FCPS Java Packets

Unit One – JKarel Programming

November 2008

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.

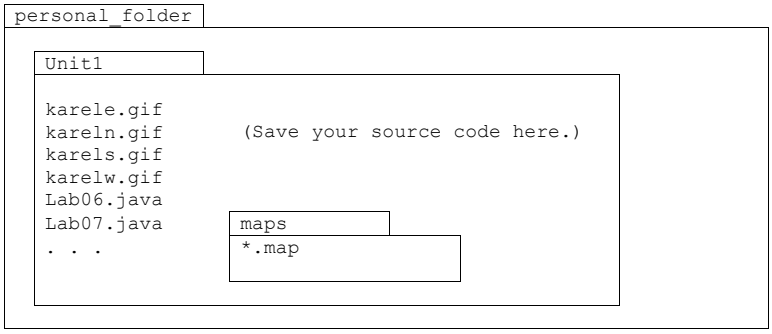
Java Instruction Plan—Unit One

Discussion

File Management

Java requires you to organize your files carefully. You must do your work in your personal folder, which might be on a network drive, a hard drive, or a removable drive. The JKarel `Unit1` folder, with all its files and folders, should be copied from a CD or a network drive into `\personal_folder\.` Ask your teacher how to do this on your school's system.

已註解 [N1]: 在電機系的 510 實驗室，我們把所有 Unit 的程式放在 Eclipse 的 workspace 目錄底下：C:\Documents and Settings\Administrator\workspace\Unit1\src

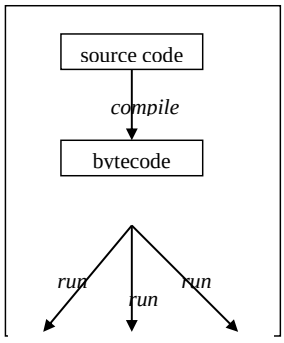


Source code is the Java program you write. Save your source code directly to your Unit1 folder; do not create sub-folders because your programs and the gif files must be in the same folder.

已註解 [N2]: 因為我們用 Eclipse 的環境，所以我們是放在 Unit 底下一個 src 的目錄。在 Unit2 之後，當教到 package 的概念時，我們就會另外建 sub-folder。

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 process was adopted in order better 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: you 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 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 sending Robot commands. 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: The class name and file name must match exactly.

Line 6: Left brace begins class Lab00.

Line 7: All applications have a *main* method. The statements execute in order.

Line 8: Left brace begins main.

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

Line 12: The compiler usually ignores whitespace.

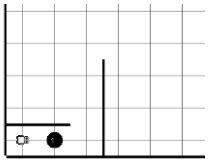
Line 13: Our *object* named karel is an *instance* of the Robot class. We have *instantiated* a Robot object.

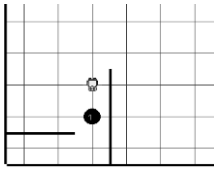
Line 15: move() is an *instance method* in every Robot *object*.

Lines 24 and 25: Right brace ends main and another right 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 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

Start: 

End: 

已註解 [N3]: 我們是使用 Eclipse 的 IDE，功能比這個 iGRASP 強，可同時編寫、編輯，與執行程式。

已註解 [N4]: Java 允許你寫的程式借用別人已經寫好的程式 (更明確的講，別人寫好的 class)。這個 import 告訴 compiler 當它看到不認識的 class 可從這些 package 去找。
在 eclipse 裏，需要對相對應的 Java Project 也就是 Unit1 先建好 Build Path，並把兩個 *.jar 檔引入。在 Package Explorer 的專案裏，你會看到一個 Referenced Libraries 的目錄，底下就有這兩個 JAR 檔。

已註解 [N5]: Display 這個 class 有那些 method 可參考 Unit1 的 API 文件：
http://academics.tjhsst.edu/compsci/CSweb/Unit1/api/index.html
中 class summary 就可看得到 Display

已註解 [N6]: 類別 Class 是描述有共同特徵的規格；其中的一個例子稱之前物件(object 或是 instance)。定義好的類別需要用 new 來「例舉實作」出一個物件。

已註解 [N7]: Robot 這個 class 有那些 method 可參考 Unit1 的 API 文件：
http://academics.tjhsst.edu/compsci/CSweb/Unit1/api/index.html
中 class summary 就可看得到 Robot

Exercises
Lab00

A Robot object's default initial conditions are to start at (1, 1) facing east with zero beepers. If you don't want the default settings, you may specify the x-coordinate, y-coordinate, direction, and number of beepers. For instance:

```
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?

已註解 [N8]: 這個 default 指得是要是沒有特別講，或異議，則就這麼作。台灣翻成「預設」；而大陸翻成「默認」，我覺得翻得其神韻。

已註解 [N9]: 給定一個 class，產生這類別裏的一個物件的方法稱之為 constructor (建構元)。你可以在 API 裏看到有幾種 **Constructor Summary** 的方法。

已註解 [N10]: 在 eclipse 裏，將 cursor 放在 Robot 上 (Hover)，然後按 F3 作 Open Declaration，就會帶出 Robot Class 的 java 定義，就可看到要是沒有 WorldBackend 時，就會帶出一個 default 的。

Discussion
Java Applications

Java applications, or programs, have a standard structure. You have already seen that the class name must match the filename. An application must also have a main method. The main method is the entry point for your program. When the program runs, the machine executes `public static void main(String[] args)` and the commands in main execute in order. Java's structure also requires matching parentheses for every block of code.

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

  }
}
```

The code above **must** be saved as LabXX.java. The compiler first checks the .java code for *syntax errors*. Then it imports all the necessary files, which are usually in different places. The compiler first searches the current folder (which is our Unit1), then the external import locations, then the automatic import locations. All this source code is put together and translated into bytecode, which is saved in Unit1 as a .class file.

已註解 [u11]: Syntax 語法；semantics 語義

Compiling is accomplished by running a separate program named `javac`. This program was written by Sun and is part of the j2sdk installation. The syntax from a command line would be **`javac LabXX.java`**. (In jGrasp, we just click a button.) The name of the bytecode will LabXX.class. The bytecode is a comparatively small file, so that it can travel quickly over the internet.

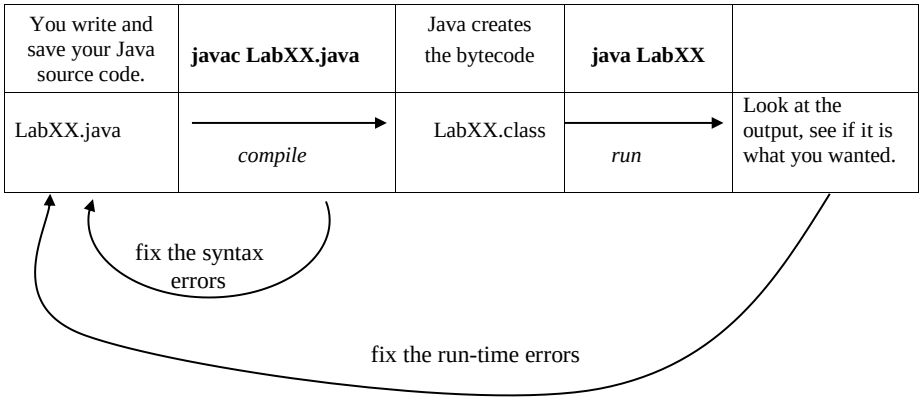
已註解 [N12]: 我們使用的是 JDK 7.15 版

To run the bytecode, maybe at a different computer, Sun uses another program named `java`. The syntax from a command line would be **`java LabXX`**, because the extension is assumed to be .class. (In jGrasp, we just click a button.) Then the JVM automatically calls your LabXX's main method. If you did not define main your program will give an error, "`not found: main`". If you did define main but the header does not say precisely:

```
public static void main(String[] args)
```

已註解 [N13]: 我們使用 eclipse 的 IDE.

your program will give an error, "not found: main". The entire process is illustrated below.

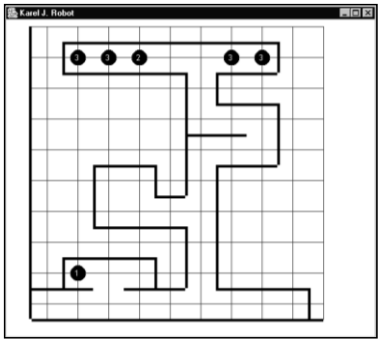


Lab01
Students and Books

Objective

Program structure. Display commands. Robot commands.

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. Method names, objects, and other variables always begin with a lowercase letter. One exception is constants, like EAST, which are written in ALL CAPS.

The word *instantiate* means to create an instance of a class—an object. Java instantiates an object with the `new` keyword.

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.

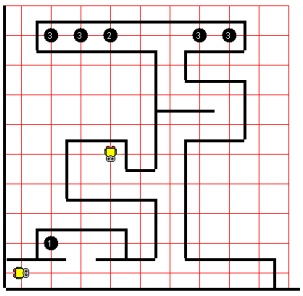
已註解 [N14]: 在 eclipse 的環境裏，在一個物件之後，只要一打上 . 之後，螢幕上馬上會跑出一個可能的 method 與 field 來，當作提示以及讓你選。除此之外，你也可以在類別上按 F3 拉出這個 class 的原始碼後，由右邊的 outline 看得到，這個類別一共定義了多少 method。或是，如果找得到這個 class 的 API 頁的話，也可以看得到更清楚的定義。

Specification

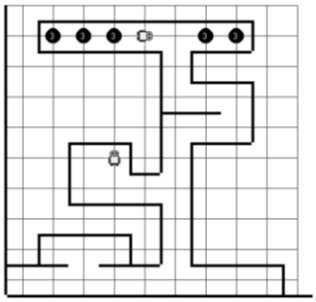
Create Unit1\Lab01.java with the "school" map at size 10x10. Instantiate two Robot objects, one named lisa using the default constructor and the other named pete starting at (4, 5) facing south with zero beepers. Have lisa pick up the "book" from the math office and bring it to pete. Have pete take the 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

start



finish



Exercises
Lab01

a. An API is a document that lists all relevant information about each class. Consult the Unit1 API found at http://academics.tjhsst.edu/compsci/CSweb/Unit1/webdemos/index.html
b. *Fields* store an object's private information about its state. List all the fields of the Robot class:

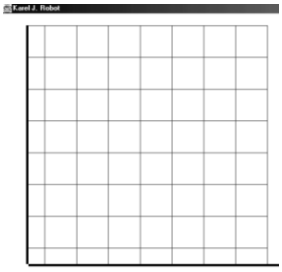
c. What methods in Robot have we used so far?

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

2) Circle the valid identifiers.

```
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 models the story from Question #6 using robots and books. Think of other classes that could have been used instead to solve the same problem.

10) What should be the attributes (private data) of each class?

10) What should be the behaviors of each class?

已註解 [N15]: 針對每個類別，對應它個自資料的部份屬於名詞，叫作 field；對應它的能耐的部份 常用動詞來描述較暢意，叫作 method。

已註解 [N16]: 應該指得是 beepers。這個應該是 persons 與 books。

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.

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, e.g. new Robot (4, 5, Display.SOUTH);			
forget to import edu.fcps.karel2.Display;			

Finally, comment out the lisa Robot and all lisa's commands. What happens to pete?

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."

已註解 [N17]: 請參考 One-17 中的 Exercise Part1: Lab04 裏，有各種不同型式的 Errors。

已註解 [N18]: ,●我們使用的 Eclipse 裏，可以在 Preferences 中設定 line numer。另外也可以用 ctrl+shift+F 鍵把程式對齊。

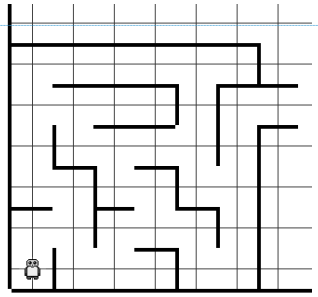
已註解 [N19]: 關於 lexical (拼字) error 以及 syntax (造句) error，如果使用 eclipse 的發展環境，在你撰寫程式的過程式，eclipse 就會持續幫你檢查，並在出錯時，用色提示。

已註解 [N20]: 除了你讓 robot 往牆走之外，你讓有上沒有 beeper 的 robot 放下 beeper，也是執行過程式才會發生的。

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.java, 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.

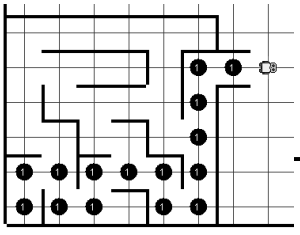
Athlete is a general *resource class* that will be useful in a wide range of applications. The *application* called Lab02, which has the main() method, will use the Athlete class. We sometimes say that Lab02 *hasa* athlete.

Specification

Load Unit1\Athlete.java. Implement the methods turnRight and turnAround, then compile Athlete.java to create the file Athlete.class. When you create an athlete object in your Lab02 application, Java will look for the Athlete.class file in order to understand what athlete objects do. 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



已註解 [N21]: 不同的類別之間有親屬關係：父輩與子輩類別。子輩會繼承所有父輩有的屬性以及會的方法。子輩還可能改寫 (override) 父輩會的方法；或是實現父輩無法實現的抽象夢想 (interface)。同一個父輩可能會有好幾個子輩，各有各的改良。之後，會講到「多型」polymorphism。

已註解 [N22]: 這個關鍵字 extends 非常重要，告訴 compiler 目前宣告的這個 Athlete 是 Robot 的子輩。一旦父子輩的類別關係建立了，他們之間的欄位與方法就有關連。

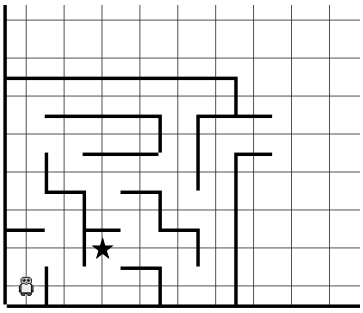
已註解 [N23]: Application 裏一定有一個 public static void main{String[], arg}並在程式中呼叫到 resource class。

已註解 [N24]: 當作 resource class，它無法自己被執行

Exercises
Lab02

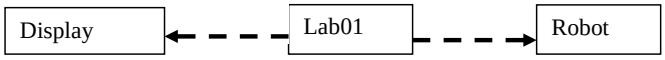
Examine the robot-map shown. Assume the Athlete's name is maria.

1) Write the commands to put one beeper at the indicated location.

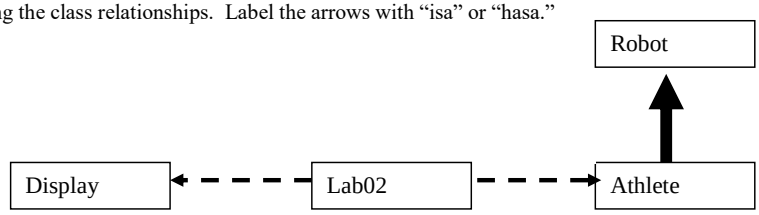


2) Use a different path from your answer from Question #1. Write the commands to put one beeper at the indicated location

3) Lab01 used three classes, Lab01, Display, and Robot. Label the boxes (it's called a UML diagram), showing the class relationships. Label the arrows with "hasa."



4) Lab02 involved four classes, Lab02, Display, Robot, and Athlete. Label the boxes in the UML diagram, showing the class relationships. Label the arrows with "isa" or "hasa."



已註解 [N25]: Unified Modeling Language 是一種用來塑模複雜軟體系統的工具

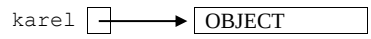
Discussion
Resources vs. Drivers

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, instantiates an object, and assigns (the "=" sign) the reference to point to the object. You should have this picture in mind:

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

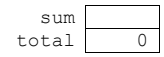


Two or more references can point to a single object. You will see this in later labs.

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.

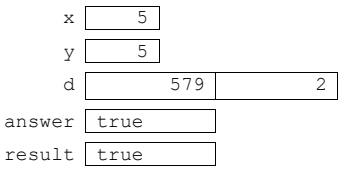
Primitive data types are not accessed by reference. Instead, the data is stored directly in memory spaces called *variables*. A *type declaration* assigns a name to a memory space, and often assigns a value, e.g.:

```
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 constructor method `public Athlete(int x, int y, int dir, int beep)` creates four memory spaces for integers, but does not assign values. The values are assigned when the actual values are *passed* to the constructor method by a call to the constructor such as:

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

已註解 [N26]: 主從式架構：供人使用的是 server (resource)，主動去用別人的是 client (application)。
當作 server 的是讓人使役，就像 Robot 一樣，會有 constructor 讓 client 用 new 來製造出一個 object，然後用來使喚。而當作 client 的一定要有一個 public static main 的 method 來啟動整個程序的執行。
※在 client 的程式中也可以另外寫 main 之外的 method。

已註解 [N27]: 計算機裏的資料以及指令長得都像是由 0101 的位元組合而成。他們都存在電腦記憶體中，你可以想成是有頁碼(地址)編號的一塊一塊記憶體。對於 primitive type 的資料而言，所需要幾頁的記憶體是可預期的，但是如果要記下一個物件或是類別的資料和指令內容，不同類別所需的總記憶體可能很不一樣，這時候，就另外盡情挖一塊來記，然後記住這大塊的頁碼，我只要記住這個索引 (reference) 的頁碼，之後就可以找得到這整大塊的內容。

已註解 [N28]: 密封 (細節)：capsule 豆莢。

已註解 [N29]: 物件在記憶體中的大小不好拿捏；Java 讓實際的物件按其大小所需在，佔某塊的某記憶體區塊來記錄，而這塊記憶體的起點位置稱之為 Reference；Reference 可想成是實際物你擺放位置的地址。

已註解 [N30]: 參考 ch4 第 18 頁的「物件的記憶體圖」。

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.

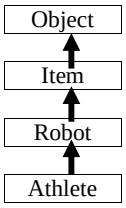
```
1   //Name_____________________________ Date_____________
2   import edu.fcps.karel2.Robot;
3   import edu.fcps.karel2.Display;
4    public class Athlete extends Robot
5   {
6       public Athlete()
7       {
8          super(1, 1, Display.NORTH, Display.INFINITY);
9       }
10      public Athlete(int 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 isn't flexible. 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 more flexible, so that we could specify an Athlete at different positions, directions, and numbers of beepers. We can do so by writing a *constructor with arguments*.

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 hard to call, but is flexible because it allows the programmer to instantiate an athlete in any state.

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 fields and 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, "constructor not found."

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.

已註解 [u31]: 建構子，就是用來按類別「起始」一個「物件」的方法。

已註解 [u32]: Athlete 的 constructor 裏的變數與其父輩 Robot (super) 所呼叫的變數並不需要一樣。但只要參數對應的關係有講清楚就不會亂了。

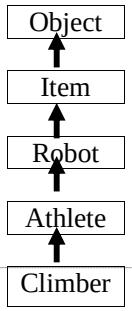
已註解 [u33]: 用到 constructor 的時候就是要 new Class()；這與要指使一個 object 作 method 不一樣。

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. A climber climbs mountains and finds treasure. Climbers always start at the bottom facing north with one beeper, but we want climbers to be able to begin at any given x-position. In order to specify the starting x-position, the Climber class will define a 1-arg constructor as well as a default constructor. How do these two constructors work?

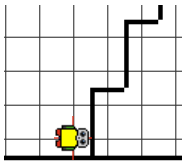
```
1  public class Climber extends Athlete
2  {
3    public Climber()
4    {
5      super();
6    }
7    public Climber(int x)
8    {
9      super(x, 1, Display.NORTH, 1);
10   }
11   public void climbUpRight()
12   {
13     //pseudocode: tL, m, m, tR, m
14   }
     //You must define three other instance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



已註解 [u34]: 這個 climber 在向右上爬之前，我們先假設他的頭是面向東的，所要要先 tL；然後，爬完之後，仍然是面向東的。
同樣的，如果是向西的，也是一開始要面向西的，所要相對應的指令是 tR, m, m, tL, m。

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 one step, going to the right.



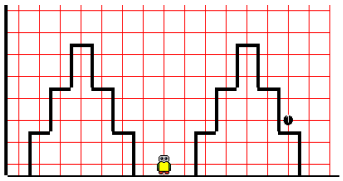
Specification
Consult the Unit 1 API to find out what methods the Climber class needs to know.

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

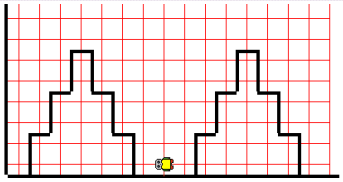
Create Unit1\Lab03.java with "mountain" map at size 16 x 16. Instantiate two climbers starting at x-coordinate 8. Put down the beeper, which is the "base camp." Staying together as much as possible, bring the treasure back to "base camp."

已註解 [u35]: 這個意思是 instantiate 兩個 climbers，都從 y = 8 開始，先放下一個 beeper 當起點的標記；然後這兩個 climbers 一起去把山後的 beeper 拿回來；過程中儘量靠近，但也不希望一個爬上，同時一個人跟他對沖在爬下。最後，兩個人都要把一個當作寶藏的山後 beeper 拿回基地。

Sample Run:
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

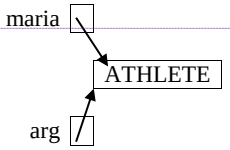
Passing Arguments

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.



已註解 [u36]: Formal argument 想成是數學函數的「變數」，而 actual argument 就想成是我們指定這變數為某個特定值。

已註解 [u37]: 參數 argument 不一定是單純的 primitive data type，其更重要的是 object 也可當 argument，只是在 pass 的時候是以其相對應的 reference 作替代。

已註解 [u38]: 對於不同的物件，用來代表這物件所需的記憶區塊大小不一定會一樣。當在提到一個物件時，不可能把整塊物件搬給呼叫的人；一個較好的方法就是，把「地址」(reference) 給呼叫的人就可以了。呼叫的人有了這個 reference 就知道怎麼去找得到整個物件了。

已註解 [u39]: 本來 arg 只知道它是指向一個為 Athlete 的物件；這在當 maria 被 passed 進來時，maria 所內含的 reference 的值被丟到 arg 裏；透過這個地址，就找到原來 maria 所指的那個 Athelete 的物件。

Lab04
Take the Field

Objective

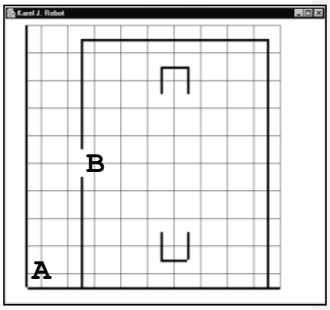
Class methods vs. Instance methods

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.
}
```



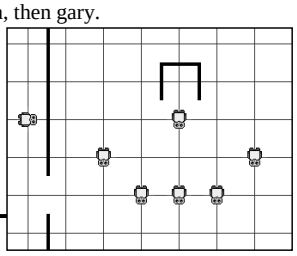
We can use this class method with any athlete object by passing the object. For instance, if we have athletes `maria` and `gary` defined in main, we can tell each to take the field with the commands:

```
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.



已註解 [u40]: 我們到目前為止所強調的類別：Robot 是屬於可以製造出好幾份重複內容物件的類別，這類別裏所定義的 method 是給物件用的 Instance method 。另外還有一種類別，宣告時前面會加一個 static，像 Display 這個類別，使用它時，不必 new 一個相對應的物件，直接用 Class 名，然後加上「.」再加上 method()；像這種不用 new 物件，而可以直接用的 method 稱之為 class method。

已註解 [u41]: 之前的 move()，turnLeft()等的 method 是在 instantiaed 一個物件之後，用來 invoke 這物件用的。現在講的這個 class method 是給當下正在執行的這個 application 的 class 所用，意即不用 instantiate 任何物件即可用的。

已註解 [u42]: Static method 在 header 有一個修飾字 static；通常是放在最前面。

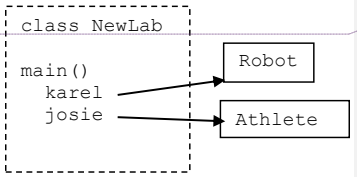
已註解 [u43]: 在 Lab04 裏一共有兩個 static class，一個是 main，另外一個是 takeTheField。其實，這個 takeTheField 也可以另外寫一個 class，比如說 Lab04A，然後在這裏用 Lab04A.takeTheField 呼叫。當然，目前這個方法放在一起是最好的。

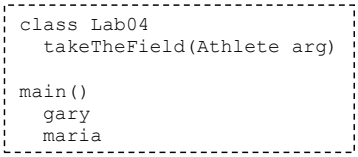
已註解 [u44]: 我們所講 class method 的意思，指的是對於目前呼叫的這整個 class 都適用。這是透過在這個 class 中 static 的宣告。

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);

已註解 [u45]: 這個 Exercise 剛好給大家試不同情況的 error。大家可以在 eclipse 上直接試指令，就可以知道是怎樣的錯法。

已註解 [u46]: 這是 class method，而不是 instance method，所以呼叫的方式不對。

已註解 [u47]: 這個 class method 是 Lab04 定義給自己的 class 用的，而不是給別人用的。

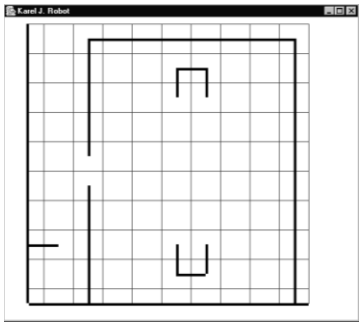
已註解 [u48]: ※karel 是 NewLab 裏宣告的，而 takeTheField 只是給 Lab04 用。如果要跨越 class 去使用別的 class 的 static method，一定要標出那個 class 的名稱之後再「.」才可使用。而這題錯的原因是，karel 不是 Athlete，不符合 formal arg 的規定。

已註解 [u49]: Arg 的數目不對。

已註解 [u50]: 呼叫時一定要傳一個 actual argument。

Exercises, Part 2
Lab04

Examine the robot-map shown. Note 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.



1. Describe how you would have to change your Lab04 application so that it would work in this map.

2. Write down the complete code of all methods that you changed, starting with the *header* for that method and including the entire method *body*.

```
public static void takeTheField(Athlete arg)
  {
      arg.move();
      arg.turnRight();
      arg.move();
      arg.turnLeft();
      arg.move();
      arg.move();
      arg.move();
      arg.turnRight();
      arg.move();
  }
```

已註解 [N51]: Static method 在 header 有一個修飾字 static；通常是放在最前面。

3. *Modular design* is the practice of breaking larger problems into smaller, more manageable pieces. Explain how modular design helped us maintain our Lab04 program when the robot-map changed.

Look at the Java API for the Math class.

已註解 [N52]: 先到 http://download.oracle.com/javase/6/docs/api/ 在左下欄位裏，找 Math 的這個 class，再看裏面的 method

4. Which Math method doesn't take any arguments? _______

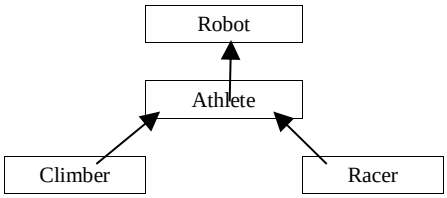
5. Name a Math method that takes two arguments. _____

6. 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++)
{
}
```

已註解 [N53]: 到目前為止，每行的程式一定會被執行一次。其實，我們可以對一段程式碼，按「需要」，讓他執行固定的 n 次，或是看狀況決定要不要執行。

已註解 [N54]: 所謂確定性的重覆指的是執行的次數確定；而不確定性的重覆則是「看情況」：根據程式執行當時的「狀況」

已註解 [u55]: 之前曾經提到 syntax (語法)，這裏的 for-loop 有明確的寫法，如果寫錯的話，就會犯了 syntax error。

已註解 [u56]: ●教大家使用 debug 的功能：我們可以控制程式一步一步的執行 (F6)，或是一直衝到某一行，然後突然停在那裏。當有呼叫其他的 method 時，也能夠「鑽進去」(F5) 一行一行的執行，或是看夠了就爬出來 (F7)。

已註解 [N57]: 這都是假設討論中，「要作的工作」只是一行指令。如果超過一行的話，就用 { 與 } 限定出來。

已註解 [N58]: 我們從小學的數學中：加減乘除，大部份都是涉及兩個數的運算；但這個運算只涉及一個變數：自己給自己加值。

已註解 [N59]: 造句的語法：要背起來！否則 compiler 會抱怨。

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 three.

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. What is the syntax for sending pete to runTheRace()?

```
 3  import edu.fcps.karel2.Display;
 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 sprint(int n)
17     {
18       for(int k=1; k <= n; k++)
19          move();
20     }
21    public void put(int n)
22     {
23       ____________________

24       ____________________
25     }
26    public void pick(int n)
27     {
28       ____________________

29       ____________________
30     }
```

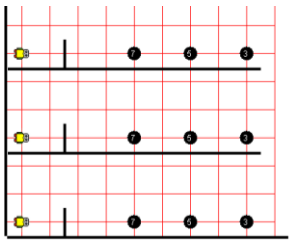
Specification

Consult the JKarel API for the methods that should be in the Racer class. Create Unit1\Racer.java. Implement the Racer class, then compile.

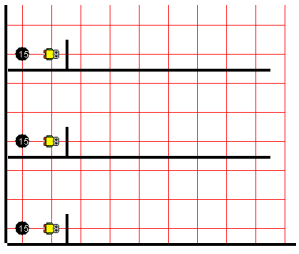
Create Unit1\Lab05.java with "shuttle" map at size 10x10. Create three racers to run the shuttle run. It's not a race. Let one racer finish the task, then send each of the others. Use a class method. Each pile of beepers must be brought individually back to the racer's starting point. At the end each racer must step away from the pile to confirm that all 15 beepers have been retrieved.

Sample Run

Start:



End:



已註解 [N60]: 一次可以跑好幾步的方法，這裏稱之為 sprint (衝次)

已註解 [N61]: 這是同一個主程式中所宣告的三個 robot 都會作的事，所以以 class method 來達成為最好。

已註解 [N62]: 其中會有一個 shuttle (int x, int n) 的 method，API 沒講功能，其應為：將橫軸上 x 步外的 n 個 beepers 搬運回來。

已註解 [N63]: 之後，在 page One-42 的 Lab14 裏，我們會教到那這三個 racer 真的比賽，誰跑得快。(用 thread 裏 run() 的作法)

已註解 [N64]: 這個 Lab 是三個 racer 依序完成，之後我們會看到三個同時進行的情況。

Discussion
Loops 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.

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

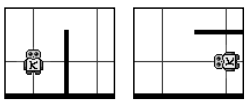
Warning

There is no such thing as an if-loop. An if-statement checks its condition only once.

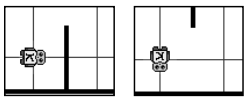
Java distinguishes between *void methods* and *return methods*. Void methods take action and change the situation, such as `public void turnRight()`. In contrast, return methods provide information about a robot's situation. Since there are different kinds of information, return methods come in several 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."

已註解 [N65]: 只要條件滿足，持續執行。

已註解 [N66]: 雖然講 next to，但指的是同在棋盤的 (x，y) 格上。

已註解 [u67]: 有一個指令叫作：do-while，語法為：
do { statements } while {logic condition}
執行時是 {statements}先執行一次，然後查核 {logic condition} 是否成立，成立的話，上面的 {statements} 再執行一次，然後再檢查；否則的話，就不再回頭了。
這個 do-while 的特性是 {statements} 至少執行一次。

已註解 [N68]: IF 與 WHILE 最大的差別在於，if 只檢查條件一次，所以最多就執行一次。

已註解 [u69]: 對於 method，我們要求它執行：某些動作、也以可要求執行的物件或是類別在執行完時，回報狀態。如果不需要的話，就把這個 method 前加 void(就是「沒有」)，但如果需要回報的話，就要先講好要回報的資料型態。

已註解 [N70]: 在看 API 時，為什麼 method 的宣告會有資料型態或是就 void 呢？

已註解 [u71]: 布林指的是：yes 或是 no，二者取一。最常搭配使用的就是 if (boolean) 的決策判斷式，然後可以分歧點：yes 的話作什麼，no 的話又作什麼。

已註解 [N72]: karel.frontIsBlocked() = ! karel.frontIsClear(). 這個驚嘆號「！」是一個改變布林值的運算符號：！yes = no；同樣的！no = yes。

Exercises

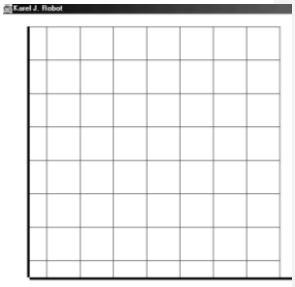
Boolean methods

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. karel.facingWest();	2. Write the commands to make an Athlete named `ann` to put down all her beepers. ann.hasBeepers();
3. Write the commands for Climber `c` to pick up a pile of beepers. c.nextToABeeper();	4. Write the commands for Racer `ray` to stop moving when it is next to (on top of) another racer. ray.nextToARobot();

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`

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

true && true →	true || true →
true && false →	true || false →
false && true →	false || true →
false && false →	false || false →

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

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

已註解 [N73]:「聰明又用功」的相反不是「不聰明又不用功」呦~
是「不聰明又不用功」

已註解 [N74]:「長得帥或是很有學養」的相反不是「長得不帥或是沒有養」呦~
而是「長得又不帥而且又沒學養」。

Lab06
A Half-Dozen Tasks

Objective

while-loops and if-statements

Background

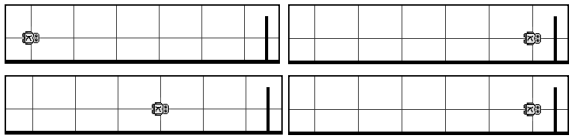
This lab instantiates 6 Robots to perform 6 different tasks. For convenience, we code each task as a class method. Each task uses one or more indefinite loops. Some tasks also use if-statements.

Row	Task
1	Go to the end of the row of beepers.
2	Go to the beeper.
3	Go to the wall.
4	Go to the wall, picking up all the beepers (max one beeper per pile).
5	Go to the wall, picking up all the beepers.
6	Go to the end of the row of beepers; there is one gap somewhere.

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

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

Before / After



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, your program will not contain the command:

```
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

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.
2. Create a valid robot-map for this lab using MapBuilder.jar and test your program with that map.

已註解 [N75]: 在寫程式時，發生「無窮迴圈」不是好事；如果電腦笨笨的，你的程式就會一直霸佔 cpu，浪費資源。

已註解 [N76]: 這是你與自己寫出來的程式作互動的方式。我們在 Unit2 會教到更花俏的方法。

已註解 [u77]: 在 Unit1/maps 的目錄底下就可以看得到。記得 SAVE 時，檔案名稱要有「.map」的副檔名，這是因為在 `Display.openWorld("maps/" + filename + ".map");` 的這個指令中，已經規定完整的檔案名稱有副檔名。照道理，你建完地圖之後，就存在 maps 的目錄底下，但在 eclipse 的 Package explorer 並不會馬上看得到，這是 eclipse 的 bug，有時你用 F5 (Refresh) 馬上就可以看得到。你只要確定在 maps 底下，上面的指令就一定找得到的。

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.
{

}

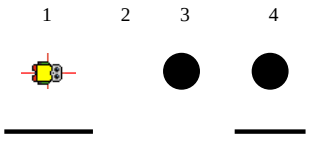
public static void task_10() //fill in gaps with a beeper. Stop when you reach a wall.
{

}

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
{



}

Using MapBuilder.jar, make your own maps for tasks 7 through 12 above.

Lab07
Exploration

Objective

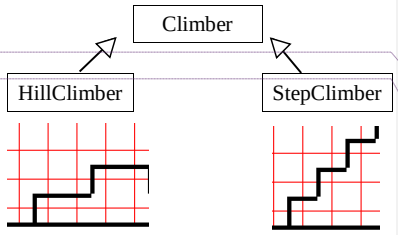
Polymorphism.

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

已註解 [u78]: ☆多型
對同樣一個名稱的 method，允許不同的「實現」方法。這是 Java 裏的一個重要的特色。會發生在三種情境：一個是子輩 override (改寫)父執輩的原本作法；二是兄弟姐妹間彼此對同一個 method 不同的作法；三是只有「虛名」的界面(接口) interface 所列出來的虛名 method。

已註解 [u79]: 名義上宣告是父輩的屬性，最後真的被 new 時，竟然是兒輩的來頂替！
記得：在 Java 的國度裏，反是父輩會(聽得懂)的 method，子女輩都一定會或是有對策執行。但是，子女會的新把戲，父母輩可不一定會。就像我現在說：開始寫 Java 程式，一般學生的家長恐怕就搞不定。

已註解 [u80]: 這要先寫爬山 Mountain 這個 class 說起。原來山只有一種，但之後，較不陡的，這時怎麼辦？

已註解 [u81]: 凌駕、蓋過之意。或簡單講：改寫：這是第一種多形。

已註解 [u82]: 在 `Lab07` 中的 `Mountain.explore( new Climber(x) )` 要如何解讀？
原本 `Mountain.explore( Climber arg)` 的定義是這個 method 先有一個預會存 `Climber reference` 的 `arg`，但是實際使用時，也就是 `new HillClimber(X)` 確 `new` 出一個子輩的來。這時候，按多型的系運作原則，原來 `arg` 也就收了。
這感覺就是 `Climber karel = new HillClimber()` 一樣。

已註解 [u83]: 原先針對特定「山的形狀」寫了反應「爬山」的動作的： `Mountain.explore( Climber arg)` 的方法，是給 `Climber` 爬的。之後，發現有不同陡度的山岐地。然而因為基本爬法一樣，所以，我們就以「改良」原 `Climber` 的方法，先 `extend` 一種新的類別，讓它們在爬上爬下時能順應不同的陡度。同名稱，不同作法，這就是「多型」的意義，

Exercises

Lab07

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

1. Does Mountain contain class methods or instance methods?
2. Which direction does the method explore() explore first?
3. How does explore() know not to search the other mountain if the treasure has already been recovered?
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 do we know that we've reached the summit?
7. If the treasure isn't there, 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. What is the difference between explore_west() and explore_east()?
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?

已註解 [N84]: 要先把 Mountain.java 仔細看過，再作答。

已註解 [u85]: ●在 Eclipse 的環境裏，試 debug，要學會設定中斷點 (break point)，然後停在這個中斷點上，之後，再用 F5(step into)、F6 (step over)、或是 F7(step return)。

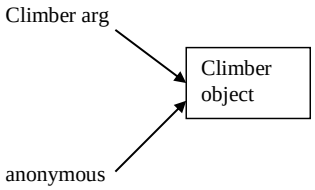
已註解 [u86]: 結論就是先寫一個給基本的 class 所使用的 method code，然後，再 extend 這個基本的 class，就不同本事的 method 重寫即可達到程式碼，像是 exploe()，共用的目的。

Discussion

Polymorphism

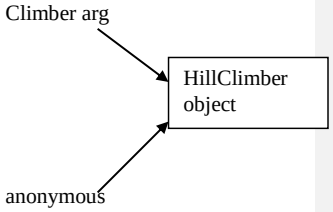
The formal argument to the method Mountain.explore() is a reference of type Climber. When an actual argument, either Climber, HillClimber, or StepClimber, is passed to Mountain.explore(), a copy of the reference is made. This means that the formal argument, named arg, will point to the same object as 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 Exercises 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*.

已註解 [N87]: 原先我們只是針對較陡的山寫了 explore 的 method，是要 Climber 爬的。結果後來發現還有山勢較不陡的 step 和 Hill。為了要容納新的狀況：多形的作法是保持 explore 的這個 method 不動，但另外寫 HillClimber 以及 StepClimber 當作 Climber 的子輩。以後，所有的狀況仍然讓 explore 來解決，只是如果真有要用到這兩種狀況，就讓 Climber 的兩個子輩出面應對。這是應變措施中，儘量不改即有系統的最佳安排。

已註解 [N88]: 對於複雜的資料格式，我們不可能把整個 object 傳過去，而是傳相對應的 reference 的值。

已註解 [u89]: 還記得在 Lab04.java 裏，除了 static 之外，另外還「埋」了一個 static 的 takeTheField()。我們之前說也可以分開成另外一個 class 寫。這裏的 explore() 就是另外寫一個 Mountain() 的 class。

已註解 [u90]: 與上個註釋同。

已註解 [N91]: 原本預期父輩的物件，結果確跑出他的子輩來頂替。但因為 explore 的這個 method 會用到的物件 method 都事先已經講好了，所以，一旦頂替上來的是子輩，那麼在真要執行 explore 的這個 class method 裏各個原先為 Climber 的 instance method 這時也由子輩的 instance method 來取代。

已註解 [N92]: 只能夠由兒輩來頂替，而不能由父輩來頂替兒輩！

已註解 [u93]: 這裏出問題的原因是：原本 Robot Class 就沒有定義 climbUpRight() 的 method()

已註解 [N94]: 實際執行的程式碼不一開始就 binding，而是要用的時候再 bind。

已註解 [u95]: ☆現在大家可能比較理解為什麼使用 class method 的那種 class，又叫作 static。因為這類的 class 不會有物件，也不會有「多形」的變化。另一方面，即然(不是 static) 的類別有可能會有反逆的子輩出來，而且又可能會 override，那 compiler 就「以逸待勞」，並不會急著預設立場要執行那一種可能的 method，而是「火燒屁股」真的要執行的時候才把真會用到的版本拿出來用。

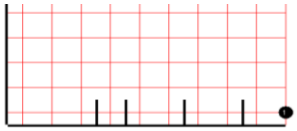
Lab08
Hurdle Racing

Objective

Polymorphism. The if-else control structure.

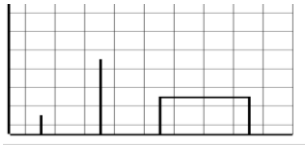
Background

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 the algorithm:



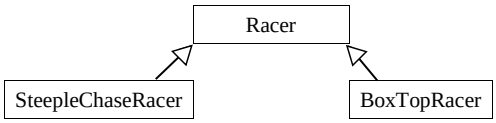
```
while(!arg.nextToABeeper())
  if(arg.frontIsClear())
    arg.move();
  else
    arg.jumpRight();
```

In English, the algorithm is if there's no hurdle, then move, otherwise jump over the hurdle. Keep going until you hit a beeper. In the code, the `jumpRight` in the else-clause is executed when the condition `frontIsClear` is false.



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

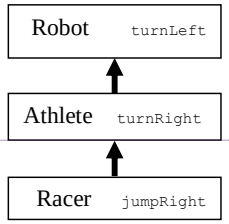
已註解 [N96]: 這個 race() 會用到 turnRight() 的這個指令。Robot 是不會 turnRight()的。

Discussion

Casting

Polymorphic behavior is resolved and executed at run-time. At compile-time, the compiler is only able to see if the superclass reference has access to ("knows about") the method calls. If the superclass does not know about the methods, the code won't compile. If the superclass does have access to the methods, then the code will compile. The polymorphic behavior appears only at run-time.

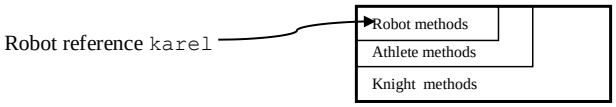
For example, in the previous lab, Robot does not know how to `jumpRight`. `jumpRight` is a completely new method in Racer. Yet we are sometimes forced to use the Robot reference when the object is actually a Racer. Can the Robot reference access Racer's new commands, such as `jumpRight`? No, unless you *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 computer does not allow access. Casting changes the reference, never the object itself! Here is an example of a cast: `Racer arg2 = (Racer)arg;`



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

Study the code for the Knight (which moves like a knight in chess) subclass. Then we create 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?

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. Let's cast the Robot `karel` reference to a Knight reference `lancelot`, then tell `lancelot` to retreat.

```
Knight lancelot = (Knight)karel;
lancelot.retreat();
```

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

5. On the drawing above, draw a lancelot reference to the entire Knight object.
6. Is `lancelot.retreat()` now a legal command? ____ Why?

7. Make the Extension to Lab08 work by casting the Robot reference. Explain what casting does.

已註解 [u97]: Casting 中文翻成「強制轉型」，其中又分為：
(1)硬把兒子當老子用的 upcasting，也就是發生 polymorhism 發生的狀況，像是：Robot karel = new Knight()，以及，
硬把老子當兒子的 downcasting，像是這裏的 4) 與 6)。
Downcasting 時，父輩保留其原有的 method，但是接受子輩所有額外新的 method。

已註解 [u98]: upcasting:
明明宣告是 Robot，但塞給他的卻是孫輩的 Knight 。

已註解 [u99]: Yes. turnLeft() 是 Robot 所知道的 instance method 之一。

已註解 [u100]: 這發生在 late binding，所以真的執行時，會用到 Knight 所定義的 turnLeft()。

已註解 [u101]: No. turnRight () 並不是 Robot 所知道的 instance method 之一，硬要塞，compiler 會抱怨：
The method turnRight() from the type Robot is not visible。

已註解 [u102]: Knight's turnLeft().
原本是 Robot 的 type，現被強制轉型為 Athlete：老伙丫被強迫要染髮穿垮褲？！這叫作「阿公變孫仔」downcasting；但別緊張，karel 還是 Robot，我們只是「按 karel 的樣子」另外作出一個叫 maria 的 Athlete：原來屬於 Robot 的部份會由 Knight 的方法定義；但對於 Athlete 多出 Robot 的 method，就按 Athlete 的定義。其中，對於 Athlete 而言，turnRight()就應該知道了。真正執行的 turnRight()也會是原本 Athlete 所定義的。

已註解 [u103]: Yes. Lacelot 已經被宣告為一個 Knight 了，所以它聽得懂 retreat() 這個 method。

Lab09
Shifting Piles

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

已註解 [u104]: 演算法：早晚 (eventually) 必然能解出問題的解法

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.

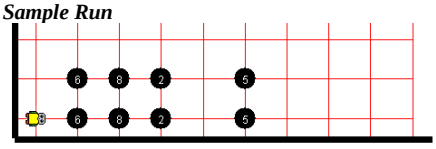
4. How can you proceed from one step of the problem to the next step of the problem?

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

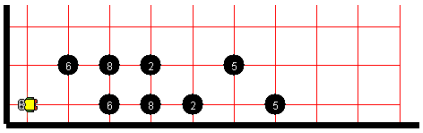
Good luck.

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".

Sample Run



before

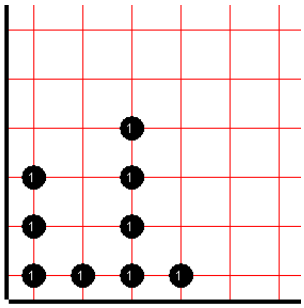


after

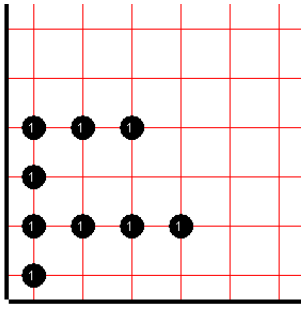
Extension
Use MapBuilder.jar to create valid robot-maps to test your program.

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. Assume that the robot is in position, facing east. Assume that the robot has already picked up and stored the number of beepers in that row in the variable *rowCount2*. Now write the code to put down *rowCount2* number of beepers in a 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.turnRight();
  karel.move();
}
else
{
  karel.turnLeft();               //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. (Can you use factoring to rewrite the last example?) When three-way branching is needed, we use an else-if ladder:

已註解 [u105]: 一定會發生一個分枝，而且也只有一個分枝會發生。

```
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.

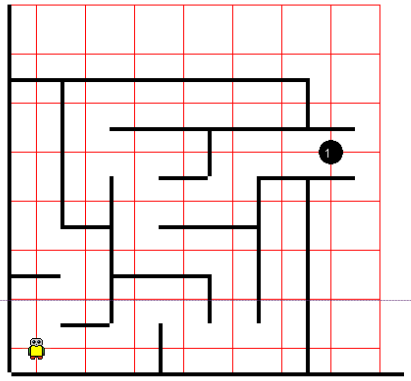
Lab10
Maze Escaping

Objective

Algorithms.

Background

You'll get very few hints on how to solve this problem. Your Athlete robot 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 are the initial conditions of the problem?

2. What are the terminating conditions of the problem?

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?

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

Good luck.

Specification

Create Unit1\Lab10.java at size 10x10. Declare one Athlete robot at (1, 1) facing north with an infinite number of beepers. Test your program on "maze1", "maze2", and "maze3". Escape the maze.

Test Data

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

Extension

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

已註解 [u106]: 自己試著沿牆邊走走看，靠左或靠右都行，但別臨時換邊

已註解 [N107]: 參考我準備有關 rightIsClear()以及 frontIsClear() 的「四種組合想一想」

已註解 [N108]: (加分題) 請設計一個 7 X 7 的 map {出口與牆由你決定}使得右派的 athlete 要花最遠的路程才能走到出口。修改你的程式印出摸索過程中所走的步數。

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();
karel.climbUpRight();
lisa.jumpRight();
lisa.climbUpRight();
horatio.move();
Lab04.takeTheField(pete);
Climber ophelia = new Robot();
Athlete kyla = (Athlete) karel;
Climber ann = (Climber) maria;
```

```
pete.pick();
pete.sprint(-10);
pete.put(5);
pete.climbUpRight();
pete.move(10);
Lab08.race(karel);
Robot hamlet = new Racer(3);
Climber ray = (Climber) horatio;
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: if the Robot karel is not carrying beepers, and it is on top of a beeper, then pick it up.

Discussion
Abstract Methods

An *interface* tells you **what** methods are available. An API is an interface, listing the methods that you can use. The *implementation* specifies **how** the methods work. Often, you don't know and don't care how the methods work. Other times, it will be your job to implement the interface.

An *abstract method* specifies an interface without its implementation. All you see is the header of the method plus the keyword `abstract`. For example, in Lab11 the abstract method is `public abstract void display();`. Notice the semicolon: there is no code in the body of an abstract method!

Since no implementation is given, an object of this class would not know how to respond when the method is called. Thus, it is not permitted to create objects of a class containing any abstract methods. A class with an abstract method is called an *abstract class*, labeled with the keyword `abstract`. The class could contain many methods with implementations, but so long as there is even one abstract method, the class must be abstract. The opposite of an abstract class is called a *concrete class*, but there is no keyword. All the classes we have used up till now have been concrete classes. All objects are instantiated from 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.

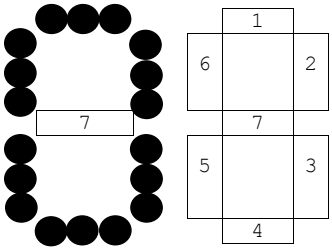
Look at the code on the right. Digit is an abstract class because it has the abstract method `display`. Notice that `display` has no implementation code in its body, but only a semicolon.

```
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. Therefore, Zero inherits `segment1_On`, as well as the private Robot object. On the other hand, if we want to instantiate a Zero object, we 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"?

已註解 [u109]: 界面 (接口)

已註解 [u110]: 實現

已註解 [u111]: 阿爸自己也作不來的心願 abstract method，要子輩來完成！

已註解 [u112]: 我們要在棋盤上利用七段顯示器的原理把 0 到 9 的十個數字顯示出來。其過程不外乎就是在這七個段距上決定要 on 或是 off。我們可以每個數字都寫一遍，或是將這些共用的部份建一個「父輩」稱之為 Display，各別的數字要顯示時，再將與別的數字迥異的 display() 的這個方法另外實現。其好處是：簡潔有層次。

已註解 [u113]: 這跟你指定 robot 爬的順序有關！

Lab11
Your ID Number

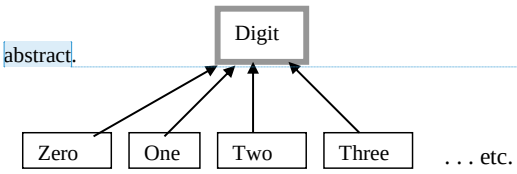
Objective

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

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.

已註解 [N114]: 在 UML 的符號中，代表 abstract 的 class，用灰色框來表示

已註解 [N115]: 他每次去 instantiate 一個 subclass 的 object 時，其實都是用一個 superclass 的宣告，這就是「多形」的作法

已註解 [N116]: 一開始 new 一個 object 就馬上指派給他工作作

已註解 [N117]: 逆向工程：你從系統的輸入與輸出間的關係反推出其中運作的道理與實現。
※你怎麼了解 Digit ?

Discussion

Interfaces

An interface is like an abstract class that has only abstract methods. No method in an interface actually does anything. All the methods are abstract. 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 there are no methods that do anything, it doesn't make sense to extend an interface. There is nothing to extend. Thus we use a different keyword, "implements," to indicate that our subclass is providing code for all the methods. That is, our subclass will *implement the interface.* For example,

```
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 our class defines **how** those methods work.

The power and beauty of this is that the driver application does not care about the object's implementation. The driver calls the methods and the object does its task. Object-oriented programming is so easy!

In the next lab you will define two classes that implement the Workable interface. (Since Workable has four abstract methods, you must define all four methods.) You then instantiate an object from one of those classes and pass it to the class method `work_8x8_square`. The class method `work_8x8_square` will call `work_one_row`. But what does `work_one_row` actually do? Here is the code:

```
 8
 9    public static void work_one_row(Workable arg, int n)
10    {
11       for(int k = 1; k <= n; k++)
12       {
13          arg.workCorner();
14          arg.moveOneBlock();
15       }
16    }
```

We can't tell what this code will actually do! One reason programmers use abstract classes and interfaces is to make the code more generic and re-usable, which has its good points and its bad points.

The actual behavior depends on how the methods `workCorner` and `moveOneBlock` have been defined in the subclasses. In other words, the `Workable` interface results in another example of polymorphic behavior.

Because `Workable` is generic code, we could instantiate and pass **any** object from any hierarchy, as long as that class also implemented `Workable`. The compiler guarantees that our implementing class actually has code for `Workable`'s methods. Obtaining that guarantee is another reason that code designers think in terms of interfaces.

已註解 [u118]: Interface 是「未完成心願」的列項，它只提供方法的 signature，但細節全無，「畫虎爛」的成份居多，但就會有人(類別)把他當真，並真的實現出來。如果你說真有類別把這「未完成心願」當父輩來崇拜，也不算過份。

另外要提的是：任何一個子輩只能夠有一個父輩，但可以以實現好幾個 interface。我們把這些非親非故「私淑艾者」的崇拜者，想成是這些只出口號界面的教子、教女。

● 原本要定義給這些「未完成心願」界面用的 method，只能靠這些奉行心願的教子、教女來幫化實現與執行。

命名的慣例：這種界面通常的慣例都是「able」結尾，表示，你搞得定的啦！

已註解 [u119]: Extends 是用來擴展真有或是抽象的類別；而 implements 是用來實現「未完成的心願」，我們把前者當作是生父母；後者當作是乾爹、乾媽。生父母只能夠有一個，但乾爹、乾媽可以有很多個。

已註解 [u120]: 之前我們討論過在物件導向語法中，由子輩「頂替」父輩出征的用法。這裏的 interface 本身雖然不是一個類別，但是對於「實現」這個 interface 的任何 class，可「等效」看成是這裏 interface 的子輩，並且可以用類似 頂替 superclass (這裏的 interface) 的方式傳入本來要求為 interface arg 的函式中。

已註解 [u121]: 多形發生的狀況，除法兒女覆寫 (override) 父輩的作法之外；兄弟姐妹(sibling)之間對同一個 method 不同的實現方法；最後一種就是各教子、教女對界面不同的實現方式。

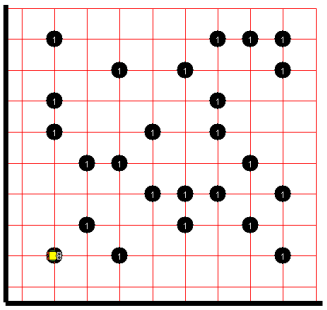
Lab12
Harvesting and Carpeting

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 when they find them whereas carpeters put beepers down where they find none. On any given corner with a beeper, a harvester will pick the beeper up but a carpeter will leave the beeper there. On any given corner without a beeper, a carpeter will put a beeper down but a harvester will not.

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

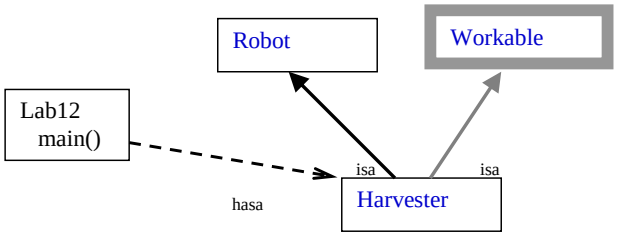
已註解 [u122]: 之前我們討論過在物件導向語法中，由子輩「頂替」父輩出征的用法。這裏的 interface 本身雖然不是一個類別，但是對於「實現」這個 interface 的任何 class，可「等效」看成是這裏 interface 的子輩，並且可以用類似 頂替 superclass (這裏的 interface) 的方式傳入本來要求為 interface arg 的函式中。

Code in the main method, shown to the right, will create either a harvester or a carpeter with equal probability. The Math class method `random` generates and returns one decimal number between zero and one. It might be zero but it will not be one. Another way to say this is that `Math.random` returns a value from the interval [0, 1).

```
34    if( Math.random() < 0.5 )
35    {
36       work_8x8_square( new Harvester(2, 2) );
37    }
38    else
39    {
40       work_8x8_square( new Carpeter(2, 2) );
41    }
```

Warning. Some of the method definitions in Harvester and Carpeter may seem overly simplistic. Remember that the interface knows nothing and doesn't want to know anything about how your class will work. For instance, it doesn't know that the inherited classes are also Robots. So if the interface requires you to specify how to move forward one block, just use the robot's `move` command. That's it.

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



Specification

Load Unit1\Harvester.java and Unit1\Carpeter.java. Implement the required methods.

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

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. 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. Dolphin s = new Mammal();
g. Fish f = new Dolphin();
```

Lab13--Project

An Original JKarel Lab

Use your imagination and your experience from this unit to create an original JKarel problem. In formulating your own JKarel 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 JKarel 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 JKarel Lab:	0----1----2----3----4----5
UML class diagram is correct:	0----1----2----3----4----5
lab is at an appropriate level of difficulty:	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

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 the robots and the 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 and implements Runnable. Implementing Runnable is important because a Thread constructor requires a Runnable object as an argument.

```
Thread(Runnable arg)
```

In Lab14, the compiler checks to make sure that weismuller, fraser, etc., are all Runnables.

```
4  public class Lab14
5  {
6    public static void main(String[] args)
7    {
8      Swimmer weismuller = new Swimmer(2);
9      Swimmer fraser = new Swimmer(4);
10     Swimmer spitz = new Swimmer(6);
11     Swimmer phelps = new Swimmer(8);
12     Thread t1 = new Thread( weismuller );
13     Thread t2 = new Thread( fraser );
14     Thread t3 = new Thread( spitz );
15     Thread t4 = new Thread( phelps );
16     t1.start();
17     t2.start();
18     t3.start();
19     t4.start();
20   }
21 }
```

Thread is a built-in java class called java.lang.Thread. Every Thread object has a `start` method that takes care of the details of running parallel processes. In Lab14, four different processes will all execute at the same time. The result will be parallel processing, i.e., different robots will perform their own methods seemingly at the same time.

Suppose we are writing a program that shows a video accompanied by audio. Obviously, we want our video and audio to execute together, at the same time. Complete the little shell at the right.

This synchronization is called *concurrency*. In order to cause independent processes to execute concurrently, in parallel, Java uses *threads*. Since threads are built in, they don't have to be imported.

```
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.

已註解 [u123]:
- A thread is a single sequence of execution within a program
- each program can run multiple threads of control within it, e.g., Web Browser
同一個時間「同步」執行的程序

已註解 [u124]: 實際的情況是這些 thread 都排隊在 cpu 中輪流被一個一個「分時」地被執行，因為分的時間很短，而人的反應很慢，所以會有感覺，他們都是「萬箭齊發」「同時同步」地再被執行中。

已註解 [N125]: 不同的程式在同一個時間一起執行，這叫同步 concurrently

已註解 [u126]: 語法

Lab14
Synchronized Swimming

Objective

Implement an interface to achieve parallel programming.

Background

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, it is Java's way to guarantee parallel behavior.

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

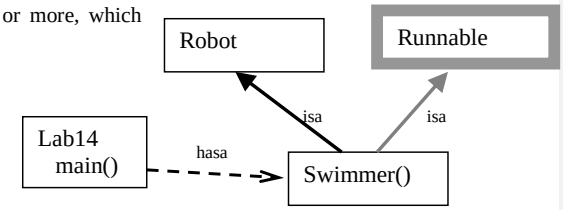
A Thread constructor requires a Runnable object as an argument. Every Thread object has a start() method that runs run() and takes care of all the parallel processes. In other words, all we have to do to produce parallel behavior is to implement the Runnable interface in our Swimmer object. That's your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

```
1   //Name_____________________________ Date_____________
2   import edu.fcps.karel2.Display;
3   import edu.fcps.karel2.Robot;
4   public class Swimmer extends Robot implements Runnable
5   {
6     public Swimmer(int x)
7     {
8       super(x, 1, Display.NORTH, 0);
9     }
10    public void run() //not swim
11    {
12
13
14
15
16
17
18
19    }
20  }
21
```

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.

已註解 [N127]: 每一個 class 只能當另一個(而且是最多一個)class 的子輩，但可以同時 implement 好幾個 interface。

Fill in the class diagram for Lab14.
Which methods in Swimmer come from Robot?
Which method comes from Runnable?



Specification

Load Unit1\Swimmer.java. Implement the Runnable interface so that a swimmer moves forward twice, 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.

Discussion
Concrete and Abstract Classes

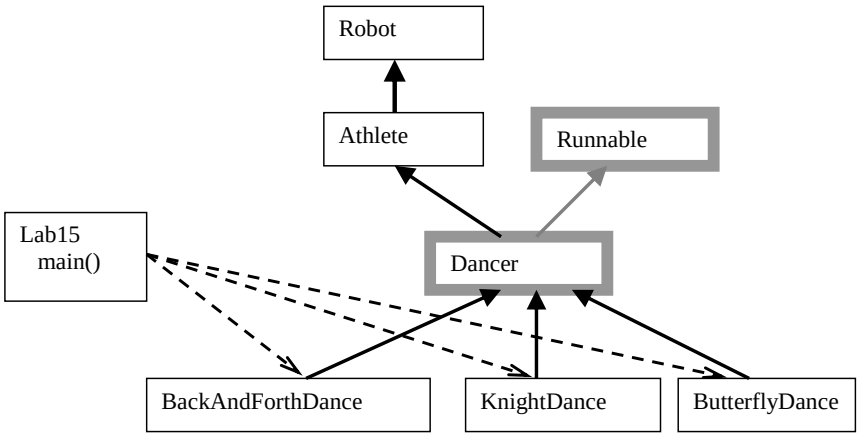
```
1  //Torbert, e-mail: smtorbert@fcps.edu
2  //version 4.16.2003
3
4    import edu.fcps.karel2.Robot;
5    import edu.fcps.karel2.Display;
6    public abstract class Dancer extends Athlete implements Runnable
7    {
8      public Dancer(int x, int y, int dir, int beep)
9      {
10       super(x, y, 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 Dancer class must be tagged `abstract` because it contains the abstract method `danceStep` on Line 16. Because Dancer is abstract, it is impossible to instantiate Dancer objects. However, we still give Dancer a 4-arg constructor (lines 8-11) in order to complete the chain of constructors. Somewhere in the hierarchy below Dancer, we know we are going to write a subclass class that will be instantiated.

The subclass immediately below Dancer normally contains the code that implements `danceStep`. It is possible, however, to write a subclass that postpones implementing `danceStep`. In that case, that subclass would also have to be abstract.

Because the `run` method (from Runnable) is implemented in Dancer, Mr. Torbert is choosing to make every subclass have the same `run` method. He could have made a different design decision, namely, not to implement the `run` method in Dancer, but to make `run`, as well as `danceStep`, abstract in Dancer. That would also work. Eventually, somewhere in the hierarchy, all the abstract methods will get properly implemented. In other words, all the abstract classes will eventually have concrete subclasses.

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



已註解 [u128]: 有一點神奇的是：抽象的類別也可以信奉某教義；abstract class implements interface。

已註解 [u129]: 原本是阿爸未完成的心願，現在發生了，他的兒子也沒完成，搞到當時未完成阿爸的心願變成是「從阿公以來就從來沒有沒成過的心願」。

已註解 [u130]: 對於號稱要 implements 某 interface 的 class，如果一時間實現不出來的話，就得在 class 的宣告前面加上 abstract 的字樣。

或是說：我的確是奉行什麼什麼主義 (interface)的，但是我現在有點忙，等我小女兒出現時再實現。

已註解 [u131]: 要用的人就要實現出來。

Lab15
Dancing Robots

Objective

Extend an abstract class and implement an interface.

Background

The class shown here may be the world's most boring dancer ever.

```
1   //Torbert, e-mail: smtorbert@fcps.edu
2   import edu.fcps.karel2.Display;
3   import edu.fcps.karel2.Robot;
4   public class BackAndForthDancer extends Dancer
5   {
6     public BackAndForthDancer()
7     {
8       super(1, 1, Display.EAST, 0);
9     }
10    public BackAndForthDancer(int x, int y, int dir, int beep)
11    {
12      super(x, y, dir, beep);
13    }
14    public void danceStep()
15    {
16      move();
17      turnAround();
18      move();
19      turnAround();
20    }
21  }
```

Notice that only the danceStep() method, which was specified abstract in Dancer, must be defined. The run() method that was defined in Dancer is simply inherited.

This means that when you create a BackAndForthDancer object and an associated thread, and when you start that thread, Java will first check here for the run() method, then move up the hierarchy to Dancer's run(), then look back here for danceStep(). Java at run-time always starts looking for a method's definition at the object on which it was invoked.

已註解 [N132]: 當被 new 時，Javac 會先由這一層的 class 裏的 method 找起，找不到的話，就會找到爸媽，就像卡債求償一樣…

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. At some point you may have had an error such as:

```
Exception in thread "main" java.lang.NoClassDefFoundError
```

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 is in every Java application.

Specification

Create at least three different dancer-type classes. Create three new Java files modeled after the example shown above. 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. Create objects from your three different dancer classes and have them "do their stuff" in parallel.

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:

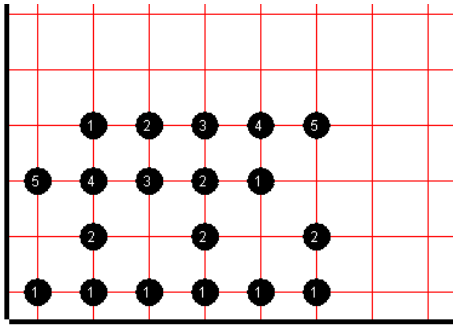
```
private int myBeepers;
```

`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.



已註解 [N133]: 每個 class 最多只能當另一個 class 的 subclass，但可以 implement 好幾個 interfaces

已註解 [u134]: 修改 Lab09 使得其變成一個同時可以跑四個 threads 的程序，這實你要作什麼事呢？

Discussion
Abstract Classes and Interfaces

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

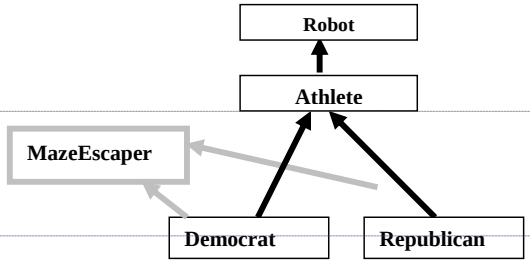
```
5       public abstract class MazeEscaper extends Athlete
6       {
7               public abstract void walkDownCurrentSegment();
8               public abstract void turnToTheNextSegment();
9       }
```

It seems like MazeEscaper might easily be an interface. Mr. Torbert chose to make it an abstract class because he wanted to guarantee that MazeEscapers were subclasses of Robots. Let's look at how the code uses MazeEscaper objects.

```
1   //Torbert, e-mail: smtorbert@fcps.edu
2   import edu.fcps.karel2.*;
3   import javax.swing.JOptionPane;
4   public class Lab17
5   {
6     public static void escape_the_maze(MazeEscaper arg)
7     {
8       arg.walkDownCurrentSegment(); //you are not at the end at the start
9       while(!arg.nextToABeeper())
10      {
11        arg.turnToTheNextSegment();
12        arg.walkDownCurrentSegment();
13      }
14    }
```

The class method `escape_the_maze` has an argument that accepts either a MazeEscaper object or a MazeEscaper-subclass object. If you try to pass something else, the code won't compile.

Now look at the code in `escape_the_maze`. Which `arg` command works **only** with Robot objects? `___nextToABeeper()___` Because that command works only with Robot objects, MazeEscaper should be in the Robot hierarchy, not simply an interface. An interface would not guarantee inheritance from Robot. The rule of thumb is, if inheriting behavior is important, then make an abstract class, like MazeEscaper in Lab17 or like Digit in Lab11. On the other hand, if two behaviors are completely unrelated to each other, like Robot and Runnable, then make an interface. The UML diagram below shows MazeEscaper as an interface.



However, we could still get this code to work even if MazeEscaper were written as an interface. We would simply have to cast the MazeEscaper reference to an Athlete reference. If you can explain why, you understand what casting is.

Suppose you were designing Flier objects. Would it be better to make Flier an abstract class or an interface?

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

已註解 [u135]: 什麼時候用 interface？什麼時候用 abstract class？

已註解 [u136]: 我猜想作者指的是：

MazeEscapter mazeEsc =
new MazeEscaper();
Athlete mazeAthlet =
(Athlete) mazeEsc;
接下來的各功能，就用這個 mazeAthlet 來作。

但我不確定的是，任何實現 MazeEscaper 界面中方法的 class 如何搞定那些原本就是 Athlete 才會的動作？

已註解 [u137]: 直覺覺得是 interface

已註解 [u138]: 直覺是 abstract class

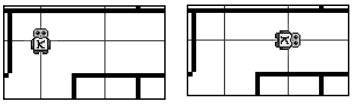
Lab17
Republicans and Democrats

Objective
Extending an abstract class.

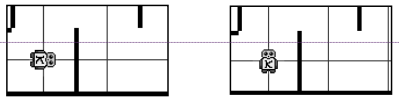
Background

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 its right-hand-side. Here is the algorithm, known as "follow walls right": it is okay to move forward if there is a wall to your right AND your front is clear. Repeat until you must stop. 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.

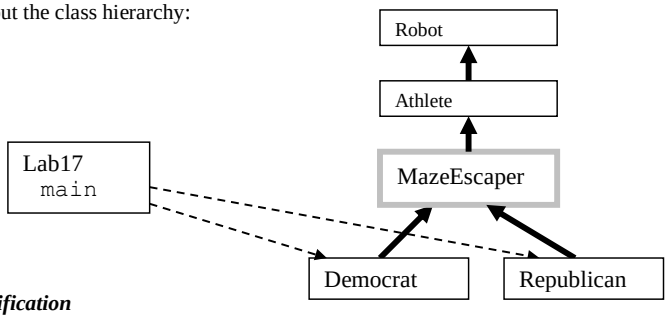


This time, we have abstracted the walking and the turning into two abstract methods in MazeEscaper.

```
5  public abstract class MazeEscaper extends Athlete
6  {
7      public abstract void walkDownCurrentSegment();
8      public abstract void turnToTheNextSegment();
9  }
```

Of course, you could just as easily "follow walls left" to escape the maze. The abstract class allows both Republican and Democrat objects to use the same static class method `escape_the_maze` in the driver.

Fill out the class 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", and "maze3".

Extension 1
Start each robot in the upper left hand corner. You must write the appropriate constructor in MazeEscaper.

Extension 2
Let MazeEscaper implement Runnable, making all necessary changes. Which politician escapes first?

已註解 [u139]: 原本我們要考慮的有四種情況：
(1)右手有牆、前無擋路；
(2)右手有牆、前有擋路；
(3)右手無牆；
(4)前、左、右都有牆(死角)

但其實，
(3) 發生時，就向右轉，然後向前走一步；(如果轉向向前走一步之後，還不是摸不到牆時，還是可以再「就向右轉，然後向前走一步」；
(4) 可以併入(2) 的考量。

已註解 [u140]: 這個非常有趣，你要實現 Runnable 的界面，那這兩種 politicians 來比快脫逃。

Exercises

Hierarchies, Abstract Classes, and Interfaces

1. Why would a programmer choose to make an abstract class, such as MazeEscaper?

2. Why would a programmer choose make an interface such as Runnable?

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

4. Are constructors inherited in Java? Y/N

5. Suppose we add this code to 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 is `super`?
   e. What happens to 1, 6, Display.NORTH, 0?

6. Define "polymorphism."

7. Explain exactly how and where the MazeEscaper lab demonstrates polymorphic behavior. (Recall that a Thread's `start` command automatically calls a `run` method.)

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() (i.e., 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.

已註解 [u141]: 目前的 method 就不再作了。

已註解 [u142]: 這是針對像 while 或是 for 這樣的 loop：break 整個 loop 不用再 loop 了；而 continue 是該 loop 不用往下作，而再重新進入 loop。

```
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.

已註解 [u143]: Break 是跳離整個 while 的迴圈

已註解 [u144]: Continue 是迴圈中剩下來的部份不執行，而立即再跑到最上面再就 loop body 的部份重新開始執行。

By convention these statement make your code less readable than 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:

已註解 [N145]: 課本原來寫 then 是錯的，應該是 thAn 才對！

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

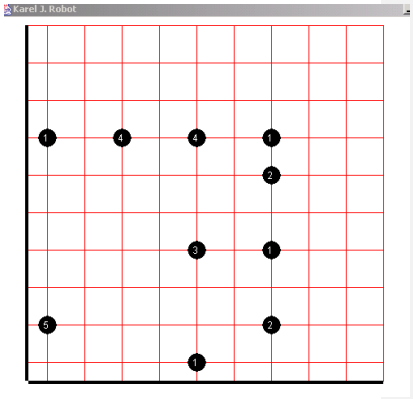
Use return, break, and continue sparingly and only for handling special cases.

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.

Algorithm for Reading a Map
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.



Be careful! You must pick up the pile after reading it. Keep all the beepers you encounter in your beeper bag. Otherwise, you may change the map.

Consult the JKarel 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.

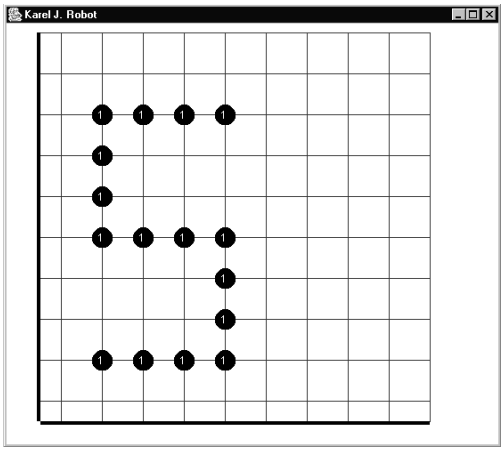
Load Unit1\Lab18.java at size 8x8. Use maps "map1", "map2", and "map3". Compile and run.

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.

已註解 [u146]:
```
    int month = 8;

        switch (month) {
            case 1:  futureMonths.add("January");
            case 2:  futureMonths.add("February");
            case 3:  futureMonths.add("March");
            case 4:  futureMonths.add("April");
            case 5:  futureMonths.add("May");
            case 6:  futureMonths.add("June");
            case 7:  futureMonths.add("July");
            case 8:  futureMonths.add("August");
            case 9:
futureMonths.add("September");
            case 10: futureMonths.add("October");
            case 11: futureMonths.add("November");
            case 12: futureMonths.add("December");
                     break;
            default: break;
        }
```

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!

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

Class Dorothy

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

```
public class Dorothy extends 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();
  }
}
```

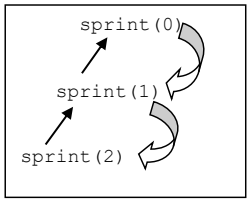
The following solutions are *recursive* because the method `sprint` calls the method `sprint`—it calls a copy of itself. The two solutions below act exactly the same, but the code on the right is often easier to read. Since the **`return`** statement forces the method to end early, the **`else`** is 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.

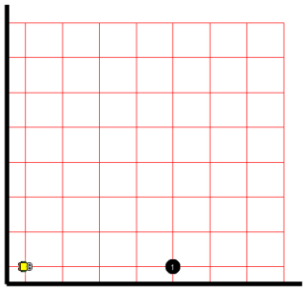
Return to Lab18's Pirate class. All three instance methods in Pirate can be written recursively.

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. This is called the base case. 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.

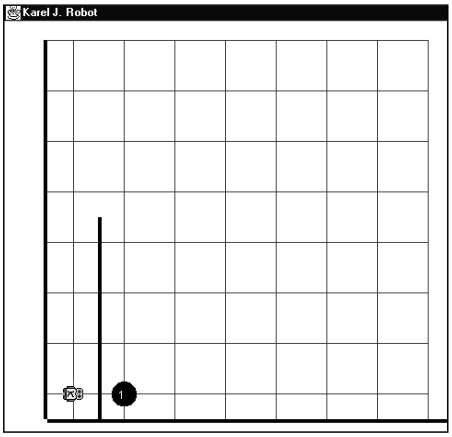
LabXX
Seeking the Beeper

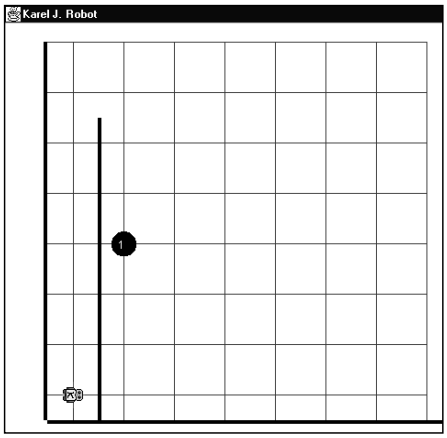
Objective
To understand recursion.

Background
Your robot begins at (1, 1) facing north. 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.

已註解 [N147]: 課本原來寫 facing east，但實際的狀況是 Seeker 的 constructor 初始即為 north

已註解 [N148]: 這個附的兩張圖，畫出來的 Robot 應該都要 facing north 才對





Consult the JKarel 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 multiple randomly generated maps—run the program a few times to make sure your method always works.

Test Data

This screen capture shows what your robot-map will look like at the end of execution, regardless of the height of the wall or the placement of the beeper on the 2nd x-coordinate.

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.

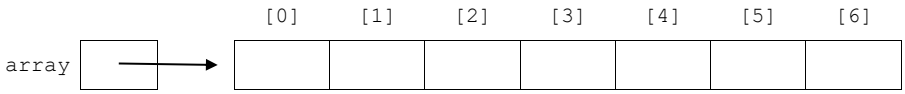
```
public void followWallsRight()
{









}
```

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 create actual robot objects use:

```
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.

已註解 [N149]: 這個 [] 就是用來宣告，我要講的是一組陣列。

接下來再來看前面的類名：Robot，
這讓 Robot[] 所代表的意思為：
我要宣告一組「內容是用來存指向 Robot 物件的陣列」

已註解 [N150]: 這一共有三個動作：
(1) Robot[] array：我要預留一個帶頭的路標名稱，這個名稱是用來存放內容為 Robot 陣列的內容的帶頭的路標；
(2) New Robot[7]：我要真的去找店面能讓我擺 7 組 Robot 物件的位置；
(3) array = new Robot[7] 我把先前的名稱和這實體掛鉤放在一起

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.

```
Display.placeBeeper((int)(Math.random()*N+1), (int)(Math.random()*N+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.