a user interface package. Assume this new field only allows users to enter numbers, and ignores letter keys. If we give this new text field object the same methods as the standard text field (polymorphism), we can substitute it into programs without changing the surrounding code.

This leads to a concept at the heart of both AI and Java: semantics. What is it that makes the car object and the truck object semantically similar? How do you write code that lets somebody swap a car for a truck? These questions introduce the notion of *inheritance*.

21.5 Inheritance

Cars and trucks are both travel vehicles. Traditional stoves and microwave ovens are both cooking appliances. To model these relationships in Java, we would first create *superclasses* for the more general, abstract things (travel vehicles and cooking appliances). We would then create classes for the more specific, concrete things (cars, trucks, stoves and microwave ovens) by making them *subclasses* of the appropriate superclasses. Superclasses and subclasses are also referred to as *parent* classes and *child* classes, respectively.

What does this buy us? Two things: consolidation and reuse. To write a class to model a car, and then one to model a truck, it should be obvious that there is considerable overlap between the two definitions. Rather than duplicating the definition of such common functions like starting, stopping, accelerating, and so on, in each object, we can consolidate the descriptions in the methods of the more abstract notion of a vehicle. Furthermore, we can then use the vehicle superclass to quickly create a motorcycle class: motorcycles will inherit the general functionality of a vehicle (starting, stopping, etc), but will add those properties that are unique to it, such as having two wheels.

In Java terminology, the classes for each of cars, trucks, and motorcycles are said to *extend* or *inherit* from the class for vehicle. The code skeletons for implementing this inheritance are shown next. In particular, note that there is nothing in the `Vehicle` class that specifies that it is to be used as a superclass. This is important, as another developer might at some point want to extend our class, even if the desire to do so never entered our mind when we originally designed the software. In Java, most classes can later be extended to create new subclasses.

```
public class Vehicle
{
    public void start()     /* code defines start*/
    public void stop()      /* code defines stop */
    public void drive()    /* code defines drive */
    public boolean equals (Vehicle a)
        /* code defines if two vehicles are same */
    public int getWheels() { return 4;}
}
public class Motorcycle extends Vehicle
{
    public int getWheels() { return 2;}
                    /* Overrides Vehicle method */
}
public class Car extends Vehicle
{
    /* define methods that are unique to Car here */
}
```

```
public class Truck extends Vehicle
{
            /*define methods unique to Truck here*/
}
```

To develop this idea further, imagine that we wanted a class to model dump trucks. Again, we have overlap with existing components. Only this time, there may be more overlap with the class for trucks than the class for travel vehicle. It might make more sense to have dump trucks inherit from trucks. Thus, a dump truck is a truck, which is a travel vehicle. Figure 21.1 illustrates this entire class hierarchy using an inheritance diagram, an indispensable design aid.

We next discuss *inheritance-based polymorphism*. When we extend a superclass, our new subclass inherits all public fields of its parent. Any code that uses an object of our subclass can call the public methods or access the public state variables it inherited from its parent class. In other words, any program written for the superclass can treat our subclass exactly as if it were an instance of the superclass. This capability enables polymorphism (Section 21.3), and is crucial for code reuse; it allows us to write programs that will work with classes that have not even been written yet.

The ability to extend most classes with subclasses establishes a semantic relationship: we know that dump trucks are trucks, and that trucks are travel vehicles, therefore trucks are travel vehicles. Furthermore, we know how to start travel vehicles, so we clearly know how to start dump trucks. In Java, if the class `Vehicle` has a method `start`, then so does the class `DumpTruck`. We next present fragments of code that illustrate how polymorphism is enabled by inheritance.

Since DumpTruck and Car are descendants of Vehicle, variables of type Vehicle can be assigned objects of either type:

```
Vehicle trashTruck = new DumpTruck();
Vehicle dreamCar = new Car();
```

Methods that take arguments of type Vehicle can take instances of any descendants:

```
if (dreamCar.equals(trashTruck))
then
    . . .
```

Finally, methods that return type `travelvehicle` can return instances of any subclass. The following example illustrates this, and also shows an example of the Factory design pattern. A Factory pattern defines a class whose function is to return instances of another class, often testing to determine variations on those instances.

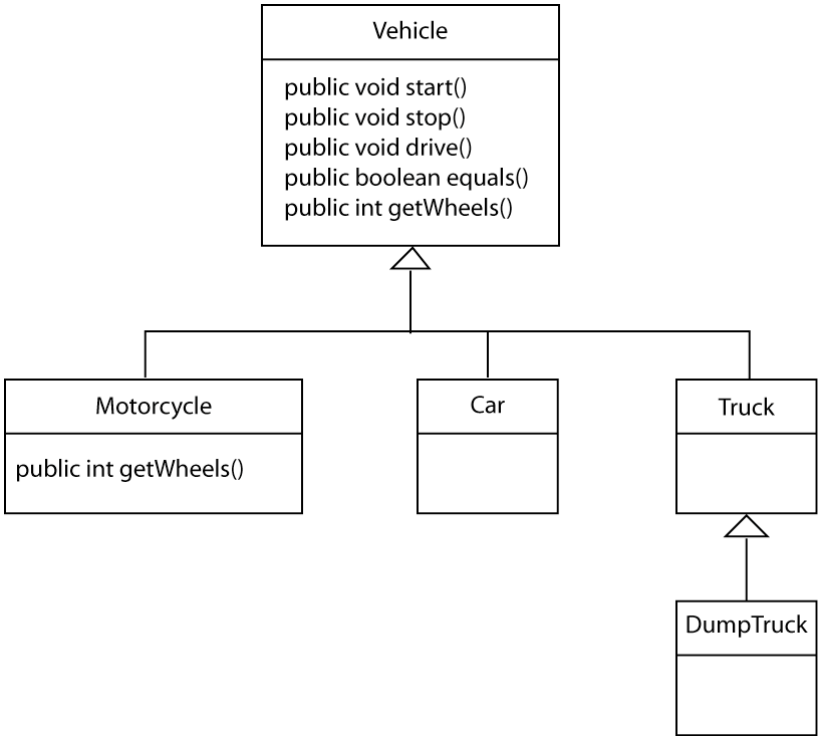

Figure 21.1: An inheritance diagram of our travelvehicle class hierarchy. Arrows denote "extends," also known as the "is a" relationship. For example, a dump truck is a truck, which is a travelvehicle.

```
class Vehicle Factory()
{
    public Vehicle getCar(Person customer)
    {
        if(customer.job = "Construction")
        then return new Truck();
        if (customer.job = "Salesman"_
        then return new Car();

        // etc.
}
```

We have seen semantic relationships and type hierarchies before. In Section 2.4 we used them to model symbolic knowledge in the form of semantic networks and frames. Semantic relationships serve the same purpose in Java. We are using classes to represent things in a hierarchically organized way. Java can credit classical AI research as the source of modeling knowledge using semantic hierarchies on a computer (see Luger 2009, Section 7.1).

Another important aspect of inheritance and polymorphism is the ability to give different classes the same methods, but to give them different definitions for each class. A common application of this is the ability to change a method definition in a descendant. For example, suppose we

want to change the start method for a motorcycle from using a starter to a simple kick-start. We would still make motorcycle a descendant of Automobile, but would redefine the start method:

```
class Motorcycle extends Vehicle
{
    public void start()
    {
        // Motorcycles use this definition of start
        // instead of Vehicle's
    }
}
```

We make one final point on inheritance. In our class hierarchy, `DumpTruck` has exactly one parent: `Truck`. That parent has only one parent as well: `Vehicle`. Every class in Java has exactly one parent. What is the parent of `Vehicle`? When a class does not explicitly state which class it extends, it automatically inherits from the generic class called `Object`.

`Object` is the root of all class hierarchies in Java. This automatic inheritance means that every single object in a Java program is, by inheritance, an instance of the `Object` class. This superclass provides certain facilities to every single object in a Java program, many of which are rarely used directly by the programmer, but which can, in turn, be used by other classes in the Java Standard Library.

In some languages (e.g., C++ and CLOS), classes can inherit from multiple parents. This capability is known as *multiple inheritance*. Java does not support multiple inheritance, opting instead for single inheritance. There is an ongoing debate between proponents of each approach, primarily related to program integrity and inadvertent creation of side effects, but we will not go into theses issues here.

21.6 Interfaces

Up to this point, the word *interface* has been used to refer to the public fields of a class. Here we introduce a special Java construct that also happens to be known as an "interface". Although these concepts are related, it is important to distinguish between the two.

A Java interface is a mechanism for standardizing the method names of classes for some purpose. It is effectively a contract that a class can agree to. The term of this agreement is that a class will implement at least those methods named in the interface definition.

A Java interface definition consists of the interface name, and a list of method names. A class that agrees to implement the methods in an interface is said to implement that interface. A class specifies the interfaces it implements in the class declaration using an implements clause. If a class agrees to implement an interface, but lacks even one of the prescribed methods, it won't even compile. For example, consider the interface:

```
interface Transportation
{
    public int  getPassengerCapacity();
    public double getMaximumSpeed();
    public double getCost();
}
```

The definition of the `Transportation` interface does not define the implementation of the methods `getPassengerCapacity()`, `getMaximumSpeed()`, or `getCost()`. It only states that any instance of `Transportation` must define these methods. If we had changed the definition of the class `Automobile` to read as follows, then we would be required to define the methods of `Transportation` in the class `Vehicle`.

```
class Vehicle implements Transportation
{
    . . .
}
```

This feels a bit like inheritance. After all, when we extend a parent class, we are promising that all public fields from that parent class are available, allowing programs to treat our new class like the parent class. Even if the inherited fields are overridden, they are sure to exist. However, there are conceptual and practical differences between inheritance and Java interfaces.

What is most important is that interfaces are only a specification of the methods in a class. They carry no implementation information. This allows us to define "contracts" between classes for how classes should behave without concern for their implementation. Suppose, for example, we wanted to define a class to manage all our vehicles. This class, which we call `Fleet`, could be defined to manage instances of the interface `Transportation`. Any class that implemented that interface could be in `Fleet`:

```
class Fleet
{
    public void add(Transportation v)
    {
         //Define add here
    }
    public double getTotalCost()
    {
         //sum of getCost on all Transportation
    }
}
```

This technique will be extremely important in using Java for building AI applications, since it allows us to create general tools for searching problem

spaces, pattern matching, and so on without knowing what types of spaces we will be searching or objects we will be matching. We simply create the interface such objects must define, and write our search engines to handle those interfaces.

At the operational level, classes can implement any number of interfaces. The methods defined in the interfaces can even overlap. Because the implementing class does not actually inherit method implementations from the interface, overlap is irrelevant. This gives us many of the benefits of multiple inheritance, and yet retains some security from unwanted side effects. Finally, interfaces provide another mechanism to enable polymorphism. As long as we know that a specific object can perform certain operations, we can ask it to do so.

Interfaces are especially useful when writing frameworks that require some specific behavior from objects of some class, but that do not care at all about the object's family. Thus, the framework does not constrain the semantic relationships of the class, which may come from a completely different developer, and might not have even been written yet. Java frameworks that impose such constraints often over-step their boundary, particularly when an interface would have been sufficient.

21.7 Scoping and Access

Java allows programmers to specify the context in which declarations and variables have an effect. This context is known as the *scope*. For example, if we declare a variable inside of a method, that variable is only accessible within that method.

Fields declared *private* are only accessible by methods in the same class; not even children of the class can access these fields. Although we often would like child classes to be able to access these variables, it is usually good style to define all member variables as `private`, and to provide public accessor functions to them. For example:

```
public class Foo
{
    private int value;
    public int getValue()
    {
        return value;
    }
    public void setValue(int newValue)
    {
        value = newValue;
    }
}
```

The reason for enforcing the private scope of member variables so strongly is to maintain the separation of interface and implementation. In the previous example, the class `Foo` does not expose the representation of the

variable `value`. It only makes a commitment to provide such a value through accessor functions `getValue()` and `setValue(int newValue)`. This allows programmers to change the internal representation of value in subtle ways without affecting child classes. For example, we may, at some time, want to have the class `Foo` notify other classes when `value` changes using Java's event model. If `value` were public, we could not be sure some external method did not simply assign directly to it without such a notification. Using this private variable/public accessor pattern makes this easy to guarantee.

In contrast, protected fields are accessible by inherited methods, and public fields are accessible by anybody. As a rule, protected scope should be used carefully, as it violates the separation of interface and implementation in the parent class.

Fields declared *static* are unique to the class, not to the instance. This means that there is only a single value of a static variable, and all instances of a given class will access that value. Because this creates the potential for interesting side-effects, it should be used judiciously. Static methods do not require an instance in order to be called. As such, static methods cannot access non-static state variables, and have no this variable.

Another useful modifier is *final*. You cannot create subclasses of a class that is declared `final`, or override a final method in a subclass. This allows the programmer to limit modifications to methods of a parent class. It also allows the creation of constants in classes, since variables declared `final` cannot be changed. Generally, `final` variables are declared `static` as well:

```
public class Foo
{
    public static final double pi = 3.1416;
    . . .
}
```

21.8 The Java Standard Library

In pursuing the goal of widespread reuse of software components, Java comes prepackaged with a substantial collection of classes and interfaces known as the *Java Standard Library*. This library provides a variety of facilities, including priority queues, file I/O, regular expressions, interprocess communication, multithreading, and graphical user interface components, to name just a few.

The Java standard library has proven to one of Java's most strategic assets. At a minimum, developers can focus on high-level objects by reusing common components, instead of building every application from a few basic commands. Using the techniques described above, standardization saves developers from having to learn multiple libraries that provide identical functionality, which is a common problem with other languages for which multiple software vendors distribute their own libraries. The Java library continues to grow driven by emerging challenges and technologies.

21.9 Conclusions: Designing in Java

As we strive to acquire the Java idiom, and not simply "learn the language", we must begin to think in Java. As stated earlier, Java and OOP emerged out of a need to improve the entire process of creating software. More than other languages, Java is linked to a particular set of assumptions about how it will be used. For example, *reuse* is an important aspect of this approach. When possible, we should not build Java programs from scratch using the basic language, but should look for reusable classes in the standard libraries, and may even want to consider commercial vendors or open source providers for complex functionality. In addition, Java programmers should always try to generalize their own code to create classes for possible later reuse.

One intention of object-oriented programming is to simplify later program maintenance by organizing code around structures that reflect the structure of the problem domain. This not only makes the code easier to understand, especially for other programmers, but also tends to reduce the impact of future changes. In maintenance, it is important for the change process to be reasonable to the customer: change requests that seem small to the customer should not take a long time. Because the final user's estimate of the complexity of a requested change reflects her or his own conceptual model of the problem, capturing this model in the computer code helps to insure a reasonable change process.

Finally, the ability to create classes that hide their internal structure makes Java an ideal language for prototyping (Luger 2009, Section 8.1). We can build a program up one class at a time, debugging and testing classes as we go. This sort of iterative approach is ideal for the kinds of complex projects common to modern AI, and programmers should design classes to support such an iterative approach. Guidelines for this approach to programming include:

> Try to make the breakdown of the program into classes reflect the logical structure of the problem domain. This not only involves defining classes to reflect objects in the domain, but also anticipating the source of pressures to change the program, and making an effort to isolate anticipated changes.
>
> Always make variables `private`, and expose `public` methods only as needed.
>
> Keep methods small and specialized. Java methods almost never take more than a full screen of code, with the vast majority consisting of less than a dozen lines.
>
> Use inheritance and interface definitions to "program by contract". This idea extends such useful ideas as strong variable typing to allow the programmer to tightly control the information passing between objects, further reducing the possibility of bugs in the code.

Finally, we should remember that this is not simply a book on programming in Java, but one on AI programming in Java. A primary goal

of Artificial Intelligence has always been to find general patterns of complex problem solving, and to express them in a formal language. Although Java does not offer the clear AI structures found in Lisp and Prolog, it does offer the programmer a powerful tool for creating them in the form of reusable objects/classes.

In writing this chapter, we faced a very difficult challenge: how can we write a meaningful introduction to so complex a language as Java, without falling into the trap of trying to write a full introductory text in the language. Our criteria for organizing this chapter was to emphasize those features of Java that best support the kinds of abstraction and structuring that support the complexity of AI programming. As with Lisp and Prolog, we will not approach Java simply as a programming language, but as a language for capturing the patterns of AI programming in general, reusable ways.

We should not simply think of Java as a programming language, but as a language for creating specialized languages: powerful classes and tools we can use to solve our programs. In a very real sense, master Java programmers do not program in Java, as much as they use it to define specialized languages and packages tailored to solve the various problems they face. This technique is called *meta-linguistic abstraction*, and it is at the heart of OOP. Essentially meta-linguistic abstraction means using a base language to define a specialized language that is better suited to a particular problem.

Meta-linguistic abstraction is a powerful programming tool that underlies many of the examples that follow in later chapters, especially search engines, the design of expert systems shells, and control structures for genetic algorithms and machine learning. We can think of these generalized search strategies, representations, or expert-system shells as specialized languages built on the Java foundation. All the features of Java described in this chapter, including inheritance, encapsulation, polymorphism, scoping, and interfaces serve this goal.

Exercises

1. Create an inheritance hierarchy for a heating and air conditioning system for a building using centralized heating/cooling sources. Attempt to make the overall control parallel and asynchronous. Capture your design with an inheritance diagram (see Figure 25.3).

2. Expand on problem 1, where each floor of the building has a number of rooms that need to be heated/cooled. Create a room hierarchy where the generic room has a fixed set of attributes and then each individual room (office, common room store room) inherits different properties. Each room should have its individual controller to communicate with the general-purpose heating/cooling system.

3. Problems 1 and 2 can be even further expanded, where each room has its own volume, insulation factor (the area of windows, say), and heating/cooling requirements. Suppose you want to design the ultimate "green" or conservation-oriented structure. How might you design such a building?

4. Create an inheritance hierarchy for an elevator control system. Take, for example a fifteen-storey building and three elevators. Capture your design with an inheritance diagram (see Figure 25.3).

5. Create Java structure code, similar to that in Sections 21.4.2, 21.5 and 21.6 for implementing the examples you created in questions you considered from exercises 1 - 4.

6. Consult the Java Library. What software components therein might support your program designs for question 3?