

Table of Contents

[Chapter 1](#) Basics: Primitive instructions, tasks, Ruby's world.

[Chapter 2](#) User defined instructions

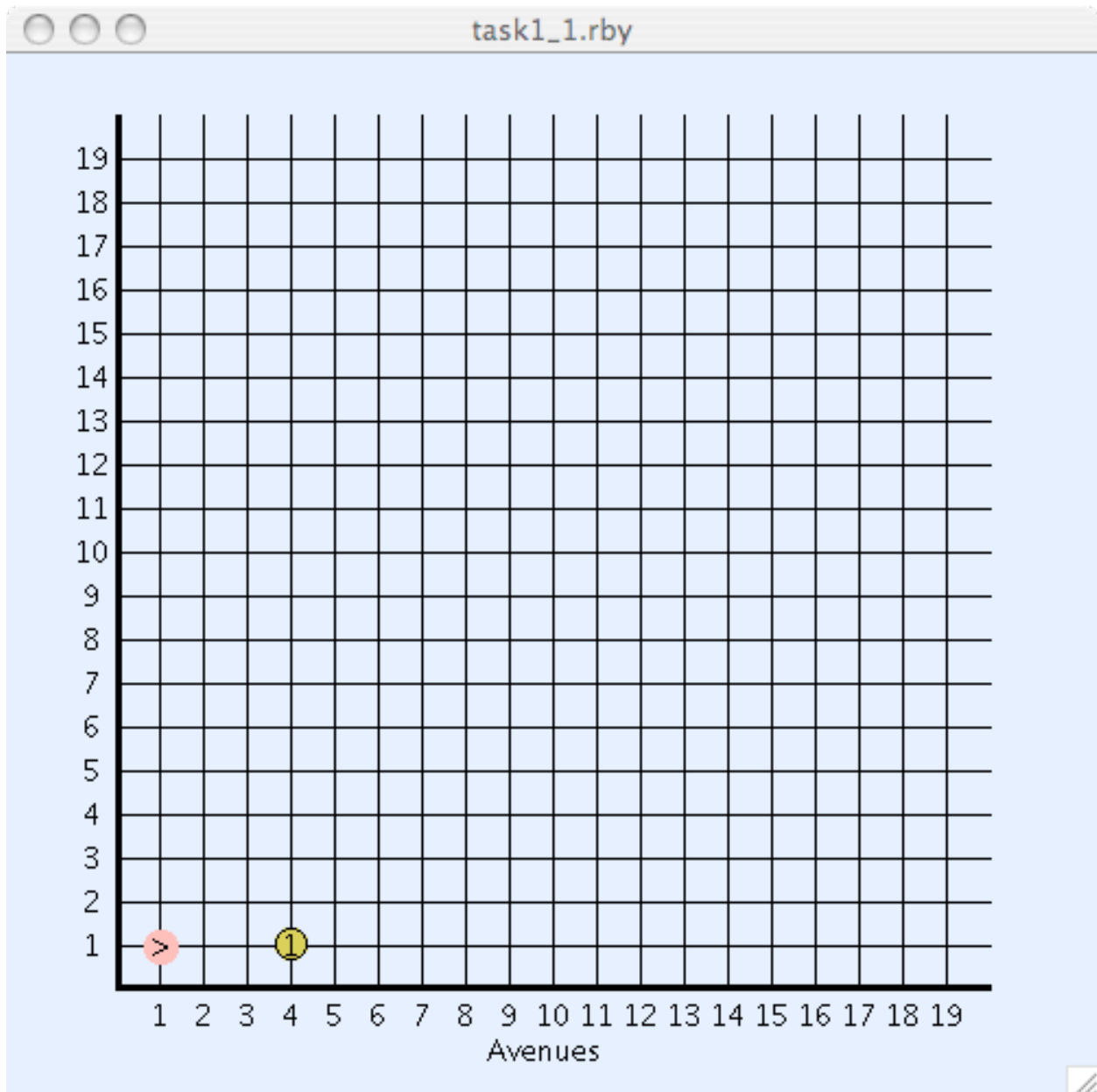
[Chapter 3](#) Conditional instructions

[Chapter 4](#) Repeating instructions

Chapter 1 - Elementary Robot Programming

Ruby's World

Ruby the robot lives in a very simple world consisting of streets and avenues. It is not as exciting as our world (no pizza shops, no cell phones, no snowboarding), but it is complicated enough that Ruby can be programmed to do some fairly complex tasks. It looks like this:



Here we see Ruby standing at the corner of 1st Street and 1st Avenue, facing east. This corner is sometimes called (1,1), for brevity. This intersection is the only one with a special name; it is called “the origin”, or sometimes, “home”. There are walls of impenetrable nopassium to the west of First Avenue and to the south of First Street; these keep Ruby from falling off the edge of the world. And there is a coin at (1,4).

Ruby’s Instructions

Ruby understands only two instructions to move around in her world:

- `move()`; -- moves one block straight ahead,
- `turnLeft()`; -- turns left 90 degrees.

These instructions must be written exactly as: `move()`; and `turnLeft()`; -- the words, `move` and `turnLeft`, must be followed by `()`; or they will not be recognized (Ruby is, after all, a mindless automaton). The capitalization must be all lowercase, except for the L in `turnLeft`; again, if not, they will not be recognized (Ruby, like other automata, is very literal-minded).

The only other thing in Ruby’s world (besides streets, avenues, and walls) is coins. There can be any number of coins at any intersection. In the diagram above, there is just one coin at (1,4). Ruby understands two instructions to interact with coins:

- `takeCoin()`; -- takes one coin from the corner where Ruby is standing
- `putCoin()`; -- puts one coin on the corner where Ruby is standing

Ruby carries a bag to hold coins. It can hold many, many, coins; you won’t have to worry about it getting full (think of it as a pocket universe, about the size of our own).

The only other Ruby instruction is:

- `halt()`; -- turns Ruby off.

You must tell Ruby to `halt()`; at the end of every Ruby program (otherwise her battery would run down).

Ruby Tasks

A task for Ruby consists of an initial state and a final state. As you could guess the initial state is the state Ruby starts in; it includes Ruby’s location, direction, the number of coins in her bag, and the number of coins on what intersections.

The final state is how Ruby and the coins in the world must be when she halts. It is specified exactly like the initial state.

Ruby Programs

A program is a series of instructions. (This is a definition worthy of remembering.) A program is written to accomplish some task. Ruby’s first task is to retrieve the coin (see

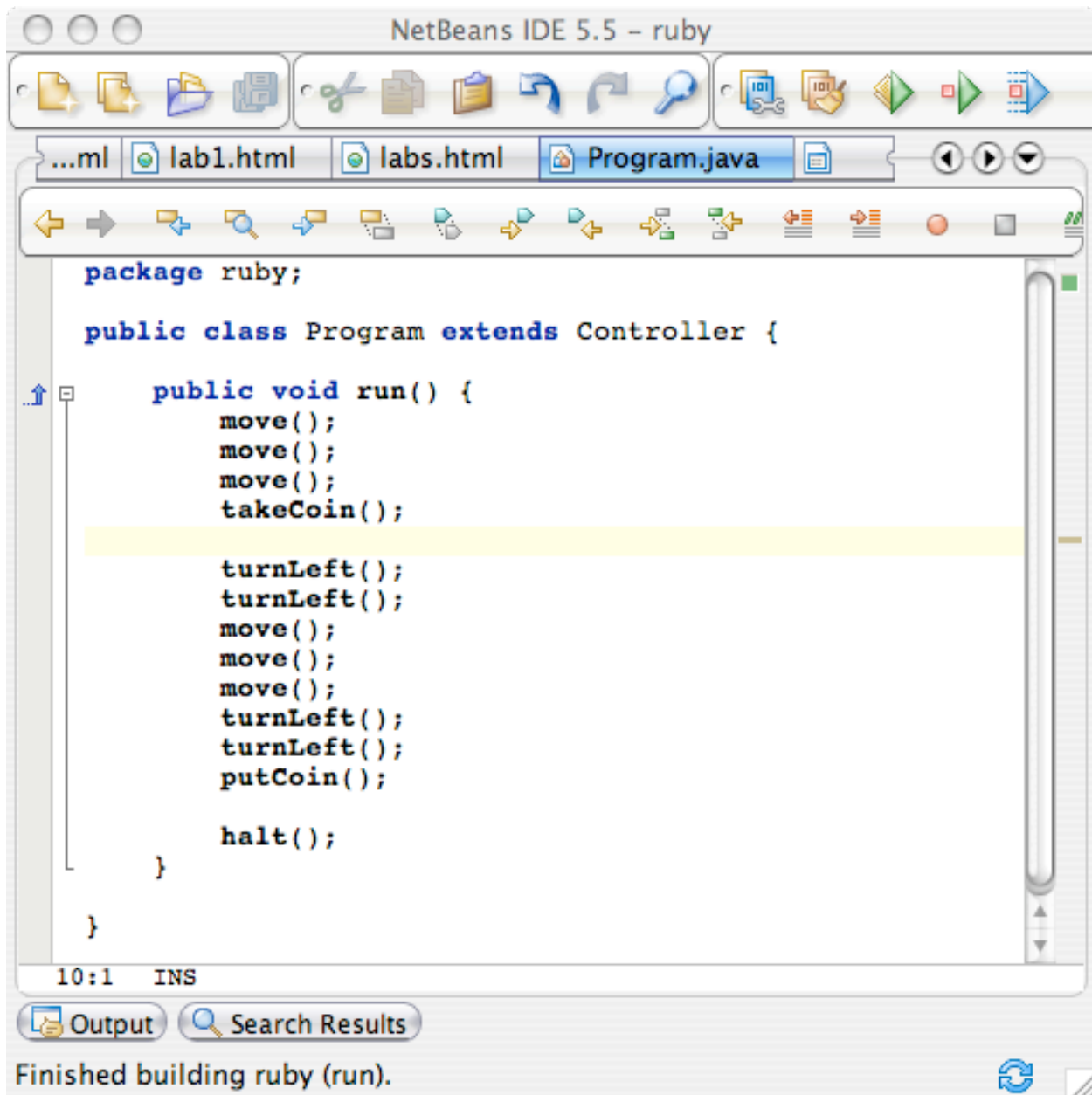
diagram above), set it down at the origin, face east and halt. That can be accomplished by the following series of instructions:

```
move();
move();
move();
takeCoin();

turnLeft();
turnLeft();
move();
move();
move();
turnLeft();
turnLeft();
putCoin();

halt();
```

To allow you to program Ruby and see what your programs do, your instructor has written a robot simulator that uses the Java programming language. It allows you to write Ruby programs in Java in the NetBeans environment. In that context, the Ruby program for her first task looks like:



To program Ruby to accomplish other tasks for your labs, you will edit the program between the {}'s after run().

What could go wrong?

Programming can be tricky, partly because it is so easy to make little mistakes and forget things -- but, mostly because computers have no intelligence, and are incredibly literal-minded. In the computer/human system that comprises the programming environment, you, the human, will have to adjust to the machine's foibles, because given the current state of computing, it will certainly *not* adjust to you. If you are willing to make the same mistake 1000 times, it will give you the same error message 1000 times. You

are the adaptive system here; for your programs to work, you will modify your behavior (since the machine is not capable of it!). It will help you to find (and thus correct) your errors if you realize that there are different kinds of errors. Errors come in four flavors: lexical, syntactic, runtime, and intent.

Lexical errors

An unabridged dictionary contains the complete lexicon of a language. Lexical errors happen when you use words Ruby does not know. Typing errors, and memory errors cause lexical errors.

- If you type `mvove();`, instead of `move();`, that's a lexical error.
- If you type `turnleft();`, instead of `turnLeft();`, that's a lexical error.
- If you type `getCoin();`, instead of `takeCoin();`, that's a lexical error.

Lexical errors will be highlighted by red marks in NetBeans.

Syntactic errors

Syntax is another word for grammar. Java has a very strict, limited grammar. Every instruction must have `()`; after it. If you omit any of those three, it's a syntax error. If you edit any part of `Program.java` outside the `{}`'s that follow `public void run()`, you are extremely likely to cause syntax errors. For now, that's all the syntax errors you are likely to see.

Like lexical errors, syntax errors will show up underlined in red.

Runtime errors

Ruby only has five instructions, but three of them can cause runtime errors by trying to make Ruby do impossible things.

- `move();` will fail if Ruby is facing a wall (1/2 a block away).
- `takeCoin();` will fail if Ruby is not standing on a corner with a coin.
- `putCoin();` will fail if Ruby has no coins in her bag.
- `turnLeft();` will never fail! How could it?
- `halt();` will never fail, either, but omitting it will cause an intent error.

Intent errors

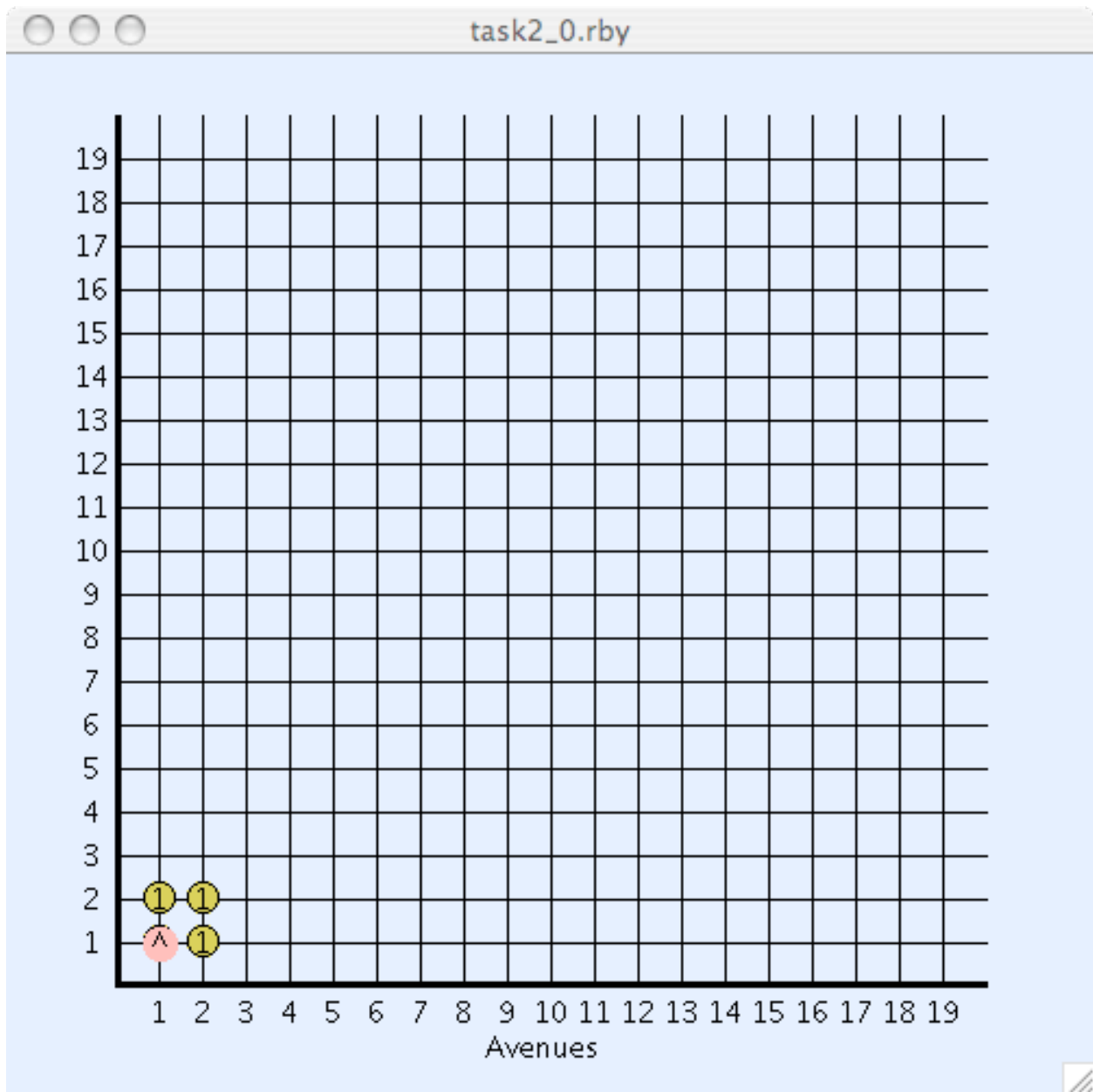
Intent errors can happen when none of the other three types of errors occur, but Ruby does not finish her task correctly. For instance, in the task above, if you removed the `putCoin();` instruction at the end, after Ruby halted, she would not have completed her task (which is to bring the coin back to the origin and put it down). The easiest intent error is forgetting to halt Ruby after completing her task.

Chapter 2 - Defining New Instructions

Your programs should be beautiful. Perhaps you would not have thought that aesthetics was part of programming, but it is. Well conceived, nicely written programs are beautiful because they not only work correctly, but also because they are easy to understand and thus to debug. Debugging (removing errors from) a program is the most difficult and frustrating part of programming.

You have learned the five primitive Ruby instructions, `move`, `turnLeft`, `takeCoin`, `putCoin`, and `halt`. They were enough to accomplish the simple tasks in Chapter 1, but the programs required were rather long. With such long programs it is very easy to make mistakes, and sometimes hard to tell exactly how to fix those mistakes. The tasks in this chapter are more difficult, but the programs will be shorter because you will learn to define new instructions.

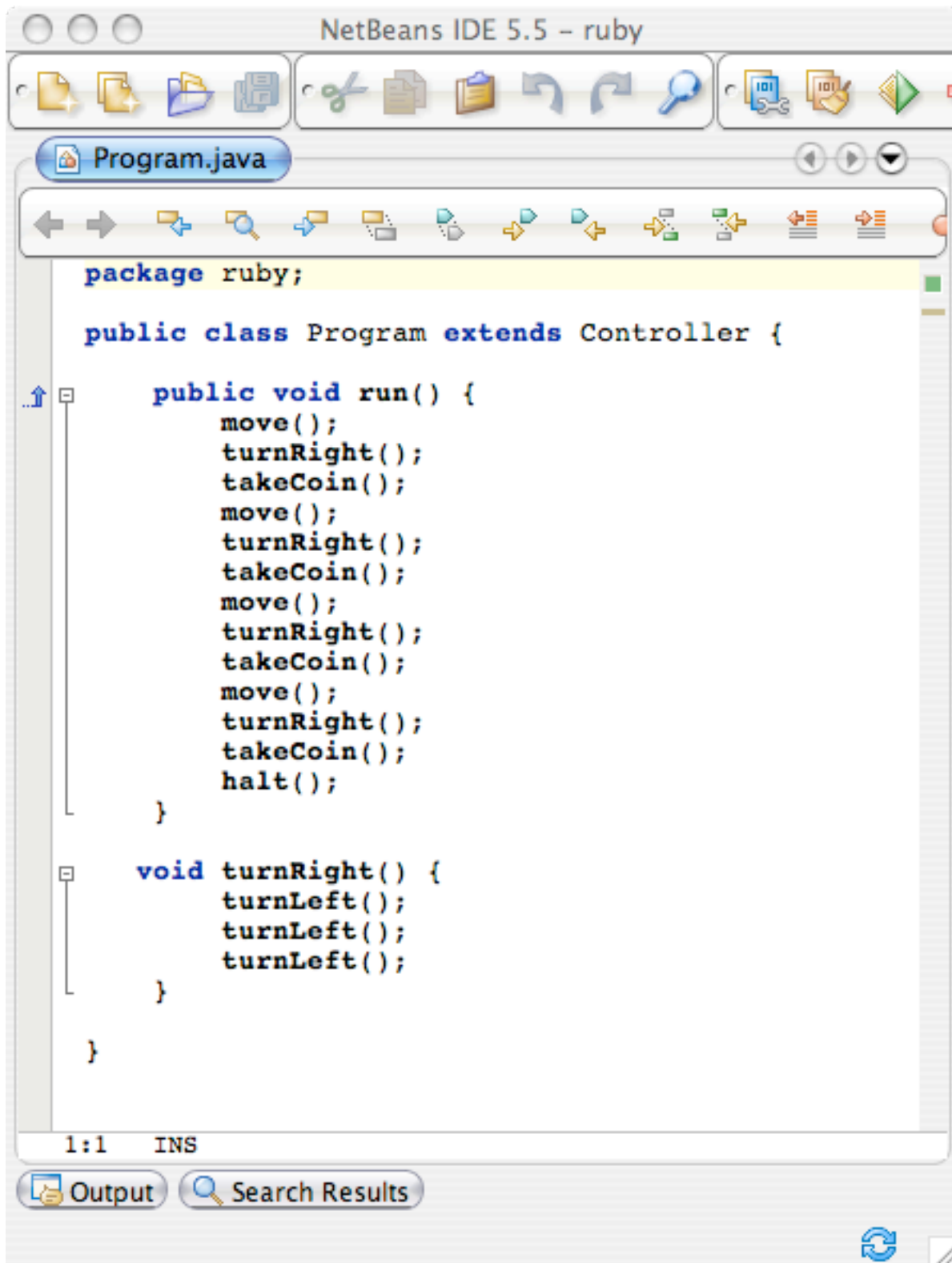
Here is task 2_0: the initial situation has Ruby at home facing north, with coins on (1,1), (1,2), (2,1), and (2,2); the final situation is Ruby at the same place, same orientation, with all four coins in her bag (see figure, below). One approach to this task would be to do the following four times: `move`, `takeCoin`, and `turn right`. But, there is no built in `turnRight` instruction. It is inconvenient to turn left three times when you wanted to turn right.



Fortunately you can define new instructions. To define a `turnRight()` instruction, you write:

```
void turnRight() {  
    turnLeft();  
    turnLeft();  
    turnLeft();  
}
```

Here's a program for `task2_0` with a `turnRight()` instruction defined.

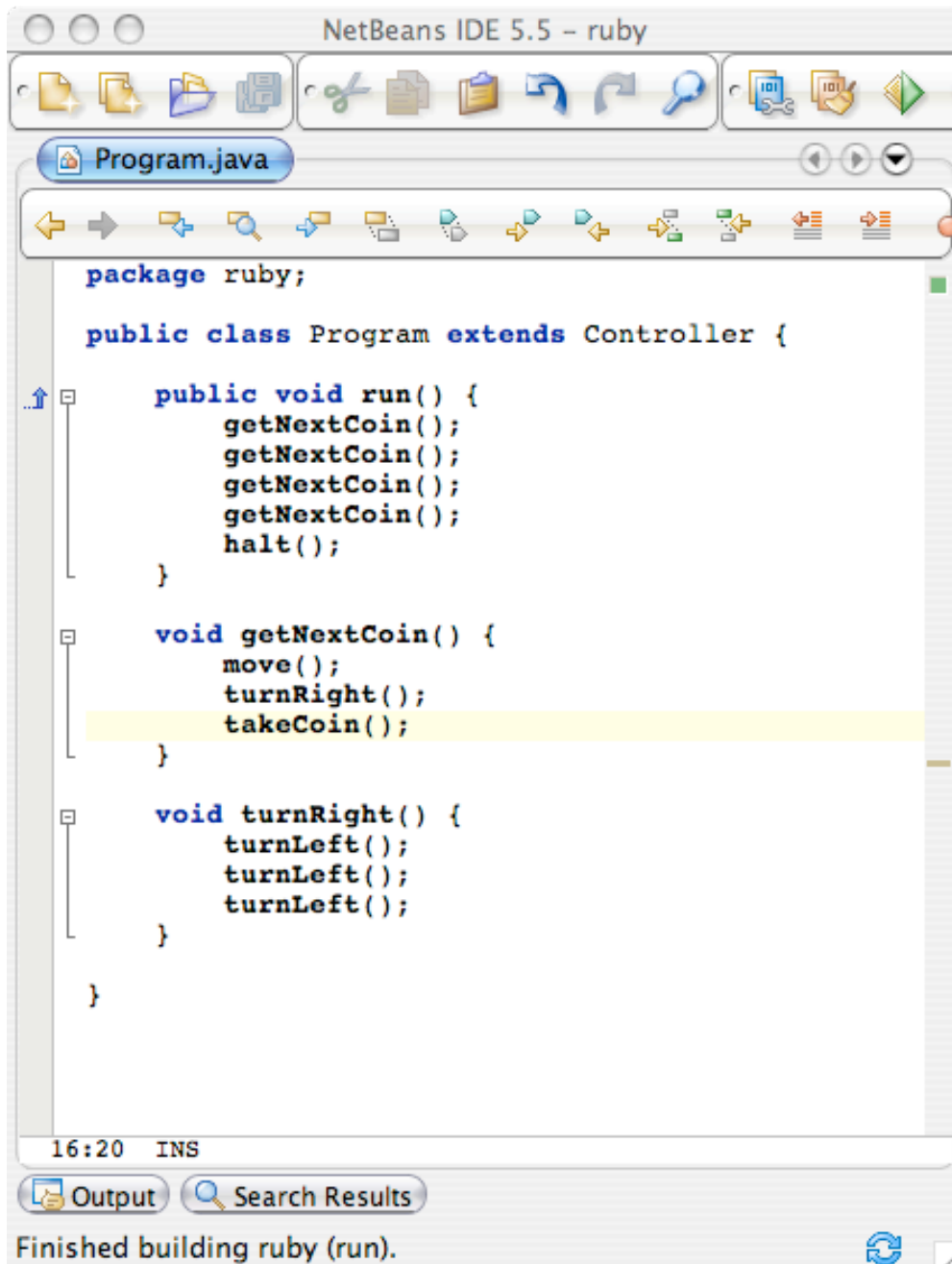


To define a turnRight instruction, starting after the closing curly bracket of run(), but before the closing curly of Program, you type, void, then the name of the new instruction,

then (), then open curly, then (on the next line) the subprogram (i.e. the series of instructions to make the new instruction do whatever you want it to do), then a close curly.

The run() method, does exactly the same thing four times (i.e. move, turnRight, takeCoin). This is a clue that you should write a new instruction. As you get used to programming, you will get good at realizing beforehand what would be good instructions to write.

It is important to chose descriptive names for new instructions so that you will not have to remember what they do. A pretty good name for this one might be getNextCoin. Here's the program rewritten with that instruction included:



```
package ruby;

public class Program extends Controller {

    public void run() {
        getNextCoin();
        getNextCoin();
        getNextCoin();
        getNextCoin();
        halt();
    }

    void getNextCoin() {
        move();
        turnRight();
        takeCoin();
    }

    void turnRight() {
        turnLeft();
        turnLeft();
        turnLeft();
    }

}
```

16:20 INS

Output Search Results

Finished building ruby (run).

Not only is it shorter than the original, it is easier to understand!

Notice that both the new instructions are between the close of run() and the close of the class (i.e. the final close curly).

Stepwise refinement (a problem solving technique)

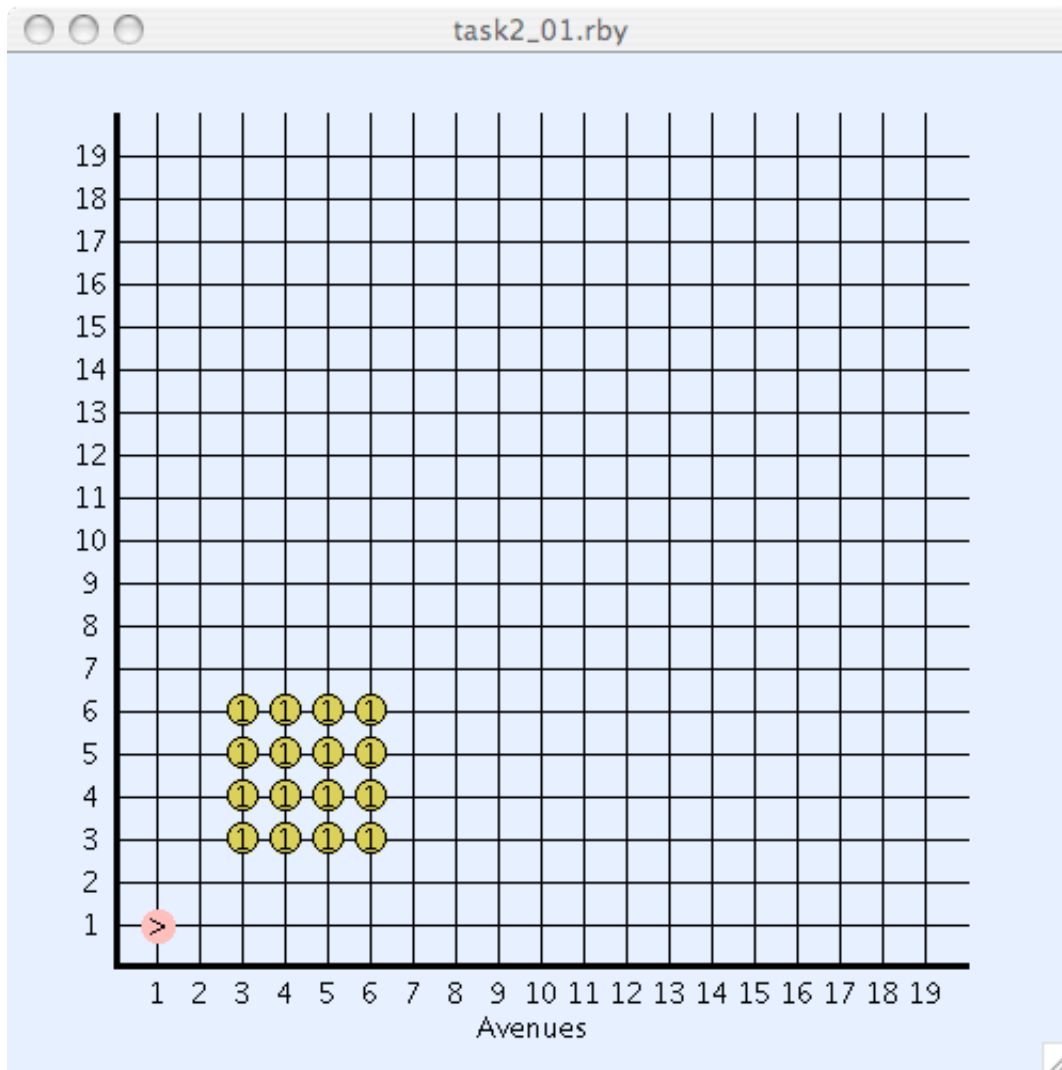
Normally, if a program is difficult at all, after reading the task description, and before beginning to write the program, programmers do what is called design. When the task is simple, and there is an obvious way to do it, there's no need for design; but when the solution is not so easy or obvious, good design can make the coding much simpler. Design is the careful thinking about how the program will be shaped. Time spent carefully designing your programs will be repaid because debugging them will be easy. A standard technique for designing programs is stepwise refinement.

Here's how to solve a problem by stepwise refinement:

```
if the problem is trivial
  just solve it
otherwise (i.e. it is too big, or too complicated to solve simply) {
  break it into 3-7 subproblems
  for each of the subproblems
    solve the subproblem by stepwise refinement
}
```

The last line might seem to threaten infinite regress, since stepwise refinement is used as part of the definition of stepwise refinement. But, since each time we restart the stepwise refinement it is with a smaller problem, it can't be infinite. Defining a procedure which utilizes itself has a special name; it is called recursion.

In her next task Ruby has taken up farming. Her tomatoes are ripe and ready for picking:



Her task is to harvest all the tomatoes, bring them back to the origin, and face east.

This task can be seen as having three parts:

```

go to the tomato field
pick all the tomatoes
go home

```

This is a common type of task, with three parts: 1) get started (or initialize), 2) do what needs to be done, and, 3) finish up.

So the run() method could just be:

```

public void run() {
    goToField();
    pickEm();
    goHome();

    halt();
}

```

The first and last steps are easy, you can write them in a few lines; the second is a little more complicated. An obvious approach is to pick up one row, four times. To pick a row, she will take a coin and then move; repeating that 4 times (maybe that instruction would be `pickAndMove()`). One difficulty is that after the first row, Ruby must turn left to the second row, and after the second row she must turn right. Stitching together instructions so that they work together, one after another is another skill you will develop. You must be very careful to pay attention to the precise state of the robot at the end of each instruction, so the next one will execute properly.

The `pickEm()` instruction might look like:

```
void pickEm() {
    pickARow();
    positionForNextRow();
    pickARow();
    positionForNextRow();
    pickARow();
    positionForNextRow();
    pickARow();
}
```

Except, positioning is different for every other row, so it would have to look like:

```
void pickEm() {
    pickARow();
    positionForNextRowLeft();
    pickARow();
    positionForNextRowRight();
    pickARow();
    positionForNextRowLeft();
    pickARow();
}
```

So maybe a better way would be:

```
void pickEm() {
    pickARow();
    pickARow();
    pickARow();
    pickARow();
}
```

...and write `pickARow()` to move to the beginning of the next row after picking this one...

There are many ways to write a program to accomplish a particular task, and while some are definitely better than others, there is not one right way.

New instructions -- conclusion

Well designed new instructions can make your programs shorter and easier to debug. Well named new instructions can help you (or someone else) understand and appreciate the beauty of your program. Here are two maxims for writing new instructions:

1) The body of every new instruction should have 5 ± 2 instructions (i.e. 5 plus or minus 2; from 3 to 7). That is because people can only keep that many things in mind at once, and you want people (including yourself) to be able to easily grasp what each new instruction does at a glance. Long sequences of primitive instructions are not easily understandable at a glance.

2) The body of your `run()` instruction should be readable by your mother, or your roommate. That means it should be 5 ± 2 instructions with meaningful names. For instance, the baseball diamond cleaning task might have a `run()` method like:

```
public void run() {
    goToDiamond();
    pickUpTheCoins();
    goHome();

    halt();
}
```

Anybody could understand what that does (even though the context and details are missing).

3) Use stepwise refinement to create simple, easy to understand programs; such programs are beautiful.

Chapter 3: Conditional execution

So far all your programs did exactly the same thing each time they were executed. When Ruby was working as a waitress, the tips were always in exactly the same place; when she was a hotel maid the guests never moved the wastebaskets. In the real world, things are not quite so predictable. Customers might leave tips anywhere on the table; hotel guests may move wastebaskets anywhere in the room, or even take them home. No hotel manager will put up with a robot that turns itself off whenever the wastebaskets are moved.

Conditions Ruby can test

To deal with the vagaries of life, Ruby has some primitive senses. She can test the following conditions:

- `facingEast()` - true if Ruby is facing east
- `facingWest()` - presumably the others are self-explanatory?
- `facingNorth()`
- `facingSouth()`

- `frontIsBlocked()` - true if there is a wall immediately in front of Ruby
- `rightIsBlocked()` - true if there is a wall immediately on Ruby's right
- `leftIsBlocked()`

- `coinInBag()` - true if Ruby has at least one coin in her bag
- `nextToACoin()` - true if Ruby is standing on a corner that has at least one coin

The if statement

When you want Ruby to do different things under different conditions you use the if statement. It allows Ruby to either do something or not, depending on circumstances.

If Ruby were standing on a corner where there might be a coin, she could pick one up if it were there by:

```
if (nextToACoin())
  takeCoin();
```

This if statement would either pick up a coin, or do nothing, depending on if there were a coin on the corner with Ruby.

Problem Solving Technique -- Analysis By Cases

It is very common when writing programs, and in problem solving in general, that one must do different things in different cases. For instance, if you are running under a fris-

bee, if it was thrown forehand, you expect it to tail off one way, if it was thrown back-hand, the other. If it was thrown as a hammer (up-side-down) you expect it to slow down rapidly and tail off abruptly. In each case you do different things to catch it.

PS: Analysis By Cases (ABC for short) is a problem solving technique designed especially for problems with multiple cases.

Analysis By Cases (ABC)

Identify the various cases. For each, answer the following questions (making a table if it is complicated): 1) How can you distinguish this case? 2) What action do you wish to take in this case?

Once you have identified each case, decided how to distinguish each case from the others, and what action to perform in each case, you are ready to write code. The examples will illustrate the use of this technique.

faceEast() instruction

You can use ifs to write an instruction to make Ruby face east (which would be useful at the end of task where she must end up facing east). This instruction must work in four different cases; where Ruby is initially facing east, west, north or south. Using ABC, you can distinguish between these cases directly by using `if (facingNorth())` and etc. What action must Ruby take in each case? If facing east, she's done; if west, Ruby needs to turn around, if south, `turnLeft`, and if north, `turnRight`.

One way to write this is to use three ifs; like this:

```
void faceEast() {
    if (facingNorth()) {
        turnLeft();
        turnLeft();
        turnLeft();
    }
    if (facingWest()) {
        turnLeft();
        turnLeft();
    }
    if (facingSouth())
        turnLeft();
}
```

Notice that when you wish to do more than one thing in the conditional part of an if, you must surround them by `{}`s, but if there is only one thing the `{}`s are optional.

That `faceEast` will work, but it can be shortened as:

```
void faceEast() {
    if (facingNorth())
        turnLeft();
    if (facingWest())
```

```
        turnLeft();
    if (facingSouth())
        turnLeft();
}
```

If Ruby is initially facing north, the first if will face her west, the second, south, the third, east.

If Ruby is initially facing west, the first if will do nothing, the second will face her south, the third, east.

If Ruby is initially facing south, the first and second will do nothing, and the third will face her east.

If Ruby is initially facing east, none of the three will do anything.

Which version do you like better?

if-else

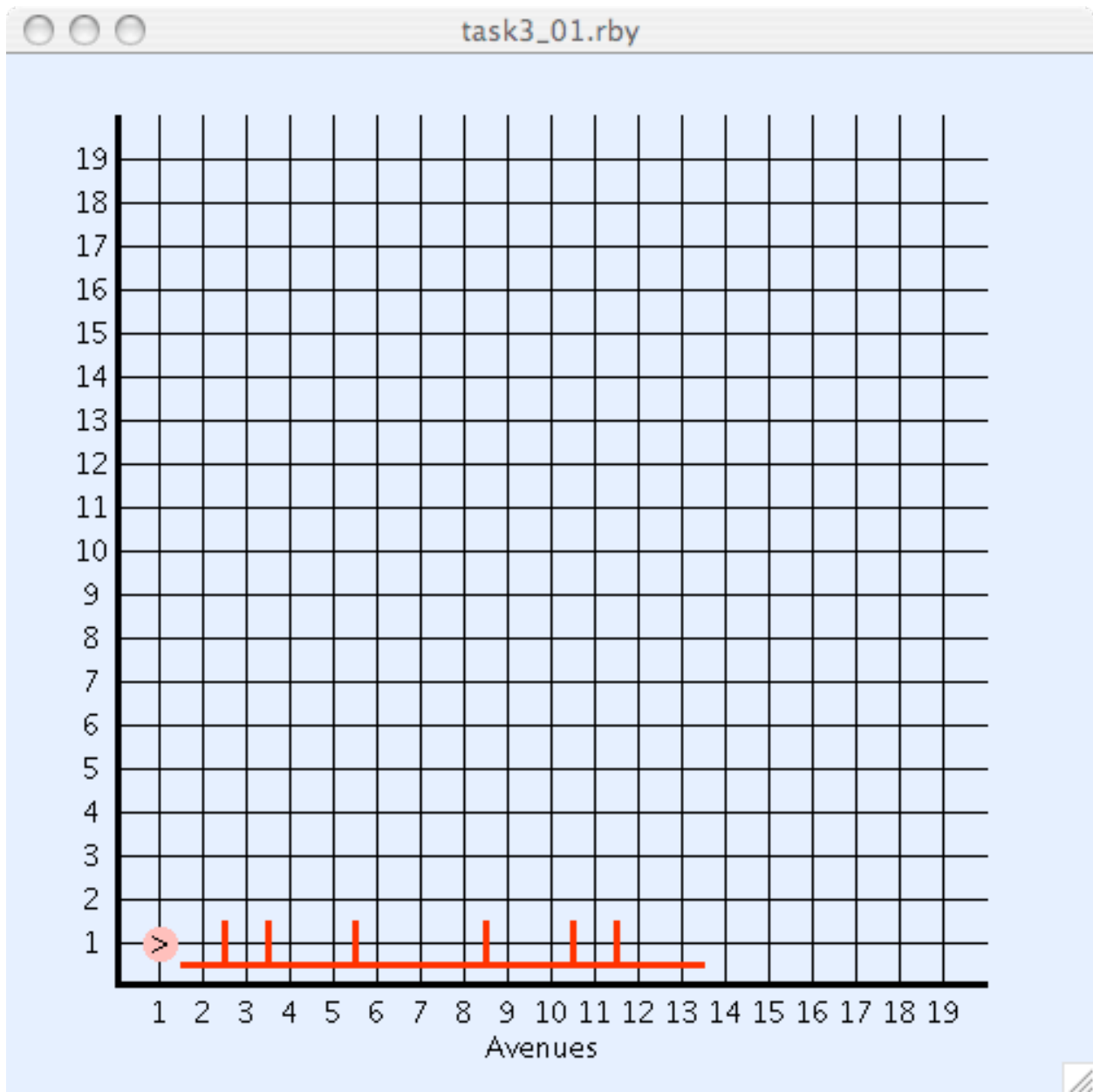
The if statement either did something or did nothing. That is fine if it solves the problem you are addressing, but sometimes you want to do either one thing or another. In that case you use an if-else statement. It looks like this:

```
if (condition)
    doOneThing();
else doAnother();
```

It starts out just like the plain if statement; if the condition is true, it executes the statement that comes next. If, on the other hand, the condition is false, it executes the statement that comes after the else. So, above, if condition is true it would do one thing, else it would do another. Here's an example:

Task 3_01: Ruby has gone out for track and is running a 12 block hurdle race. Each block there is either a hurdle (one block high) or no hurdle. If there is no hurdle, Ruby should just move forward; if there is a hurdle she should jump over it, coming down right on the other side. Of course she should not jump higher than one block (as that would slow her down) and she is not a flying robot, so she must come down after each hurdle.

Pranksters move the hurdles at unpredictable times, so Ruby does not know which blocks will have hurdles in a particular race.



To accomplish this task Ruby must advance one block 12 times (each time either jumping a hurdle, or just moving forward depending). You could either explicitly use the ABC method, or simply write the code. Like this:

```
void advanceOneBlock() {  
    if (frontIsBlocked())  
        jumpHurdle();  
    else move();  
}  
  
void jumpHurdle() {
```

```

    turnLeft();
    move();
    turnRight();
    move();
    turnRight();
    move();
    turnLeft();
}

```

To complete the task, Ruby must do this 12 times, either like this:

```

public void run() {
    run12Hurdles();
    halt();
}

void run12Hurdles() {
    run4();
    run4();
    run4();
}

void run4() {
    advanceOneBlock();
    advanceOneBlock();
    advanceOneBlock();
    advanceOneBlock();
}

```

or using a for loop, like this:

```

public void run() {
    for (int i=0; i<12; i++)
        advanceOneBlock();

    halt();
}

```

Syntax and semantics of the if statement

BNF, Java and Adaptive Systems

When you write code for the Ruby simulator, you are secretly writing Java code. The Java compiler, like any contemporary compiler, is very literal and rigid. It insists that source code match its grammar, symbol by symbol. Any deviation will result in compiler errors; and errors prevent the compiler from producing byte code, and without byte code you can't execute your program.

Syntax errors are just details, but they are details that can stop you. If you omit, or misuse a comma in an English paper, you may be corrected by your teacher, or lose points, but your English teacher can still understand your meaning. The compiler, by contrast, will cut you no slack. If a program is missing a semicolon, or has a misspelled word, no matter how many times you compile it, it will still generate an error, and will still not run. In the person/compiler system, the person must make the adjustment, the compiler will not. Fortunately, once you know the BNF, you will at least know what the compiler is looking for.

The compiler will only recognize code that conforms to a simple, strict grammar. That grammar is described in what is called BNF (Bachus-Naur Form). Here is some BNF Notation:

::= means is defined as
<x> means one thing of type x
| means or
[x] means optional x
[x]* means 0 or more xs

Learn this BNF notation summary if you wish to write code without mysterious errors!

Here is the syntax and semantics of the if statement (my abbreviation for statement is stmt).

Syntax:

<if stmt> ::= if (<condition>) <stmt> [else <stmt>]

Semantics:

1. Evaluate the <condition>
2. If the value of the condition is true, execute the <stmt> after <condition>
3. If the value is false and there is an else part, execute the <stmt> in the else part.

Conclusion

Conditional instructions allow Ruby to accomplish tasks where she (and her programmer) does not know the exact situation before starting. This provides much more power, since she can then solve infinitely many different tasks with a single program.

If is used to do a thing or not.

If-else is used to do one thing or another.

If and recursion is used to do something over and over.

Chapter 4: Repeating instructions

Sometimes Ruby needs to do things an indefinite number of times. For instance if Ruby's task is to collect all the coins from a room, return them to the origin, and put them all down, she cannot know beforehand how many there are. Thus, her program cannot tell her to put down an exact number. But, there are two ways she can put down however many coins she is carrying. The first way is to use an if statement and recursion.

Recursion

Let's say we are trying to write a `dropAllCoins()` instruction. It would start by checking if Ruby has any coins to drop, and if so, put one down:

```
if (coinInBag()) {
    putCoin();
}
```

That's a good start, but now she has to put the rest down; i.e. drop all the coins she has left. But! we have an instruction that does that, the one we are writing; namely, `dropAllCoins()`!

```
void dropAllCoins() {
    if (coinInBag()) {
        putCoin();
        dropAllCoins();
    }
}
```

That will do it. It looks like maybe it will be an infinite loop since the definition of `dropAllCoins()` includes an invocation of `dropAllCoins()`, but if you think carefully you will realize that each time Ruby goes to `dropAllCoins()` recursively she will have one less coin (since the `putCoin()` is right in front of the recursive call).

Iteration

The other way to achieve repeated action is iteration. Two iterative statements are `for` and `while`. Here's `dropAllCoins()` using a `while` loop:

```
void dropAllCoins() {
    while (coinInBag()) {
        putCoin();
    }
}
```

That's it. The syntax is just like an `if`, and the semantics, very similar, except, if the condition is true, after executing the statement, control is returned to the top of the loop (i.e. the `while`), and it starts over. Plus, there is no `else` with a `while`!

Here is the syntax and semantics of the while statement.

Syntax: `<while stmt> ::= while (<condition>) <stmt>`

Semantics:

1. Evaluate the `<condition>`
2. If the value of the condition is true, execute the `<stmt>` after `<condition>` and then return to 1.

When to use while, when to use recursion

Iteration and recursion are functionally equivalent; that means each can do all and only what the other can do. In that case, which should you use? Your choice. Whichever seems more natural and understandable to you is the right choice.