FCPS
Java
Packets

Unit One – fcpsKarel Programming

June 2019

Developed by Shane Torbert
Answer Key by Marion Billington
under the direction of Gerry Berry
Thomas Jefferson High School for Science and Technology
Fairfax County Public Schools
Fairfax, Virginia

Contributing Authors
The author is grateful for additional contributions from Marion Billington, Charles Brewer, Margie Cross, Cathy Eagen, Anne Little, John Mitchell, John Myers, Steve Rose, John Totten, Ankur Shah, and Greg W. Price.

The students' supporting web site, including all materials, can be found at http://academics.tjhsst.edu/compsci/
The teacher's FCPS Computer Science CD is available from Stephen Rose at srose@fcps.edu

Acknowledgements
Many of the lessons in Unit 1 were inspired by *Karel++: A Gentle Introduction to the Art of Object-Oriented Programming* by Bergin, J., Stehlik, M., Roberts, J., and Pattis, R.

Java Instruction Plan—Unit One

Section One – Classes and Objects **Page**

Section Two – Inheritance and Methods

Section Three – Algorithms and Polymorphism

Section Four – Abstract Classes and Interfaces

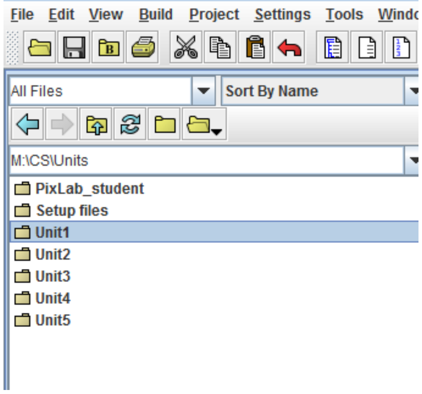
Section Five – Return, break, and continue

Appendix

Discussion

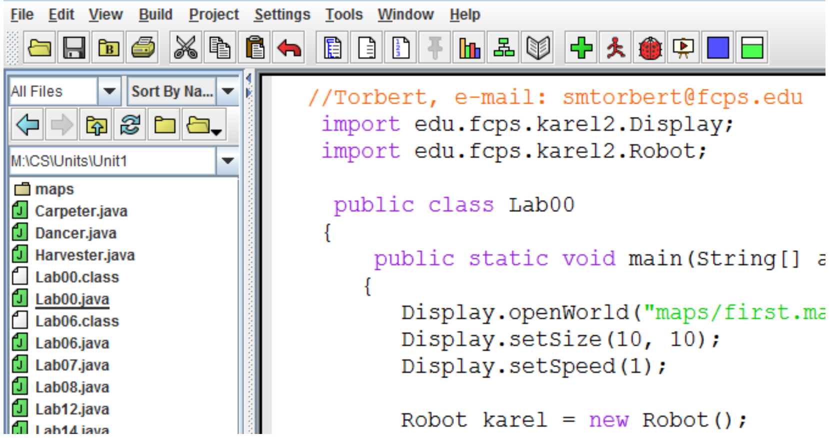
File Management

Java requires you to organize your files carefully. Most students make a personal `CS` folder, which might be on a network drive, a hard drive, or a removable drive. Inside `CS`, copy `Setup_files` and the Unit folders. Then inside Unit1, do all your Unit1 work. Do not create subfolders inside Unit1. Ask your teacher how to do this on your school's system. Here is a typical directory structure, as seen from jGrasp:



M:\CS\Units

PixLab_student
Setup files
Unit1
Unit2
Unit3
Unit4
Unit5



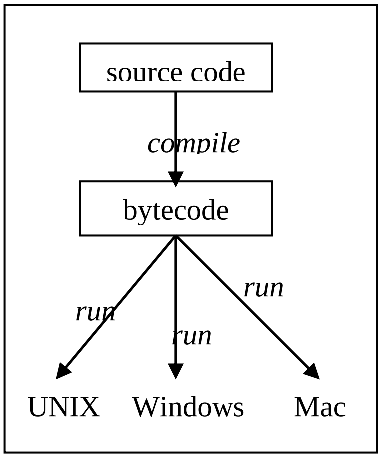
M:\CS\Units\Unit1

maps
Carpeter.java
Dancer.java
Harvester.java
Lab00.class
Lab00.java
Lab06.class
Lab06.java
Lab07.java
Lab08.java
Lab12.java
Lab14.java

```
//Torbert, e-mail: smtorbert@fcps.edu
 import edu.fcps.karel2.Display;
 import edu.fcps.karel2.Robot;

  public class Lab00
 {
    public static void main(String[] a
   {
      Display.openWorld("maps/first.ma
      Display.setSize(10, 10);
      Display.setSpeed(1);

      Robot karel = new Robot();
```

Source code is the Java program you write. Running a Java program is a two-step process: *compiling* and then *running*. Compiling is the translation of the source code into a lower-level code called *bytecode*. Compiling also checks for some kinds of errors, which is helpful. If the program successfully compiles, then you run the bytecode and see what happens. If you edit your source code, and you do not re-compile, then you are running the old bytecode. Accordingly, every time you make a change, you must re-compile in order to create an updated bytecode.

The two-step structure was adopted in order to accommodate different kinds of computers connected by the World Wide Web. The bytecode is sent over the internet, which is then run on the local system. The result is that Java is portable, or platform independent, because you can compile once and run anywhere. The alternative structure, which is used by C++ and other languages, is to produce compiled code directly for each type of computer. This second kind of code has an advantage in that it runs faster than the Java kind of code.



Java, and *object-oriented programming*, can be difficult (but fun) to learn. Unit 1 uses karel robots and maps to introduce the concepts of object-oriented programming.

Why do we use the name "karel"? According to *Karel++: A Gentle Introduction to the Art of Object-Oriented Programming* by Joseph Bergin, Mark Stehlik, Jim Roberts, and Richard Pattis:

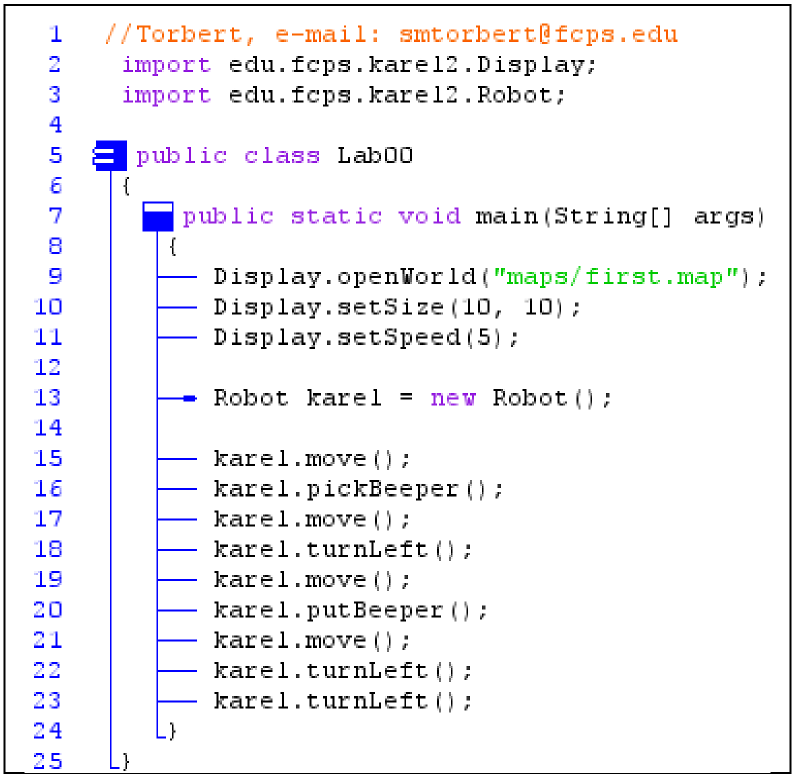
> The name "karel" is used in recognition of the Czechoslovakian dramatist Karel Čapek, who popularized the word *robot* in his play *R.U.R.* (Rossum's Universal Robots). The word *robot* is derived from the Czech word *robota*, meaning forced labor.

Lab00
Hello Robot

Objective
Classes, objects, and calling methods. Program structure.

Background
Ask your teacher about the text editor and Java environment you will be using. A nice text editor for a beginning student is jGRASP, available for free from Auburn University.

Specification
Go to File, New, Java file. Enter the source code shown below.



```
 1  //Torbert, e-mail: smtorbert@fcps.edu
 2   import edu.fcps.karel2.Display;
 3   import edu.fcps.karel2.Robot;
 4
 5  public class Lab00
 6  {
 7     public static void main(String[] args)
 8     {
 9        Display.openWorld("maps/first.map");
10        Display.setSize(10, 10);
11        Display.setSpeed(5);
12
13        Robot karel = new Robot();
14
15        karel.move();
16        karel.pickBeeper();
17        karel.move();
18        karel.turnLeft();
19        karel.move();
20        karel.putBeeper();
21        karel.move();
22        karel.turnLeft();
23        karel.turnLeft();
24     }
25  }
```

Line 1: This line states the programmer's name and date, beginning with //. The // starts a *comment* line, which is not compiled into bytecode; it is ignored.

Line 2: The `import` keyword makes classes in other packages accessible.

Line 5: A class (in Java) is a container for code. This class is public, not private.

Line 6: Left curly brace begins the class.

Line 7: All Java programs have a `main` method. The statements execute in order.

Line 8: Left curly brace begins `main`.

Line 9: `openWorld` is a *class method* in the Display *class*.

Line 12: The compiler ignores whitespace.

Line 13: This line *instantiates* a Robot object from the Robot class. The *object* named karel (or referenced by karel, or pointed to by karel) is an *instance* of the Robot class. A class is like a blueprint or a plan for a house. An object is like the actual house.

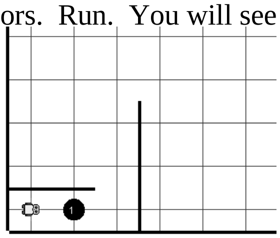
Line 15: `move` is an *instance method* in every Robot *object*. On Line 15 we *call* the `move` method.

Lines 24 and 25: Right curly brace ends main and another right curly brace ends class Lab00. When you click CSD, the braces are all indented properly and the blue lines are generated.

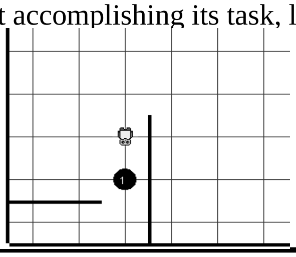
Go to Save As. . . and save it as Lab00 (the file name must exactly match the class name) in the Unit1 folder. CSD. Compile and fix any errors. Run. You will see a map and a robot accomplishing its task, like this:

Sample Run
`Unit1Lab00.jar`

Start:



End:



Exercises
Lab00

A Robot object's *default* initial state is (1, 1) facing east with zero beepers. If you don't want the default state, you may send the *values* of the x-coordinate, y-coordinate, direction, and number of beepers as *arguments* to the constructor. For example:

```
Robot ophelia = new Robot();        //calls the default constructor

Robot horatio = new Robot(5, 4, Display.SOUTH, 37); //calls the 4-arg constructor
```

1) What does ophelia know? What does horatio know? What do both robots know?

2) Write the command to create a robot named pete starting at (4, 3) facing west with 50 beepers.

3) Complete the main method to have lisa move one block, put down a beeper, then move another block. Since we have not set-up a specific map, the *default robot-map* will be used. The default map is empty except for the two infinite-length walls on the southern and western edges.

```
public static void main (String[] args)
{
  Robot lisa = new Robot(7, 7, Display.SOUTH, 15);





}
```

4) Complete the main method to have martha move forward five blocks and "hand-off" her beeper to george. Have george move forward two blocks and put the beeper down.

```
public static void main (String args[])
{
  Robot martha = new Robot(1, 1, Display.NORTH, 1);
  Robot george = new Robot(1, 6, Display.EAST, 0);







}
```

5) Question #3 has no `Display.openWorld`. In that case, what map is used?

Discussion
Java Applications

Java *applications*, or programs, require a specific structure. A Java application is required to have a **`public static void`** `main` method, which is the entry point for running the code. When the program runs, the computer executes the commands in `main`, one after another. To repeat: if you do not define **`public static void`** `main(String[] args)`, your code is not runnable. That code is not an application, even though it compiles. If you try to run your code, jGrasp will give an error, "not found: main". (It may interest you to know that all the Android "apps" on your cell phone require an <action android:name="android.intent.action.MAIN"/> as well as some other very specific methods.)

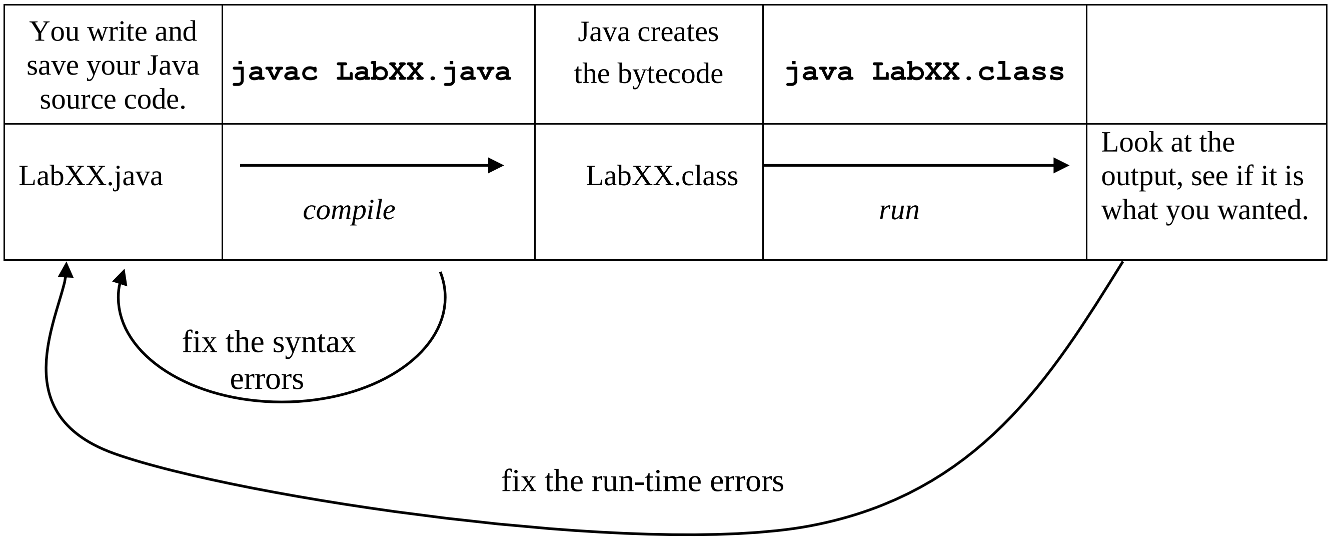
Java uses matching curly braces to indicate blocks of code. You already know that the class name must match the filename, so that the code below must be saved as `LabXX.java`.

```
public class LabXX
{
  public static void main(String[] args)
  {

  }
}
```

When you click the jGrasp "compile" button, you are actually running a separate program called `javac`. (This program was written by Sun, now owned by Oracle, and is part of the j2sdk installation.) The compiler first checks the `LabXX.java` code for *syntax errors*. Then it imports all the import files, looking first in the current folder (which is our Unit1), then in the external import locations, then the automatic import locations. All this source code is translated into bytecode and saved as `LabXX.class`. The bytecode is a comparatively small file, so that it can travel quickly over the internet.

When you click on a Java applet in a web browser, or on the jGrasp "run" button, you are actually running a separate program called `java` . (This program was written by Sun, now owned by Oracle, and is part of the jre installation.) The "run" button calls the `main` method in the application and executes the commands.

The write-compile-run process is illustrated below.


You write and save your Java source code.
javac LabXX.java
Java creates the bytecode
java LabXX.class
LabXX.java
compile
LabXX.class
run
Look at the output, see if it is what you wanted.
fix the syntax errors
fix the run-time errors

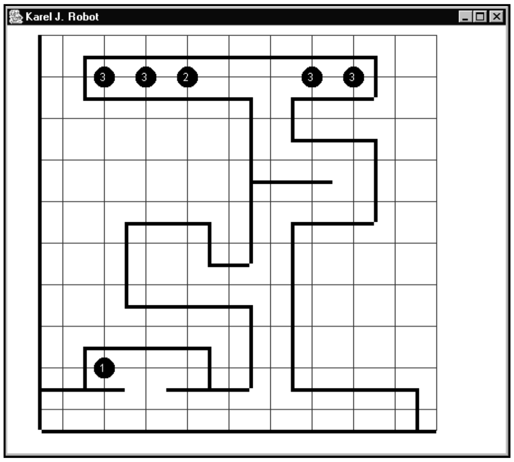
Lab01

Students and Books

Objective

Program structure. Display's methods. Robot's methods.

Background



Maps are mapped according to the Cartesian coordinate system. The corner at the bottom-left of the graphics window is (1, 1).

Maps create the context for solving robot problems. Pre-defined maps are stored in the folder Unit1\maps

An *identifier* is the name of a class, an object, or a method. An identifier can be any unique sequence of numbers, letters, and the underscore character "_". An identifier cannot begin with a number. An identifier cannot be a Java keyword, like `class`. Identifiers are case-sensitive, so lisa is not the same as Lisa. As a convention, only class names begin with an uppercase letter. Methods, objects, and variables by convention begin with a lowercase letter. Constant values, such as EAST, are written in ALL CAPS.

The word *instantiate* means to create an instance of a class—an object. It is like saying that you instantiate the house from its blueprints. Java instantiates an object using the `new` keyword and calling the constructor.

The object "knows" how to do things in its *methods*. We can *call* or *invoke* a method, or *send a message* to a robot object, by using *dot-notation*. For instance, a robot named pete will move forward with the call `pete.move();.` The identifier "pete", before the dot, is the name of an object and the identifier "move", after the dot, is the name of a method. All methods in Java are marked by parentheses, which sometimes are empty and sometimes have values. The pete object will attempt to follow the calls in its `move` method.

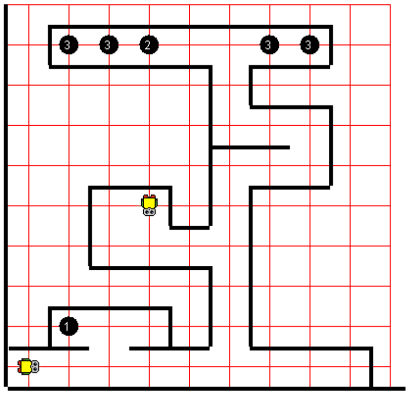
Specification

Create Unit1\Lab01.java with the "school" map at size 10x10. Instantiate two Robot objects, one named lisa using the default constructor and the other named pete starting at (4, 5) facing south with zero beepers. Have lisa pick up the "book" from the math office and bring it to pete. Have pete take the book to the storage room and place it on the pile that currently has only two books. Then have pete step away from the pile.

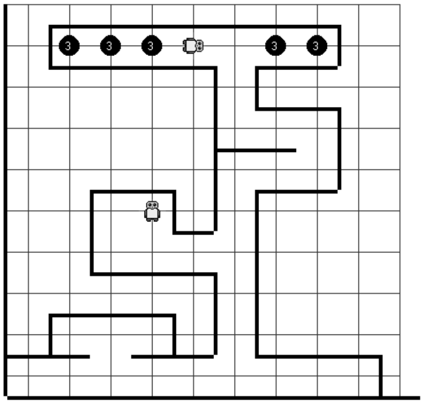
Sample Run

```
Unit1Lab0.jar
```

start:



end:



Exercises

Lab01

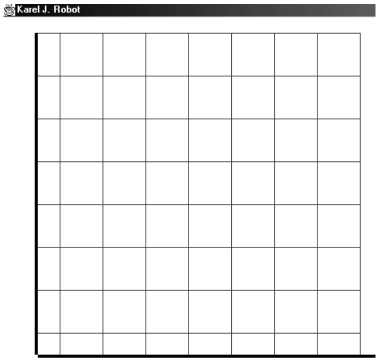
1) An API tells you everything you need to know about how to use the methods and constructors for each class. Go to the Unit1 API at https://academics.tjhsst.edu/compsci/CSweb/index.html
   a. *Fields* store an object's *private* information about its *state*. List all the fields of the Robot class:

   b. What methods in Robot have we used so far?

   c. What methods in Display have we used so far?

2) Circle the identifiers that will compile. Put a star by the identifiers that by convention identify a class.

```
mrsAdams   mr.chips    class      r2_d2      c-3po     Hal9000    7_of_9
```



3) Mark the following points A(1, 1), B(6, 7), C(5, 2), D(3, 8), E(8, 1) on the robot-map shown.

4) What has the size of this robot-map been set to?

5) Give the command to set that size.

6) Lisa is a teacher at the local high school. She needs to store some books in the storage room downstairs. Lisa takes the books from the math office to the student lounge, where eager students wait to help their teachers. Lisa gives the books to pete, who cheerfully stores the books on the smallest pile.

Identify the nouns in the story above:

Identify the verbs in the story above:

7) In objected oriented programming, nouns turn into _______________.

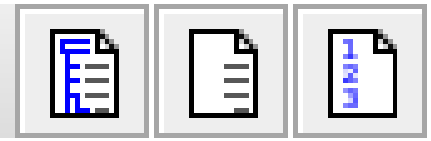
8) In objected oriented programming, verbs turn into _______________.

9) Our Lab01 program modeled a story about students and books in terms of robots and beepers. Write another similar story that could have been modeled in terms of robots and beepers.

10) In terms of your new classes, what should be the fields (private data) of each class?

11) Thinking about your new classes, what behaviors would it be useful to have?

Discussion
Errors



A Java compiler checks your work for errors whenever you compile a program. For instance, if you misspell `Robot` as `Robt` in your source code, you will receive a *lexical error* message because `Robt` is undefined. The compiler can't understand that you meant to say `Robot` but spelled it incorrectly. (The error message will tell you on which line your error occurs; use the Line Numbering button to help you locate the exact line.)

A second kind of error is a *syntax error*, meaning that it breaks the rules of forming valid commands in Java. It is like making a grammatical error in English.

A third kind of error can occur when the program runs. If your code tells a robot to walk through a wall, pick up a beeper that isn't there, or place a beeper that it doesn't have, then a *runtime error* will occur. Your run-time environment generates an error message saying why the program crashed.

A fourth kind of error occurs when your program executes all the commands, but doesn't accomplish the required task. Such *logic errors* are not the program's fault. They are the programmer's fault, usually due to unclear thinking. You'll be spending a lot of time correcting logic errors.

Please get in the habit of generating CSD and compiling after you write a few lines of code. That way, you know that any syntax errors are in the last few lines that you wrote.

To help correct logic errors, you might try to *comment out* parts of the code. That way you can check portions of the code by itself. The commented out portions will turn orange (in jGrasp), and the compiler ignores those lines. You can either use // at the beginning of each line, or you can use /* to start a block of code and */ to end it. In jGrasp, you can highlight some code and press Ctrl-/. That's nice.

Using Lab01, create these errors and write down what error messages, if any, they generate.

common errors	**CSD error messages**	**compiler error**	**run-time error**
omit a semicolon			╳
put in an illegal semicolon, say after public `static void main(String[] args);`			╳
omit a `}`			╳
lexical error in "SSSSchool"			
pass the wrong number of *arguments* (or inputs) into a method, e.g. `new Robot(4, 5, Display.SOUTH);`			╳
forget to import edu.fcps.karel2.Display;			╳

Finally, comment out all lisa's commands. What happens?

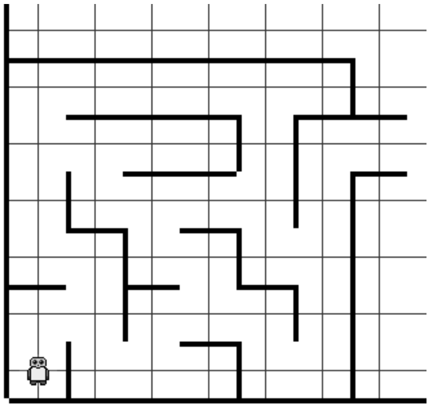
The process of finding errors and correcting them is called "debugging" and the errors themselves are called "bugs." In 1951, Grace Hopper, who would eventually rise to the rank of Rear Admiral in the United States Navy, discovered the first computer "bug." It was a real moth that died in the circuitry. Hopper pasted the moth into her UNIVAC logbook and started to call all her programming errors "bugs."

Lab02

Escape the Maze

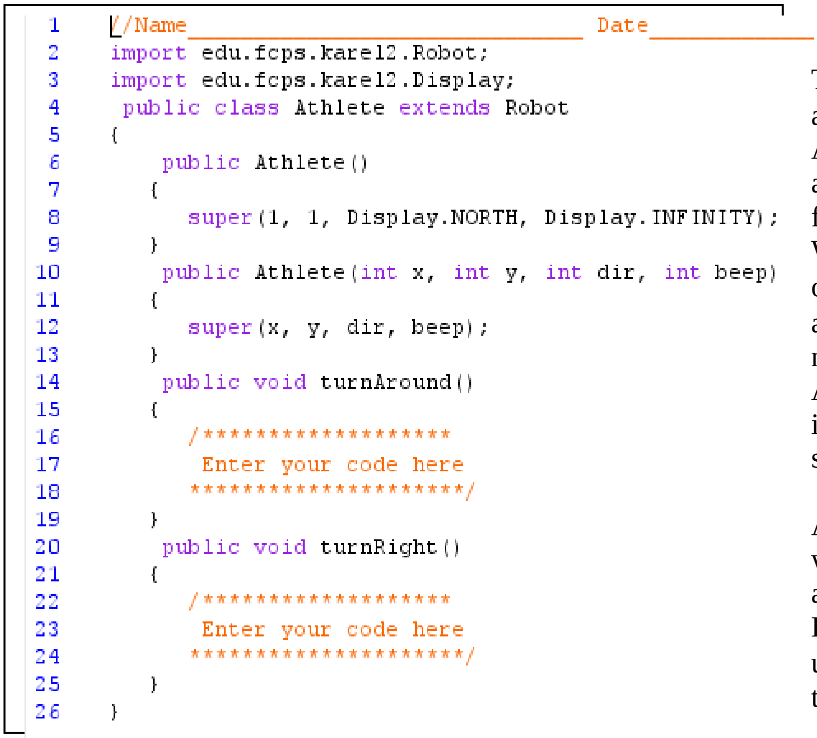
Objective

Inheritance. Defining instance methods.



Background

You may have noticed that Robots don't know how to turn right. In object oriented programming, it is common practice not to change the classes that are given to you. Instead, you extend Robot and define the new class which will have the power to turn right. Study the structure of Athlete, especially lines 14 to 24. In these *instance methods*, we “teach” Athlete how to turn around and how to turn right. (We will talk about lines 6 to 13 later.)

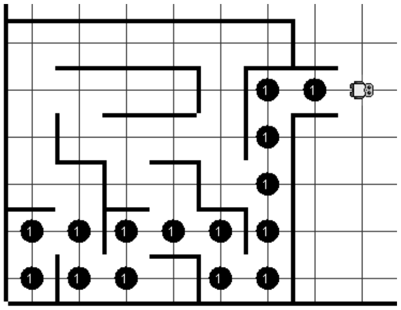


```
1   //Name______________________________ Date____________
2   import edu.fcps.karel2.Robot;
3   import edu.fcps.karel2.Display;
4    public class Athlete extends Robot
5   {
6       public Athlete()
7      {
8         super(1, 1, Display.NORTH, Display.INFINITY);
9      }
10      public Athlete(int x, int y, int dir, int beep)
11     {
12        super(x, y, dir, beep);
13     }
14      public void turnAround()
15     {
16        /*******************
17         Enter your code here
18        *********************/
19     }
20      public void turnRight()
21     {
22        /*******************
23         Enter your code here
24        *********************/
25     }
26  }
```

The keyword `extends` means that an athlete *isa* robot. "Isa" means that the Athlete class inherits the behaviors and attributes of the Robot class; the methods from Robot do not have to be re-written. We can use `turnLeft` in our definition of `turnRight` and `turnAround`, and any athlete object can use any robot method. When you write methods in Athlete, think of programming an Athlete in general, not of programming any specific athlete object.

Athlete is a general *resource class* that will be useful in a wide range of applications. The *application* called Lab02, which has the `main` method, will use the Athlete class. We sometimes say that Lab02 *hasa* athlete.

Specification



Create Unit1\Athlete.java as given above. Implement the methods `turnRight` and `turnAround`, then compile. Do not run Athlete.

Create Unit1\Lab02.java with “maze” map at size 8x8. Instantiate one athlete object and direct it to escape the maze. Leave a trail of beepers to mark the path. By default, athletes begin with an infinite number of beepers.

Sample Run

`Unit1Lab02.jar`

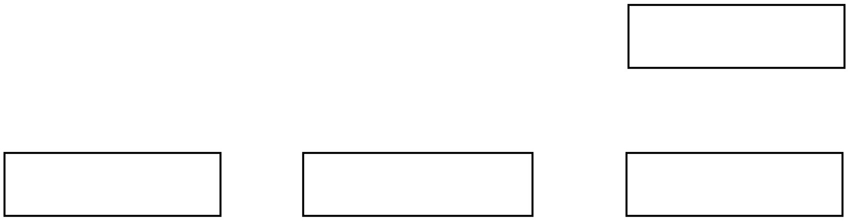
Extension

Write a `putAndMove` method in Athlete then use it in Lab02.

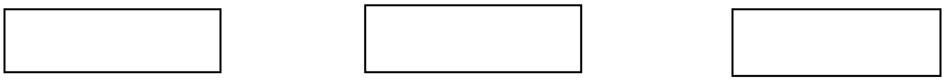
Discussion

UML Diagram

A UML diagram shows the relationships between interacting classes. (See page 44 for a complicated example.) The convention is to use a dashed, horizontal arrow to show the `hasa` relationship. A thick, vertical arrow shows the `isa` relationship. For example, Lab02 was composed of four different classes, Lab02, Display, Robot, and Athlete. Label the boxes in the UML diagram, showing the class relationships. Label the arrows with "isa" or "hasa."



In contrast, Lab01 used three classes, Lab01, Display, and Robot. Label the boxes showing the class relationships. Label the arrows with "hasa."



Exercises

Lab02

1. What is *compiling*? ____

2. Define *bytecode*. ____

3. To call the superclass constructor, you need to use the _____________ keyword.

4. Making a ______________ error in your code is similar to making a grammatical error in English.

5. The ____________ keyword signifies that a relationship of inheritance exists between two classes.

6. T/F If a line in the code is preceded by //, that line of code is neither compiled nor run.

7. T/F In the Karel-the-Robot hierarchy, a Robot isa Athlete.

8. T/F A resource class must always have more than one constructor.

9. In which direction does a default Athlete face? ______________

10. By passing different __________________ through the arguments in the constructor we can instantiate objects in different _____________.

11. Write the code to instantiate a default Athlete: ______________________________

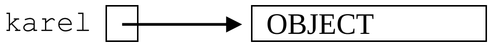
Discussion
Resources vs. Drivers

In Lab01 you wrote one class while in Lab02 you wrote two classes, a *driver class* and a *resource class*. The driver class contained the `main` method while the resource class contained the code for Athlete. Using a different terminology, we could say that the driver class is the *client* and the resource class is the *server*. The client sends messages or commands while the server knows how to respond to those messages. Notice also that the driver class did not have to be instantiated to be used. The driver class had a `static main` method, which contained code that just lies around, waiting to be used. We could just run the `main` method. In contrast, the resource class did have to be instantiated (using the `new` keyword) as an object. Then the client could send messages to that object, the server, directing the object to perform some task.

Discussion
Class Types vs. Primitive Types

A *data type* specifies how the different kinds of data, e.g., numbers, or words, or Robots, are stored and manipulated. (This is a big topic in computer science. Here we just introduce the topic.) Classes define *complex types* since they *encapsulate* data and methods, i.e., an object's data and methods are private and wrapped up inside the object. Objects are accessed by *references*, which act like little pointers. The code `Robot karel = new Robot();` does three things: creates a reference (`karel`), instantiates an object (`new Robot()`), and assigns (the "=" sign) the reference to point to the object. You should imagine the reference pointing to object, like this:

```
Robot karel = new Robot();
```

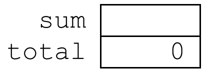


Two or more references can point to a single object. You will see this in later labs.

In Java, some built-in data types are not complex, but are simpler, called *primitive data types*. Three of Java's primitive data types are `int` for integers, `double` for decimal numbers, and `boolean` for true-false.

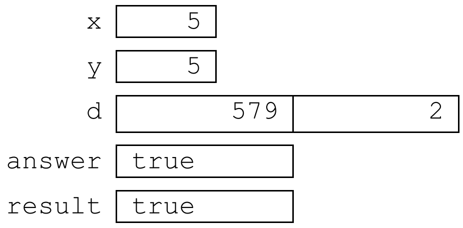
Primitive data types are not accessed by reference. Instead, the data is stored directly in memory spaces called *variables*. A *type declaration* assigns a name to a memory space, and often assigns a value, e.g.:

```
int sum;
int total = 0;
```



We may say that these two statements *declare* `sum` and `total`. Here are five more declarations:

```
int x = 5;
int y = x;
double d = 5.79;
boolean answer = true;
boolean result = answer;
```



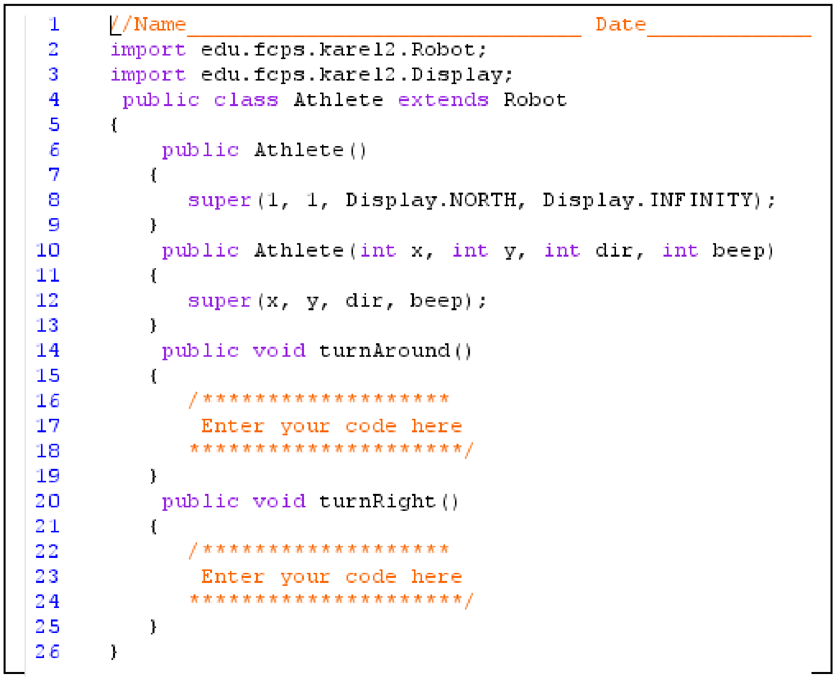
The eight declarations on this page create a reference to a Robot object, seven different variables, and assign values to six of those variables.

The 4-arg constructor `public Athlete(int x, int y, int dir, int beep)` creates four variables for integers, but does not assign values. When you call the constructor with the `new` keyword and *pass* values, then the values are assigned to the fields of the Robot object. Here is an example call:

```
Athlete ariadne = new Athlete(1, 2, Display.EAST, 10);
```

Discussion
Constructors

Lab02 instantiated an athlete at (1, 1, north, infinity) with the call `Athlete karel = new Athlete();` That call executed lines 6-9 in the Athlete class below, which is one of Athlete's constructors. A *constructor* method is a method that creates an object and initializes its private data. *Initializing the private data* is done in this case by the `super` command which specifies this athlete's x-position, y-position, direction, and beepers.



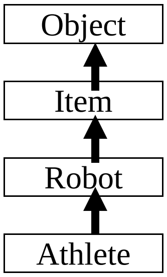
```
1   //Name______________________________ Date_____________
2   import edu.fcps.karel2.Robot;
3   import edu.fcps.karel2.Display;
4    public class Athlete extends Robot
5   {
6       public Athlete()
7      {
8         super(1, 1, Display.NORTH, Display.INFINITY);
9      }
10      public Athlete(int x, int y, int dir, int beep)
11     {
12        super(x, y, dir, beep);
13     }
14      public void turnAround()
15     {
16        /*******************
17         Enter your code here
18        *********************/
19     }
20      public void turnRight()
21     {
22        /*******************
23         Enter your code here
24        *********************/
25     }
26  }
```

The no-arg (or *default*) constructor is easy to call, but it always instantiates an object in the default state, which in the case of an Athlete is 1, 1, north, and infinity. It would be nice to be able to create an Athlete at different positions, directions, and numbers of beepers. We can do so by writing a *constructor with arguments* (with inputs.)

Lines 10 through 13 contain the code for the *4-arg constructor*, because in line 10, the (int x, int y, int dir, int beep) creates room for 4 values. The 4-arg constructor is harder to call, but it allows the programmer to instantiate an athlete in any state, which is useful.

Java allows us to write as many constructors as we want, each instantiating objects in slightly different states. You know from Lab01 (and from page One-5) that Robot also has two varieties of constructors.

In Java, a subclass inherits methods but not constructors. The subclass constructor needs an explicit call, which is the `super` command, to the superclass's constructor. The `super` command calls the constructor of the class above the current class. Thus, Athlete's `super` calls Robot's constructor. Robot also has a `super` method, which calls the constructor of the class above it, which happens to be Item (you only know this because it is in the API). Item calls the constructor of Object. Eventually, all the `super`s in the hierarchy instantiate all the pieces that make an Athlete object.



There are several rules regarding constructors. First, the name of the constructor method must match the name of the class, i.e. `Athlete` on lines 6 and 10 must match `Athlete` on line 4. Second, constructors are **not** labeled with a keyword, such as `void` or `static`. Third, the call to `super`, including the number and types of arguments, must find the corresponding constructor in the class in the hierarchy above it. For example, the Athlete's `super(1,1,Display.NORTH,Display.INFINITY)` must find a corresponding 4-arg constructor method `public Robot(int x, int y, int dir, int beep)`. If the compiler is unable to find the corresponding constructors in the class above, it reports an error, "no suitable constructor."

If you choose not to write a constructor, then Java automatically generates a hidden no-arg constructor for you! This is very nice but it is also confusing to beginners, because it appears there is no constructor at all. If you do write a constructor with arguments, then you should, as a general rule, also define a no-argument constructor, just in case the chain of `super` constructors needs it later. Notice that we follow this rule with the Climber class in Lab03, giving Climber two varieties of constructors.

Lab03

Climb Every Mountain

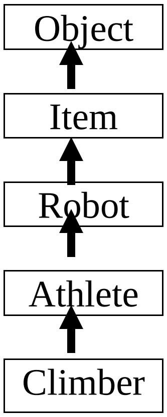
Objective

Constructors are special methods that create an object and initialize its private data, often by the `super` command.

Background

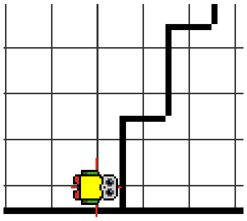
A Climber is a specialized Athlete that climbs mountains and finds treasure. Climbers always start on the bottom row (y=1), facing north, with one beeper, but we want climbers to be able to begin at any given x-position. The x value will be passed as an argument in the constructor. Therefore, we will give the Climber class a 1-arg constructor as well as a default constructor. How do these two constructors work?

```
 1 public class Climber extends Athlete
 2 {
 3   public Climber()
 4    {
 5      super();
 6    }
 7   public Climber(int x)
 8    {
 9      super(x, 1, Display.NORTH, 1);
10    }
11   public void climbUpRight()
12    {
13      //pseudocode: tL, m, m, tR, m
14    }
    //You must write three other instance methods.
```



We might instantiate a climber with `Climber tenzing = new Climber(100);`. Note that we pass one argument to Climber's one-arg constructor. Climber's own `super` (on line 9) passes four arguments to Athlete's 4-arg constructor. Athlete's constructor passes those four to Robot, and so on up to Object.

The Climber's instance methods all have to do with climbing the mountain. Each method should work for one step only. Plan how you will teach the climber to climb up exactly one step, going to the right, under `public void climbUpRight()`



Specification

Consult the Unit 1 API to find out what methods the Climber class needs to know.

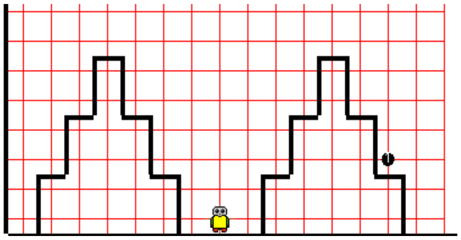
Create Unit1\Climber.java. Implement the methods in the Climber class, then compile.

Create Unit1\Lab03.java with "mountain" map at size 16 x 16. Instantiate two climbers starting at x-coordinate 8. Put down the beeper, which is the "base camp." Staying together as much as possible, bring the treasure back to "base camp."

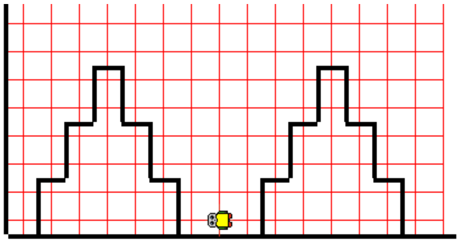
Sample Run

`Unit1Lab03.jar`

start:



end:



Exercises
Lab03

1. Notice which class Elf inherits from. Write a default constructor for Elf. All elves begin at (1, 90), face south, and carry an unlimited supply of beepers.

```
public class Elf extends Robot
{
```

2. Notice which class Spartan extends. Write a constructor for Spartan that takes two arguments specifying the position to start on. All Spartans face east with one beeper.

```
public class Spartan extends Athlete
{
```

3. Write a second constructor for Spartan. This constructor is a default constructor. It instantiates a Spartan in an Athlete's default state.

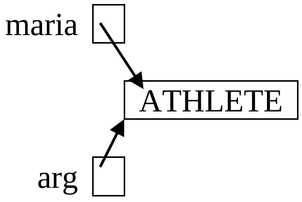
Discussion
Passing Arguments (or inputs)

We said above that we might instantiate a climber with `Climber tenzing = new Climber(100);`. The number 100, inside parentheses, is called an *actual argument*. The program then goes to the method's code, sees the `public Climber(int x),` creates room for the `x,` and assigns the 100 to the `x`. The variable x is called the *formal argument*. Whenever the `x` is used after that, its value will be 100. Similarly, when the call to `super(100, 1, Display.NORTH, 1)` is executed, then those four integers are *passed* up to Athlete. There, the `x` is 100, the `y` is 1, the `dir` is Display.NORTH, and the `beep` is 1. The four actual arguments must match up with the four formal arguments, or the class will not compile. Whenever we write or call a method, we need to think about the method's arguments, both their type and their number.

x	100

x	100
y	1
dir	3
beep	1

In the next lab, we will be *passing objects* as arguments with `takeTheField(maria);` The computer then goes to the method's code, sees `public static void takeTheField(Athlete arg),` and creates and assigns a new reference. Because Java objects are referenced, what gets passed is the reference, meaning that `maria` and `arg` point to the same object. In effect, the athlete has two names. Commands invoked on `arg` affect the same object to which `maria` points.



Lab04
Take the Field

Objective

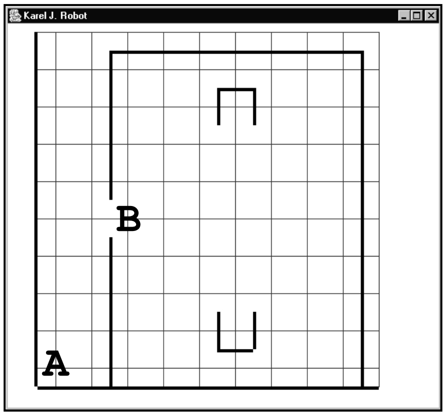
Class methods vs. Instance methods

Background

A *class method* conveniently stores code that is accessible to anyone, without first creating an object (the class "knows"). Mathematical formulas, such as those in the Math class, are examples of class methods. Often class methods control the environments for other objects. You have already seen `Display.setSize` and `Display.openWorld`. All class methods are marked by the keyword `static.` Class methods are usually placed before the main method, but they don't have to be.

An *instance method* is a method written in a class that must be instantiated (the object "knows") before being used. Every Athlete should know how to `turnRight`. Therefore, turning right is appropriately defined as an instance method within the Athlete class--the object knows. In contrast, taking the field is something that is specific to this lab only. Therefore, taking the field is appropriately defined as a class method within the Lab04 class--the class knows. All the athletes in Lab04 will go from the locker room (Point A) to the field (Point B) by executing the code in the class method `takeTheField`.

```
public class Lab04
{
  public static void takeTheField(Athlete arg)
  {
     arg.move();
     arg.move();
     arg.move();
     arg.move();
     arg.turnRight();
     arg.move();
     arg.move();
  }
  public static void main(String[] args)
  {   //You must define main.
```



We can use this class method with any athlete object by passing the object. For instance, if we have athletes `maria` and `gary` defined in main, we can tell each to take the field with the commands:

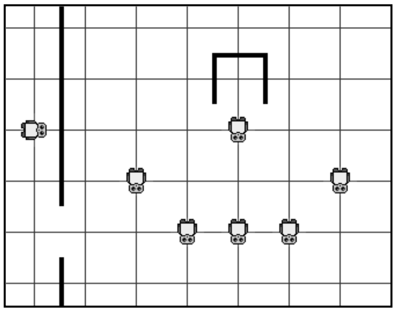
```
      takeTheField(maria);
      takeTheField(gary);
```

Notice that we do NOT say `maria.takeTheField()`. This is because `takeTheField` is not part of an Athlete object. It is "known" by the Lab04 class. We call `takeTheField` and pass the object to the argument `arg`, which points to whatever Athlete has been passed; first maria, then gary.

Specification

Create Unit1\Lab04.java with "arena" map at size 10x10. Create six athletes starting in the locker room; use the default constructor. Create one object with the 4-arg constructor to put the coach at the side of the field. Have your team get positioned for the start of the game as shown

.



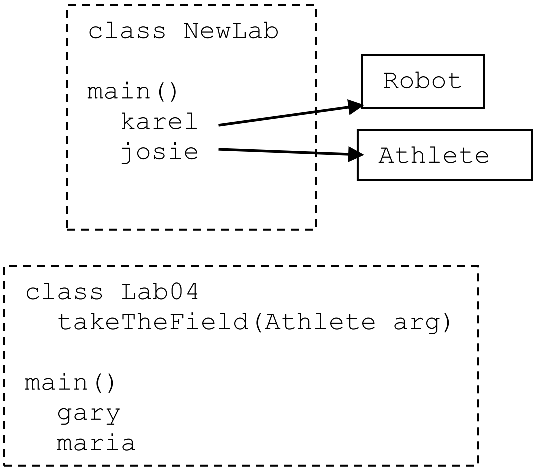
Sample Run

`Unit1Lab04.jar`

Exercises, Part 1

Lab04

```
public class NewLab
{
  public static void main(String[] args)
  {
   Robot karel = new Robot();
   Athlete josie = new Athlete();
   //Assume each command below is inserted here.
  }
}
```



Explain what is wrong with each command shown below.

```
1.  karel.turnRight();


2.  karel.putBeeper();


3.  maria.pickBeeper();


4.  josie.turnLeft();
    josie.move();


5.  josie.takeTheField();


6.  takeTheField(josie);


7.  Lab04.takeTheField(karel);


8.  Lab04.takeTheField(josie, karel);


9.  Lab04.takeTheField(Athlete arg);
```

Exercises, Part 2

Lab04

1. Looking at your code for Lab04, write a call to a *4-arg constructor*: ______________________

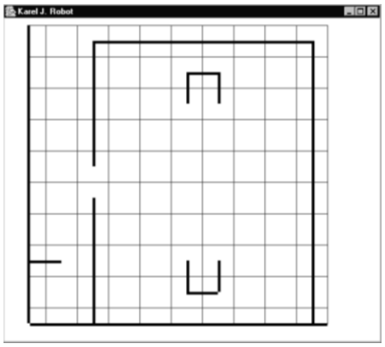
2. Looking at your code for Lab04, write a call to a *default constructor*: ______________________

3. Looking at your code for Lab04, write a call to a *class method*: ______________________

4. Looking at your code for Lab04, what lines contain the *definition* of the class method? ____________

5. Looking at your code for Lab04, write a call to an *instance method*: ______________________

6. Examine the robot-map at the right. Notice that this is not exactly the same arena map from Lab04. Our facilities are being renovated and the construction crew has temporarily blocked off our normal passageway; there's a horizontal wall north of (1, 2) that wasn't there originally.



7. What method do you have to change so that it would work in this map?

8. Write code for that new method, including the *header* and the *body*.

9. *Modular design* is the practice of breaking larger problems into smaller, more manageable pieces. Explain how modular design helped you fix your Lab04 program when the robot-map changed.

Look at the API for the Math class in the current version of Java. (In jGrasp, go to Help→Java API.)

10. Which Math method doesn't take any arguments? ______________________

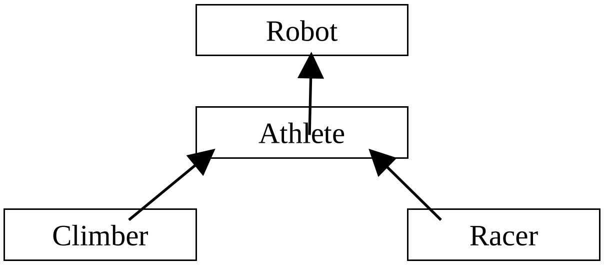
11. Name a Math method that takes two arguments. ______________________

12. Why does it make sense that all the Math methods are marked `static`?

Discussion
Class Hierarchy

In the *class hierarchy*, or inheritance hierarchy, shown in the UML diagram at the right, we say:

- Robot is the *superclass* of Athlete and Athlete is the *subclass* of Robot.
- Athlete is the superclass of Climber and Climber is the subclass of Athlete.
- Athlete is the superclass of Racer and Racer is the subclass of Athlete



Other names for a superclass are *base class* or *parent class*. Other names for a subclass are *derived class* or *child class*. Since a climber isa athlete and an athlete isa robot, we can also say that a climber isa robot. Likewise, since a racer isa athlete and an athlete isa robot, we can say that a racer isa robot. Not only do the subclasses inherit methods and fields, but they also inherit the name of the superclass.

1) For each attribute, circle the classes that **must** have that attribute.

A. `Athlete`s have green feet. `Robot` `Athlete` `Climber` `Racer`

B. `Climber`s have blue scales. `Robot` `Athlete` `Climber` `Racer`

C. `Robot`s have curly tails. `Robot` `Athlete` `Climber` `Racer`

D. `Racer`s have sharp teeth. `Robot` `Athlete` `Climber` `Racer`

2) Given the declarations below, circle all the commands that are legal.

```
Robot karel = new Robot();
Athlete gary = new Athlete();
Climber lisa = new Climber();
Racer mary = new Racer();    // Racer defines jumpHurdle()
public static void runAway(Athlete arg)
```

karel.move();	gary.move();	lisa.move();	mary.move();
karel.turnRight();	gary.turnRight();	lisa.turnRight();	mary.turnRight();
karel.turnLeft();	gary.turnLeft();	lisa.turnLeft();	mary.turnLeft();
karel.climbUpRight();	gary.climbUpRight();	lisa.climbUpRight();	mary.climbUpRight();
karel.jumpHurdle();	gary.jumpHurdle(),	lisa.jumpHurdle();	mary.jumpHurdle();
karel.main();	gary.setSize();	lisa.openWorld();	mary.pickUpBeeper();
karel.runAway();	gary.runAway();	lisa.runAway();	mary.runAway();
runAway(karel);	runAway(gary);	runAway(lisa);	runAway(mary);

Discussion
Loops

We often want to repeat (or *iterate*, or loop) either one command or a sequence of commands. For instance:

```
  karel.move();
  karel.putBeeper();
  karel.move();
  karel.putBeeper();
  karel.move();
  karel.putBeeper();
  karel.move();
  karel.putBeeper();
  karel.move();
  karel.putBeeper();
  karel.move();
  karel.putBeeper();
```

```
  pete.pickBeeper();
  pete.pickBeeper();
  pete.pickBeeper();
  pete.pickBeeper();
  pete.pickBeeper();
  pete.pickBeeper();
  pete.pickBeeper();
  pete.pickBeeper();
  pete.pickBeeper();
  pete.pickBeeper();
```

In the first example, we want to repeat moving and putting exactly 6 times. In the second example, we want to pick exactly 10 beepers. Since we know beforehand exactly how many times to iterate (*definite iteration*), we will use a *for-loop.* (We will study *indefinite iteration*, the while-loop, in Lab06.)

```
int k;
for(k=1; k<=6; k++)
{
  karel.move();
  karel.putBeeper();
}
```

This loop will cause karel to move and put down a beeper six times (once when k has each of the values 1, 2, 3, 4, 5, and 6).

```
int k;
for(k=1; k<=10; k++)
  pete.pickBeeper();
```

This loop will cause pete to pick up a beeper ten times (once when k has each of the values 1, 2, 3, 4, 5, 6, 7, 8, 9 and 10).

Braces are optional if the process to be repeated consists of only a single command.

Each for-loop has three parts, where k begins, where k ends, and how k increases. The command k++ causes the value of the variable k to increase by one. If k was one, it is now two; if k was two, it is now three; and so on. (Plus-plus (++), the *unary increment operator*, gave us the name of the language C++.)

The table to the right shows both the values that the integer variable k takes and the values that the boolean expression k <= 6 takes inside karel's loop.

k	k <= 6
1	true
2	true
3	true
4	true
5	true
6	true
7	false

The boolean expression evaluates to false only when the value of k reaches seven. Since the loop continues until this condition is false, the value of k is seven when the loop stops. What is the value of k when pete's loop stops?

The loop control variable, k, can actually be declared within the for-loop. The syntax of a for-loop that repeats n times can then be written as:

```
for(int k = 1; k <= n; k++)
{
}
```

The loop control variable, k, does not have to be k. You can name it anything that makes sense.

Lab05
Shuttle Run

Objective: for-loops

Background
Racers always start on the left side at x=1, facing east with an infinite number of beepers, but they may begin on any y-coordinate. The Racer constructor with an argument that specifies the starting y-position is on lines 6-9.

Racers also improve upon the powers of Athletes and Robots. Racers will be able to move many steps at a time, pick up piles of beepers, and put down piles. These instance methods will use for-loops and pass a variable. One is done for you; complete the other two.

If pete is a Racer, then `pete.sprint(100);` will move pete way off the screen. How does the `sprint` method work?

Since three racers will complete the same task, which depends on the map that is used, this is a perfect place to define a class method. Let's call it `runTheRace`.

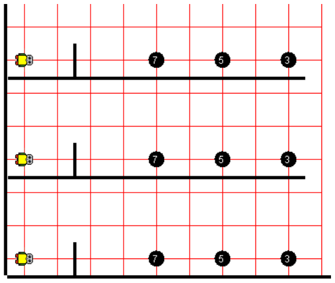
```
 4  public class Racer extends Athlete
 5   {
 6    public Racer(int y)
 7     {
 8       super(1, y, Display.EAST,
                               Display.INFINITY);
 9     }
10    public void jumpRight()
11     {
       //pseudocode:  tL, m, tR, m, tR, m, tL
15     }
16    public void jumpLeft()
17     {

25     }
26    public void sprint(int n)
27     {
28       for(int k=1; k <= n; k++)
29          move();
30     }
31    public void put(int n)
32     {
33       __________________

34       __________________
35     }
36    public void pick(int n)
37     {
38       __________________

39       __________________
40     }
41    public void shuttle(int spaces,
                             int beepers)
41     {
42
43
44     }
```

Specification
Create Unit1\Racer.java. `jumpRight` and `jumpLeft` move the racer over a 1-block tall wall. `sprint`, `put`, and `pick` each take one argument *n* and do their thing *n* number of times. `shuttle` moves the racer to a pile of beepers, picks up the pile, moves back to the beginning, puts down the pile, and gets ready to go again. The two arguments to `shuttle` specify how far to move and how many beepers to carry.

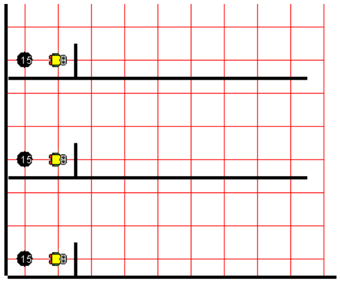
Create Unit1\Lab05.java with "shuttle" map at size 10x10. Create three racers to run the shuttle run. It's not a race. Let one racer finish the task, then send each of the others. Use a class method called `runTheRace` , which calls the `shuttle` method several times, passing the appropriate arguments. At the end, each racer should move off the pile to confirm that all 15 beepers have been retrieved.

Sample Run
`Unit1Lab05.jar`

start:

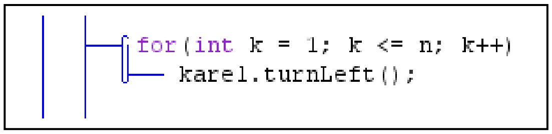


end:

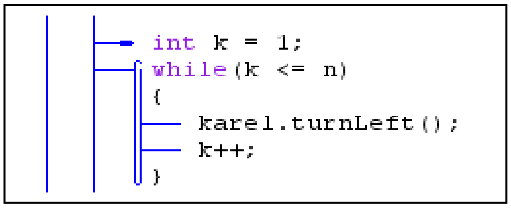


Discussion
for-loop, while-loop, and if

Any for-loop can be re-written as a while-loop.



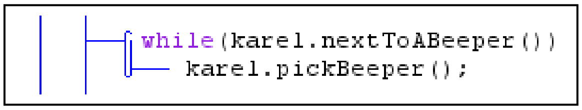
```
for(int k = 1; k <= n; k++)
   karel.turnLeft();
```



```
int k = 1;
while(k <= n)
{
   karel.turnLeft();
   k++;
}
```

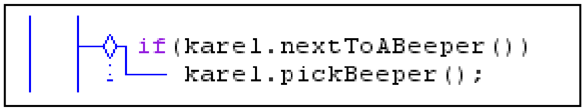
Both loops as written will try to make karel turn left for *n* times, no matter what. However, it has become conventional to use the for-loop for definite loops and the while-loop for indefinite loops, where you don't know beforehand exactly how many times to loop.

The while-loop below is a better example of an indefinite loop, for we don't know when it will stop. It keeps picking up a beeper as long as `nextToABeeper` is `true`. The result is to make a robot that is standing on top of a pile to pick up that entire pile of beepers.



```
while(karel.nextToABeeper())
   karel.pickBeeper();
```

An if-statement is similar to a while-loop, but only makes one decision. The if-statement below will cause karel to pick up one beeper if he is currently standing on top of one. If karel is not currently standing on top of a beeper, the code just moves on. Thus, the if-statement acts as a guard to prevent the program from crashing in certain situations. In other words, an if-statement checks its condition only once.



```
if(karel.nextToABeeper())
   karel.pickBeeper();
```

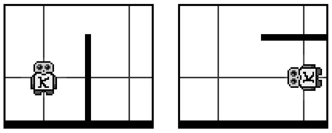
Warning

There is no such thing as an if-loop. Don't say "if-loop," say "if-statement."

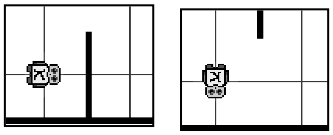
Java distinguishes between *void methods* and *return methods*. Void methods take action and change the situation, such as `public void turnRight()`. In contrast, return methods provide information about a robot's situation. Since there are different kinds of information, return methods come in several varieties. One variety returns *boolean* values, either true or false. Think of booleans as answering yes-no questions. An example of a boolean method, used above, is `public boolean nextToABeeper()`.

Another boolean method defined in the Robot class is `frontIsClear`. It determines whether or not a wall blocks a robot's path. Look at the examples below; you can see that this method makes no distinction whatsoever as to the cardinal-direction in which the robot is facing.

`frontIsClear()` returns `true`



`frontIsClear()` returns `false`



There is no method `frontIsBlocked`. However, there is the exclamation mark (!), called the *`not`* operator, and follows the rules not-true is false and not-false is true. The `not` operator goes in front of the whole expression, for example, `!karel.frontIsClear()`. We might read it as "not karel dot frontIsClear."

Exercises before Lab06

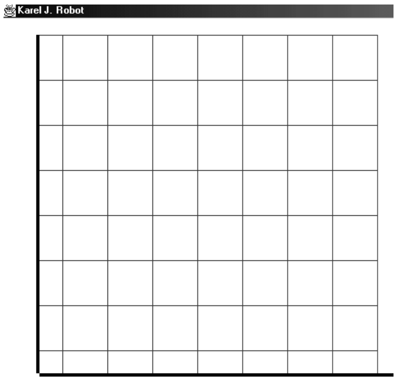
Booleans

Look at the Robot's API to see what boolean methods might be helpful here.

1. Write the commands to force a Robot object named `karel` to face west no matter what direction it is initially facing.	2. Write the commands to make an Athlete named `ann` to put down all her beepers.
3. Write the commands for Climber `c` to pick up a pile of beepers.	4. Write the commands for Racer `ray` to stop moving when it is next to (on top of) another robot.

5) Given the declarations and the default map, indicate the boolean value of each statement.

```
Robot pete = new Robot(5, 1, Display.WEST, 37);

Robot lisa = new Robot(2, 6, Display.SOUTH, 0);
```



`!pete.frontIsClear()`	true	false
`pete.hasBeepers()`	true	false
`!lisa.hasBeepers()`	true	false
`!pete.facingWest()`	true	false
`lisa.facingWest()`	true	false

6) The logical AND (in Java) is the symbols **`&&`**. The logical OR (in Java) is the symbols **`||`**. Evaluate:

true && true →

true && false →

false && true →

false && false →

true || true →

true || false →

false || true →

false || false →

7) Extra Credit: make a truth table that shows that De Morgan's Laws are true:

```
!(a && b) == !a || !b          !(a || b) == !a && !b
```

Lab06
A Half-Dozen Tasks

Objective

while-loops and if-statements

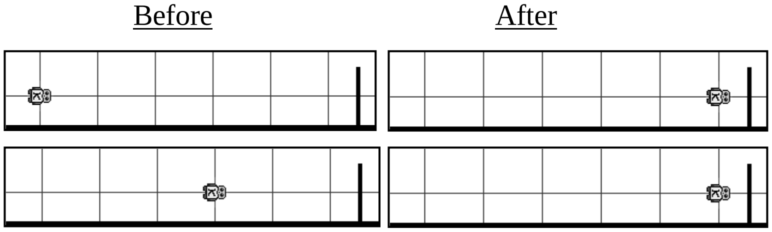
Background

This lab instantiates 6 Robots to perform 6 different tasks. For convenience, we code each task as a class method. Each task uses one or more indefinite loops. Some tasks also use if-statements.

Row	Task
6	Get across the gap (one step wide) and go to the end of the entire row of beepers.
5	Go to the wall, picking up all the beepers.
4	Go to the wall, picking up all the beepers (max one beeper per pile).
3	Go to the wall.
2	Go to the beeper.
1	Go to the end of the row of beepers.

If `temp` (for temporary) is a Robot object, the solution to task number three is:

```
while(temp.frontIsClear())
{
  temp.move();
}
```



This loop will repeat an unknown number of times, as long as temp's path is not blocked. The pictures show two such situations. Warning: if there is not a wall somewhere along the path then this segment of code will repeat forever; this is called an *infinite loop* and your program will never end.

This is the first lab in which your program must work correctly for many different robot-maps, all of which have the same basic structure. Thus, the name of the map will not be *hard-coded*:

```
Display.openWorld("maps/tasks1.map");
```

because your program cannot be fully tested with only the `tasks1` map. Instead you must import the javax.swing.JOptionPane class and use the commands:

```
String filename = JOptionPane.showInputDialog("What robot map?");
Display.openWorld("maps/" + filename + ".map");
```

When your program runs, an input dialog box will open prompting you for the name of the map to use. Run your program three times, entering either "tasks1", "tasks2", or "tasks3". Your program does not work unless it runs successfully for all three maps.

Specification

This is your first program that has shell code provided for you. Be sure to load Unit1\Lab06.java. Set the size to 10x10. Notice the 6 class methods, each instantiating a Robot object. Accomplish all six tasks. The same code must work with all three maps "tasks1", "tasks2", and "tasks3".

Extensions

1. Modify task_04 and task_05 so that they count and print the number of beepers. Look at the code on the next page for a hint. In tasks1.map, task_04 will count 4 beepers and task_05 will count 11.
2. Create a valid robot-map for this lab using `MapBuilder.jar` and test your program with that map.

Sample Run `Unit1Lab06.jar`

Exercises

Lab06

How can the robot keep count of its beepers? One way is to use an integer variable that starts at zero and increments every time. This code counts and prints the number of beepers in a pile.

```
int count = 0;
while(karel.nextToABeeper())
{
  karel.pickBeeper();
  count++;
}
System.out.println("Count is "+ count);
```

Complete the methods below.

public static void task_07() //go to the beeper or the wall. Count and report the number of steps you took.
{

}

public static void task_08() //go until you are next to another robot, then put all your beepers down.
{

}

public static void task_09() //put down 5 different piles with 4 beepers in each pile. Use definite loops.
{

}

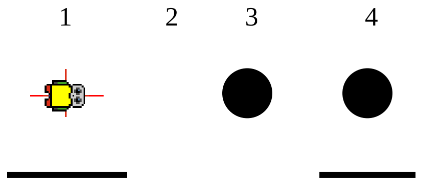
public static void task_10() //fill in gaps with a beeper. Stop when you reach a wall.
{

}

public static void task_11() //while there is a wall to your right, put down one beeper at each step
{

}

public static void task_12() //go until there is a wall to your right and you are standing on a beeper
{



}

Extension Using `MapBuilder.jar`, make your own maps for tasks 7 through 12 above. Figure out how to place and remove walls and beepers. When you save, you must type in the name and the extension `.map`

Lab07
Exploration

Objective

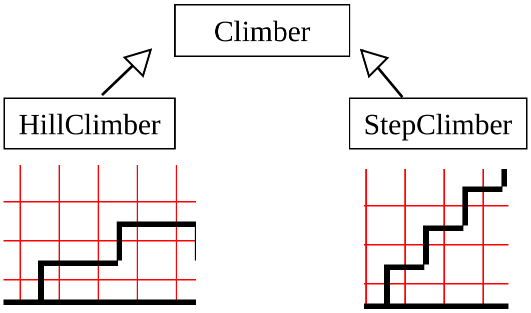
Polymorphism.

Background

In object oriented programming, if we set things up correctly, we can call the “same” methods, but by passing different subclass objects, we can get (slightly) different behaviors. This is a very powerful technique. As a real-world example, suppose the student council tells the students to wear their class colors during Homecoming. Every subclass of student, freshmen, sophomore, etc., acts on that command, but in its own way, meaning that freshmen know to wear red and sophomores white, and so on. That is polymorphic behavior.

To get polymorphic behavior in Java, we give commands to a reference, but pass subclass objects that actually perform those commands. Any time we have a superclass reference to a subclass object, we have the possibility of polymorphic behavior. Here is an example of a superclass reference (`Robot`) to a subclass object (`Athlete`): `Robot karel = new Athlete();`

In this lab you are going to define two subclasses of Climber, one to climb hills and one to climb steps. Four instance methods (`climbUpRight`, etc.) in each subclass will *override* the four similar instance methods in Climber. We will be giving commands to the Climber reference, but objects of the Climber, HillClimber, and StepClimber classes will act on those commands.



Let's study this simpler example:

```
Climber karel = new HillClimber();
karel.climbUpRight();
```

Which `climbUpRight` command will be followed, Climber’s or HillClimber’s? The answer is that Climber `karel` will execute `climbUpRight` as it is defined in the HillClimber class. `karel` “thinks” it is a Climber, but it behaves like a HillClimber. In the actual Lab07, we will call `explore(Climber arg)` and pass a HillClimber as an argument.

Specification

Load Mountain.java and study its method `explore`. Complete the Exercises on the next page.

Create Unit1\HillClimber.java and Unit1\StepClimber.java. Consult the Unit 1 API for more information regarding the HillClimber and StepClimber classes. Implement both the HillClimber and StepClimber classes. Recall that the actual arguments of the `super` in the HillClimber constructor must match the formal arguments accepted by the Climber constructor.

Load Unit1\Lab07.java. Compile and run. The application will prompt you, using the JOptionPane methods, to specify the map, the type of climber, and the x-coordinate starting position as follows:

mountain1, mountain2, mountain3	Climber	8
hill1, hill2, hill3	HillClimber	10
step1, step2, step3	StepClimber	12

Sample Run `Unit1Lab07.jar`

Exercises
Lab07

Looking at Unit1\Mountain.java, answer the following questions.

1. Does Mountain contain class methods or instance methods? _________ How do you know?

2. Explain the difference in syntax between `explore(arg)` and `arg.move()`.

3. How does `explore` know not to search the eastern mountain if the treasure was on the western one?

4. Explain the purpose of the first `arg.putBeeper()` and last `arg.pickBeeper()` commands of `explore`.

5. In `explore_west`, what is the variable `n` actually counting?

6. How does the Climber know that it has reached the summit?

7. If the treasure isn't on the east summit, how do we know not to try to pick it up?

8. Why can we use a **for-loop** going down the mountain when we needed a **while-loop** going up?

9. How are we able to find base camp?

10. Which specific method call(s) result in different behaviors, depending on the object?

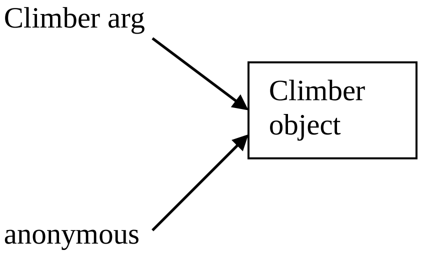
11. Does the steepness of the mountain change our code to go up and down? Why or why not?

12. How can different types of objects make use of a single, common code?

Discussion
Polymorphism

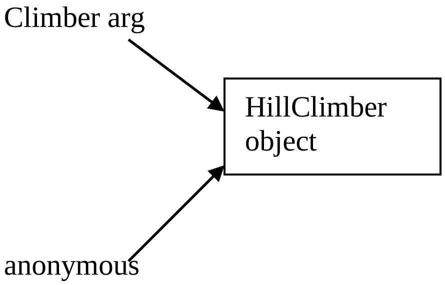
The formal argument to the method `Mountain.explore` is a reference of type Climber. When an actual argument, either Climber, HillClimber, or StepClimber, is passed to `Mountain.explore`, a copy of the reference is made. This means that the formal argument, named `arg`, will point to the same object as the actual, anonymous, argument. In other words, we *pass an object by reference*:

```
public class Mountain
{
  public static void explore(Climber arg)
  {
      //method definition here
  }
}
public class Lab07
{
  public static void main(String[] args)
  {
     Mountain.explore( new Climber(x) );
  }
}
```



So far, strictly speaking, there has been no polymorphism involved. Now we get to:

```
public class Mountain
{
  public static void explore(Climber arg)
  {
     //method definition here
  }
}
public class Lab07
{
  public static void main(String[] args)
  {
     Mountain.explore( new HillClimber(x) );
  }
}
```



In this situation, the Climber reference `arg` will point to a subclass HillClimber object. Is a HillClimber a legal argument to the method `explore`? Yes, the compiler (in this case) allows a superclass reference to a subclass object. The compiler checks to see if the reference has access to the method. The computer at run-time actually runs the method's code, whatever it may be. Therefore, when we give "the same" commands to `arg`, the subclass's methods are executed and we get polymorphic behavior from the code. `climbUpRight` is one of the *polymorphic methods* in this lab. Note we could pass a StepClimber to `explore`, but it would not pass the compiler to pass a Robot object, because Robot objects do not know how to `climbUpRight`.

Suppose the header were `explore(Robot arg)`. In that case, the compiler does not allow the superclass reference to the subclass object. However, we can fix it if we *cast* the reference. See Discussion after Lab08.

Polymorphism works because the code calls a superclass's method, but the subclass's (overridden) method is the one that actually gets executed. It is the type of the object and not the type of the reference that determines the behavior. The algorithm in `explore` is the same, but the objects are slightly different--but not totally different, for the objects must stand in a superclass/subclass relationship. Polymorphism is also called *dynamic method binding*, or *late binding*.

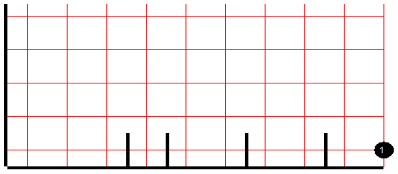
Lab08
Hurdle Racing

Objective

Polymorphism. The if-else control structure.

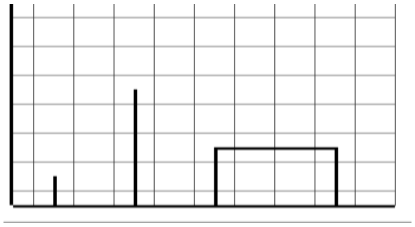
Background

The hurdles shown below are all one-block tall hurdles. A beeper marks the finish line. An object of the Racer class can traverse that race course using this class method:



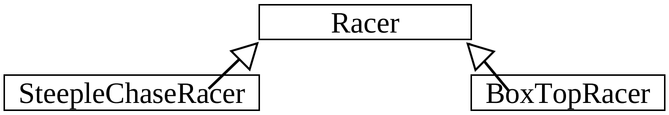
```
public static void race(Racer arg)
{
   while(!arg.nextToABeeper())
      if(arg.frontIsClear())
         arg.move();
      else
         arg.jumpRight();
}
```

In English, the algorithm is: if there's no hurdle, then move, otherwise jump over the hurdle. Keep going until you hit a beeper.



Now let's change the race-course so that instead of one-block tall hurdles we must jump over hurdles of any height. We do not change the algorithm above! The method `arg.jumpRight` is a perfectly good command—in the abstract—to jump hurdles of any height. All we need to do is to write our own `jumpRight`. In object-oriented programming we look around for a class that does approximately what we want (i.e. Racer) and the extend it. When we extend a class, we give the new class the powers that we really want—namely, how to jump hurdles of different heights.

What about jumping hurdles of differing widths? Again, our original algorithm still works! All we have to do is to write a new subclass that knows how to jump these new kinds of hurdles.



Each of the classes SteepleChaseRacer and BoxTopRacer will *extend* Racer and *override* `jumpRight`.

Specification

Consult the Unit 1 API for more information regarding the SteepleChaseRacer and BoxTopRacer classes. Create Unit1\SteepleChaseRacer.java and Unit1\BoxTopRacer.java. Implement both the SteepleChaseRacer and BoxTopRacer classes as described above, then compile.

Load Unit1\Lab08.java. Compile and run. With maps “hurdle1”, “hurdle2”, and “hurdle3”, use an object of type Racer. With maps “steeple1”, “steeple2”, and “steeple3”, use a SteepleChaseRacer object. With maps “boxtop1”, “boxtop2”, and “boxtop3”, use a BoxTopRacer object.

Extension

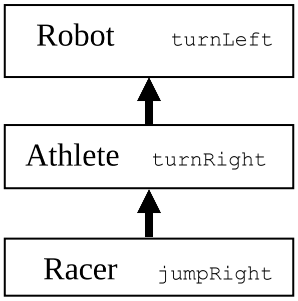
Have the class method `race` accept a Robot argument, not a Racer argument. That is, change the header to be `public static void race(Robot arg).` Explain why the code won't compile.

Sample Run `Unit1Lab08.jar`

Discussion

Casting

Polymorphic behavior is useful because the programmer can give one, general command to a superclass reference which is executed differently by each subclass. However, there sometimes is a glitch: if the superclass does not have access to a method, the code won't compile. Then it is the programmer's responsibility to *cast the reference*, thereby making the method available.



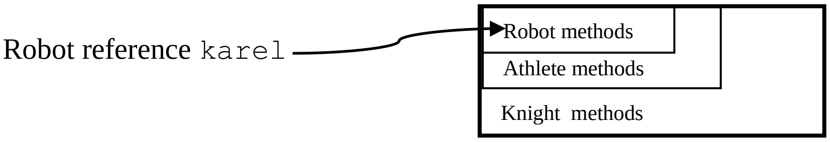
For example, in the previous lab, the Robot superclass reference does not know how to `jumpRight` because `jumpRight` is a completely new method in Racer. Yet the programmer is sometimes forced to use the Robot reference to talk to the Racer objects. The way to do this is to *cast the reference*. Casting creates a new "larger" reference pointing to the entire object. Casting uncovers powers that you know are already there, to which the compiler does not allow access. Casting changes the reference, never the object itself! Here is an example of a cast, which allows `arg2` to access the new methods: `Racer arg2 = (Racer)arg;`

Here is a real-world example (sort of): Clark Kent does not know how to fly. But if you cast Clark Kent to his other name, then he does know how to fly. You have uncovered his hidden powers. In Java we would write:

```
      Person clarkKent = new Kryptonian();
      clarkKent.fly();   //error!
      Kryptonian superman = (Kryptonian)clarkKent;
      superman.fly();
```

Let's begin again. As always, we start with code that contains a superclass reference to a subclass object:

```
      Robot karel = new Knight(4,3,Display.NORTH,6);
```



1. Is `karel.turnLeft()` a legal command at compile-time? _____ Why?

```
public class Knight
          extends Athlete
{
  //a 4-arg constructor
  public void turnLeft()
  {
  }
  public void turnRight()
  {
  }
  public void retreat()
  {
  }
}
```

2. Which `turnLeft` is actually executed at run-time in Question #1, the Robot's `turnLeft` or the Knight's `turnLeft`? ____________ Why?

3. Is `karel.turnRight()` a legal command?_______ Why?

4. Cast the Robot `karel` reference to a Knight reference `lancelot`, then tell `lancelot` to retreat.

5. Is `lancelot.retreat()` now a legal command? ____ Why?

6. On the diagram above, show that the new `lancelot` reference points to the entire Knight object.
7. Make the Extension to Lab08 work by casting the Robot reference. Explain what casting does.

Lab09

Shifting Piles

Objective

Algorithms. An algorithm is a sequence of steps, usually repeated, that eventually solves the problem.

Background

You'll get very few hints on how to solve this problem. Your robot is standing on the first row at (1, 1) facing east with zero beepers. Each block between (1, 1) and (7, 1), inclusive, has a pile of beepers. One or more of these piles may contain zero beepers. It is possible that all of the piles contain zero beepers. It is guaranteed that each pile contains a finite number of beepers. Your task is to shift each pile one block to the right. The pile that started at (2, 1) should move to (3, 1), the pile that started at (3, 1) should move to (4, 1), etc.

(Each block on the second row, between (1, 2) and (7, 2), also contains piles of beepers, but they DO NOT NEED TO BE MOVED! They only serve as a reference, so that when your program runs, you can easily check whether it has run properly.)

Ask yourself the following questions:

1. What are the initial conditions of the problem?

2. What are the terminating conditions of the problem?

3. List the sub-tasks that the robot must perform.

4. How can you proceed from one step of the problem to the next step of the problem?

5. Will your answer to Question #4 get you from the initial to the terminating condition?

Good luck. If you are using jGrasp, ask your teacher how to use the debugger. The debugger helps to see where your program is going wrong.

Specification

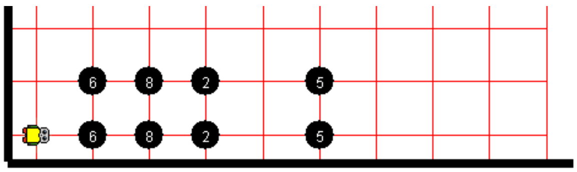
Create Unit1\Lab09.java at size 10x10. Declare one Athlete at (1, 1) facing east with zero beepers. Shift each of the piles on the first row one block to the right. Test the program with maps "pile1", "pile2", and "pile3".

Extension

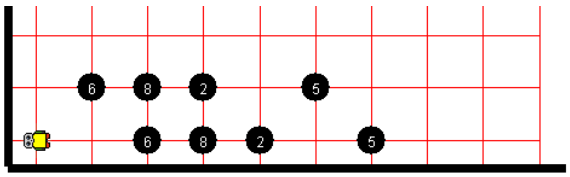
Use MapBuilder.jar to create valid robot-maps to test your program.

Sample Run

```
Unit1Lab09.jar
```



before

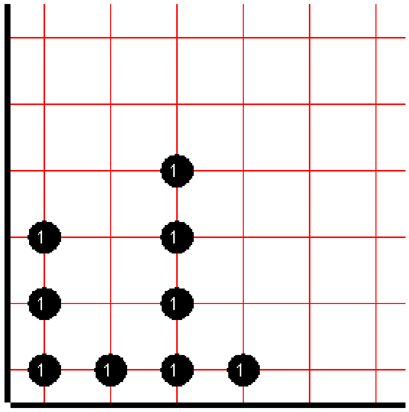


after

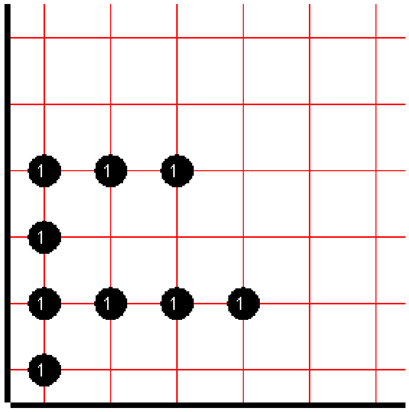
Exercises

Lab09

In this exercise you also have to pick up beepers, counting as you pick, then carry the beepers to another place and put them down. Change the vertical pictograph ("before") into the horizontal pictograph ("after"). There is only one beeper at each corner. There are no more than 4 columns of beepers. You do not know how tall each column is, but there are no gaps within a column.



Before



After

1. Plan the tasks, in English, not in code, which your Athlete must do.

2. Would it help to make a subclass of Athlete? Y/N. If so, what would you name it and what would it do?

3. Here is a useful task: assume that the robot has already picked up the column and stored the number of beepers in that column in the variable *rowCount2*. Assume that the robot is in position, facing east. Now write the code to put down *rowCount2* number of beepers in that row.

4. Your teacher may wish to turn this exercise into a lab. If so, you may use maps "picto1.map", "picto2.map", and "picto3.map".

Discussion
Decision Structures

The simplest decision structure, which you have already seen, is the **`if-`**statement.

```
if(karel.nextToABeeper())
  karel.pickBeeper();

if(total == 0)                // tests for equality, i.e. "is equal to"
{
  karel.turnRight();
  karel.move();
}
```

(Given this code fragment, we may assume that karel is an Athlete and `total` has previously been declared as an integer.) The if-statement is useful when we have one branch that may or may not happen. When we have two branches, only one of which will happen, we use the **`if-else`** statement.

```
if(karel.nextToABeeper())
  karel.pickBeeper();
else
  karel.putBeeper();          //only happens if !karel.nextToABeeper()
```

Here is an if-else example that requires curly braces:

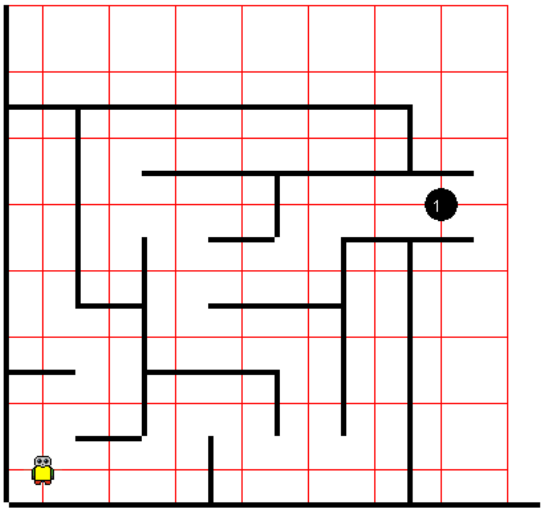
```
if(total != 0)                // tests for "is not equal to"
{
  karel.turnLeft();
  karel.move();
}
else
{
  karel.turnRight();           //only happens if total == 0
  karel.move();
}
```

In an if-else statement, exactly one of the choices will occur. It is impossible that neither will occur. It is impossible that both will occur. When three-way branching is needed, we use an else-if ladder:

```
if(total < 0)
  karel.turnRight();
else if(total > 0)
  karel.turnLeft();
else
  karel.turnAround();
karel.move();          //move goes here because it will definitely move no matter what
```

Of course, this three-way branching can be generalized to n-way branching by using as many else-if statements as are required. Make sure you end the ladder with an else statement to ensure that exactly one of the methods will be executed—and in some cases just to keep the paranoid Java compiler happy.

Lab10
Maze Escaping



Objective
Algorithms.

Background
You'll get very few hints on how to solve this problem. An Athlete is standing at (1, 1) facing north with an infinite number of beepers. Your robot is in a maze of unknown size. The maze has a continuous sequence of walls connecting the start of the maze to the end of the maze--that is, by keeping a wall on one hand, you will reach the end, which is a single beeper. A good problem-solving strategy is to imagine yourself in a corn maze. What do you do to exit the maze?

Ask yourself the following questions:

1. What is the initial state of the Athlete?

2. How does the Athlete know it is finished?

3. What does an Athlete know about walls?

4. At each step, the Athlete needs to make one decision and either move, turn left, or turn right. What sorts of situations could the Athlete be in? Draw them here:

Specification
Create Unit1\Lab10.java at size 10x10. Declare one Athlete at (1, 1) facing north with an infinite number of beepers. Write and use a class method `followWallsRight` which contains your code to escape the maze by following the walls on the right. Test your program on “maze1.map”, “maze2.map”, and “maze3.map”.

Sample Run `Unit1Lab10.jar`

Extension 1
Use `MapBuilder.jar` to create valid robot maps to test your program.

Extension 2
Make a class method `followWallsLeft`. Look up `Math.random` and figure out how to make the athlete follow either the left walls or the right walls with a 50-50 probability.

Extension 3
Mark the path from start to end. The final path should not mark detours that were tried and then later abandoned. <u>Hint</u>: The last sentence was another hint.

Exercises

Lab10

1. Given the declarations shown, circle the legal commands.

```
Robot karel = new Robot();
Athlete maria = new Athlete();
Climber lisa = new Climber(4);
Racer pete = new Racer(1);
Robot horatio = new Climber(8);
```

```
maria.facingEast();                  pete.pick();

karel.climbUpRight();                pete.sprint(-10);

lisa.jumpRight();                    pete.put(5);

lisa.climbUpRight();                 pete.climbUpRight();

horatio.move();                      pete.move(10);

Lab04.takeTheField(pete);            Lab08.race(karel);

Climber ophelia = new Robot();       Robot hamlet = new Racer(3);

Athlete kyla = (Athlete) karel;      Climber ray = (Climber) horatio;

Climber ann = (Climber) maria;       Racer jerry = (Racer) lisa;
```

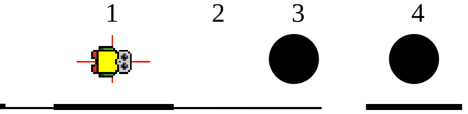
2. What is wrong with the instantiation:

```
Climber ophelia = new Climber(1, 1, Display.SOUTH, 10);
```

3. Write a constructor for a Roofer class that extends Robot. Roofers can start anywhere, but all Roofers face SOUTH and carry 100 beepers.

4. Write the code for: if you are on top of a beeper, pick them all up, and report how many there are.

5. Write the code for: move forward until there is a wall to your right and you are standing on a beeper



Discussion
Abstract Methods

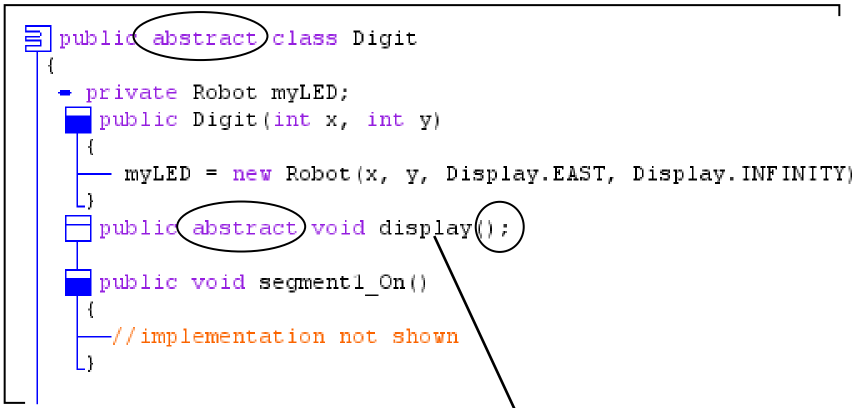
As a general term, an *interface* tells you **what** you can do to or with a thing. (What is the interface to a car?) It is a different matter to know **how** the thing works, which is the *implementation* of the interface. (How does a car work?) In object-oriented programming, often all you need to know is **what** methods are available, and you don't much care **how** the methods work. Other times, it will be your job to *implement the interface*.

An *abstract method* specifies an interface but not its implementation. For example, in the *abstract class* Digit the *abstract method* is `public abstract void display();`. Notice the semicolon: there is no code in the body of an abstract method!

The code (or implementation) for the abstract method has to be given elsewhere, namely, in the class that extends the abstract class. This new class is called a *concrete class*, since you can instantiate objects from it. It is impossible to instantiate objects from abstract classes. All the classes we have used up till now have been concrete classes.

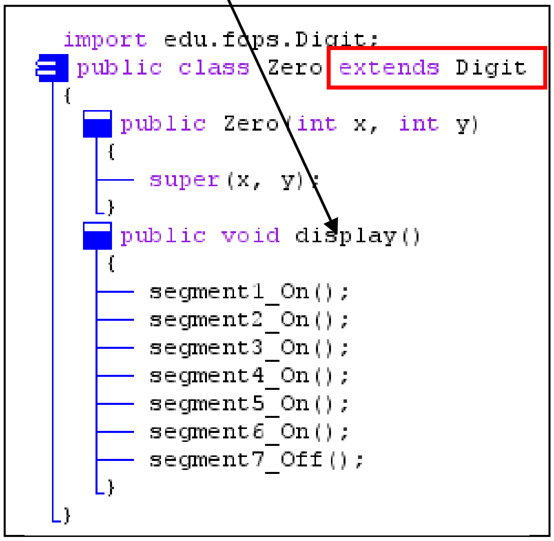
Abstract classes are useful when designing a class hierarchy. Usually, when you are given an abstract class, you extend it. Usually, when you extend an abstract class, you make it concrete, i.e., you write the implementation code for every abstract method. The compiler checks to see that any concrete subclass actually does override the abstract method. If not, the compiler will generate an error.

Look at the code on the right. Digit is an abstract class because it has the abstract method `display`. Notice that `display` has no implementation code in its body, but only a semicolon.



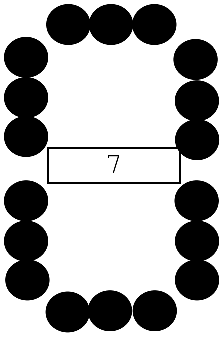
```
public abstract class Digit
{
   private Robot myLED;
   public Digit(int x, int y)
   {
      myLED = new Robot(x, y, Display.EAST, Display.INFINITY)
   }
   public abstract void display();

   public void segment1_On()
   {
      //implementation not shown
   }
```

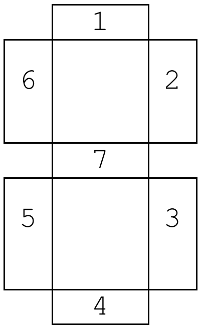
Notice that the class Zero extends the abstract class Digit. You know that Zero inherits `segment1_On` as well as the private Robot object. If you want to instantiate a Zero object, which drops beepers in the shape “0”, you must write code that implements the abstract method `display`.



```
import edu.fcps.Digit;
public class Zero extends Digit
{
   public Zero(int x, int y)
   {
      super(x, y);
   }
   public void display()
   {
      segment1_On();
      segment2_On();
      segment3_On();
      segment4_On();
      segment5_On();
      segment6_On();
      segment7_Off();
   }
}
```

Digits are formed from beepers in seven segments, each of which is either turned on or turned off. These methods must be called in order from one to seven, as shown in Zero's code.





If segments 1 to 6 are turned on, the rows of beepers will form the digit “0”. How do you display “1”?

Lab11
Your ID Number

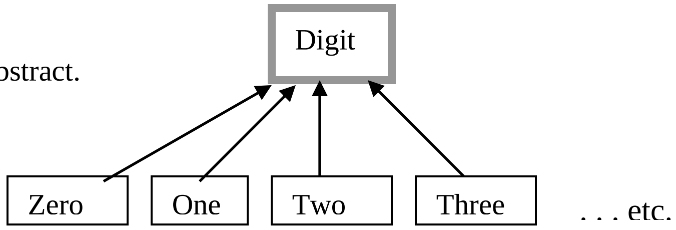
Objective

Extending an abstract class.

Background

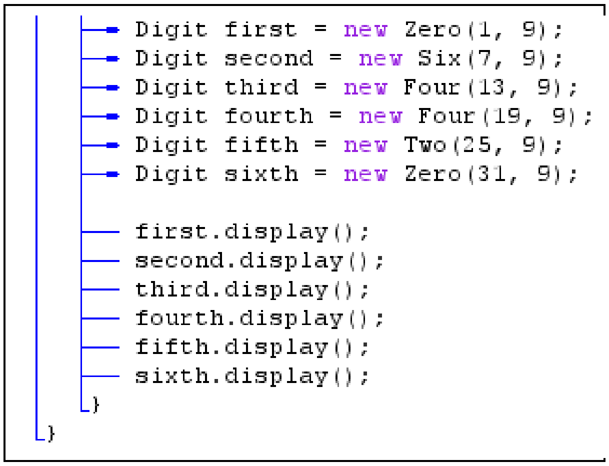
Digit is an abstract class. It provides code to turn on segments of beepers, but its `display` method is abstract. When you extend Digit, you will implement (provide the code for) `display` that is appropriate to each concrete class Zero, One, Two, etc.

The class hierarchy is familiar, although Digit is abstract.



This hierarchy models the concept of a digit, which is a higher abstraction than your concept of, say, a five. Your concept of a five in turn is a higher abstraction than your concept of this five here—5. The various degrees of abstraction illustrated here relate precisely to the relationship between an abstract class, a concrete class, and an object.

Mr. Torbert's student ID number is 064420. He wants to display his ID number in a robot-map using Digit objects. Part of his driver program is shown at the right. The first digit in his ID is a zero, the second digit is a six, the third digit is a four, and so on. Notice that each digit is modeled by its own concrete class. Look how he instantiates each object. Notice that the arguments to the constructor somehow control the horizontal spacing of the digits. He later calls the superclass's `display` method, which is abstract, but is overridden by each subclass's concrete method. Here is another example of polymorphism at work.



```
      Digit first = new Zero(1, 9);
      Digit second = new Six(7, 9);
      Digit third = new Four(13, 9);
      Digit fourth = new Four(19, 9);
      Digit fifth = new Two(25, 9);
      Digit sixth = new Zero(31, 9);

      first.display();
      second.display();
      third.display();
      fourth.display();
      fifth.display();
      sixth.display();
   }
}
```

Specification

Create Unit1\Lab11.java with a default-map at size 36x32 or 42x37. Import both `edu.fcps.karel2.Display` (why?) and `edu.fcps.Digit` (why?). Display your student ID number using appropriate Digit objects. For each different numeral, you will have to define a concrete subclass.

Extension 1

Since we only ever call one method for each object, we do not need *named* references. Instead of using

```
       first.display();
```

we could have used *anonymous* references, such as

```
       new Zero(1, 9).display();
```

Rewrite the code to use anonymous references.

Extension 2

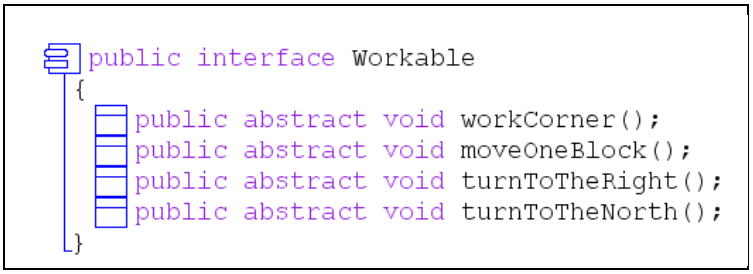
You have quite a bit of evidence about the behavior of Digit. Reverse engineer the Digit class. Make sure your version works.

Sample Run `Unit1Lab11.jar`

Discussion

Interfaces

As we said before, an *interface* tells you **what** methods are available. Java's special class called `interface` contains a list of abstract method(s), all of which are to be defined later in a subclass. At the right is the Workable interface, which is in the Unit 1 folder as `Workable.java`.



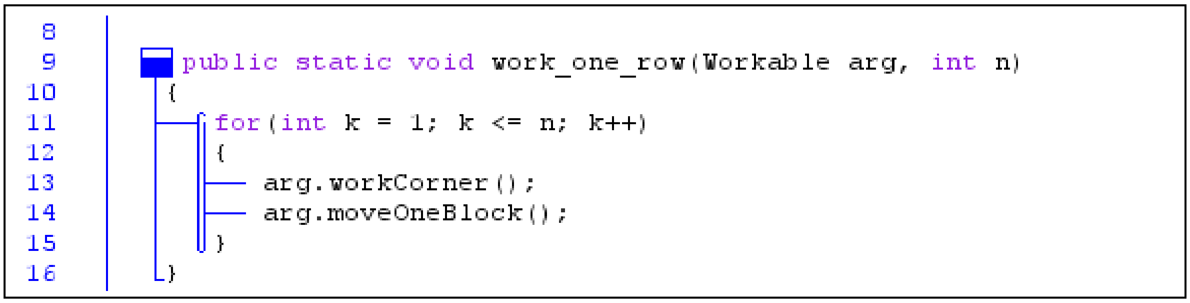
```
public interface Workable
{
   public abstract void workCorner();
   public abstract void moveOneBlock();
   public abstract void turnToTheRight();
   public abstract void turnToTheNorth();
}
```

Since an `interface` does not supply any implementation code, there is nothing to inherit from classes above in the hierarchy, and thus nothing to extend. Java uses a different keyword, `implements`, to promise that our subclass will provide code for all the abstract methods. That is, our subclass will *implement the interface.* A standard outline for a resource class looks like this:

```
public class Harvester extends Robot implements Workable
   {
     //private data
     //constructors
     //new instance methods in Harvester
     //code for the four abstract methods from Workable
   }
```

Again, the interface specifies **what** the methods are and other, unrelated, subclasses define **how** those methods work. Specifically, a `Workable` reference "knows" it can do `workCorner,` etc., but it does not "know how to do" `workCorner,` etc. It is the subclass objects that "knows how to do" `workCorner,` etc.

Mr. Torbert wrote this class method (from the next lab) to work with any `Workable` object:



```
public static void work_one_row(Workable arg, int n)
{
   for(int k = 1; k <= n; k++)
   {
      arg.workCorner();
      arg.moveOneBlock();
   }
}
```

Remarkably, we can't tell by looking only at this code what it will do. All we know, because it compiles, is that the `arg` object has implementation code for `workCorner`, etc. The actual behavior depends on how the methods `workCorner`, etc., are defined in the subclasses, which might not even be Robot classes at all.

In the cases of `Harvester` and `Carpeter`, the compiler guarantees that the `Workable arg` reference has access to `workCorner` , etc., but how they work depends on which object the `arg` is pointing to. Does that sound familiar? It should, because here is another example of polymorphic behavior.

The power and beauty of this is that the driver application does not care about the object’s implementation. The driver calls the methods (that are conveniently listed in the interface and also in the API) and the object does its task. Object-oriented programming is so easy!

Lab12

Harvesting and Carpeting

Objective

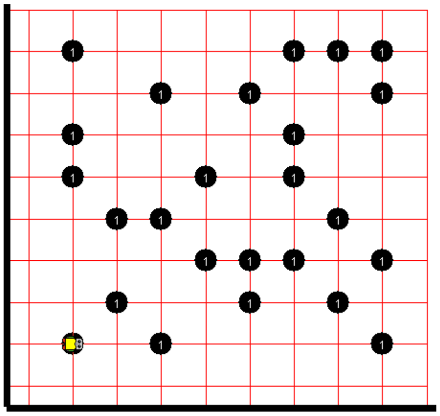
Implementing an interface.

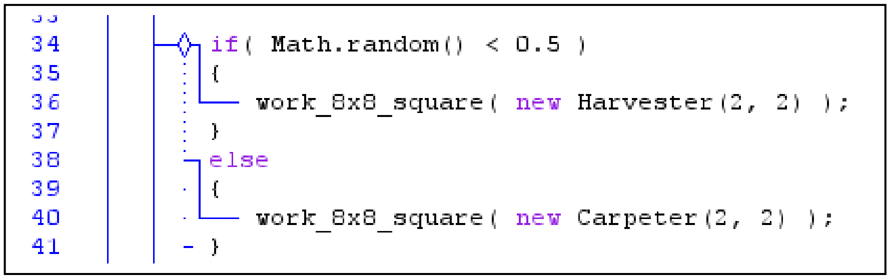
Background

You are going to write two classes named `Harvester` and `Carpeter`. Each of these classes will implement the `Workable` interface, but in slightly different ways. Polymorphism!

So how are harvesters different from carpeters? Harvesters pick beepers up if they find them, thus clearing the map of beepers. Carpeters put beepers down if the corner is empty, thus filling the map with single beepers.

The driver class will use a class method `work_one_row` and pass a `Workable` reference. Therefore, this lab is another example of a superclass reference to a subclass object.

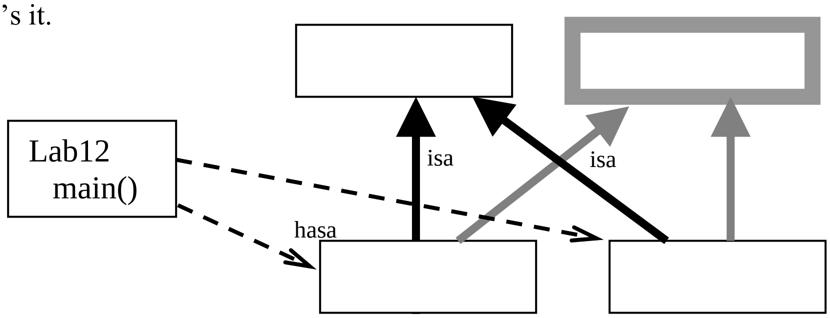




```
if( Math.random() < 0.5 )
{
   work_8x8_square( new Harvester(2, 2) );
}
else
{
   work_8x8_square( new Carpeter(2, 2) );
}
```

Code in the main method, shown to the right, will create either a harvester or a carpeter with equal probability. The Math class's `random` method generates and returns one decimal number between zero and strictly less than one. Other ways to say this are that `Math.random` returns a value such that 0 <= value < 1, or that `Math.random` returns a value from the interval [0, 1).

Important: The `Workable` interface, like all interfaces, is outside any given hierarchy. An interface knows that it can run the specified methods (the **what**), but does not know how those methods work. Accordingly, the method definitions (the **how**) in `Harvester` and `Carpeter` may seem too simplistic. Therefore, if a `Workable` method is supposed to move the robot forward one block, just tell it what "move" means according to the robot hierarchy. That's it.



Fill in the UML diagram at right.
Which methods come from Robot?
Which methods come from Workable?

Specification

Load Unit1\Harvester.java and Unit1\Carpeter.java. Implement the `Workable` interface.

Load Unit1\Lab12.java. Study the two class methods carefully. This application walks a spiral square with either a harvester or a carpeter. Run the program over and over until you see that it works with both types of workers.

Sample Run `Unit1Lab12.jar`

Exercises

Lab12

```
1       public interface Swims
2       {
3          void swim();
4       }
5       public abstract class Fish implements Swims
6       {
7          public Fish()
8          {  /* some code  */  }
9          public void swim()
10         {  /* some code  */  }
11         public abstract void breed();
12      }
13      public class Catfish extends Fish
14      {
15         public Catfish()
16         { /* some code  */  }
17         public void breed()
18         { /* some code  */  }
19      }
20      public class Dolphin extends Mammal implements Swims
21      {
22         public void swim()
23         { /* some code  */  }
24      }
```

1. Draw the UML diagram for the classes above.
2. Does Line 3 compile without error? __ Why or why not?
3. Does Fish implement Swims correctly? __ Why or why not?
4. Does Catfish extend Fish correctly? __ Why or why not?
5. Does Dolphin implement Swims correctly? __ Why or why not?
6. Fish and Catfish both have constructors (Lines ____ and ____). Does Dolphin need to have a constructor too? ___ Why or why not?
7. What do Catfish "know" how to do?
8. Which of the following will compile without error?

```
a. Fish f = new Fish();
b. Catfish c = new Fish();
c. Fish f = new Catfish();
d. Swims s = new Catfish();
e. Catfish c = new Swims();
f. Mammal m = new Dolphin();
g. Swims  s = new Dolphin();
h. Fish f = new Dolphin();
```

Lab13--Project

An Original fcpsKarel Lab

Use your imagination and your experience from this unit to create an original fcpsKarel problem. In formulating your own fcpsKarel problem, consider the following:

1. How many robots are involved?
2. What are the initial conditions?
3. What is the goal?
4. How many robot-maps do you need to make, and what do they look like?
5. What can the programmer assume? For instance, will there always be a beeper at (5, 4) or do we have to be flexible?
6. What control structures (`while`, `for`, `if`, `if-else`) are needed?
7. What existing fcpsKarel class or classes may be appropriate to use in the solution?
8. What extended classes may be helpful (or necessary) in solving the problem?
9. What abstract classes or interfaces are helpful?
10. What story can you devise to suit this lab?

You may use previous labs as guides to choosing an appropriate problem, but try to be creative as well. You should code your own solution that works before writing up the lab assignment for your peers.

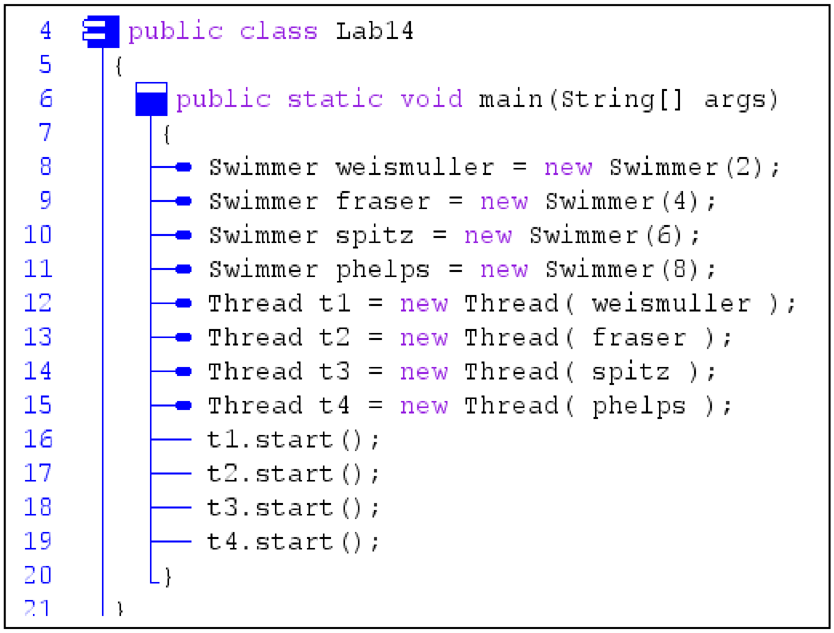
Here is one possible grading rubric. Check with your teacher for sure.

Word document looks just like a fcpsKarel Lab:	0----1----2----3----4----5
lab assignment for peers is appropriate	0----1----2----3----4----5
lab is at an appropriate level of difficulty:	0----1----2----3----4----5
UML class diagram is correct:	0----1----2----3----4----5
some elements of the task require the robot to make decisions, i.e., the program works in different maps:	0----1----2----3----4----5
maps are appropriate:	0----1----2----3----4----5
uses a class method to solve the problem:	0----1----2----3----4----5
uses inheritance appropriately:	0----1----2----3----4----5
uses polymorphism appropriately:	0----1----2----3----4----5
uses an abstract class or implements an interface:	0----1----2----3----4----5
the solution works:	0----1----2----3----4----5
the lab is creative and original:	0----1----2----3----4----5

Discussion
Threads

Programmers use abstract classes and interfaces to guarantee that subclasses have certain methods. The example in Lab14 is Thread and Runnable. Java uses Thread and Runnable to allow several different objects all to execute their own methods at the same time, in *parallel.* Java has a `Thread` class that takes care of the details as to **how** it all works. You don't need to know how, but you do have to *implement the Runnable interface* in the resource subclass.

At first glance at Lab14 (shown here to the right) you may be wondering if this is really a robot lab at all. Where are Robot objects and Display? All we see are Thread and Swimmer objects. We have to look at the Swimmer class (or its API) to see that Swimmer `extends Robot implements Runnable`. Implementing Runnable is important because a Thread constructor requires a Runnable object as an argument. In Lab14, the compiler checks to make sure that weismuller, fraser, etc., are all Runnables. If you implement Runnable correctly, then the Thread's `start` command will cause all four Swimmers to swim laps in parallel.



```
public class Lab14
{
   public static void main(String[] args)
   {
      Swimmer weismuller = new Swimmer(2);
      Swimmer fraser = new Swimmer(4);
      Swimmer spitz = new Swimmer(6);
      Swimmer phelps = new Swimmer(8);
      Thread t1 = new Thread( weismuller );
      Thread t2 = new Thread( fraser );
      Thread t3 = new Thread( spitz );
      Thread t4 = new Thread( phelps );
      t1.start();
      t2.start();
      t3.start();
      t4.start();
   }
}
```

Here is another example that requires parallel processing. Suppose we are writing a Java program that shows a video with audio. Obviously, we want our video and audio to execute together, at the same time. If you understood the words in the paragraph above, then you can complete the little shell at the right.

```
public class Video ______________________
{ }
public class Audio ______________________
{ }
public class MyShow
{
  public static void main(String[] args)
  {




  }
```

Here is a syntax point. We did not have to give our swimmers names. We could have started the threads with anonymous swimmer objects, as follows:

```
Thread t1 = new Thread( new Swimmer(2) );
t1.start();
```

In fact, we didn't even have to name the threads: `new Thread( new Swimmer(2) ).start();`

Which do you think is the clearest and most readable? Ultimately, convention dictates how to write the most readable code.

Lab14
Synchronized Swimming

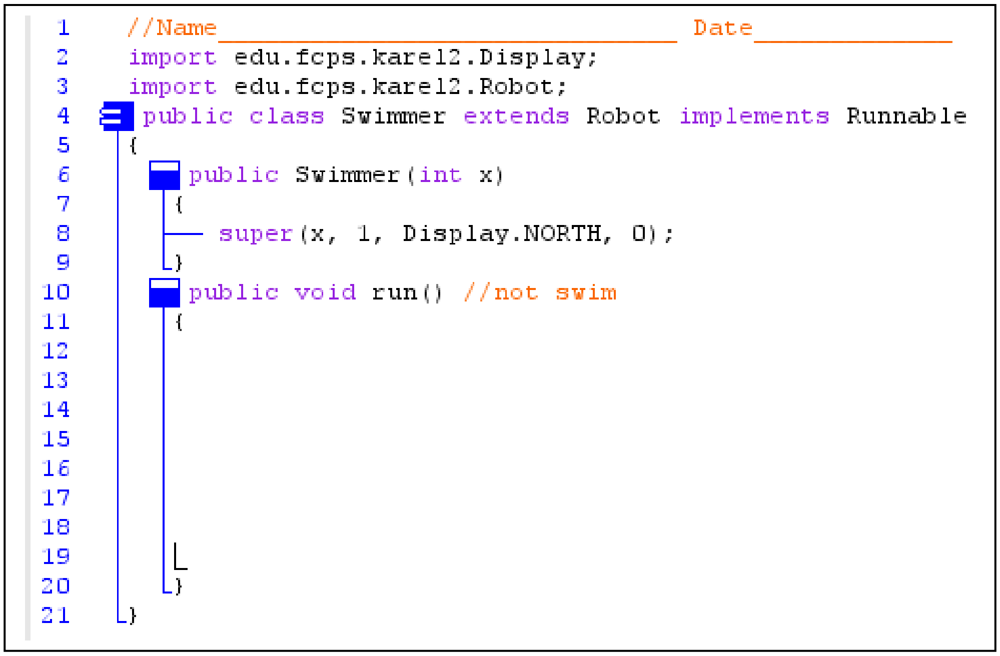
Objective

Implement an interface to achieve parallel programming.

Background

```
public interface Runnable
{
   public abstract void run();
}
```

Workable was an interface that Mr. Torbert created. Unlike Workable, Runnable is a built-in interface in java.lang.Runnable. (Therefore, don't type it in!) The Runnable interface specifies a single abstract method `run`. Although it doesn't look very useful, this interface is Java's way to guarantee parallel behavior.

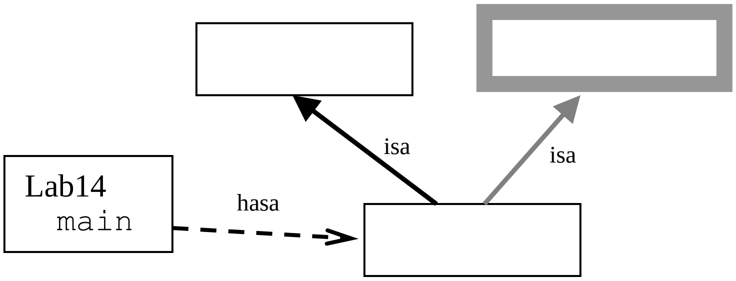


```
//Name______________________________ Date_____________
import edu.fcps.karel2.Display;
import edu.fcps.karel2.Robot;
public class Swimmer extends Robot implements Runnable
{
   public Swimmer(int x)
   {
      super(x, 1, Display.NORTH, 0);
   }
   public void run() //not swim
   {








   }
}
```

A Thread constructor requires a Runnable object as an argument. Every Thread object has a `start` method that runs `run` and takes care of all the parallel processes. In other words, all we have to do to produce parallel behavior is to implement the Runnable interface in our Swimmer object. That's your job.

In Java, a class may implement as many interfaces as desired, but is allowed to extend only one class. This rule guarantees that there is only one definition for each method, not two or more, which would lead to conflicts.

Fill in the UML diagram for Lab14. Which methods in Swimmer come from Robot? Which methods come from Runnable?



Specification

Load Unit1\Swimmer.java. Implement the Runnable interface so that a swimmer moves forward eight steps, twirls around, then moves back to its starting position to prepare for the next iteration—like one lap in a pool. Make a swimmer do ten laps.

Load Unit1\Lab14.java. Notice there are four Threads and Swimmers. Compile and run.

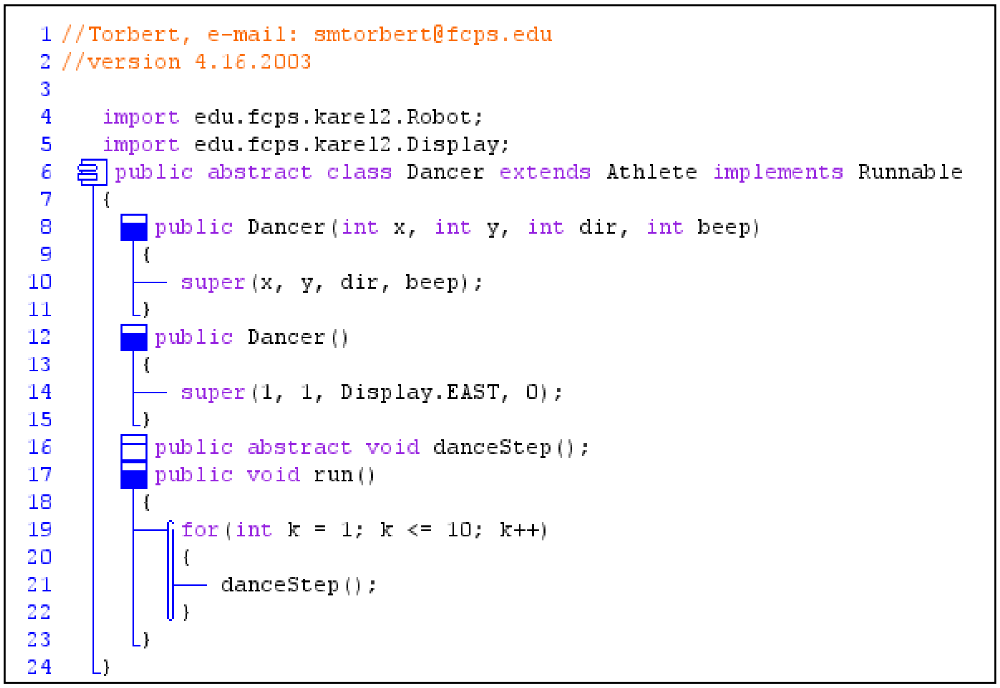
Extension

Return to Lab05, the hurdle problem. Have the Racer class implement the Runnable interface. Modify Lab05 so that the racers run in parallel.

Sample Run `Unit1Lab14.jar` and `Unit1Lab05ext.jar`

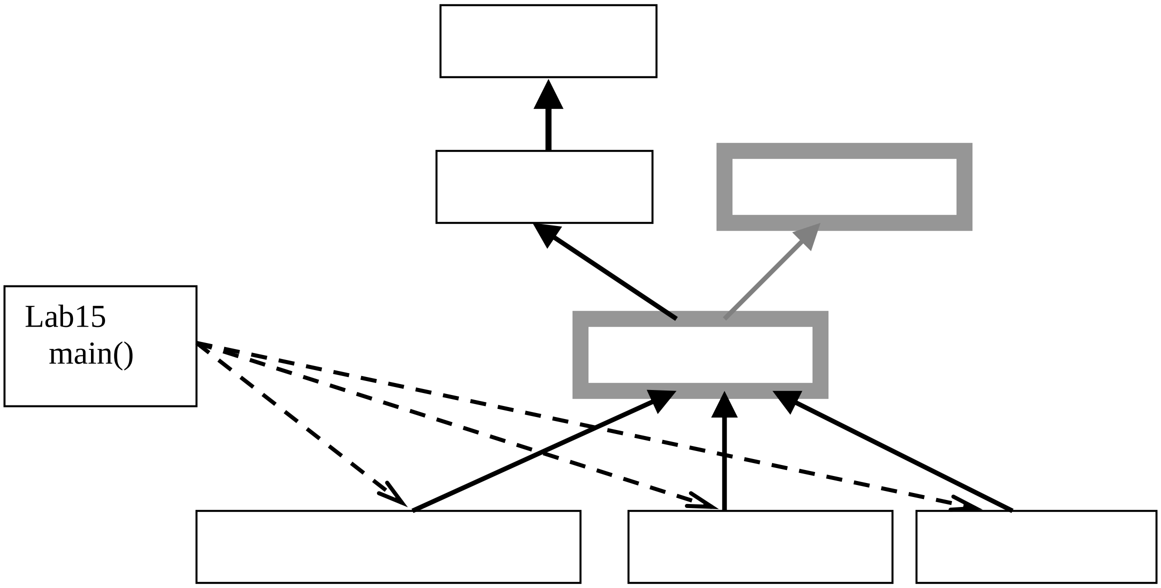
Discussion
Abstract Classes, Interfaces, and Concrete Classes

Mr. Torbert wants you write several Robot that classes that dance in parallel in different ways. Because he is a good programmer, he first writes an abstract Dancer class with an abstract method `danceStep`. He implements Runnable in Dancer, because he wants all the sub-Dancers to be able to dance in parallel. Because Dancer is abstract, it is impossible to instantiate these Dancer objects. However, he needs to provide Dancer two constructors (lines 8-15) because it is part of the hierarchy to `super` up to these from Dancer's subclasses. Somewhere in the hierarchy below Dancer, we know we are going to write at least one subclass class that will be concrete. That is, somewhere in a class that extends Dancer, we will eventually have to implement all the abstract methods and interfaces.



```
 1 //Torbert, e-mail: smtorbert@fcps.edu
 2 //version 4.16.2003
 3
 4    import edu.fcps.karel2.Robot;
 5    import edu.fcps.karel2.Display;
 6     public abstract class Dancer extends Athlete implements Runnable
 7    {
 8        public Dancer(int x, int y, int dir, int beep)
 9       {
10          super(x, y, dir, beep);
11       }
12        public Dancer()
13       {
14          super(1, 1, Display.EAST, 0);
15       }
16        public abstract void danceStep();
17        public void run()
18       {
19          for(int k = 1; k <= 10; k++)
20          {
21             danceStep();
22          }
23       }
24    }
```

The class hierarchies can become quite complicated. Try filling out this one for Lab15.



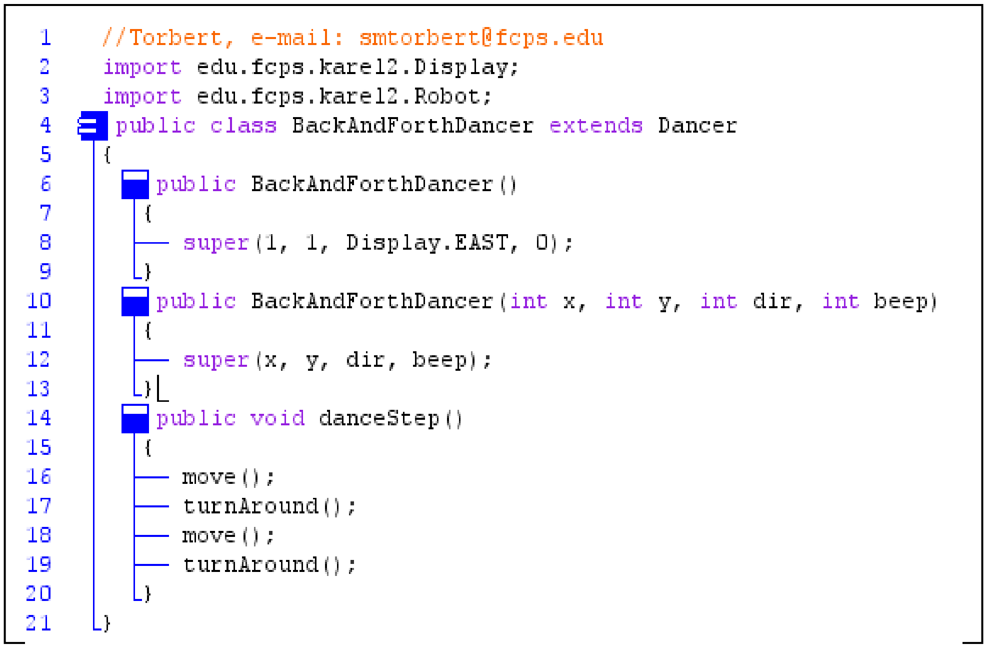
Lab15
Dancing Robots

Objective

Extend an abstract class.

Background

Here is an example of a class that extends Dancer and implements `danceStep`. Notice how simple this class looks, since it appears to contain only one instance method. Of course, this dancer inherits lots of functionality from the entire Robot hierarchy. This is an example of good object-oriented programming.



```
//Torbert, e-mail: smtorbert@fcps.edu
import edu.fcps.karel2.Display;
import edu.fcps.karel2.Robot;
public class BackAndForthDancer extends Dancer
{
   public BackAndForthDancer()
   {
      super(1, 1, Display.EAST, 0);
   }
   public BackAndForthDancer(int x, int y, int dir, int beep)
   {
      super(x, y, dir, beep);
   }
   public void danceStep()
   {
      move();
      turnAround();
      move();
      turnAround();
   }
}
```

You should be aware that, although you have used multiple threads in the last few labs, your programs have been using one thread since the first day, because the main thread of any Java application exists by default. The main thread executes the commands in the `public static void main(String[] args)` method, which must be present, exactly as written, in every Java application.

Specification

The class above may be world's most boring dancer ever. Using the code above as a model, write at least three different Dancer subclasses, but make your dancers more interesting than just back and forth. For instance, you might have a square dancer, a line dancer, a break-dancer, a waltzer.

Create Unit1\Lab15.java. Model your Lab15 `main` on the Lab14 `main`. Instantiate objects from your three different dancer classes and have them "do their stuff" in parallel.

Sample Run

`Unit1Lab15.jar`

Lab16
Shifty Robots

Objective

Implementing multiple interfaces.

Background

Isn't it amazing that your Lab14 and Lab15 programs executed in parallel, even though you don't know how parallelization works? Some one else did all that work. It works for your code in Lab16 because the parallelization algorithms in the Thread class were written for any Runnable object. That is, when you implement the Runnable interface, then you automatically get parallelization. Oh, the power of object-oriented design.

Let's return to the problem of shifting piles of beepers, as in Lab09. The algorithm most people used was to repeat 6 times: pick up a pile while counting the beepers, put down the previous pile, and move one block. Since we are working each corner, it might be nice to implement the Workable interface. Which methods are in the workable interface? Study Lab12, Harvester and Carpeter, especially `workCorner` and `moveOneBlock`. What does it mean in Lab16 to `workCorner`? What does it mean to `moveOneBlock`?

Since we want to shift four rows at a time, in parallel, we of course will need to implement the Runnable interface. Which method is in the Runnable interface? See Lab14, Synchronized Swimming. What do we want `run` to do in this lab?

One purpose of this lab is to let you implement two interfaces. The Shifter class has the header:

```
public class Shifter extends Robot implements Runnable, Workable
```

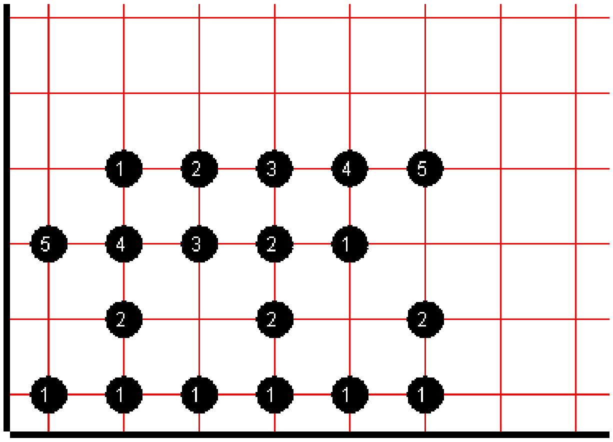
Another purpose of this lab is to show you a private variable. The Shifter class initializes a private variable (or private field) with the command:

```
private int myBeepers;
```

`myBeepers` stores integers. When the Shifter class is instantiated, each shifter object has its own copy of this variable—that's why it is private. In this lab, we have four shifter objects and therefore four different `myBeepers` values. One shifter might be storing the number "3" while another is storing the number "1".

Specification

Load Unit1\Shifter.java. The problem is to shift four rows of piles of beepers, in parallel. Of course, implement all the methods from the two interfaces. Use the field `myBeepers` to help keep count. You may assume that the rows to be worked are always six blocks long (when you shift, of course, you will then go into the seventh block).



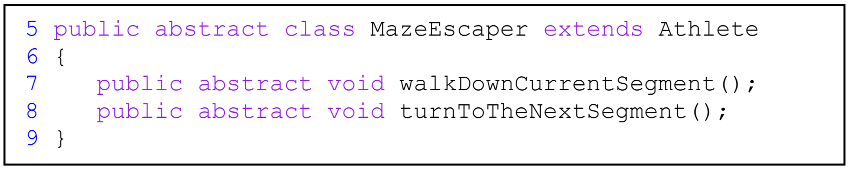
Load Unit1\Lab16.java. Compile and run. The application uses the "shifty" map.

Sample Run

`Unit1Lab16.jar`

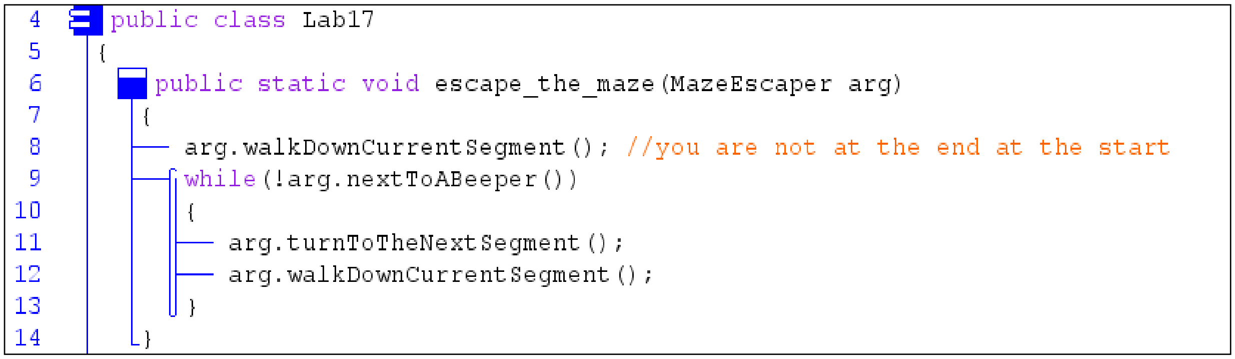
Discussion
Abstract Classes and Interfaces

When do programmers write an abstract class and when do they write an interface? MazeEscaper, for example, contains only abstract methods.



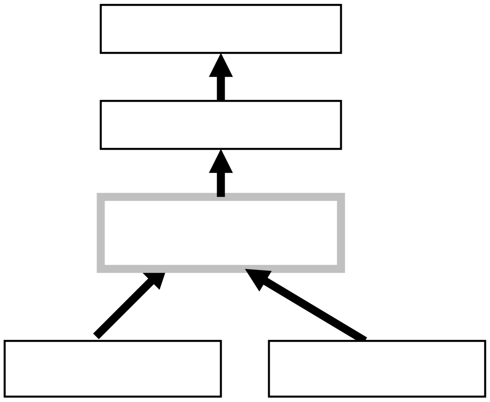
```
public abstract class MazeEscaper extends Athlete
{
   public abstract void walkDownCurrentSegment();
   public abstract void turnToTheNextSegment();
}
```

It seems as though MazeEscaper could have been written as an interface. However, Mr. Torbert chose to make it an abstract class in order to guarantee that MazeEscapers "know" (or have access to) Athlete's methods. Let's look at how the driver code calls an Athlete's method when passing a `MazeEscaper arg`.



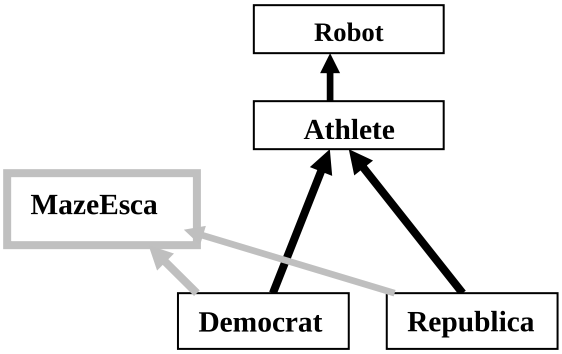
```
public class Lab17
{
   public static void escape_the_maze(MazeEscaper arg)
   {
      arg.walkDownCurrentSegment(); //you are not at the end at the start
      while(!arg.nextToABeeper())
      {
         arg.turnToTheNextSegment();
         arg.walkDownCurrentSegment();
      }
   }
```

Which command used above is a command that Athletes know? ______________________ Because MazeEscapers need to know that command, MazeEscapers ought to be a part of the Robot hierarchy. An abstract MazeEscaper can extend Athlete, and a concrete Democrat can extend MazeEscaper. Fill in the hierarchy:



The rule of thumb is, if inheriting behaviors is important, then insert an abstract class in the hierarchy, like MazeEscaper, Dancer (Lab14), or Digit (Lab11). On the other hand, if the desired behavior is completely unrelated to the hierarchy, like `run` in Runnable and the Robot hierarchy, then write an interface.

If Mr. Torbert had written MazeEscaper as an interface, it would not guarantee inheritance from Athlete. Suppose he wrote it anyway as an interface, like `Republican extends Athlete implements MazeEscaper`, shown in the UML diagram. If we instantiate `MazeEscaper pol = new Republican();`, the `pol` reference does not have access to any Athlete commands. However, we could still get the `pol` object to escape the maze if we *cast* the old `MazeEscaper arg` reference to a new `Republican arg2` reference, like this:



```
Republican arg2 = (Republican) arg;
```

Now the `Republican` object's "hidden methods" have been uncovered. The new "larger" `arg2` reference is able to access all of the Republican's methods. If you understand this, then you understand casting.

Lab17
Republicans and Democrats

Objective

Extending an abstract class.

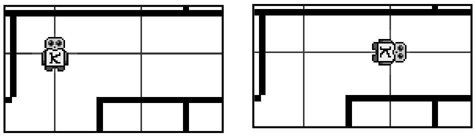
Background

A maze is a robot-map containing walls but no islands. A beeper marks the exit. A robot can escape from any maze by constantly keeping a wall to one side. This time, we have abstracted the walking and the turning into two abstract methods in MazeEscaper. Does this algorithm always work, in the abstract?

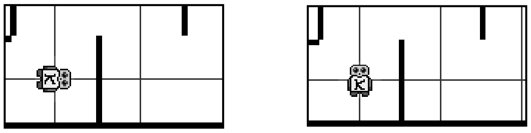
```
public abstract class MazeEscaper extends Athlete
  {
    public abstract void walkDownCurrentSegment();
    public abstract void turnToTheNextSegment();
  }
```

Your job, of course, is to make the abstract methods concrete in two subclasses. Here is the “follow walls right” algorithm: it is okay to move forward one step if there is a wall to your right AND your front is clear. Then, if your right is clear, turn right and take a step; otherwise, simply turn left. The algorithm does not need a special case for getting stuck in dead ends because the robot will simply make two left turns and thus effectively turn itself around.

Turning the corner to the right.



Turn left.



Obviously, we could just as easily “follow walls left” to escape the maze. The abstract class allows both Republican and Democrat objects to polymorphically use the same static class method `escape_the_maze` in the driver. We don't need to cast because MazeEscaper is now in the Robot hierarchy.

Specification

Create two concrete subclasses of MazeEscaper, a Republican who follows walls right and a Democrat who follows walls left, each implementing the abstract methods in MazeEscaper. You don't need to write constructors, for in this case Java automatically generates default `super` constructors.

Load Unit1\Lab17.java. In the `main`, comment out the `escape_the_maze` lines that you are not testing. Test the program using “maze1”, “maze2”, “maze3”, and “maze4”.

Extension 1

Start each robot in the upper left hand corner. You must write the appropriate constructor in MazeEscaper.

Extension 2

Write MazeEscaper as an interface, making all necessary changes.

Extension 3

Let MazeEscaper implement Runnable, making all necessary changes. Which politician escapes first?

Sample Run

`Unit1Lab17.jar`

Exercises
Lab17

1. List three advantages to writing abstract methods, either as an abstract class or as an interface.

2. Suppose you were designing objects that can fly. Would it be better to make Flier an abstract class or an interface, and why?

3. Suppose you were designing Mammal objects. Would it be better to make Mammal an abstract class or an interface, and why?

4. Generalizing from #2 and #3 above, when do programmers write interfaces and when do they write abstract classes?

5. Do MazeEscaper, Democrat, and Republican have constructors? Y/N Why or why not?

6. Are constructors inherited in Java? Y/N

7. Suppose we write this constructor in the abstract class `MazeEscaper`:

```
public MazeEscaper()
{
   super(1, 6, Display.NORTH, 0);
}
```

   a. Is it legal to put a constructor in an abstract class? Y/N
   b. Does this count as a default constructor? Y/N
   c. Does every class in the hierarchy still compile? Y/ N
   d. What does `super` call?
   e. What happens to 1, 6, Display.NORTH, 0?

8. Define "polymorphism."

9. Explain how and where Lab17 demonstrates polymorphic behavior.

Discussion

Return, Break and Continue

One way to approach the Lab17 problem is:

```
public void walkDownCurrentSegment()
{
   while( SOMETHING && !SOMETHING_ELSE ) {
      if( SOME_OTHER_THING) {
         DO_SOMETHING;
         return;
      }
      DO_SOMETHING_ELSE;
   }
}
```

The `return` statement causes the method to end immediately. Control of execution is then "returned" to whatever method called `walkDownCurrentSegment`, namely, to the `escape_the_maze` method in the Lab17. The `return` statement is a useful tool for handling special cases in an algorithm. Two other sometimes-useful tools are the `break` and `continue` statements, which work in loops.

```
while(karel.frontIsClear())
{
   if(karel.nextToARobot())
      break;
   if(karel.nextToABeeper())
   {
      karel.pickBeeper();
      continue;
   }
   karel.move();
}
```

This while-loop will repeat so long as karel's front is clear. If karel should ever be next to another robot, the break statement will cause the loop to end immediately. If karel should ever be next to a beeper, karel will pick up that beeper and the loop will continue back at the top; karel will not have moved forward. The continue statement ends the current iteration of the loop but then continues the loop again at the start.

By convention these statement make your code less readable then simply having methods that end appropriately, loops that stop appropriately, and decision statements that control program flow appropriately. The example shown above is probably better written as:

```
while(karel.frontIsClear() && !karel.nextToARobot())
{
   if(karel.nextToABeeper())
      karel.pickBeeper();
   else
      karel.move();
}
```

Use return, break, and continue sparingly and only for handling special cases.

Lab18

Treasure Hunt

Objective

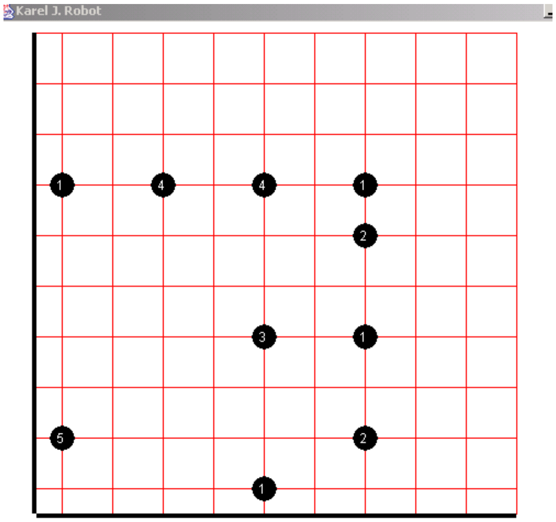
Implement an algorithm.

Background

There are treasures hidden in various robot-maps. Piles of beepers will guide your robot to the treasures. The treasure your robot is searching for is marked by a pile of exactly five beepers. To read the maps, your robot must walk forward until it encounters a pile of beepers. If that pile is the treasure, then you're done. If not, have your robot turn a certain way based on the number of beepers in the pile. Then continue searching for the treasure.

<u>Algorithm for Reading a Map</u>

1. Continue forward until you encounter a pile of beepers.
2. If you are at a pile with exactly five beepers, you've found the treasure.
3. Otherwise, if there is exactly one beeper, then turn left.
4. Otherwise, if there are exactly two beepers, then turn around.
5. Otherwise, if there are exactly three beepers, then turn right.
5. Otherwise, maintain your current heading.
6. Repeat as needed.



You must pick up the pile in order to count the beepers. Don't put the beepers down again.

Consult the Unit 1 API for more information regarding the Pirate class.

Specification

Load Unit1\Pirate.java. Pirate extends Athlete and has only a no-arg constructor starting each pirate at (1, 1) facing east with no beepers. There are three methods in Pirate that you must implement so that Lab18 will work. The method `approachPile()` should give you no trouble. The method

```
        public int numOfBeepersInPile()
```

has to count and return the number of beepers in that pile. Ask your teacher how to *return* a value from a method. The method `turnAppropriately(int beepers)` can be implemented with an if-else ladder, a for-loop, or a switch statement. If you want to use the switch statement, ask your teacher about its syntax.

Load Unit1\Lab18.java at size 8x8. Use maps "map1", "map2", and "map3". Compile and run.

Test Data

Look in the DOS window when your program ends to make sure you followed the map correctly. There are 24 total beepers in map one, 32 total beepers in map two, and 33 total beepers in map three.

Extension

Rewrite each method recursively. See the appendix for more details on recursion.

Sample Run `Unit1Lab18.jar`

Lab19

Yellow Brick Road

Objective

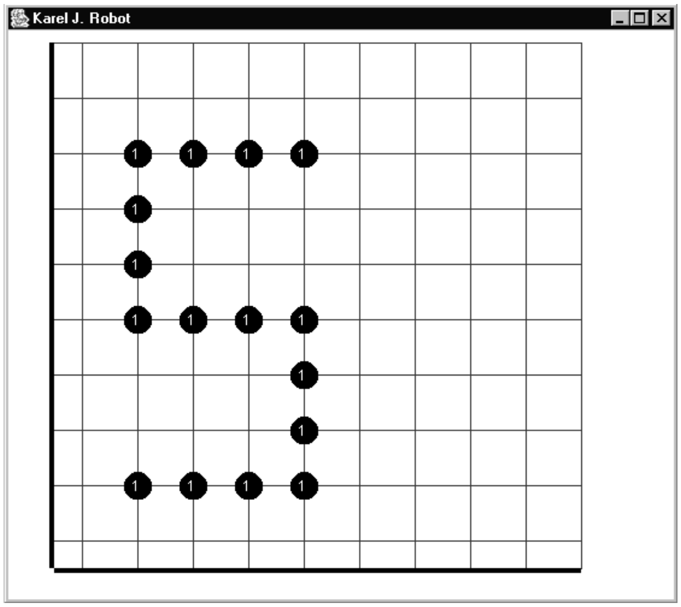
Implement an algorithm in a resource class.

Background

Follow, follow, follow, follow, follow the yellow brick road. Your job is to get your robot from (2, 2) to the end of the path of beepers, wherever that may be. Be careful! Your robot must stop when it gets to the end of the path.

Here is a good algorithm: while the robot is on the path, follow it. Nice!

Or: if the robot has found the path, then take a step. Repeat.



Specification

Create Unit1\Dorothy.java. The Dorothy API is shown at the right. What should go in the default constructor? What methods are in Dorothy? Notice that the `findPath` method returns a boolean. Use it.

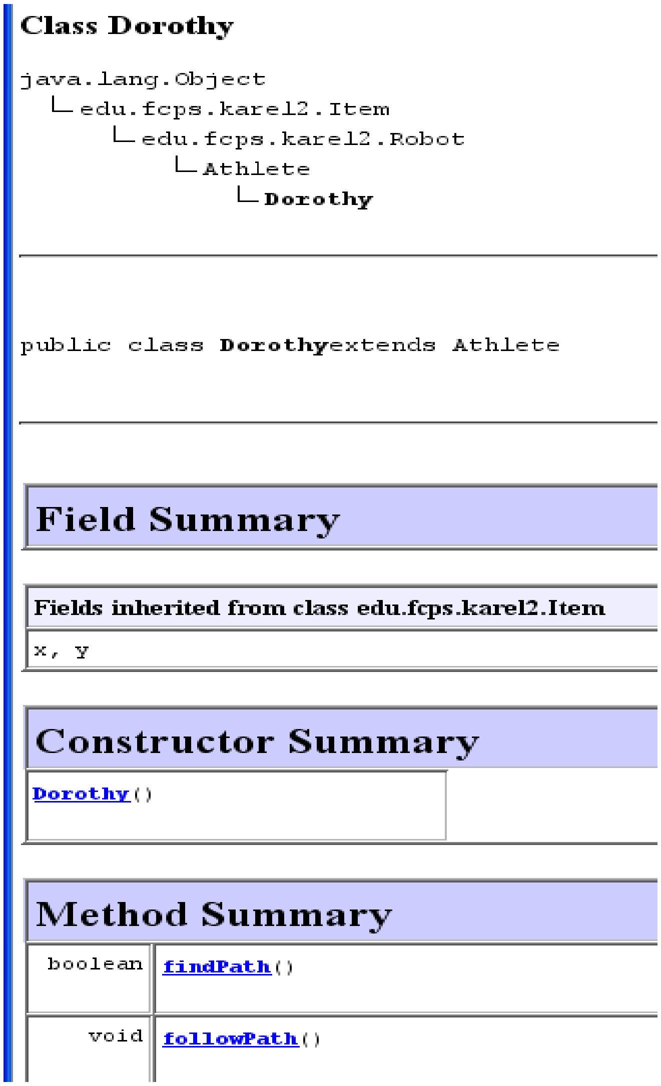
Create Unit1\Lab19.java. Instantiate a Dorothy object and send her on her way. Use “path1”, “path2”, and “path3”. The map "path3" has an extra complication for you to deal with!

Extension

Use `MapBuilder.jar` to create other valid robot-maps to test your program.

Sample Run

`Unit1Lab19.jar`



Class Dorothy

```
java.lang.Object
  └edu.fcps.karel2.Item
      └edu.fcps.karel2.Robot
          └Athlete
              └Dorothy
```

```
public class Dorothyextends Athlete
```

Field Summary

Fields inherited from class edu.fcps.karel2.Item
x, y

Constructor Summary
Dorothy()

Method Summary	
boolean	findPath()
void	followPath()

Discussion

Recursion

In Racer we defined an *iterative* method `sprint` as follows:

```
void sprint(int n)
{
  for(int k = 1; k <= n; k++)
  {
    move();
  }
}
```

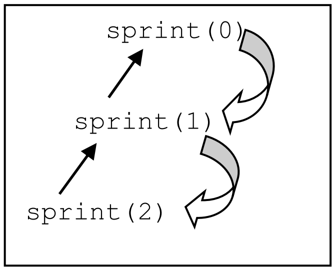
The following solutions are *recursive* because the method `sprint` calls the method `sprint`—it calls a copy of itself. The two solutions below act exactly the same, but the code on the right is often easier to read. Since the **`return`** statement forces the method to end early, the **`else`** is not required.

```
void sprint(int n)
{
   if(n <= 0)         //the base case
       return;
   else
   {
      move();
      sprint(n - 1); //recursive call
   }
}
```

```
void sprint(int n)
{
   if(n <= 0)      //the base case
      return;
   move();
   sprint(n - 1);  //recursive call
}
```

Let's trace the call `sprint(2).` It is useful to imagine the recursive calls as stacking on top of one another, as shown. Eventually, the code reaches the base case and all the calls come back out, in reverse order.

Since n =2, which is not equal to zero, the robot will move and then call `sprint(1).` In `sprint(1)` the value of n is 1, which is also not equal to zero, so the robot will move and call `sprint(0).` In `sprint(0)` the value of n is 0, which is equal to zero, so the robot would do nothing and the call to `sprint(0)` would `return` , thus ending the method. Then the call to `sprint(1)` would end, then the call to `sprint(2)` would end. In all, our robot moved twice.



Be careful with recursion, as infinite recursive calls are common (resulting in a "stack overflow" error). The following code has such an error:

```
void sprint(int n)
{
   sprint(n-1);       //ERROR:  makes the recursive call before
   if (n <= 0)        //        checking the base case!
      return;         //        Keeps calling sprint "forever"
}
```

To avoid infinite recursive calls, it is smart to think of recursion as a two-step process: 1) check the base case first to see when the recursive calling should stop, then 2) if the base case is not met, call the recursive method again with a "one-step smaller" argument so that the base case will eventually be reached.

Assignments:

Return to Racer from Lab05. Re-write the `sprint`, `put`, and `pick` methods as recursive methods.

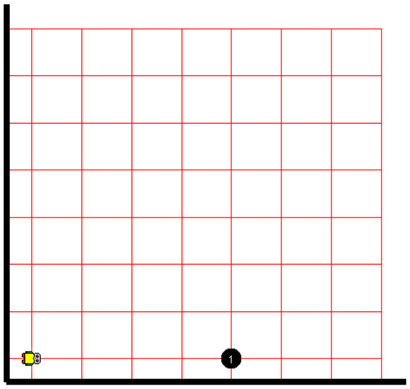
Return to Lab18's Pirate class. All three instance methods in Pirate can be written recursively.

Discussion

Stored Recursive Calls

Imagine a beeper is on the same y-coordinate as your robot, some unknown distance due east of your robot's current location. You need to retrieve the beeper, turn left, and move north the same distance and drop the beeper. The following *iterative* solution counts how many blocks we travel east during the while-loop, then uses that number to control the for-loop, which travels north:

```
int n = 0;
while (!karel.nextToABeeper())
{
  karel.move();
  n = n + 1;
}
karel.pickBeeper();
karel.turnLeft();
for(int k = 1; k <= n; k++)
   karel.move();
karel.putBeeper();
```



A more elegant solution uses a *recursive* method:

```
void recur()
{
  if(nextToABeeper())  //base case
  {
     pickBeeper();
     turnLeft();
  }
  else
  {
     move();
     recur();           //recursive call
     move();            //this command is stored; eventually it is executed
  }
}
```

As long as `nextToABeeper` is false, we move and call `recur`. Each call to recur will test `nextToABeeper`, only one block further east. At some point `nextToABeeper` will be true. The robot picks the beeper up and turns left—now facing north. This call to `recur`, the one that found the beeper, will end. When it ends, the previous call to `recur`, the one that made the call to the one that found the beeper, will continue where it left off, i.e, there is a second move that has to execute.

This move will now be made north. When each call to recur ends, the one before it continues, moves one block farther north, and ends itself. Every move before the recursive call goes east and every move afterwards goes north. Since there is the same number of moves before and after the recursive call, the robot travels the same distance north as it had traveled east.

If this doesn't make sense, imagine that the beeper was originally located only one block in front of karel and trace through the execution of the method. What if the beeper was originally located at the exact same intersection as karel?

Use the technique of stored recursive commands for the next lab.

LabXX

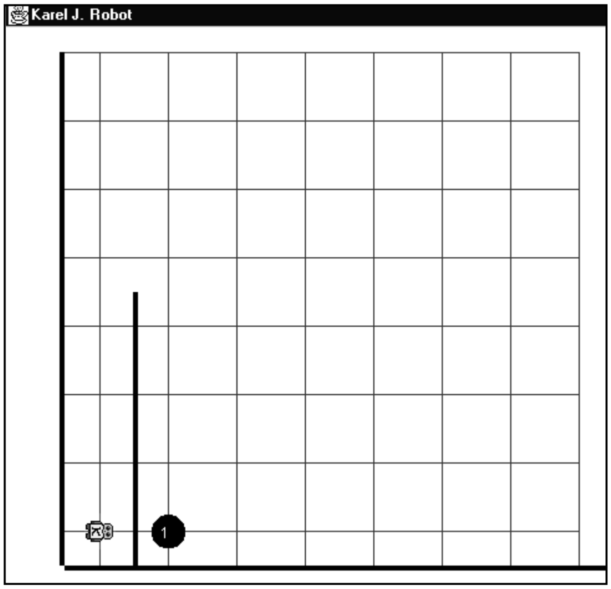
Seeking the Beeper

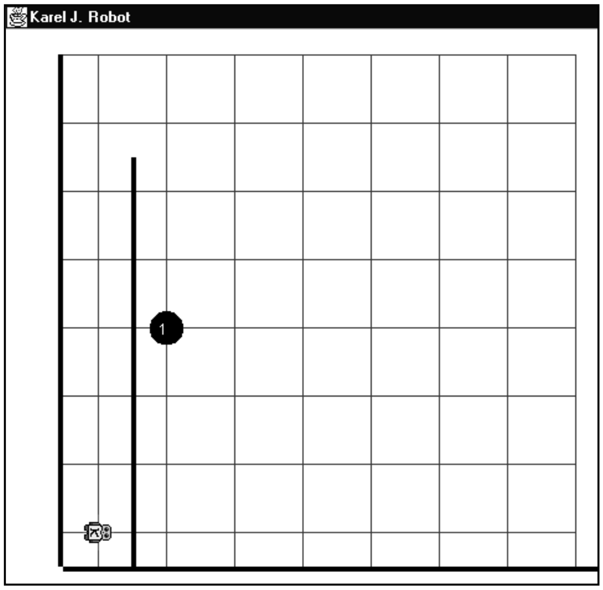
Objective

To implement stored recursion.

Background

Your robot begins at (1, 1) facing east. There is a north-south wall of unknown length blocking your robot's path. Directly on the other side of that wall, on the 2nd x-coordinate, there is a beeper. The beeper is against the wall but you do not know what y-coordinate that beeper is located on.   You must retrieve the beeper, bring it back to (1, 1), and put it down.





Consult the Unit 1 API for information regarding the Seeker class.

Specification

Filename Unit1\Seeker.java. Implement the Seeker class. Compile.

Load Unit1\LabXX.java. Compile and run. You must test the LabXX program (really you are testing your `fetchBeeper` method) for many randomly generated maps. Run the program a few times to make sure your method always works.

Sample Run

`Unit1LabXX.jar`

Exercises

Recursive Solutions

1) Imagine your robot is on 1st y-coordinate. You do not know what x-coordinate the robot is on and you do not know what cardinal-direction the robot is facing. There is a beeper at (1, 1). Use loops to get the beeper and bring it back to your starting location. Put the beeper down at the end.

2) Complete the definition of a recursive method `getBeeper`. This method is designed to be part of a solution to Question #1. The robot should search its path to the front until it finds a beeper, then go back to its starting place.

```
public void getBeeper()
{
  if(                 )
  {



  }
  else
  {



  }
}
```

3) Write the solution to Question #1 using your recursive method from Question #2. Assume that `getBeeper` does what you intended, regardless of the correctness of your answer to Question #2.

Exercises

Recursive Solutions (continued)

4) Imagine the problem from Question #1 is modified so that you do not even know what y-coordinate your robot is on—it could be at any intersection! Your solution makes use of a recursive function `findBeeper` as follows:

```
karel.faceWest();
karel.findBeeper();
karel.putBeeper();
```

Complete the definition of the recursive method `findBeeper`. You may use getBeeper.

```
public void findBeeper()
{
  if(                  )
  {



  }
  else
  {



  }
}
```

5) Your maze escaping solution can be simplified to the following:

```
karel.followWallsRight();
karel.pickBeeper();
```

Complete the definition of the recursive method `followWallsRight`.

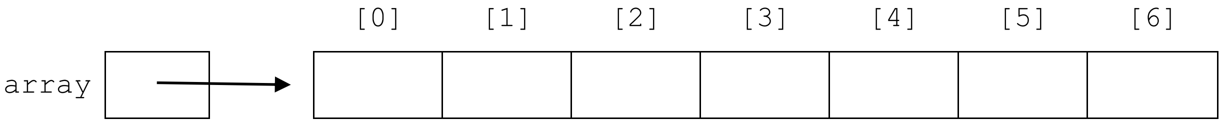
```
public void followWallsRight()
{









 }
```

Discussion

Arrays

Multiple objects can be stored with a single identifier in an “array.” An array is a linear data structure of fixed length. For instance:

```
Robot[] array = new Robot[7];
```



This declaration creates an array of seven Robot references. Each of these references is initialized to `null`. To put robot objects into each cell use a for-loop like this:

```
for(int index = 0; index <7; index++)
{
  array[index] = new Robot(index + 1, 1, Display.EAST, 0);
}
```

Arrays are zero-indexed, which means the boxes inside an array are numbered starting from zero. Don’t start counting at one with a variable that is meant to indicate a certain cell in an array. Notice that index is part of an expression that is passed as an argument to the Robot constructor. The loop control variable can be used for more than just repetition and the access of array cells.

The basic idea is that the objects stored in an array can be manipulated with a for-loop:

```
for(int index = 0; index < 7; index++)
{
  array[index].putBeeper();
}
```

Imagine you want all seven of these robots to move forward five blocks.

```
for(int index = 0; index < 7; index++)
{
  for(int count = 1; count <= 5; count++)
  {
    array[index].move();
  }
}
```

This is a very common technique—using a for-loop to access the cells of an array. Note the importance of the int-type variable index; as it changes value, the robot we are dealing with also changes, because each robot is identified by a cell-number in the array. Traversing an array with a for-loop works no matter how many objects are being stored: a hundred, a thousand, etc. Also note that the name of an array can be any valid identifier (not just `array`).

As shown, these robots will not all move at the same time. Rather, they will take turns as the outside for-loop repeats. In order to have parallel processes execute at the same time you must use threads.

LabYY

Seeking the Beeper, Part II

Objective

Solve a problem.

Background

Set the robot map to some large, random size, call it NxN. For instance:

```
   final int N = (int)(Math.random()*50+25);
   Display.setSize(N, N);
```

The keyword `final` indicates that the value of N will not change. As a convention, final variables are written in ALLCAPS, like Display.EAST.

Place a beeper randomly in the world.

```
   WorldBackend.getCurrent().putBeepers(
                 (int)(Math.random()*N+1), (int)(Math.random()*N+1), 1);
```

Your job is to get the beeper.

There are a number of different ways to approach this problem. You could have a robot start at (1, 1) and search the entire map itself. You could have a bunch of robots, maybe even an array, start along 1st and march east. You could use recursion or just loops.

Specification

Create Unit1\LabYY.java. Use the commands shown above to place a beeper randomly in a large, but randomly sized, robot map. Somehow, get the beeper.

Test Data

However you decide to approach this problem, if the beeper gets picked up, then your program works.

Glossary

Abstract: A Java keyword indicating that a method does not have a definition and must be implemented either by an extending class (for an abstract class) or by an implementing class (for an interface).

Actual Argument: A data item specified in a method call. An argument can be a literal value, a variable, or an expression.

Anonymous: An object or class that is used but not named. For instance, new Robot().move();.

Body: The commands that actually get executed when a method is called.

Class: In Java, a type that defines the implementation of a particular kind of object. A class definition defines both class and instance methods, as well as specifying the interfaces the class implements and the immediate superclass of the class. If the superclass is not explicitly specified, the superclass will implicity be java.lang.Object.

Class methods: Methods that are invoked without reference to a particular object. Class methods are tagged with the keyword `static` and do not have an implicit argument `this`.

Formal Argument (parameter): The variable representation of an argument in the definition of a method.

Header: The first line of a method definition.

Inherited Method: A method defined in a superclass that is available to a subclass (keyword extends).

Instance methods: Methods that are invoked with reference to a particular object (an instance). Instance methods affect a particular instance of the class, which is passed as the implicit argument `this`.

Isa: A phrase to express the relationships of classes in a hierarchy.

Object: The building block of an object-oriented program. Each object consists of data and functionality.

Package: A group of classes in the same folder. Packages are declared with the keyword `package`.

Polymorphism: If a method defined in a superclass is overridden in a subclass, then the subclass method is invoked at runtime. Used with abstract classes and interfaces we can write code generically to be used later without modification for various other classes.

Reference to an Object: A symbolic pointer to an object.

Root: In a hierarchy, the one class from which all other classes are descended. The root has nothing above it in the hierarchy.

Signature: A method’s name and argument list, but not the method’s return type.

Subclass: A class that is derived from a particular class.

Superclass: A class from which a particular class is derived.