This is intended for instructors with some background in both programming and teaching. There is no need to hide it from students or others, however, as it has no hints about exercises. It does assume some background and might not make sense to someone keeping a day ahead of a student

Version of August 2023

Monty Karel Instructor's Guide and Teaching Notes:

In which, Joe Bergin gives away many of his secrets for successful object-oriented programming

Note that the solutions are not provided electronically and may only be obtained from Joseph Bergin. There will be a cost involved in printing and mailing.

This is based on the Instructor's Guide to Karel ++ which was, itself, based on the IG of the second edition of Karel the Robot. That earliest version was by Stehlik and Roberts, from which Bergin prepared the more recent versions.

General Comments

Unlike some textbooks this one does not assume that the instructor is an idiot and every word uttered by the instructor must be in the book. I assume you are a brilliant teacher, who wants some guidance for new material. Be creative in using this material. Bring in other things you know about programming and computing. Fill in your own knowledge of computing. You are encouraged, also, to explore the Pedagogical Patterns that can be found on my web site: http://csis.pace.edu/~bergin. This is especially true if you are a new teacher. I spent many years as a bad teacher learning how to become a better one.

There are a few places where I will make recommendations about what you might want to avoid if you want to keep to the same philosophy I used when I wrote the book. I will explain that philosophy somewhat as the opportunity arises here. In particular, if you want to use the material out of the order presented or introduce other things early (like variables and assignment), I will try to discourage you. Some of the things that we (you and I) do require the student to be very creative but are made trivial and the student will miss an important lesson if there is an easy path.

I left out a lot of Python from the book. Some of that was just to try to make the book as simple as possible while not losing essential computational power, but some of it was very intentional to set a certain stage upon which we can maximize student problem solving skills.

We (the authors) have left out nearly everything about Python primitive data (int…) because including it complicates the student's life and doesn't teach them about OO. There is one computational model: message passing. State is implicit in the world. In this regard, KJR is much like a functional system. The students don't program with explicit state and variables. The book doesn't depend on the von Neumann architecture. Objects *do* things and they *remember* things, just like people. The only variables are reference variables (other than what is needed to support the for loop). We use these variables to "deliver robots" and to set up delegation (a big topic) in Ch 4.

The overall idea of this book and its simulator is actually applicable to all computer programming. This deep idea comes from Common Lisp, but is easily done in Python and similar OO languages. **When you are faced with a problem to solve, don't just solve it in the language at hand. First design a new language in which the original problem would be easy to solve. Then implement the language in the one at hand and finally solve the original problem in the new language**. Monty Karel is a language in this sense. The problem it tries to solve is how to teach students computer problem solving in a compelling way while moving them in the direction they really need to go.

Building a robot (or other) class changes the language. Since we really use only the language of message passing (not manipulation of primitives, etc), there is one computational model and so a new class really extends that language. So, writing a class raises the level of the language toward that of the problem, but without changing or complicating the syntax. Then you solve the problem.

Throughout the book we depend on metaphor to help us teach. I don't try to show how this stuff is implemented on a lower level machine. Instead, I relate it to what the student already knows about the real world. The set of metaphors we use is expanded and explained more in the following sections as needed. The main guiding metaphor is objects (like people) *do* things and they *remember* things. Like people, they are autonomous and control their own actions (polymorphism). The flaw in the metaphor is that with people message passing is two-way. With objects it is one-way unless you explicitly set up the other direction. A reference variable gives you a one-way channel only. You might want to read my OO story that explains this metaphor. It is in the Ch 9 material of the original Java manuscript (not in the book).
http://csis.pace.edu/~bergin/KarelJava2ed/ch9/index.html

The book is actually the first cycle of a spiral learning approach to programming. None of the topics is covered exhaustively, but each is done enough to enable serious problem solving. After you finish the book, you take up other topics with your students, but return to these in deeper detail. Many teachers have said that they often remind students how the new topic they are then teaching, far beyond the robot world, relates back to what the students learned while playing here. You can use it as a framework for going farther: arrays of robots, etc.

Note that the essence of the computational model here cannot be captured in the *main* method, nor can it be captured in the common "algorithm" of getting up in the morning and making breakfast. Rather it is best viewed as a collection of independently acting agents that communicate with each other with messages, but not controlling what the others do.

Chapter 1

This chapter tries to set a metaphor in which the student can think when things get tough. The helicopter pilot is awkward, but something like it is needed so that the student will accept the fact that one instruction completes before the next begins. This is especially true when we have multiple robots. Unless we put them in threads (ch 8) they don't run simultaneously. If one robot sends a message to another, it will wait until the second completes the instruction to continue. This is counter-intuitive without the pilot's interventions. The helicopter pilot must interact instruction by instruction when commands come from outside the robot. Streets and avenues come from New York City, though there, First Ave. is to the east, not the west. Starting from 1 not 0 is historical with Karel and was kept because many instructors wanted to migrate their existing materials. The same is true of listing streets before avenues (also typical in New York), though this is backwards from Euclidean usage. (Note that the two malapropisms in the book, entomology and mathemagician, are intentional).

When I teach this I spend about one class period or less on this chapter. Spend some time doing a role play using the primitives that will be introduced in Ch 2. Orient your classroom as the robot world. You can even do some methods (turn around = left turn, left turn) anticipating Ch 3. If your students are dramatic, "turn off" can be a theatrical collapse to the floor. Beepers can be snacks/treats if you think this advisable, though beepers disappearing from the beeper bag is not part of the metaphor.

Note that while the robots come from a "factory" the constructors (Ch. 4) are like production lines in this factory for individual "kinds" of robots. The **new** command is a delivery instruction from some specific production line to some specific initial world situation. If you eventually want to extend the metaphor (ch 4) you can think of a subclass spec as a production line that begins where another ends: emphasizing specialization as discussed below (ch 4 notes).

1.1 The Robot World

The world is flat—sometimes— unless we stand it upright so Robots can climb some steps or a mountain. Some students have a very difficult time with the idea of an up direction when the world is flat. We have encountered this confusion in junior high and high school students as well as undergraduates. We think it is best to discuss this "transformation" of the world now. One way that we do it is to take a flat piece of paper on the desk and, as a group mark it north, east, south, west. Then tape the paper to the board and mark it up, right, down, and left. It is simple but seems to help.

We have added the idea of remembering that avenues run north and south by the visual cue of the pointing of the letter "A" northward and the letter "v" southward.

We have added the idea of relative location as well as absolute location (street/avenue addresses). You can use Figure 1-4 to expand on this idea. We found early on that using an absolute address lead some students to solve problems in later chapters in absolute terms for the sample world given in the problem statement. We have seen new instruction names such as moveFrom_4thStreetTo_14thStreet when what was needed was an instruction called move_10Blocks. The idea of using relative location for Robots and the objects in the world seemed to parallel the idea of writing general instructions.

Concerning beepers, there can be an infinite number of beepers in a pile. We mention this to you now because a problem in Chapter 5 uses this situation. If you have written your own simulator (we have met several teachers that have) and are wondering how to code an infinite number of beepers, try using -1.

There can also be several robots in the world simultaneously. Each robot will have a name. Not all robots are named karel, though we usually use that name when there is only one robot. Eventually we will see that the same name can be used to refer to different robots at different times and that different names can be used to refer to the same robot (even at the same time.) Emphasize that the name is not inherent in the robot. (It isn't in people either, actually.) The name is *associated with* the robot.

1.2 Robot Capabilities

This is fairly straightforward. We don't use the testing capabilities until Chapter 5 so you may want to touch them lightly here and save the details for later. Note that the robot only "sees" forward. This is a change from the earliest versions of Karel the Robot.

1.3 Tasks and Situations

At the request of reviewers, we have included Figure 1-4 that shows six different worlds as initial situations and briefly states a robot named karel's task for each. If you want to include some problem solving discussions in the course, you can ask students to make some assumptions about each world and the task it presents. Making such assumptions is part of the redefining of the problem that is part of the problem solving process. It is likely that your students will have no assumptions to make at this point. That is OK. Some may offer assumptions that karel can move or can pick up a beeper. You need to point out that these are not assumptions (because these are facts about the robot). As an example, in Figure 1-4A it might be assumed by the student that karel has no beepers. This is OK, but does it have any bearing on the problem?

What about Figure D? Does karel have to pick the beeper? There is nothing in the task that indicates this must be done. If karel does pick the beeper, is that an error? We will let you decide the answer to this question. Consider it carefully because similar situations will occur later. Whatever you decide, be consistent throughout the course.

1.4 Robots and Objects

This is our first introduction to the larger world of programming. The *remember* and *do* capabilities will be returned to. Note that *remember* doesn't always mean a field of an object. Objects do lots of things by delegation (Ch 4) but they *seem* to remember things. Really, it means we have procedural (methods that don't return a value) methods and functional (methods that return a value) ones. We won't do much with fields here in this book and you are encouraged not to introduce them until you finish this book. In particular, if you add an int field for street and avenue location and have the robot count its way around the world students will overuse this tool to their detriment. There are so many other ways to do things that this will just get in the way. We will use fields to hold references to robots and strategies to enable delegation (Ch 4).

You will want to start assigning exercises in each chapter before you finish the chapter. These maps give an indication of which sections should be covered before assigning a problem. For example, in Chapter 1 cover the first three sections before assigning problem 4, but you can assign problem 3 after only the first two sections.

Section	Problem map
1	1, 2, 5
2	3
3	4, 6

4 I omitted the following 7th exercise from the (preliminary edition) text for reasons of space: What two general behaviors are objects, including robots, capable of?

Chapter 2

This chapter introduces the students to programming robots. We show the students how to move and turn a robot and how to handle beepers. The need for the turnOff instruction is explained and we show a complete program for the first time. The execution of this program is explained and the form of a robot program is explained. Error shutoff and programming errors are discussed.

Be aware that this chapter overuses *main*. It's purpose, however, is to motivate Ch 3 and show that programming like this doesn't scale well. Long sequences of instructions are error prone and difficult to live with. Moreover, main is not, and cannot be, polymorphic. The thing to emphasize in this chapter is not the use of main, but the sending of messages to robots. Each message must be directed to a specific robot. The message must be recognized by the robot. There is a bit on the metaphor for static things in the introductory material for Chapter 4 in this guide.

Primitives of the robot programming language are introduced as well as the class mechanism. Your students will need to set up their environments and try out some code here.

It is strongly recommended that you and your students put all your code into Python modules (files with suffix ".py") so that it can be imported into other modules. If all of a student's modules are in the same folder it will be easy to import things. Such a folder is called a package. The simulator code as supplied must be findable from the package. One package is probably sufficient for the student, with a module per exercise.

Oddly, while everything in Python is an object, not all code needs to be built within a class. Some auxiliary functions, perhaps those that specify worlds might be functions defined at the "top level" in a module.

To avoid putting too much directly in main, one can also define a *task* function to contain the solution to a robot task and simply invoke this from main.

Suggested format of a Python module used to solve Monty Karel exercises.

I suggest that a Python module for exercised from this book have the following format. The order of listing is helpful if not always required.

a) all imports: Note that you should always import the world and the window from karel.robota. Without these some things can fail.

b) one or more classes

(c) a *task* function that follows all classes, but isn't part of any of them. The solution, including the creation of robots goes here.

d) the __main__ function in which you create the world (or read it) and tell the world to execute the task function.

Sample: Stair Sweeper – from Chapter 3:

```
from karel.robota import East
from karel.robota import window
from karel.robota import world
from karel.robota import UrRobot

class StairSweeper(UrRobot):

    def turnRight(self):
        "Robot turns right by executing three turnLeft instructions"
        self.turnLeft()
        self.turnLeft()
        self.turnLeft()

    def climbStair(self):
        "Robot climbs one stair"
        self.turnLeft()
        self.move()
        self.turnRight()
        self.move()

def task():
    alex = StairSweeper(1, 1, East, 0)
    alex.climbStair()
    alex.pickBeeper()
    alex.climbStair()
    alex.pickBeeper()
    alex.climbStair()
    alex.pickBeeper()
    alex.turnOff()


if __name__ == '__main__':
    world.setDelay(50)
    world.readWorld("../stairs.kwld")
    window().run(task)
```

The above uses the inherited (from UrRobot) constructor.

Note that Python is a multi-inheritance language. So, a student might build a special “mixin” class in a separate module to contain often used methods like turnRight and turnAround. This can then be imported and added to the inheritance list of the class(es) used to solve the exercise.

2.1 Changing Positions
Move is discussed and the idea of an error is introduced. This is changed from the original Karel the Robot. If you have not taught with Karel before, it does not matter, but die-hard robot fans should check the new description of *move* carefully. If a robot attempts to move when its front is blocked, it no longer executes the move by turning off. It executes an error shutoff instead of a move.

2.2 Turning in Place
Method turnLeft is discussed and there is little change in this edition.

It is important when teaching both the move and turnLeft instructions that you emphasize where a robot must be. You must stress that a robot is always on a corner or intersection and always facing one of the four directions. Having experience with students ranging from junior high through graduate students and even school teachers, it helps to use visual aids. In lecture, we have been known to play Robot and use chairs for walls and soda cans for beepers.

2.3 Finishing a Task
This is the same as earlier versions. Note that it is not an error to omit turnOff. Neither is it an error to send a message to a robot that has turned off, either because it has executed turnOff or has performed an error shutoff. Any such message will simply be ignored.

2.4 Handling Beepers
Stress that they are on the same corner as the robot; the robot cannot reach over to an adjacent corner to pick a beeper. Also stress that beepers do not block a robot’s movement or turning.

2.5 Robot Descriptions
This is the introduction of the class concept. A class describes the capabilities of a type of robot. Emphasize that ellipsis is not part of the language, but is being used in the ordinary sense.

2.6 A Complete Program
This introduces the students to a complete program with the required reserved words and punctuation.

You want to stress that there are few problems in programming that have only one correct solution. Frequently there are many solutions to a problem and the goodness or badness of the solution is often subject to debate. Encourage these debates!

2.6.1 Executing a Program
The ideas of simulation and tracing are presented and an annotated program is included to walk the student though the tracing of a robot's execution of the program. For reinforcement, we suggest that you present a new world and a new task, ask for suggestions of a program to solve the task, write the program, and trace its execution.

2.6.2 The Form of Robot Program
This section presents the grammar rules for the robot programming language. There are several kinds of things: punctuation marks, robot descriptions, tasks, delivery specifications, instructions, messages, delimiters, and reserved words. These are discussed in detail.

2.7 Error Shutoffs
Error shutoffs are discussed, justified and the three situations that can cause them are enumerated. Note that the absence of a turnOff instruction is not an error.

2.8 Programming Errors
At the request of reviewers, we have included examples of each of the programming errors discussed. Some of these are annotated. The discussion of an intent error includes an initial situation and a program that can be traced to find the error. The last paragraph is very important and must emphasized now and later on as the problems become more complex.

We recommend that you give students a complete, correct, robot program that they can play with. After they get it running, they should try to make a single error (say a lexical error) and then try to compile and run, just to see what happens. Then they won't be surprised when it happens to them. Then try a syntactical error, etc.

2.8.1 Bugs and Debugging
This presents the idea of bugs. We prefer to call them errors and would like to create the term deerroring but this sounds like another animal (deerroring). There are some new ideas in Chapter Three that can help students reduce the number of errors. One of these is stressing the need to plan and analyze thoroughly before implementing (keying in the program). Many of our students (especially the hackers) feel they think best at the keyboard. This attitude is just plain wrong unless you use a testing framework such as JUnit, which the Monty Karel simulator does support. For a novice such an attitude might be avoided if we begin stressing now the idea of, "Read in English, Think in English, Discuss in English, Plan in English, then, and only then, Program in the robot programming language."

2.9 A Task for Two Robots
We just introduce the topic. Much will be made of it in Ch. 4. We show a simple main that uses two robots.

2.10 An Infinity of Beepers

Again just an introduction for completeness. There are some interesting problems later. Note that a robot can carry an infinity of beepers (metaphor – a beeper creation engine in its beeper bag). We can also find such on a corner. The robotWorld module gives access to the word *infinity*. When creating a world with a text editor, -1 beepers indicates an infinite number on a corner.

You need to cover the first 6 sections before assigning any of the problems here. Most of them are then accessible.

Section	Problem map
1	
2	
3	
4	
5	
6	1,3,4-10
7	
8	2
9	11
10	12, 13

Chapter 3

Programmer-written methods are the key topic here along with inheritance. Note that when doing stepwise refinement, the helper methods should almost always be "private" (names beginning with an underscore). Actually "hidden" might be a better name than "private" in Python. Nothing is actually reserved to a class if you know enough tricks.

We make an exception, however, for certain "helpers", such as *turnRight* and *turnAround*, which seem to be natural robot actions applicable to many problems rather than those tailored to a specific problem. It isn't something to obsess over at this level. In a follow on book (Beyond Monty Karel) such considerations are more important.

Actually stepwise refinement is much less used in OO than in procedural programming. It is one technique among many and delegation to a different object is usually preferable. This is taken up in Ch 4 along with polymorphism, as the latter depends on delegation.

When you build an object in an OO language, the goal is to have it as simple as possible. Each object should perform a single service for its clients. When you ask what can an object "do" for another object, the answer should be very simple. There is a problem with this book that will arise if you are not careful. It is all too easy to build a DoEverything robot class, by putting all of the methods of all of the discussion and exercises into one class. Such a thing has no coherence (low *cohesion*). The goals of OO programming are the same as the goals of other programming: encapsulation, information hiding, low coupling, cohesion, modularity, etc. Remember, as you go, that the end all of OO programming is not "reuse". In fact, reuse as a goal is overused. Much more important is having a structure that you can grow into something else. Successful big programs were once successful small programs that grew. OO makes this growing easier.

When writing robot programs in Python (unlike Java) you seldom need a constructor for a new class. The inherited constructor from UrRobot will usually do. The exceptions, seen later in the book, are when a robot needs to “remember”, in an instance variable, a strategy or, perhaps some helper robots. Monty Karel has a section in the Appendix on constructor details.

Note how we depend on metaphor for explanations of things rather than detailed technical explanations. "main" is the helicopter pilot, etc. Since we don't use primitive data (int…) in the book, you don't really need to discuss the von Neumann architecture and register transfer computation. Let the computational model of Karel/Python speak for itself. If you try to express everything in terms of a lower level model you complicate things for the students since they then need to understand both levels and the mapping. This is an important idea, of course, but it can wait until later – even until a later course. If you want a metaphor for a robot "remembering" (which often uses instance variables) recall that a robot has its own dictionary. A robot can remember something by putting an entry into its own dictionary, or it can remember a reference to another object that is responsible for the thing "remembered". You don't need to discuss memory/ ram/ addresses…

Note that inheritance (extends) is not Python's most powerful tool. Interfaces are much more useful. The main tools, in fact, are
Inheritance
Interface (only simulated here, as it isn’t a formal Python construct)
Composition
Delegation
You are strongly encouraged to use inheritance only for specialization. A subclass object should logically be a special kind of superclass object. Don't think that a Cylinder is a subclass of a Circle. You save nothing by this but a bit of typing, and typing is not what makes programming difficult. If you don't follow this advice your programs will get difficult to reason about as you will need detailed information at too many places. On the other hand if you do follow it then when you have a Foo variable you can just think of the object as *being* a Foo even if the reference really points to a Bar. If Bar is correctly implemented as a special kind of Foo then you really DO have a Foo and can forget the distinction. You avoid a lot of casting when you do this also.

Note that the simulator for Monty Karel is a bit richer than the book. In particular, there are additional optional parameters in the constructor, naming a *fill* color and an *outline* color. If you use these parameters, the robot representation distinguishes it from other robots if you create several with different colors. See the Python docs for the complete description of what is available in the robots and in the world. Color names are strings, ‘red’, ‘blue’, etc. You will need to give your class a constructor to take advantage of the optional parameters.

3.1 Creating a More Natural Programming Language

This section sets up the need for new classes and instructions. Ask your students to refer to the problems done in Chapter Two to identify any new instructions in addition to turnRight that would have been helpful.

3.2 A Mechanism that Defines New Classes of Robots

This gives the syntax of a class and also explains its purpose. We also show the import that is required to use the simulator for this book.

3.3 Defining New Methods

This explains the structure of a new instruction definition. Because we do not yet discuss the structured statements (if, while), the definitions are very limited. But there is power here, as shown in Chapter 4.

This is a good time to stress the indenting of programs. Python is very picky about indentation. A “tab” is normally four spaces. If your editor can make the substitution automatically, it will be helpful in typing programs.

You need to stress the following about instruction blocks: 1) a Robot treats an instruction block as a single instruction and 2) the Robot must execute every instruction within the instruction block before the program is finished unless an error shutoff or a turnOff is executed. Caveat: This will change when we introduce the *return* instruction in Chapter 4.

3.4 The Meaning and Correctness of New Methods

This chapter brings up a major point concerning the difference between “what we say and what we mean.” In this case it is the difference between what we write as the definition of the new instruction and what we name the new instruction. Strive to stamp out any idea your students might have about Karel or the computer on which Karel is running having any awareness of what is occurring.

3.5 Defining New Methods in a Program

This section presents a complete program with new method definitions. They are annotated to help students trace Karel’s execution of the program. Use the idea of looking up a word in a dictionary to explain how this works. This program requires a robot to look up the definition of climbStair, which then requires a robot to look up another definition of turnRight.

3.6 Modifying Inherited Methods

This gives an introduction to a topic that might be surprising to students--we can redefine an inherited method. The new robots will use the new method. The old robots will continue to use the original. It all depends on which class they belong to. Stress the logical nature of programs here. It wouldn't be good to override move with a method that also turned, unless you were implementing a KnightPiece in a chess simulation. A reference variable gets its type in its declaration, but an object, including a robot, gets its type when it is created with new. It is the type of the object, not that of the reference that determines what is done when a robot receives a message.

3.7 An Ungrammatical Program
This section explains the consequences of making various kinds of syntax errors.

3.8 Tools for Designing and Writing Robot Programs
This is a long section that presents several methods for using new instructions. Polya's problem solving model is briefly introduced here. We kept it brief because Polya's model is not the topic of concern. Feel free to expand as you see fit. It can only enhance the material. We also introduce the idea that programs must be easy to read, debug, and modify to perform a slightly different task. This idea is very important and must be stressed over and over again. Some of your hackers (and colleagues) will argue that speed of execution or shortest programs are important. They are, but in our view, only after these three criteria are met!

Note that there are two kinds of decomposition in OO programs. Method decomposition, as discussed here is the less important. The other is decomposing the problem into its object/actors. This will be taken up in Chapter 4.

3.8.1 through 3.8.4 Stepwise Refinement—a Technique for Planning, Implementing, and Analyzing Robot Programs
This follows almost the same path through the planning process as the earlier versions. We have cast it into Polya's model and have added a question/answer format to try and show to beginners how we use stepwise refinement. The only real difference between this discussion and the original is that we first present a plan for solving the program that is analyzed and discarded in favor of a different plan (which is the same as the one in the first edition).

Emphasis should be on thinking, discussing, planning, and analyzing at each step of the way. Point out that most of the work should be mental and on paper. The total time spent at the keyboard should be greatly reduced if this is done carefully and thoughtfully!

Note, however, that an OO programmer spends much less time doing this than a procedural programmer. It is the essence of procedural (top down) development. It is only one tool among others, most more important, in OO. Note also, that these "helper" methods are almost always private (name begins with underscore). The public methods set the services of a class and maintain the invariants. Chapter 4 will complete this "tools" picture by showing a set of objects cooperating on a task.

3.9 Advantages of Using New Methods
This presents the ideas that using new methods combined with stepwise refinement usually leads to programs that are easy to read and can help reduce the time spent fixing errors. Programs written when this two framework is properly applied are usually quite easy to modify as shown in Section 3.9.2.

Section 3.9.3 remakes the points about program style and presents a solution to the problem using only primitive instructions. There is an error in this code: Karel executes three turnlefts after the last putBeeper instruction instead of just two.

3.10 Writing Understandable Programs

This section restates the idea of using good names for new instructions. We strongly urge the use of verbose names without abbreviations.

Section	Problem map
1	
2	
3	1
4	
5	2
6	10, 11
7	
8	3-9
9	
10	13, 14

Chapter 4

In my view, this is the most essential chapter, though it is pretty steep. You can quit half way through if you like and come back to it later. Try to get through Section 4.6, at least, as we introduce some more abstract objects there. Objects are things, but they can be abstract things too. However, this chapter really belongs before Ch 5 and not after. Once you understand and believe that, you are a real OO programmer. I can't emphasize enough that OO programming is not about classes. It is about objects communicating with polymorphic message passing. Classes are only a very small step in the needed direction. In this sense Karel++ was not object-oriented, but only "object-based." Likewise C++ is not truly an object-oriented language. While it is a multi paradigm language it is usually programmed as a data encapsulation language. OO is quite a bit richer than that. In Python, note that literally everything is an object. This is unlike Java, in which integers and booleans and such are primitives, not objects.

There are two kinds of polymorphism: ad-hoc (ch 5) and dynamic (ch 4). Dynamic is both more powerful and more natural (once you are used to it). It is more powerful in the sense that it is a better vehicle for maintaining complex and changeable programs, not just in inherent computational power. The experienced OO programmer will "see" a dynamic solution to a problem about as readily as an ad-hoc one. An experienced procedural programmer will see only the ad-hoc solution in too many situations. My goal is to bring the students to the point at which the two have equal weight so they can choose intelligently and not by default. Dynamic polymorphism is not a difficult topic, but you need to get used to using it. The robot's dictionary is a pretty good metaphor. The robot itself remembers how it responds to a message. Polymorphism really is just that simple. It has deep consequences, of course.

If you are a procedural programmer you get used to the idea of calling functions and knowing what code will be executed. OO programming isn't like that. In OO you send messages (NOT "call a method on an object" – UGH). The receiver of the message, not the sender, "decides" what code to execute. Actually it doesn't decide. It just does the only thing it knows how to do, by consulting its dictionary. People send messages to each other. The receiver decides how to interpret and respond to the message. The sender does not.

To be successful with polymorphism you have to recognize opportunities for it. These occur whenever you would normally write an if statement. A dynamic solution in such a situation has two parts. First you must put the different behaviors (two or more) into different objects. These are often in different classes, but sometimes you can just parameterize the construction of different objects. Each object knows how do do one thing. The second part is to bring the right object to bear at the right time. This is usually harder. State change diagrams can help with this. Hash maps can also. Sometimes an if or switch early in the program can pick an object to be passed around, saving many if and switch statements later. If the problem changes add a new class or method, or sometimes modify a method. You don't need to hunt down a lot of if/switch statements and bring them all into compliance with the change.

More on metaphor. An abstract class is like a template with holes for later customization. An interface is like a user manual for something. It tells how to *use* the thing. A class is like an architectural drawing. It tells how to *build* something. Objects instantiated from the same class (using one of its constructors) form a "family". Static (classmethods, global variables, etc. as discussed in Beyond Monty Karel) things are owned by the family. Ordinary (non-static) things are owned by the individuals. Think of static as the exception. Use it only as NECESSARY,

The web site has an optional section on how to use this material to build a linked list with no IF statements. It might be beyond most students, but will give instructors additional familiarity with this material.

The first author (Bergin) has two additional books (Java based) about polymorphism. They are amenable to beginning students: Polymorphism: As It Is Played, and Polymorphism Companion. They are available at Amazon.

4.1 Robot Teams

This version of the work permits several robots to coexist--with the purpose of cooperating in a problem solution. It is easy to get lots of robots working, but harder to interleave their work. The main technique for interleaving is a choreographer that directs the operations of other robots. In effect one robot behaves like the helicopter pilot with respect to other robots. You can also apply the first two sections of Chapter 8, if you like, to get quite interesting things to happen.

4.2 Similar Tasks

We introduce the abstract class and abstract methods. The metaphor is that of a partly built template object that can be tailored later for a more specific use. It emphasizes inheritance where different branches do different but similar things. Note that in Python, you can instantiate an object of an abstract class, but it's methods will fail when some things remain undefined.

4.3 Choreographers

We introduce instance variables (fields) of an object. In this book, these are always object valued (references) and we emphasize the nature of a reference. We need this for delegation, which is letting one object use another to carry out part of its service. Python variables always refer to objects, Even ints are objects. The book doesn't depend on the student manipulating integer values. They are used only for robot instantiation (street and avenue values in the constructor, say. As we have said earlier, if you introduce int fields, too many problems become trivial and students will miss an opportunity to learn problem solving for the rather thin benefit of learning a language feature. Note that we will introduce some serious algorithmic thinking in Chapters 6 and 7 that does not depend on having primitive data like int. A robot can, for example, compute the GCD of two values with no such data. Pattis is to be thanked for the deep insights that went into this.

Overriding methods is reinforced here as well.

4.4 Object-Oriented Design – Clients and Servers

This introduces the topic by examining the various robots (objects) that might cooperate in the solution of a large problem. Overall design is not a decomposition of a complex OPERATION or process into sub-processes. Instead it is the discovery of the ACTORS that can participate in the solution. These actors are given (simple) tasks. This means that when you get to decomposing procedures you are already working with smaller units. We emphasize clients and servers. Note that client-server describes a relationship that lasts for the sending of one message and the returned value if any. It is not a permanent property of an object. However, to define a class we usually think in terms of "what (single) service will this class define?" More accurately – what services will its *objects* provide?

The Python interface concept is introduced. It isn't a natural feature of Python, however. But they are an extremely useful way to think about programs. They are much more important and useful than inheritance, actually. When I teach this material, I spend half a day before we start the book doing role play exercises. The students are the objects and they have cards with their "script". The instructions conform to an interface and I show them this, so an interface is the first actual code my students see. The metaphor for an interface is a "user manual" for something like an automobile. It tells you how to use something, not how to build it. A class is like a production line (or sometimes like a set of engineering drawings).

A Python “interface” is just a class in which none of the methods are implemented, either defined as *pass* or as raising an exception. An “abstract class” is a class in which some of the methods have implementations, perhaps with undefined parts.

4.5 Using Polymorphism

One key idea here is encapsulating either empty or default behavior in an object (usually called a Null-Object, which is part of the design pattern by that name). We emphasize object invariants here. The constructor establishes them and the public methods maintain them.

We also emphasize that to get polymorphism you need to get different behaviors into different objects. We use different classes for the different objects here, but that isn't always necessary. Sometimes you can parameterize.

4.6 Still More on Polymorphism – Strategy and Delegation

Here we introduce objects that are not robots. These objects are much more abstract. Emphasize that "things" can be abstract as well as concrete and we can model them in a class. Strategies are important in their own right, but it is *delegation* that is the really big idea here. This forms the basis of sophisticated OO thinking, moving beyond objects as just the "nouns" in a problem. Objects can be simple if they delegate behavior to other objects. *Keep the objects simple. Put the complexity in the interactions.*

Recall that in Chapter 3 we had the harvester pick two rows at a time to solve the problem of alternating turns at the ends of rows. Note that if we had a left turn strategy and a right turn strategy and swap these at the end of rows (as part of executing them, actually), then we could pick one row at a time. After covering decorators in Section 4.8, you might try the pin-setter problem (Chapter 3, problem 2) with strategies (a set of strategies for placing the beepers and another for turning). Once you can do this, try the baseball diamond problem (Chapter 3, problem 4). It might help in the latter to have a method in your decorator that returns it's *decorated* to you.

Now we start to see the need for constructors, to create and initialize instance variables; the strategy objects in this case.

4.7 Python List and More on Strategies

The curve is getting steep here, but we thought the Spy example was strong enough to carry student interest. We shall see. We introduce a way for robots to communicate with each other without having references hard coded into their classes. The idea of an anonymous object is very important, actually. The robots have to rendezvous on the same corner to get started, but after obtaining a reference there, they can move about and still maintain contact. Later in your course you might be introducing arrays (or ArrayList). You can use this framework there, by enumerating a robot's neighbors and then itself carrying around a list of the robots (really references to the robots) it found at the rendezvous. It could then send messages to all of them even after they had parted. Return values are introduced here, but not the return statement. That is introduced in Chapter 5.

Lists are used within the simulator for many purposes. The world keeps a collection of the robots in it, for example, and enumerates over this collection to draw the world. Note that the *pop* action of a list removes an item. Iteration over a list is discussed in later chapters.

4.8 Decorators

The difference between naïve and sophisticated OO programming is design patterns. If you think this is just a buzzword, then I invite you to explore deeper. Patterns are what experts know how to do naturally. Design patterns are all about structure. In this book, the patterns introduced are extremely simple. This and the observer are the only two that are sophisticated enough to be thought of as having structure, and even here it is minimal. A decorator requires an interface and a special class that implements it (just two things). The decorator not only implements that interface but it keeps a reference to another object that also implements it. Thus both objects (decorator and decorated) have the same protocol. The same messages can be sent to either. I can send the message to the decorated (if it is visible) and get a simple thing. I can send the same message to its decorator and something nicer will happen, though the decorator is guaranteed to execute the *doIt* method of it's decorated object also. This is a simple introduction to a very powerful idea.

What is best about this is that we can change the behavior of something at run-time. Actually polymorphism is what underlies this ability and this is just another example of it.

As I mention elsewhere, if you want to understand modern language libraries you must learn to understand decorator. That is another reason I included it. A (Java) buffered reader is just a decorator of an arbitrary reader. When you read through a buffered reader, it first reads from its (decorated) reader and then buffers the input before giving it to you. This is a very cool and powerful idea. It actually simplifies the libraries, once you understand the principle. The alternative would have been a combinatorial explosion of options. You can read more on this topic at:
http://csis.pace.edu/~bergin/sol/java/gui/javaio.html.

Here is another reason to think about decorators and especially decorated strategies. Suppose some robot wanders around and builds up a complex strategy by repeatedly decorating some simpler version. Over time, the strategy can get very complex; arbitrarily complex, actually. Now, what the robot actually carries around is *a method encapsulated as a piece of data*. This method can be later executed. It can be saved away. It can be passed to some other robot for execution. It can be analyzed as data, etc. Some of the exercises in this chapter point in this direction, though a lot more can be done with this idea.

4.9 Observers

Observers are another key to understanding modern language libraries. We don't discuss the libraries in the book, but once you understand observers you understand how buttons and button listeners work. We present the simple case and try to dispel a misconception about observers as the actual implementation of observers is somewhat different from the meaning of the words used to describe them. That is why we have the metaphor discussion. There is more at: http://csis.pace.edu/~bergin/Java/eventfundamentals.html and especially at: http://csis.pace.edu/~bergin/Java/JavaEventStory.html.

Within the Monty Karel simulator, Observers are used in several places. For example, the world is an observer of each of its robots. When they change position or direction, the world is informed through the signaling mechanism. Also, the window in which the world is drawn listens for an event that signals that the window has been resized by the user. When this occurs the robot images are rescaled. This is a time consuming process. If we had to do it every time we drew one of the images the simulation would skip or run too slowly.

4.10 Final Words on Polymorphism

Here I try to push toward the idea that inheritance (and implementing interfaces) should be treated as a logical thing, not just a programming language tool. If you forget this rule then you will build un-maintainable code. We aren't smart enough (Miller's rule of 7) to keep all the necessary detail about a program in our heads if we don't treat programming logically. This means, primarily, using abstraction. A class or interface defines an abstraction. Subclasses should conform to that abstraction and, when they don't, we need to remember all the details about why they don't. Never subclass just to save re-typing an instance variable. It will cost you much grief in the future. A Cylinder is NOT a specialized Circle. Subclassing circle to get cylinder buys you nothing and costs you coherency of your code. Just say no.

Section	Problem map
1	1-6, 8
2	
3	7
4	13, 14, 18
5	
6	9, 12, 20, 21
7	10, 11, 17, 19
8	
9	15, 16, 22
10	

Chapter 5

This material is traditional IF and IF-ELSE selection – ad-hoc decision making. It is less important to OO than polymorphism so is presented after it. There is a pedagogical reason for the ordering also. If students learn this first, from instructors who know it best, it will be difficult for them to reach the point of seeing the polymorphic solution as easily and quickly as the selection solution. We therefore want to emphasize and reinforce polymorphism as much as possible.

Up until now, robots have had no way to test their own state, nor have we been able to test it externally. However, keep in mind that a program even without the new capabilities discussed in this chapter and the next may be able to do sophisticated things. In general, there are three ways to "know" something about the state of a robot. The program itself may contain information about the state of its objects. For example, if a robot starts with three beepers and has laid two then it must still have one. Often, but not always, a reading of the program will tell you what the state is at most points. I once had a student who never used such information, but always set a flag that he later checked. It took me a bit to show him that wasn't needed, but now he has a Doctorate.

The second way to "know" is to build up the knowledge and encapsulate it in an object as the program executes. This is what strategies are all about, especially the strategies we modify and exchange as we go. This is a very powerful technique. The third way is to have the object make specific tests to "gain the knowledge" that it may have once had, but now has lost. That is the subject of this chapter.

I often describe executing an IF statement as "climbing knowledge hill," as the amount of knowledge inside the IF is more than you had outside it. But, usually, you then exit the loop and tumble down the other side of knowledge hill. In Section 5.9 we will see how to retain the information you had at the top of the hill, and the previous chapter shows how to capture the knowledge when it first arises.

Note that only enough of selection is covered to be complete. We leave out switch statements as a result. The pedagogy here is due to Pattis, Stehlik, and Roberts.

This chapter introduces the conditional instructions IF and IF/ELSE. The idea of a test that evaluates to either True or False is introduced. Nesting conditionals is presented, some transformations are explained for simplifying code. Note that there is no dangling else in the Robot programming language since normal Python indentation makes clear which IF an ELSE belongs to.

5.1 The IF Instruction

This section presents the form of the IF instruction, discusses the <test>, and the THEN clause which must be an indented block.

5.2 The Conditions Robots Can Test

The 8 conditions that Robots can test are listed. The *not* operator is also introduced here. Everything is introduced via the new class: Robot. The new class emphasizes inheritance, which is one of our main lessons.

5.2.1 Writing New Predicates

A predicate is similar to an instruction except that it returns a boolean value (*True* or *False*). The return statement is introduced as well. Only predicates have return statements here, though we introduced other return values in Chapter 4 (Enumerations). They can be used to effect short circuit evaluation of AND and OR constructs. Note that return causes immediate termination of the predicate in which it appears. A return value (in general) can be remembered in a field (the dictionary), computed on the spot, or delegated to another object.

5.3 Simple Examples of the IF Instruction

Three examples are developed in this section and checking the correctness of the IF is explained.

5.3.1 The harvestOneRow Instruction

The harvesting problem from Chapter Three is revisited. This time the field is missing some beepers. The rationale for pickBeeperIfPresent is explained and the new instruction is developed using an IF instruction.

5.3.2 The faceNorthIfFacingSouth Instruction

A mythical problem called the Lost Beeper Mine is presented. An example of this problem is shown later in this document. This problem is used as a recurrent framework for examples. There is no such problem in the book. It is at the end of this document. Using the question/answer format, a plan is developed and analyzed and the new instruction written. Again we use this pattern to try and show the novice how we think about using the IF instruction.

5.3.3 The faceNorth Instruction

Again we use the Lost Beeper Mine problem to generate the need for a new instruction. The question/answer format is used to plan and analyze two completely different new instructions to solve the problem.

5.3.4 Determining the Correctness of the IF Instruction

Spend some time here. It should help clear up any misconceptions your students might have concerning how a robot executes the IF instruction. It will be helpful when we add the ELSE clause.

5.4 The IF/ELSE Instruction

This section presents the form of the IF/ELSE instruction in terms of a hurdle race.

5.5 Nested IF Instructions

The idea of nesting is introduced and explained in terms of a beeper replanting problem. The nested IF's are carefully discussed to show how a robot executes the instructions. Again, we suggest that you spend some time on this part of the section. We also suggest using a new instruction to "hide" the nested IF thus making the new instruction easier to read.

Beginners often write something like

```
if someTest():
   return True
else:
   return False
```

Show them that this is equivalent to

```
return someTest()
```

Similarly for a negated test. The above is a sure give-away of a naïve programmer.

5.6 More Complex Tests

The Robot programming language now has the AND (*and*) and OR (*or*) operators for making more complex tests. This discussion returns to the Lost Beeper Mine and shows a way to "simulate" the AND operator. You may want to ask your students how to "simulate" the OR operator (put the disjunct in an ELSE clause). AND and OR were purposely omitted from earlier versions to permit these discussions and a thorough examination of the ideas. We do both here: show the equivalence and introduce the operators.

5.7 When to use an IF Instruction

This section presents half of a paper tool we call the decision map. The decision map is used after the plan is thoroughly developed and a student is trying to figure out which piece of the robot programming language to use. Novices frequently say something like this, "I understand the syntax of the IF and how it works, but when do I use it?" If a beginner has done a good job of planning, the decision map can help answer this question. The map asks questions about what a robot needs to do and directs them along a series of branches until a particular language construct is reached. You might want to look at the entire map which is printed at the end of Chapter Six. The map was introduced to this series of books by Roberts and Stehlik. Here is a more complete version of the map that will appear in a future printing of the book. In includes polymorphism as well as the usual if statements.

5.8 Transformations for Simplifying IF Instructions

This section presents four transformations and code examples of the transformations. Code such as this is seldom written on purpose. It usually originates when students start fixing bugs. They eventually get the program to run correctly but they are so involved in the code that they lose sight of the style issues. Time spent here is will spent as the problems get more complex.

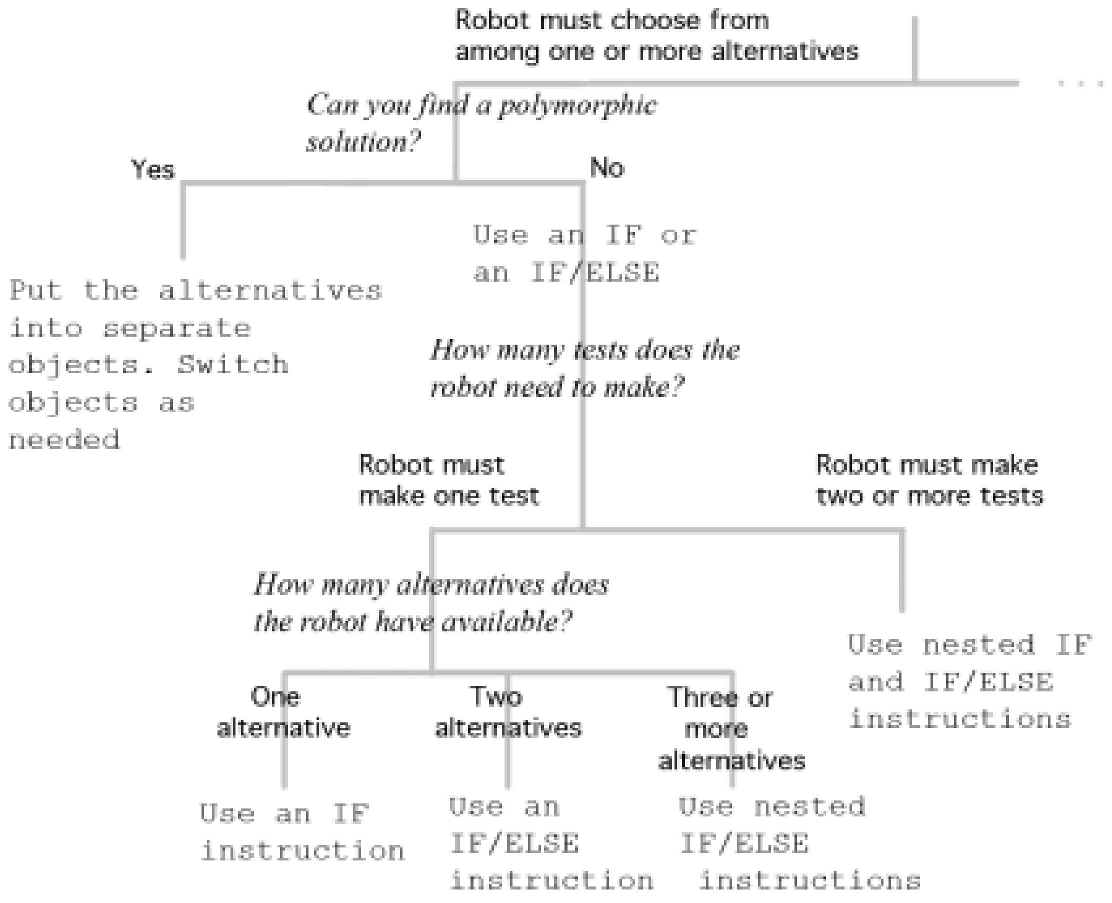
Robot must choose from among one or more alternatives
. . .
Can you find a polymorphic solution?
Yes
No
Put the alternatives into separate objects. Switch objects as needed
Use an IF or an IF/ELSE
How many tests does the robot need to make?
Robot must make one test
Robot must make two or more tests
How many alternatives does the robot have available?
Use nested IF and IF/ELSE instructions
One alternative
Two alternatives
Three or more alternatives
Use an IF instruction
Use an IF/ELSE instruction
Use nested IF/ELSE instructions

Fig 5.4

5.9 Polymorphism Revisited

Here we give some ideas about how one can use if statements early in the program to set up objects that behave polymorphically throughout. This is an important idea that is widely used in OO programming. The idea (not in the text) is that an object can be thought of as a "flag with behavior." At the point where a procedural programmer would set a flag, an object capturing the state is created instead. Create different objects (maybe from different classes) for different "flag values," all having the same interface. This object is passed around instead of the (int) flag. Then later, there is no test necessary. You just send that object an appropriate message and it will carry out one of the behaviors you need. Such a *flag* might appear in different contexts, with different local needs. Capture each of these in a different method of the objects.

Note: Some instructors might like to introduce recursion at this point, before iteration. Bergin's web site has an optional section on this idea that can be used here.

Section	Problem map
1	
2	
3	
4	1
5	3- 6, 9-12
6	13
7	
8	2, 7, 8, 14, 17
9	15, 16

Chapter 6

This chapter presents the FOR-LOOP and the WHILE loops. The FOR-LOOP is only a small part of the chapter. We spend most of our time on the WHILE loop. Careful work with the sections on the four step process, errors to avoid, and the discussion of the loop invariant will be very useful for your students, especially those that will continue to take additional programming and computer science courses in the future.

If you have (or can get) the 2nd edition of Karel the Robot (it is in the first edition also) the room escape stuff in Section 5.7 (5.5 in 1ed) is wonderful. It emphasizes stepwise refinement of methods (rather than polymorphism) a bit too much for the current text, but it is worth a look and can be adapted to Monty Karel also. Its real beauty is in the beyond-the-horizon opportunities. In this book we have the material in Problems 6.6 and 6.7. Similar material with more of an OO flavor appears in Section 6.7

6.1 The FOR-LOOP Instruction

This section explains the form and execution of the for loop instruction within the context of the new instructions, turnRight and harvestOneRow (from Chapter Three). It is the simplest counting idiom of the Python for loop, using the range construct. There are more sophisticated versions of range, not covered here. We only count upward and never use the index except for counting. The for loop variable is not available after the loop ends, of course. It is local to this statement. You can (should) avoid an external declaration until later. We won't use the value of the loop control at all, nor a variable upper limit of iteration. We suggest you only use the simple form of the for loop in this initial pass and return to it later as needed in the course. It is here to show that there are two kinds of iteration, a fixed number of iterations (for loop) and a variable number (while loop). While we could do counting loops with while, it would require even more discussion of primitive data and assignment than we want to teach now.

6.2 The WHILE Instruction

This section presents a need for the WHILE instruction, explains the form, and provides a four step process for building WHILE loops. Again, we require a braced block, not a single statement.

6.2.1 Why WHILE is Needed
The example of (a robot named) karel having to walk forward to find a lone beeper somewhere directly ahead is presented. Remember that the robot world is infinite to the east and north (theoretically) so a for loop will not always work, no matter how big the chosen number of repetitions. Note that the implementation of the world in the simulator actually wraps around (like a globe) with 2^{32} streets and the same number of avenues. If your robot walks far enough it could actually walk into the back side of a boundary wall.

6.2.2 The Form of the WHILE Instruction
The form of the WHILE instruction is shown in this section. A robot's execution of the WHILE loop is also explained. Careful attention to this section will pay off very soon.

6.2.3 Building a WHILE Loop - the Four Step Process
This section offers a guideline for building a WHILE loop. You must emphasize that this works only if the planning has been thorough and careful. Successfully building a WHILE loop requires that the students know exactly what a robot must do.

6.2.4 A More Interesting Problem
Using the question/answer format and the four step process we write another new instruction that uses a WHILE loop. This instruction will be used for several sections as we modify the task, which the robot must perform. You can emphasize that our new instruction definitions must be easy to modify so they will work in an ever-changing world. Note, however, that OO programmers prefer to work by extending existing classes and overriding methods, rather than modifying them.

6.3 Errors to Avoid with WHILE Loops
Two different errors and a common misconception are discussed in this section.

6.3.1 The Fence Post Problem
This problem is defined and then presented in a slightly modified version of the world used in Section 6.2.4.

6.3.2 Infinite Execution
The problem of infinite execution is explained in this section and some advice is offered for planning the body of WHILE loops.

6.3.3 When the Test of a WHILE is Checked
This section discusses a beginner/novice misconception that the test of the WHILE loop is continually checked as karel executes the body of the WHILE loop. This discussion provides a nice transition to the next section on nesting.

6.4 Nested WHILE Loops
This section examines a good example and a bad example of nesting WHILE loops. The bad example is long and detailed but worth the time. It provides some good insight to the debugging process as well as how to carefully plan your loops.

6.4.1 A Good Example of Nesting
The question/answer format is used to build a new instruction that uses nested WHILE loops to good advantage. We also show a better style of coding that "hides" the nested loop. We recommend the style of hiding the nested loop because, in our judgment, it is easier to read.

6.4.2 A Bad Example of Nesting
This section builds another new instruction using nested WHILE loops. Our reasoning is faulty in several places and this leads us astray. One error in our reasoning is assuming that a beeper is always in a corner of the room. Using this implied assumption we go off on the wrong track. The question/answer format works well here as it allows us to revisit our reasoning and see where the error was. This is a long discussion but it is worth taking the time to go through it with your students to help prevent their making similar errors.

6.5 WHILE and IF Instructions
This section presents some common errors made by novice programmers when combining these two instructions.

6.6 Reasoning about Loops
This is section was new in 2ed of Karel the Robot and is retained here. It restates the four step process and the informal way to reason about the correctness of WHILE loops. Then a discussion about loop invariants is presented. For simplification (from both the teacher's and the beginner/novice student's points of view) we have limited the definition of invariant to be, "an assertion which is true after each iteration of the loop." You may wish to expand this, as is typically done, to include the state immediately before and after the loop, but we feel this limited definition is better for the novice crowd. This section explains what we mean when we say we want to consider something that is "interesting" about the loop. The entire discussion is a good prelude to the next section.

6.7 A Large Program Written by Stepwise Refinement
The problem has a more object-oriented flavor than the original from Karel the Robot. It opens up possibilities of robot teams in the solution. It is a good discussion about building OO programs, making assumptions, and testing our programs. This section is time well spent. However, the original was brilliant as mentioned above, though highly procedural.

6.8 When to Use a Repeating Instruction
This is a continuation of the decision map tool introduced in Section 5.7. First we revisit the original map, then we add the right side for repeating instructions and finally show the entire map. Again, this is designed for beginners/novices that are having trouble "seeing" how to use these structures.

Section	Problem map
1	5, 13
2	1, 2, 4, 26
3	
4	
5	3, 6, 7 – 11, 13, 15, 19, 20, 22, 28
6	18, 24
7	12, 16, 17, 21 – 23, 25
8	29 – 32

Chapter 7

This will be brief. Few teachers use this chapter for many reasons. There is some good stuff here. We urge you to consider covering some or all of the topics.

7.1 Recursion

We feel that this is a good discussion about recursion and will benefit those students that are going to take additional computer science courses. Some problems are extremely awkward without recursion and very natural with it. The web site has an optional section that can be used to introduce recursion before iteration if you wish to use it. Note that if you have introduced int variables, especially counters, prior to this, it will become difficult to motivate the material. Too many interesting problems are made trivial using counters. I suggest you save counters for later to make this more interesting and challenging.

7.2 More on Recursion

This gives a more complete treatment--similar to the four step analysis of loops.

7.3 Tail Recursion and Looping

The relationship between the two is explained. Some have questioned why we included it. It turns out that tail recursion can be removed from programs by compilers, increasing efficiency and stack usage. It is an important technique in functional languages. We like it as it is fun and makes you think about the meaning of a loop. It also sets up the next section.

7.4 Going Formal

A formal definition of WHILE is given in terms of recursion. Hopefully this will elucidate both recursion and WHILE loops.

7.5 Searching

This section comes from the original Karel the Robot. It is brilliant, actually. Note that we deviate after this section from the first and second editions. If you have one of those editions you can adapt that material. We have chosen to put most of it into the exercises. See the next section, which is our more OO way of doing arithmetic.

7.6 Doing Arithmetic
This section is the quite different from the first and second editions of Karel the Robot but also appears in Karel++. It treats arithmetic numerically rather than positionally. It permits teams of robots to work together.

7.7 Polymorphism--Why Write Many Programs When One Will Do?
Here we try to make the implications of polymorphism as explicit as possible. The meaning is still simple (each object does only what it knows how to do) but the implications are deep. Emphasize the difference between a *reference* and the *object* that it points to. It is the object that "knows" what to do, not the type of the reference that determines what to do.

7.8 Dynamic Python
This section emphasizes both that everything in Python is an object and that variables and the entries in a list don't have static types. Typing is done entirely at runtime when a message is sent to an object and that object knows whether or not it can respond. The "compiler" doesn't check for correctness prior to execution. This makes for some difficulties in proving programs correct as you need to do much more testing than with statically typed languages such as Java. The flexibility of Python, thus, has a cost.

7.9 Conclusion
We address in a simple way the question of when you should write a new class.

Section	Problem map
1	
2	1, 2, 12, 17 – 22, 26, 28
3	
4	
5	3, 4 – 11, 13 – 16, 27
6	24
7	25
8	29

Problem 23 admits an iterative, if messy, solution. I'm trying to think of a nice recursive one. My solution looks like bubble sort. We don't have arrays here and we also don't have a way to refer to a robot by the street on which it stands, so it gets quite ugly. Try to do it for just two robots first.

Additional Exercises

35: A robot named leonardo wants to compute a Fibonacci sequence such as 1, 1, 2, 3, 5, 8, … where each number after the first two is the sum of the two previous ones. Teach it how to do so. Each number is represented by a "pile" of beepers, one on each corner of an avenue starting at the southern boundary wall. For example, on avenue 6 there should eventually be a beeper on each of the first eight streets. Start with beepers at (1, 1), and (1, 2), only, representing the first two data items: both equal to one. It will be helpful to have a method that will duplicate a pile of beepers on a given corner onto another corner. (My solution uses something from every chapter so far, including strategies and recursion.)

Chapter 8

This is here primarily so that you have something interesting as an extension to the essential material. Note that it only introduces concurrency and some of its problems. It does not provide solutions to the deep problems of concurrency. The Python solutions (synchronization) are awkward at best – obsolete at worst. The simulator has sophisticated ways to handle these problems but they are not discussed in the text (material available online).

Note that the first two sections of this chapter can be introduced much earlier. You can then write more interesting multi-robot programs.

Note that if you want to see the graphic window with moving robots, you must also write the standard top level task() function and create a window that you will send the run(task) message to. Without this, the simulation runs, but invisibly. You can use a tracer or "display" the robots to see what is going on, but the graphic window is likely what you want. The examples in the chapter don't show this detail.

8.1 Simple Concurrent Programs

This section introduces the run method that is required for threads. All robot classes implement the *Runnable* interface (but with an empty run method) so they are all ready to use with concurrency if you just override *run*. To make the simulation work, however, you also need to tell the world about the thread by passing a reference to your robot as a parameter of the World's *setupThread* method. You can put any code you like into the run method. Note that *setupThread* takes a *Runnable* argument, not necessarily a robot. You can write arbitrary classes, have them implement *Runnable* and use this to help use them in our simulator. It is quite general, in fact. Note that while you override the run method, you never call it.

The speed slider at the top of the page will speed up or delay the threads as required.

8.2 Robot Runs in its Own Thread

All we really do here is to move the *setupThread* message to the constructor. This makes every robot created in the class run in its own thread. Note that with Python concurrency once you have a *Runnable* object (like a Robot) that implements a run method, you get it to actually run by creating a new thread with a reference to your *Runnable* and then starting the thread. All of this is hidden in the *setupThread* method here. The speed control is used to speed up or slow down the thread, also hiding some detail.

8.3 Cooperation

Here we show a simple blocking protocol to permit a robot in a thread to do nothing for a while as it waits for something to occur. Notice that it can wait forever if the unblocking condition never occurs. But simple relay races can be fun.

8.4 Race Conditions

Race Conditions are one of the classic difficulties with multi-threaded programs. It is nice that we can simulate them with actual races. Finding and solving race conditions is difficult in general and we offer no advice in this simulation. We just introduce the idea.

8.5 Deadlock

Deadlock is another classic difficulty. (Starvation is another – the solutions manual has a bit more to say about that.) Also, the Dining Philosophers (Dining Robots) is a classic example of both deadlock and starvation. The code and idea is quite accessible and it is fun to watch it run. Running at a high speed makes deadlock more likely to occur. We also use a Die object. The supplied class permits dice with any number of sides, not just those physically possible, so is a general (integer valued) random number generator. Note that the key idea of sections 8.4 and 8.5 is to write a program that will fail in an interesting way.

Section	Problem map
1	
2	
3	1-4
4	Experiment with the given code
5	Experiment with the Philosopher code

Note that problem 5 is very difficult (impossible?). 6 is the ultimate beyond the horizon.

Additional Material

The Lost Beeper Mine Adventure

This was invented on a "dark and stormy night" by Jim Roberts, with (Jim says) a lot of help from Mark Stehlik. Your students can thank them. heh heh heh.

Karel is at the origin, facing an unknown direction. Karel must

-- Walk North to a wall.
-- Walk East to a wall.
-- Using Treasure Map Rules (below) follow the trail to the end.
-- At the end of the trail karel must

Face South if it arrives facing North
Face North if it arrives facing West
Face East if it arrives facing South or East

-- Walk ten blocks
-- Turn left until the front is clear
-- Choose one of the following

move to a wall if the left is clear
move 10 blocks if the left is blocked

-- Turn right.
-- Walk to a wall.
-- Face to the East.
-- Move to a beeper.

-- From this beeper: move North a distance that is half the distance just walked from the wall to the beeper, and then move East a distance that is equal to the distance just walked from the wall to the beeper.

-- This spot marks the mine. Pick up all of the beepers at the mine.

BEWARE. The mine is mostly surrounded with piles containing an infinite number of beepers.

Treasure Map Rules

Karel is standing on a corner with at least one beeper. The search will continue until karel is on a corner with exactly five beepers. From any corner karel finds that has a beeper (including the original) karel must

-- Face North if there is exactly one beeper and move to the next corner with a beeper.
-- Face West if there are exactly two beepers and move to the next corner with a beeper.
-- Face South if there are exactly three beepers and move to the next corner with a beeper.
-- Face East if there are exactly four beepers and move to the next corner with a beeper.

Note to instructors. This exercise is meant to be tailored rather than just used as is. In particular, you might want to use the Spy-Accomplice ideas of Chapter 4 to put an accomplice somewhere on the path to let the treasure seeking robot connect two otherwise unconnected parts of the path.

Appendix: Teaching Notes and a Bit of Philosophy

"May all your objects be small, and all your messages polymorphic."

This section appears on the web in only slightly different form. It was my first attempt at an instructor's guide, and has some valuable information. I include it for completeness, though it overlaps with the material above.

Introduction

This current version of Monty Karel goes beyond the original Karel the Robot and even Karel++ in two ways. First it is much more Object-Oriented than Karel++. Second, it has additional material that moves the student from the Karel World to a richer universe of programming. For an instructor learning to be an object-oriented programmer while teaching, the fourth chapter is critical, including the optional section on linked lists(on web). The first four chapters show something of the power of object-oriented thinking. Note that the IF statement has not yet been introduced or used and yet we can show various kinds of behavioral differences and even changes in behavior. We can even see a linked list implemented entirely without IF statements.

Throughout its history, from Richard Pattis' first edition to the current one, the authors have striven to make Karel as simple as possible while still teaching the essential idea of the time. The original was nearly perfectly designed to teach procedural programming using the key idea (procedures) by introducing them first among all the topics so that students have maximum exposure to them and maximum opportunity to gain skill in their use. The book was written at a time in which there was still some controversy as to whether it was good or even possible to teach procedures early. Rich showed that it was not only possible, but desirable to do so. This version takes the same approach to object-oriented (OO) programming. The key ideas here are encapsulation, message passing, and polymorphism. These three ideas are taught together from the very beginning of the book. The authors know that there is still some controversy about this as well: can/should we teach polymorphism early? The answer is yes, and Monty Karel shows how it can be done, without terminology overload. (For some thoughts and examples on the differences between procedural and object-oriented programming, see http://csis.pace.edu/~bergin/patterns/ppoop.html)

We did not go quite as far as this in Karel++ due to the nature of C++ which we were working towards at the time. It is rather difficult to get polymorphic action in C++ as you need both virtual methods and pointers. C++ is used more as a data encapsulation language (Abstract Data Types) than it is as a true OO language, though you can do OO there. Python, on the other hand, uses references ubiquitously for objects and only virtual methods (non-static ones, anyway -- static stuff is never polymorphic). So polymorphism and OO is natural in Python and we exploit it. Classes are not just Abstract Data Types. Polymorphism is the difference.

While we did not use the term Design Pattern, chapter four introduces five (at least) important, but elementary, design patterns that have proven to be both useful generally and also used within the Python libraries that we assume will be studied later. These are

Pattern:	**How Used Here:**	**Python Libraries:**
Null Object	NullStrategy class	
Strategy	Strategy class	default.strategy.py
Decorator	StrategyDecorator class	Python I/O Libraries
Observer	RobotListener class	Listener Structure, Buttons...
Iterator	Enumeration (neighbors method of UrRobot)	Enumeration, Iterator, Collections

Most of these are discussed in *Design Patterns*, by Gamma, Helm, Johnson, and Vlissides (Addison-Wesley), the original book on software patterns, though the treatment, along with a fuller treatment of object-orientation suitable for instructors, may be found in *Design Patterns Explained*, by Shalloway and Trott (Addison-Wesley). While we have not used the word "pattern" in the text, you may want to with your students after some period of study yourself. I sometimes describe the difference between naïve and sophisticated object-oriented design as being precisely design patterns. Naïve OO design looks for objects and hence classes at the level of nouns in the problem description. While this is a useful way to begin, it is design patterns that let us refine those initial attempts and create a truly usable, maintainable, and extendable program or system. We note for completeness, that the structure of the Monty Karel simulator isbased on Model-View-Controller (MVC) architecture, which is a form of Observer. (For more on Strategy and Decorator see http://csis.pace.edu/~bergin/patterns/strategydecorator.html. For more on observer and MVC, see http://csis.pace.edu/~bergin/mvc/mvcgui.html and for the use of observers in Event handling, see http://csis.pace.edu/~bergin/Java/javaeventsummary.html.)

We have included these, not to present material on design patterns to the students, but because they are all generally useful and lead to good, maintainable, and extendable programs. They are "big ideas" that can be leveraged at other parts of the curriculum to teach other things. In particular, the StrategyDecorator makes it possible to do some interesting things early, but also makes the Python I/O libraries less opaque. Instead of looking on those libraries as a problem, think of them as embodying good design decisions that can and should be emulated.

Monty Karel's core (the first six or seven chapters) is intended for only the first few weeks of an introductory course. There are two optional sections there (on the web site) that should be attempted only with care. The first introduces linked lists of strategies using something like a null object (rather than the null value) to terminate lists naturally. This material is more advanced and may be difficult to grasp for novices in their second week. It is, however, the right way to build a list in the OO style, without pervasive IF tests for the end. The second optional section is there just for those instructors who prefer to teach recursion immediately after IF and before any discussion of iteration. This material is presented again, slightly differently, in Chapter 7.

Even before starting Monty Karel, you can spend a few days teaching ideas of objects by building metaphors that the students can rely on to guide their thinking. You can use Role Play exercises to show how objects interact with each other, send polymorphic messages, and enforce encapsulation. (See http://csis.pace.edu/~bergin/Java/RolePlay.html) You can try to solve some "design" problems at the nouns and verbs level, looking for "object candidates" in a problem statement. If you do this, you can depend less on terminology and much less on technical details. It can all feel very natural when based on good metaphor. Section 9.1 (on the web, not printed in the text) is actually an attempt to work on building such a metaphor, but this can come much earlier.(See http://csis.pace.edu/~bergin/Java/OOStory.html)

Monty Karel alone is probably not yet suitable for a complete course and would need to be supplemented with additional material. Be aware that many introductory text books are not especially compatible with this approach, as many treat object-orientation incorrectly (as an add-on to procedural programming), incompletely (giving too little attention to dynamic polymorphism), or late. There is evidence from industry as well as education that these strategies fail to develop the skills needed to be an object-oriented programmer in modern languages. Some suggestions about companion materials can be found below. There is now a second volume: Beyond Monty Karel. It has enough material to complete a first course in programming using Python as a vehicle, but, again, it isn't intended as a Python reference. It extends OO programming beyond robots.

Monty Karel is best used with a teaching philosophy that admits Spiral teaching. In this style, you do not try to teach everything about any given topic at the first introduction. Instead, you teach enough to enable problem solving, knowing that you can return to the topic later at a deeper level showing more variations. So, while Monty Karel collects most of what it has to say about IF in a single chapter, that chapter does not discuss switch statements at all. Additionally, the chapter is intended to be covered relatively quickly (two hours, say, of class room time) permitting students to get started integrating the IF with other material (polymorphism, say). Note that a long chapter of a book that collects everything about IF in one place is usually pretty boring and does not permit interesting problems to be solved. The combination of IF and polymorphism is much richer, and you will soon take up iteration in another short chapter, so that you can solve really interesting problems.

Some of the ideas here are deep, but we have also striven hard for simplicity. In particular, we try to make each class very simple. We want each class as a whole, and each method in a class to be very simple. We want each object to have a single purpose: supply a single service that other objects can use. Complexity and sophistication comes from simple things used in combination, not from building complex things.

Why Not Introduce More of the Language? Problem Solving!

Monty Karel, like its predecessors does not try to introduce the complete language; Python in this case. We have been very careful to leave things out. However, Monty Karel is still a universal computing machine: a Turing Machine. Theoretically, any computational problem whatever can be solved using this system, though it is not especially convenient to to so in some ways. However, this has a distinct advantage if you want to teach students how to solve problems, and not just teach them the syntax of some language.

In particular, there is very little use of either assignment statements or of the built in data types in Python, such as int and char. We don't touch on integer arithmetic, for example, nor do we permit the programmer to ask a robot what its street number is, nor how many beepers it has in its beeper bag. To permit these things would lessen the pedagogical benefits of using Karel, not increase them. We shall explain below.

Note that in earlier versions of the Karel system we used a specialized application to simulate the robot behavior. This made it easy to control what students were able to do and not do. This time, however, we have used pure Python for the simulator. Therefore the instructor is free to introduce additional material as he or she chooses, such as integer-valued fields within the robots. We firmly believe that it is a serious mistake to do so.

On the other hand, with the current text, you can teach a course that emphasizes algorithmic thinking. We have indicated quite a bit of this in Chapter 7, especially. See the problem section for many examples.

Fundamentally, Monty Karel, like its predecessors, is about problem solving, not about language syntax. We ask the students to program in a system with only a small but complete set of tools so that they can increase their mental capabilities in solving problems in a restricted world. At some level, every language has this characteristic, of course. In Monty Karel we just make the tool set as small as possible while teaching the essential problem solving concepts: Object-Oriented problem solving concepts, actually. Karel is actually more about thinking than programming.

We think that integer fields, and the associated integer arithmetic, is an especially dangerous tool to introduce early. For one thing, integers are objects, but not "interesting" objects, and so have nothing to tell you about object-oriented programming. Even Integer objects, being final, do not permit polymorphic programming; an essential lesson of the book. Moreover, many of the most interesting exercises in the book become trivial if the student has integer fields to work with. This doesn't help them to think deeply. On the other hand, many instructors are thoroughly familiar with int fields and so it seems natural to introduce them early. We strongly suggest that you do not, and that doing so actually gets in the way of student learning.

Many of the problems in the exercise section are interesting precisely because you can't count your way around the world. Teaching recursion, especially, is enhanced because for some problems recursion is the only tool that works within this framework. There will be plenty of time to teach counting later when you introduce array-like lists and tuples, for example.

Note that one point of philosophy in the book is that we don't keep introducing new language features to solve new problems. The language introduced is very minimal. The reason we think this is important is that any language will eventually run out of features. If the student learns that whatever problem arises is solved by the next available syntax feature, she or he will never learn to use the tools at hand properly.

Polymorphism and Especially Chapter 4

Polymorphism should not be thought of as a difficult topic. At base, it only means that the object sent a message determines what to do. Actually, it doesn't **decide**. It just does what its own class definition says it should do. In Python, as in most (not all) object oriented languages, objects get their type when they are created (UrRobot(...)) and that type can never change. Independent of this type is the reference variables used to point to objects, which itself has no type. The rules are simple. It is the type of the object as defined in its class (or possibly its superclass) that determines what is done. That is pretty much the complete definition of polymorphism. In particular, if I say:

```
karel = MileMover(...)
...
karel.move();
```

then karel will move a mile, since move was defined in MileMover.

To exploit and teach polymorphism does not depend on having students memorizing this definition, however. It appears nowhere in Monty Karel. But understanding it does depend on having lots of objects with interesting interactions, in which you can mostly forget about the actual type of the objects you point to and just use interface names to declare references. In addition to making polymorphism natural, it also makes the software flexible.

This need for having lots of interacting objects led us to introduce Strategy objects as defined by the Strategy interface. This is in addition to Robots, which are, at the time Strategy is introduced, the only other kind of interesting object in the Karel World. But then Robots can interact in some ways with each other and with Strategies, and Strategies can interact with each other, etc. Note, of course, that everything in Python is an object, but that is a deeper topic than we explore in this book.

Additionally, Chapter 4 shows how many uses of IF statements in procedural programs can be (should be) replaced by polymorphism in OO programs. We don't need to test things to switch back and forth between two behavioral states, and we should not. If we start to do so, then our programs will quickly become hard to maintain as these kinds of IF statements tend to be replicated throughout programs, wherever we need to distinguish between the states. Once they are replicated, program maintenance becomes a nightmare, since program updates require these IF statements to be revisited individually to see if they need change, and the tools we use typically give us no help in finding the points in the program that require change. Instead, the polymorphic OO way captures the difference between the behaviors in objects (like Strategy objects) and delegates all action to whichever object is current. We can add new behaviors by creating new classes and we can modify old behaviors by changing (or preferably extending) old classes.

The reason that Chapter 4 appears where it does is that polymorphism is more fundamental to OO than are IF statements. Therefore it is useful to teach it early so that students both become familiar with it and have maximum opportunity to learn, practice, and exploit it. If they first learn to solve all such problems with IF statements they will have almost no incentive to learn polymorphism since few students work on programs large enough, or long enough, for polymorphism to become essential. And by that time they will have ingrained habits that will inhibit their easy absorption of the OO way. While people with little OO experience often dispute this last claim, it is well recognized in the OO community and often used to explain the difficulty of, and length of time needed for, training good procedural programmers to become OO programmers. So Chapter 4 tries to short circuit this difficulty by starting students with a good grounding in polymorphic thinking before they begin solving problems with ad-hoc selection methods.

Another good reason for this chapter here is that it emphasizes programming with objects and object interactions. We write quite a few classes here and the classes are all simple. The objects, then, are simple as well but they interact. This is a good lesson about what real OO programs look like and gives the students additional practice with it. An OO program is a swarm of interacting objects. Like a swarm of bees. Not like Godzilla meets King Kong. Chapter 4 brings the three main ideas (encapsulation, message passing, and polymorphism) together and starts to build good thought habits with them. In your own programming and class preparation, I suggest the following. When you prepare code to show your students, find any if statements in it and see if you can find a way to remove them using polymorphism before you show it to your students. You can't always do this, and it will take you a while to get used to it. You can't do it if you are programming with (non-object) primitives like int. But if you program with interesting things (objects) then you can usually remove the *if* statements and you usually get a better program. (Another trick is to see if you can't make a class you are writing smaller and more "single purpose" by delegating some of its work to other objects.)

Again, we admit that Chapter 4 introduces deep ideas and some complexity, we have tried to manage this by keeping each of the classes and methods very small. This is desirable in any case, but notice how even trivial classes can introduce deep and important ideas. For example, the LinkStrategy class (in the optional section) is almost completely trivial, and yet is the basis of a linked list--a deep idea. Combine this with the idea of a Null Object terminating the list and we have polymorphic behavior rather than testing for end. Each node in the list gets the same message. LinkStrategy nodes do something and pass it on. At the end is another strategy that just fails to pass it on without any test necessary.

Some may argue that we leave the program open to error without the test. It IS possible, for example to create a new MoveStrategy passing null for the parameter, rather than a real strategy. In this case, we maintain, the program is broken, and this flaw will appear on the first test, when we try to send null a message, though it refers to no object. The program should therefore be fixed, and we supply the NullStrategy for this purpose if there is no better choice. If you DO want to put the test for None into the constructor, what will it do when the user passes None? The program is broken, so you inform the user, of course, but do you do this with a message sent to the console? But this is just exactly what the system will do without the IF when the exception is raised. Is it good to protect the students from seeing the system provided effect of program errors: exceptions? Most likely not, since even if you try hard to protect them, they will see them eventually on their own and if you don't show them what to do, they will be puzzled. Better you show them what can occur and then what it means, and then what to do about it. In this case, replace the None with a NullObject. Replace the IF with polymorphism. It makes a better program anyway. After all, how many if statements do you typically need in a linked list built procedurally? Generally it is more than one and therefore introduces maintenance problems in a growing program.

Actually, there is one thing you can do with an IF in the above situation. When the user passes None, test for this and set the instance variable to a new NullStrategy instead of the passed value. You are, in effect, correcting a programmers mistake here. You should carefully consider whether this is a good idea and whether to teach it to novices, however. Most of the software I use thinks it is smarter than I am, and tries, on occasion, to do something "for" me. Unfortunately it is almost always wrong, and then I need to figure out how to make the thing do what I really wanted, rather than what its programmer thought I "must have" wanted. Formatting in word processors is a frequent villain in this regard, of course.

Decorator, and Observer

Design patterns have become very important in modern OO programming and are exhibited in the libraries of many languages, but in Java more than in Python. The Monty Karel simulator is based on the observer pattern. Robots are Observable and notify the

RobotWorld when they change (move, turn, …). The graphics system in TK is also an observer. The importance of these ideas led to incorporation of the basic ideas of a few such patterns in the book, especially Decorator and Observer. This will be continued in Beyond Monty Karel.

Interfaces

Python doesn’t have formal interfaces. But interfaces are a really good idea. An interface defines a type and its protocol without defining the implementation. Any class that inherits the interface can be guaranteed to conform to the protocol. An interface is actually the first Python code I show my students (during a Role Play exercise before we begin programming). Conceptually it is key to abstract thinking about programming, since it is completely free of implementation. In some ways UrRobot should be an interface, but is not. The reason is that I need to hide lots of implementation for UrRobot to make the graphic simulator work. Since it is a class, I can do the implementation of this rather than having students need to repeat it.

As proof of this, notice that, aside from naming, there is no difference between the Controller interface and the Strategy interface. The only difference is in intent. Yet we can cover these rather different ideas with the same structure. Likewise, our decorators and the optional LinkStrategy are identical in structure. This is alluded to in one of the exercises. We give different diagrams for the two ideas, but they are equivalent, as your best students will probably grasp immediately. In fact, the kind of linked list we build in the optional section can be fruitfully thought of as a nested structure, since we don't show any way to get access to the individual nodes. In this regard it is like a list in Scheme. If you use TeachScheme! before Monty Karel, you can exploit this, perhaps.

Linked Lists and Recursion

In the optional section on Linked Lists, the doIt method of the LinkStrategy calls doIt. The question arises whether this is recursion or not. I usually take the position that it is not. In this case doIt is a message to another object. By the principle of polymorphism we therefore don't know what code will be executed. It might be this same code, executed by another object, or it might be different code altogether. Indeed it **must** be different code eventually if the sequence of messages is to end. I usually only call it recursion if a method sends the same message to the same object (self). In reality this doesn't matter, since you need to guard against infinite execution in all circumstances anyway. Here we don't need an if to guard, since we just guarantee that eventually we execute code that doesn't try to pass on the message.

Object Think

I'd like to think that the book puts students on the road to becoming good object-oriented programmers, while giving them an interesting world in which to play and learn. But the instructor should have some idea about where this road leads. To program naturally in an OO way requires a different way of thinking than that of a typical C or Pascal programmer. Let me point out some of the characteristics.

- Prefer polymorphism to if statements
- Use interfaces define protocol for the important classes.
    - Avoid complex class hierarchies.

    - Inheritance is a logical concept, not a code saving device.
- Treat delegation (Chapter 4) with just as much respect as inheritance (Chapter 3).
- You seldom write helper methods, but you often build helper objects. (delegation again)
- Composition of parts is not a good use of inheritance. Composition and delegation are the same, actually.
- Completely initialize all your objects in the constructor. Establish any invariants there.
    - Maintain invariants in all public methods.
- Prefer immutable objects, without mutators. (eases program analysis)
- Use strategy objects to encapsulate behavior that you may want to change.
- Reuse does not mean building hierarchies of classes that reuse "code" from the super class. Instead it means putting in hooks and holes into which you can drop a variety of things. Reuse is not reusing the *stuff*, but reusing the *holes*. (interfaces and strategies again).
- Prefer lots of simple objects to a few complex objects. Complexity is in the interactions, not the objects.
- To build a big program, build a simpler small program first and then extend it.

Notes. Maybe you were wondering...

Python has default values for parameters, including in constructors. We haven't made essential use of these for the most part. Robots have optional "fill" and "outline" arguments in the constructor (strings) which can distinguish robots when there are more than one.

In Chapter 4, where we introduce instance variables, note that we are very careful never to construct an object that has an unusable state. These are defined in the constructor and usually have a name beginning with an underscore, indicating they are for use within the class only. We generally avoid initialization with the None value. While our purpose here was to avoid if tests for None later, this has other benefits as well, such as simple but safe construction. Of course it IS possible for a user to pass None as a constructor parameter and defeat our careful plans. This is one situation in which a null test would be preferable in production code, but at this point we are trying to build the habit of supplying objects, not None, for parameters and have not mentioned the possibility of passing None to the student. Our position on this would be that doing this breaks the program and it should be fixed. The exception that will be thrown will indicate the error and a NullStrategy or other Null Object can then be used instead of null to fix the problem.

Notice that we always give initial values to references and other variables we create. This is just a good habit. Don't assume the system will do the right thing.

One of the teaching styles used in the book is Metaphor. In many cases a new topic is introduced via a metaphor that attempts to give the key features of the new idea in a familiar way. Care is taken to choose metaphors that aren't misleading, and to point out the limitations of metaphors we choose. For example, the term Observer (Chapter 4) can be misleading as the usual meaning isn't just what we actually build in Python. Therefore both the usual meaning and the actual meaning needed here are explained with a real-world example that we hope is meaningful to students. Generally this metaphor is given before the technical details. Robots themselves are an elaborate metaphor for objects, actually. And physical robots are a metaphor for our robots as well.

There are other teaching styles, however. One that we have seen to be effective is to introduce a new topic by showing how the currently known skills don't quite solve some new problem. They might almost do so, but not quite. In this teaching style, a new topic is introduced with an example and a solution is attempted. The solution can be shown to have flaws. The students and instructor can discuss the good and bad points of the proposed solution. Then the instructor can introduce some new material that improves the solution. Eugene Wallingford uses this technique frequently and effectively. By preparing some material ahead, you can do this too.

Bibliography

Bergin, Polymorphism: As It Is Played, Slant flying Press, 2015

Bergin, Polymorphism Companion, Slant flying Press, 2015

Bergin, Beyond Monty Karel, Slant Flying Press, 2023

Pedagogical Patterns Editorial Board (Bergin editor), Pedagogical Patterns: Advice for Educators, Joseph Bergin, Software Tools, 2012

Shalloway and Trott, *Design Patterns Explained: A New Perspective on Object-Oriented Design*, Addison-Wesley, 2002

Gamma, Helm, Johnson, and Vlissides, *Design Patterns*, Addison-Wesley, 1995

Hunt, Java *for Practitioners*, Springer, 1999

Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2000

Some textbooks should not be considered. Some do objects too late. Some do Python GUI programming using procedural programming (lots of if statements in the listeners, rather than separate listeners -- or even hide the listeners altogether and just use if statements to dispatch or hide events.). Some authors have serious misconceptions about how objects and especially inheritance should be used. In particular, Circle is NOT a proper subclass of Point, and Cylinder is NOT a proper subclass of Circle. Don't use books that suggest these as examples.

Many people have the misconception that OO is all about Classes. It is not. It is about polymorphic run time dispatch. Many people have the misconception that OO is all about code reuse. It is not. It is all about piecemeal growth of programs. To write a big program, write a small program first and then grow the big one. To do this well requires that you use polymorphic techniques and design patterns. See the, extremely helpful, Refactoring book for more. You can design a sequence of exercises for your students that teach this, by the way. Early exercises yield parts of the solution of larger problems. Even better, have your students solve these problems in groups. Still better is to have them exchange early work for later extensions. There are additional suggestions about such things in the pedagogical patterns papers on Bergin's web site. See especially Active Learning (http://csis.pace.edu/~bergin/patterns/ActiveLearningV24.html), and Feedback (http://csis.pace.edu/~bergin/patterns/FeedbackPatterns.html).

More on Elementary Patterns, Pedagogical Patterns, and teaching and using Objects can be found on Bergin's home page: http://csis.pace.edu/~bergin. All of the papers pointed to above are directly linked from this home page.

You are encouraged to submit ideas on good teaching practice to the Pedagogical Patterns Project. The author will be happy to show you how good ideas are turned into patterns and then verified by the community as good practice. All pedagogical patterns (as other patterns) require an extensive peer review process before being accepted. All good ideas require an abstraction process to enhance usability by others in contexts beyond what any one user may envision. Thus good patterns are not specific to a certain time/place/course, but can be transferred.

Note. Bergin is the principle author of Monty Karel and made the key decisions on its form. The "we" above, usually means Bergin. However, the ideas presented are widely shared in the OO community and also in the Elementary Patterns Working Group and the Pedagogical Patterns Project. This is not Bergin's sole work, however, as it is built on a firm foundation begun by Rich Pattis and extended by Mark Stehlik and Jim Roberts. Pattis provided a key core for teaching procedures properly and simply and Stehlik and Roberts added nice pedagogy. Bergin's contribution has been to morph the concepts so they fairly and completely represent OO programming.

"As simple as possible, but no simpler."