# Part I

# Understanding the Object-Oriented World View

# Chapter 1

# Object-Oriented Thinking

This is a book about object-oriented programming. In particular, this is a book that explores the principle ideas of object-oriented programming in the context of the Java programming language. Object-oriented programming has been a hot topic for over a decade, and more recently Java has become the commonly perceived embodiment of object-oriented ideas. This book will help you *understand* Java. It makes no pretensions to being a language reference manual; there are many other books that fall into that category. But knowing the syntax for a language should not be confused with an understanding of why the language has been developed in the way it has, why certain things are done the way they are, or why Java programs look the way they do. This book explores this issue of *why*.

Object-oriented programming is frequently referred to as a new programming *paradigm*. The word paradigm originally meant example, or model. For example, a paradigm sentence would help you remember how to conjugate a verb in a foreign language. More generally, a model is an example that helps you understand how the world works. For example, the Newtonian model of physics explains why apples fall to the ground. In computer science, a paradigm explains how the elements that go into making a computer program are organized and how they interact with each other. For this reason the first step in understanding Java is appreciating the object-oriented world view.

## 1.1 A Way of Viewing the World

To illustrate the major ideas in object-oriented programming, let us consider how we might go about handling a real-world situation and then ask how we could make the computer more closely model the techniques employed. Suppose I wish to send flowers to a friend who lives in a city many miles away. Let me call my friend Sally. Because of the distance, there is no possibility of my picking the flowers and carrying them to her door myself. Nevertheless, sending her the flowers is an easy enough task; I merely go down to my local florist (who

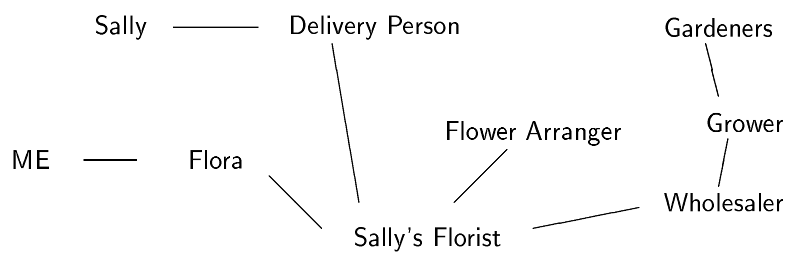


Figure 1.1: The community of agents helping me

happens to be named Flora), tell her the variety and quantity of flowers I wish to send and give her Sally's address, and I can be assured the flowers will be delivered expediently and automatically.

### 1.1.1 Agents and Communities

At the risk of belaboring a point, let me emphasize that the mechanism I used to solve my problem was to find an appropriate *agent* (namely, Flora) and to pass to her a *message* containing my request. It is the *responsibility* of Flora to satisfy my request. There is some *method*–some algorithm or set of operations–used by Flora to do this. I do not need to know the particular method she will use to satisfy my request; indeed, often I do not want to know the details. This information is usually *hidden* from my inspection.

If I investigated however, I might discover that Flora delivers a slightly different message to another florist in my friend's city. That florist, in turn, perhaps has a subordinate who makes the flower arrangement. The florist then passes the flowers, along with yet another message, to a delivery person, and so on. Earlier, the florist in Sally's city had obtained her flowers from a flower wholesaler who, in turn, had interactions with the flower growers, each of whom had to manage a team of gardeners.

So, our first observation of object-oriented problem solving is that the solution to my problem required the help of many other individuals (Figure 1.1). Without their help, my problem could not be easily solved. We phrase this in a general fashion as the following:

> An object oriented program is structured as a *community* of interacting agents, called *objects*. Each object has a role to play. Each object provides a service, or performs an action, that is used by other members of the community.

### 1.1.2 Messages and Methods

The chain reaction that ultimately resulted in the solution to my program began with my request to Flora. This request lead to other requests, which lead to still more requests, until my flowers ultimately reached my friend. We see, therefore, that members of this community interact with each other by making requests. So, our next principle of object-oriented problem solving is the vehicle by which activities are initiated:

> Action is initiated in object-oriented programming by the transmission of a *message* to an agent (an *object*) responsible for the action. The message encodes the request for an action and is accompanied by any additional information (arguments) needed to carry out the request. The *receiver* is the object to whom the message is sent. If the receiver accepts the message, it accepts the responsibility to carry out the indicated action. In response to a message, the receiver will perform some *method* to satisfy the request.

We have noted the important principle of *information hiding* in regard to message passing–that is, the client sending the request need not know the actual means by which the request will be honored. There is another principle, all too human, that we see is implicit in message passing. If there is a task to perform, the first thought of the client is to find somebody else he or she can ask to do the work. This second reaction often becomes atrophied in many programmers with extensive experience in conventional techniques. Frequently, a difficult hurdle to overcome is the idea in the programmer's mind that he or she must write everything and not use the services of others. An important part of object-oriented programming is the development of reusable components, and an important first step in the use of reusable components is a willingness to trust software written by others.

Information hiding is also an important aspect of programming in conventional languages. In what sense is a message different from, say, a procedure call? In both cases, there is a set of well-defined steps that will be initiated following the request. But, there are two important distinctions.

The first is that in a message there is a designated *receiver* for that message; the receiver is some object to which the message is sent. In a procedure call, there is no designated receiver.

The second is that the *interpretation* of the message (that is, the method used to respond to the message) is dependent on the receiver and can vary with different receivers. I can give a message to my wife Elizabeth, for example, and she will understand it and a satisfactory outcome will be produced (that is, flowers will be delivered to my friend). However, the method Elizabeth uses to satisfy the request (in all likelihood, simply passing the request on to Flora) will be different from that used by Flora in response to the same request. If I ask Kenneth, my dentist, to send flowers to my friend, he may not have a method for solving that problem. If he understands the request at all, he will probably issue an appropriate error diagnostic.

Let us move our discussion back to the level of computers and programs. There, the distinction between message passing and procedure calling is that, in message passing, there is a designated receiver, and the interpretation–the selection of a method to execute in response to the message–may vary with different receivers. Usually, the specific receiver for any given message will not be known until run time, so the determination of which method to invoke cannot be made until then. Thus, we say there is late *binding* between the message (function or procedure name) and the code fragment (method) used to respond to the message. This situation is in contrast to the very early (compile-time or link-time) binding of name to code fragment in conventional procedure calls.

### 1.1.3 Responsibilities

A fundamental concept in object-oriented programming is to describe behavior in terms of *responsibilities*. My request for action indicates only the desired outcome (flowers for my friend). Flora is free to pursue any technique that achieves the desired objective and is not hampered by interference on my part.

By discussing a problem in terms of responsibilities we increase the level of abstraction. This permits greater *independence* between objects, a critical factor in solving complex problems. The entire collection of responsibilities associated with an object is often described by the term *protocol*.

A traditional program often operates by acting *on* data structures, for example changing fields in an array or record. In contrast, an object oriented program *requests* data structures (that is, objects) to perform a service. This difference between viewing software in traditional, structured terms and viewing it from an object-oriented perspective can be summarized by a twist on a well-known quote:

> Ask not what you can do *to* your data structures,
> but rather ask what your data structures can do *for* you.

### 1.1.4 Classes and Instances

Although I have only dealt with Flora a few times, I have a rough idea of the behavior I can expect when I go into her shop and present her with my request. I am able to make certain assumptions because I have information about florists in general, and I expect that Flora, being an instance of this category, will fit the general pattern. We can use the term Florist to represent the category (or *class*) of all florists. Let us incorporate these notions into our next principle of object-oriented programming:

> All objects are *instances* of a *class*. The method invoked by an object in response to a message is determined by the class of the receiver. All objects of a given class use the same method in response to similar messages.

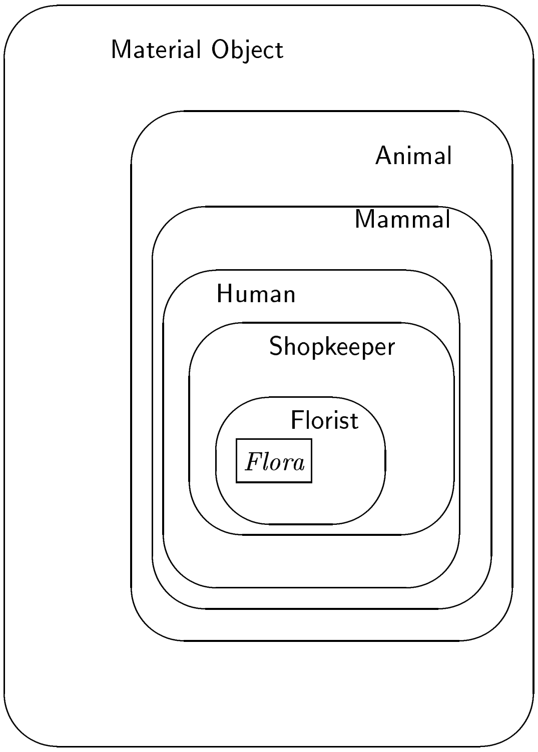


Figure 1.2: – The categories surrounding Flora.

### 1.1.5 Class Hierarchies–Inheritance

I have more information about Flora–not necessarily because she is a florist but because she is a shopkeeper. I know, for example, that I probably will be asked for money as part of the transaction, and that in return for payment I will be given a receipt. These actions are true of grocers, stationers, and other shopkeepers. Since the category Florist is a more specialized form of the category Shopkeeper, any knowledge I have of Shopkeepers is also true of Florists and hence of Flora.

One way to think about how I have organized my knowledge of Flora is in terms of a hierarchy of categories (see Figure 1.2). Flora is a Florist, but Florist is a specialized form of Shopkeeper. Furthermore, a Shopkeeper is also a Human; so I know, for example, that Flora is probably bipedal. A Human is a Mammal (therefore they nurse their young and have hair), and a Mammal is an Animal (therefore it breathes oxygen), and an Animal is a

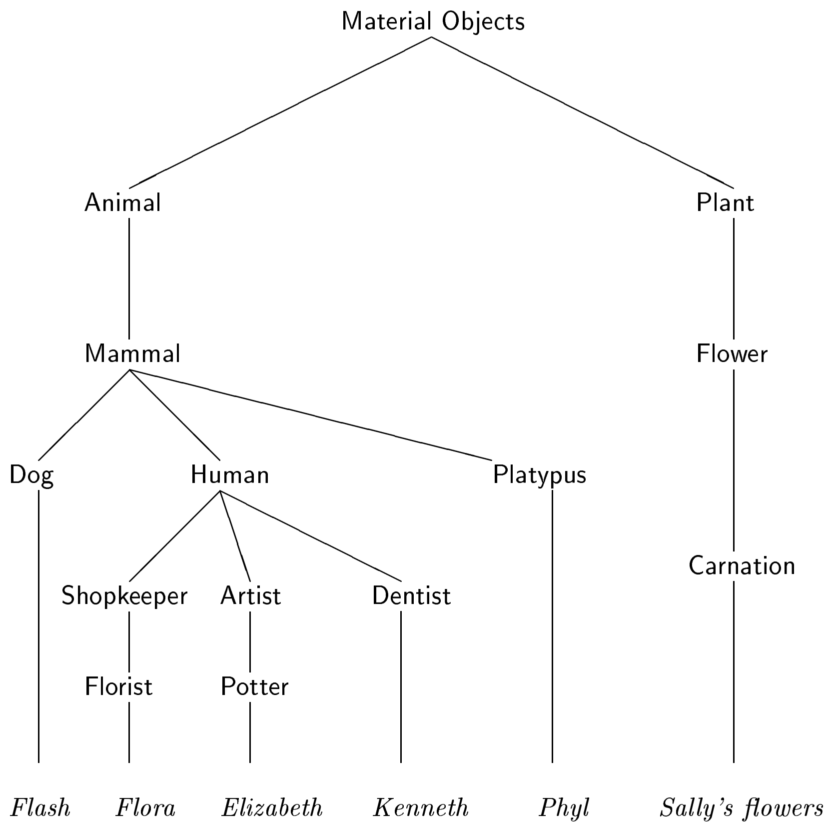


Figure 1.3: – A class hierarchy for various material objects.

Material Object (therefore it has mass and weight). Thus, quite a lot of knowledge that I have that is applicable to Flora is not directly associated with her, or even with her category Florist.

The principle that knowledge of a more general category is also applicable to a more specific category is called *inheritance*. We say that the class Florist will inherit attributes of the class (or category) Shopkeeper.

There is an alternative graphical technique often used to illustrate this relationship, particularly when there are many individuals with differing lineage's. This technique shows classes listed in a hierarchical tree-like structure, with more abstract classes (such as Material Object or Animal) listed near the top of the tree, and more specific classes, and finally individuals, are listed near the bottom. Figure 1.3 shows this class hierarchy for Flora. This same hierarchy also includes Elizabeth, my dog Flash, Phyl the platypus who lives at the zoo, and the flowers I am sending to my friend.

Information that I possess about Flora because she is an instance of class Human is also applicable to my wife Elizabeth, for example. Information that I have about her because she is a Mammal is applicable to Flash as well. Information about all members of Material Object is equally applicable to Flora and to her flowers. We capture this in the idea of inheritance:

> Classes can be organized into a hierarchical *inheritance* structure. A *child class* (or *subclass*) will inherit attributes from a *parent class* higher in the tree. An *abstract parent class* is a class (such as Mammal) for which there are no direct instances; it is used only to create subclasses.

### 1.1.6 Method Binding, Overriding, and Exceptions

Phyl the platypus presents a problem for our simple organizing structure. I know that mammals give birth to live children, and Phyl is certainly a Mammal, yet Phyl (or rather his mate Phyllis) lays eggs. To accommodate this, we need to find a technique to encode *exceptions* to a general rule.

We do this by decreeing that information contained in a subclass can *override* information inherited from a parent class. Most often, implementations of this approach takes the form of a method in a subclass having the same name as a method in the parent class, combined with a rule for how the search for a method to match a specific message is conducted:

> The search for a method to invoke in response to a given message begins with the *class* of the receiver. If no appropriate method is found, the search is conducted in the *parent class* of this class. The search continues up the parent class chain until either a method is found or the parent class chain is exhausted. In the former case the method is executed; in the latter case, an error message is issued. If methods with the same name can be found higher in the class hierarchy, the method executed is said to *override* the inherited behavior.

Even if the compiler cannot determine which method will be invoked at run time, in many object-oriented languages, such as Java, it can determine whether there will be an appropriate method and issue an error message as a compile-time error diagnostic rather than as a run-time message.

That my wife Elizabeth and my florist Flora will respond to my message by different methods is an example of one form of *polymorphism*. We will discuss this important part of object-oriented programming in Chapter 12. As explained, that I do not, and need not, know exactly what method Flora will use to honor my message is an example of *information hiding*.

### 1.1.7 Summary of Object-Oriented Concepts

Alan Kay, considered by some to be the father of object-oriented programming, identified the following characteristics as fundamental to OOP [Kay 1993]:

1. Everything is an *object*.
2. Computation is performed by objects communicating with each other, requesting that other objects perform actions. Objects communicate by sending and receiving *messages*. A message is a request for action bundled with whatever arguments may be necessary to complete the task.
3. Each object has its own *memory*, which consists of other objects.
4. Every object is an *instance* of a *class*. A class simply represents a grouping of similar objects, such as integers or lists.
5. The class is the repository for *behavior* associated with an object. That is, all objects that are instances of the same class can perform the same actions.
6. Classes are organized into a singly rooted tree structure, called the *inheritance hierarchy*. Memory and behavior associated with instances of a class are automatically available to any class associated with a descendant in this tree structure.

## 1.2 Computation as Simulation

The view of programming represented by the example of sending flowers to my friend is very different from the conventional conception of a computer. The traditional model describing the behavior of a computer executing a program is a *process-state* or *pigeon-hole* model. In this view, the computer is a data manager, following some pattern of instructions, wandering through memory, pulling values out of various slots (memory addresses), transforming them in some manner, and pushing the results back into other slots (see Figure 1.4). By examining the values in the slots, we can determine the state of the machine or the results produced by a computation. Although this model may be a more or less accurate picture of what takes place inside a computer, it does little to help us understand how to solve problems using the computer, and it is certainly not the way most people (pigeon handlers and postal workers excepted) go about solving problems.

In contrast, in the object-oriented framework we never mention memory addresses, variables, assignments, or any of the conventional programming terms. Instead, we speak of objects, messages, and responsibility for some action. In Dan Ingalls's memorable phrase:

> Instead of a bit-grinding processor...plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires [Ingalls 1981].

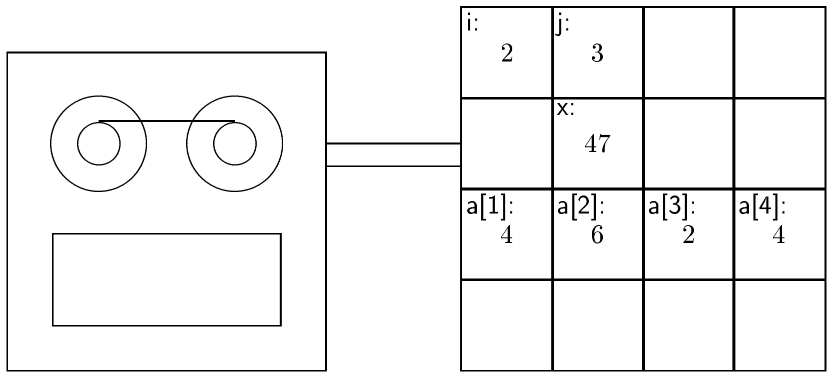


Figure 1.4: – Visualization of imperative programming.

Another author has described object-oriented programming as "animistic": a process of creating a host of helpers that form a community and assist the programmer in the solution of a problem [Actor 1987].

This view of programming as creating a "universe" is in many ways similar to a style of computer simulation called "discrete event-driven simulation." In brief, in a discrete event-driven simulation the user creates computer models of the various elements of the simulation, describes how they will interact with one another, and sets them moving. This is almost identical to the average object-oriented program, in which the user describes what the various entities in the universe for the program are, and how they will interact with one another, and finally sets them in motion. Thus, in object-oriented programming, we have the view that *computation is simulation* [Kay 1977].

### 1.2.1 The Power of Metaphor

An easily overlooked benefit to the use of object-oriented techniques is the power of *metaphor*. When programmers think about problems in terms of behaviors and responsibilities of objects, they bring with them a wealth of intuition, ideas, and understanding from their everyday experience. When envisioned as pigeon holes, mailboxes, or slots containing values, there is little in the programmer's background to provide insight into how problems should be structured.

Although anthropomorphic descriptions such as the quote by Ingalls may strike some people as odd, in fact they are a reflection of the great expositive power of metaphor. Journalists make use of metaphor every day, as in the following description of object-oriented programming from *Newsweek:*

> Unlike the usual programming method–writing software one line at a time–NeXT's "object-oriented" system offers larger building blocks that developers can quickly assemble the way a kid builds faces on Mr. Potato Head.

Possibly it is this power of metaphor, more than any other feature, that is responsible for the frequent observation that it is often easier to teach object-oriented programming concepts to computer novices than to computer professionals. Novice users quickly adapt the metaphors with which they are already comfortable from their everyday life, whereas seasoned computer professionals are blinded by an adherence to more traditional ways of viewing computation.

As you start to examine the Java programs presented in the book, as well as creating your own Java programs, you may find it useful to envision the process of programming as similar to the task of "training" a universe of agents to interact smoothly with each other, each providing a certain small and well defined service to the others, each contributing to the effective execution of the whole. Think about how you have organized communities of individuals, such as a club or committee. Each member of the group is given certain responsibilities, and the achivement of the goals for the organization depend upon each member fulfilling their role.

## 1.3 Chapter Summary

- Object-oriented programming is not simply a few new features added to programming languages. Rather, it is a new way of *thinking* about the process of decomposing problems and developing programming solutions.

- Object-oriented programming views a program as a collection of loosely connected agents, termed *objects*. Each object is responsible for specific tasks. It is by the interaction of objects that computation proceeds. In a certain sense, therefore, programming is nothing more or less than the simulation of a model universe.

- An object is an encapsulation of *state* (data values) and *behavior* (operations). Thus, an object is in many ways similar to a module or an abstract data type.

- The behavior of objects is dictated by the object *class*. Every object is an instance of some class. All instances of the same class will behave in a similar fashion (that is, invoke the same method) in response to a similar request.

- An object will exhibit its behavior by invoking a method (similar to executing a procedure) in response to a message. The interpretation of the message (that is, the specific method used) is decided by the object and may differ from one class of objects to another.

- Objects and classes extend the concept of abstract data types by adding the notion of *inheritance*. Classes can be organized into a hierarchical inheritance tree. Data and behavior associated with classes higher in the tree can also be accessed and used by classes lower in the tree. Such classes are said to inherit their behavior from the parent classes.

- Designing an object oriented program is like organizing a community of individuals. Each member of the community is given certain responsibilities. The achivement of the goals for the community as a whole come about through the work of each member, and the interactions of members with each other.

- By reducing the interdependency among software components, object-oriented programming permits the development of reusable software systems. Such components can be created and tested as independent units, in isolation from other portions of a software application.

- Reusable software components permit the programmer to deal with problems on a higher level of abstraction. We can define and manipulate objects simply in terms of the messages they understand and a description of the tasks they perform, ignoring implementation details.

## Further Reading

I said at the beginning of the chapter that this is not a reference manual. The reference manual written by the developers of the language is [Gosling 96]. But perhaps even more useful for most programmers is the annotated description of the Java class library presented by [Chan 96]. Information on the internal workings of the Java system is presented by [Lindholm 97].

I noted earlier that many consider Alan Kay to be the father of object-oriented programming. Like most simple assertions, this one is only somewhat supportable. Kay himself [Kay 1993] traces much of the influence on his development of Smalltalk to the earlier computer programming language Simula, developed in Scandinavia in the early 1960s [Dahl 1966]. A more accurate history would be that most of the principles of object-oriented programming were fully worked out by the developers of Simula, but that these would have been largely ignored by the profession had they not been rediscovered by Kay in the creation of the Smalltalk programming language. I will discuss the history of OOP in more detail in the next chapter.

Like most terms that have found their way into the popular jargon, *object-oriented* is used more often than it is defined. Thus, the question What is object-oriented programming? is surprisingly difficult to answer. Bjarne Stroustrup has quipped that many arguments appear to boil down to the following syllogism:

- X is good.

- Object-oriented is good.
- *Ergo,* X is object-oriented [Stroustrup 1988].

Roger King argued [Kim 1989], that his cat is object-oriented. After all, a cat exhibits characteristic behavior, responds to messages, is heir to a long tradition of inherited responses, and manages its own quite independent internal state.

Many authors have tried to provide a precise description of the properties a programming language must possess to be called *object-oriented.* I myself have written an earlier book ([Budd 97]) that tries to explain object-oriented concepts in a language-indepent fashion. See also, for example, the analysis by Josephine Micallef [Micallef 1988], or Peter Wegner [Wegner 1986]. Wegner, distinguishes *object-based* languages, which support only abstraction (such as Ada), from *object-oriented* languages, which must also support inheritance.

Other authors–notably Brad Cox [Cox 1990]–define the term much more broadly. To Cox, object-oriented programming represents the *objective* of programming by assembling solutions from collections of off-the-shelf subcomponents, rather than any particular *technology* we may use to achieve this objective. Rather than drawing lines that are divisive, we should embrace any and all means that show promise in leading to a new software Industrial Revolution. Cox's book on OOP [Cox 1986], although written early in the development of object-oriented programming and now somewhat dated in details, is nevertheless one of the most readable manifestos of the object-oriented movement.

## Study Questions

1. What is the original meaning of the word paradigm?
2. How do objects interact with each other?
3. How are messages different from procedure calls?
4. What is the name applied to describe an algorithm an object uses to respond to a request?
5. Why does the object-oriented approach naturally imply a high degree of information hiding?
6. What is a class? How are classes linked to behavior?
7. What is a class inheritance hierarchy? How is it linked to classes and behavior?
8. What does it mean for one method to override another method from a parent class?
9. What are the basic elements of the process-state model of computation?

10. How does the object-oriented model of computation differ from the process-state model?

11. In what way is a object oriented program like a simulation?

## Exercises

1. In an object-oriented inheritance hierarchy, each level is a more specialized form of the preceding level. Give an example of a hierarchy found in everyday life that has this property. Some types of hierarchy found in everyday life are not inheritance hierarchies. Give an example of a hierarchy that is not an inheritance hierarchy.

2. Look up the definition of *paradigm* in at least three dictionaries. Relate these definitions to computer programming languages.

3. Take a real-world problem, such as the task of sending flowers described earlier, and describe its solution in terms of agents (objects) and responsibilities.

4. Consider an object in the real world, such as a pet animal. Describe some of the classes, or categories, to which the object belongs. Can you organize these categories into an inheritance hierarchy? What knowledge concerning the object is represented in each category?

5. If you are familiar with two or more distinct computer programming languages, give an example of a problem showing how one language would direct the programmer to one type of solution, and a different language would encourage an alternative solution.

6. Argue either for or against the position that computing is basically simulation. (You may want to read the article by Alan Kay in *Scientific American* [Kay 1977].)