Hayfield Secondary AP Summer Assignment Cover Sheet

Hayfield Secondary
7630 Telegraph Road, Alexandria, VA 22315 • Phone: (703) 924-7400 • FAX (703) 924-7497

Course	AP Computer Science A
Teacher Names & Email Addresses	Daria Bergen-Hill (debergenhill@fcps.edu) Brian Oliver (bjoliver@fcps.edu)
Assignment Title	AP Computer Science A Summer Assignment
Date Assigned	June 2019
Date Due	9/13/19 (2nd Week of School)
Objective/Purpose of Assignment	• Become excited about the study of computer science • Familiarize yourself with the Java coding language • Gain a sense of accomplishment when you write, compile and run a program of your own creation. • Internet resources can be found on page 1 of the summer assignment.
Description of how Assignment will be Assessed	Grading will be based on completion of program(s) that both compile and run correctly.
Grade Value of Assignment	**If you have prior coding experience** 25 Points for creating a program in the Java coding language (equivalent of a large quiz – no more than 10% of quarter grade). **If you have no coding experience** 25 Points for completing labs 1-8 in the attached package - "Java Steps to Object Orientation"
Tools/Resources Needed to Complete Assignment	You will need access to a computer and the internet in order to complete this program. Please let Mrs. Bergen-Hill or Mr. Oliver know if you do not have access to either so accommodations can be made.
Estimated Time Needed to Complete Assignment	8 hours
Learning Goals	Student should be able to write, run, test, and debug a basic solution in the Java programming Language.

Advanced Placement Computer Science A
Instructor(s):

Daria Bergen-Hill
debergenhill@fcps.edu

Brian Oliver
bjoliver@fcps.edu

Summer Assignment

Your assignment:

1) Install the necessary tools on your computer at home as described below.
2) Read as far into Java Basics as you like, testing the example code and running it in JGrasp.
3) **a) If you have programming experience:**

   Create your own working program in JAVA. The more you read into the Primer, the more powerful things you will be able to create in your program. Also feel free to use online resources to learn even more.

   Your program should do the following:

   I. Ask for user input and read it in to variables.
   II. Process the user input to find a solution, solve a problem or perform an action. The process should include: a) At least one if-else statement, AND b) At least one loop (for or while) … Both parts a) and b) should have an effect on user input.
   III. Show the user the results of the process.

   Your program should be of your own design. It should NOT come from any of the following:

   - The examples in this document.
   - Online.
   - A book or other print source.
   - Another person.

   b) If you have no programming experience:

   Work through Labs 01-08 in "Java Steps to Object Orientation"

Here is how to set up a programming station at home:

Here is how to set up a programming station at home:

Java

- Go to: http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html and download the Java JDK for your specific operating system.
- Install the JDK.
- If the link is inactive, search for "JDK 8 oracle download".

jGrasp

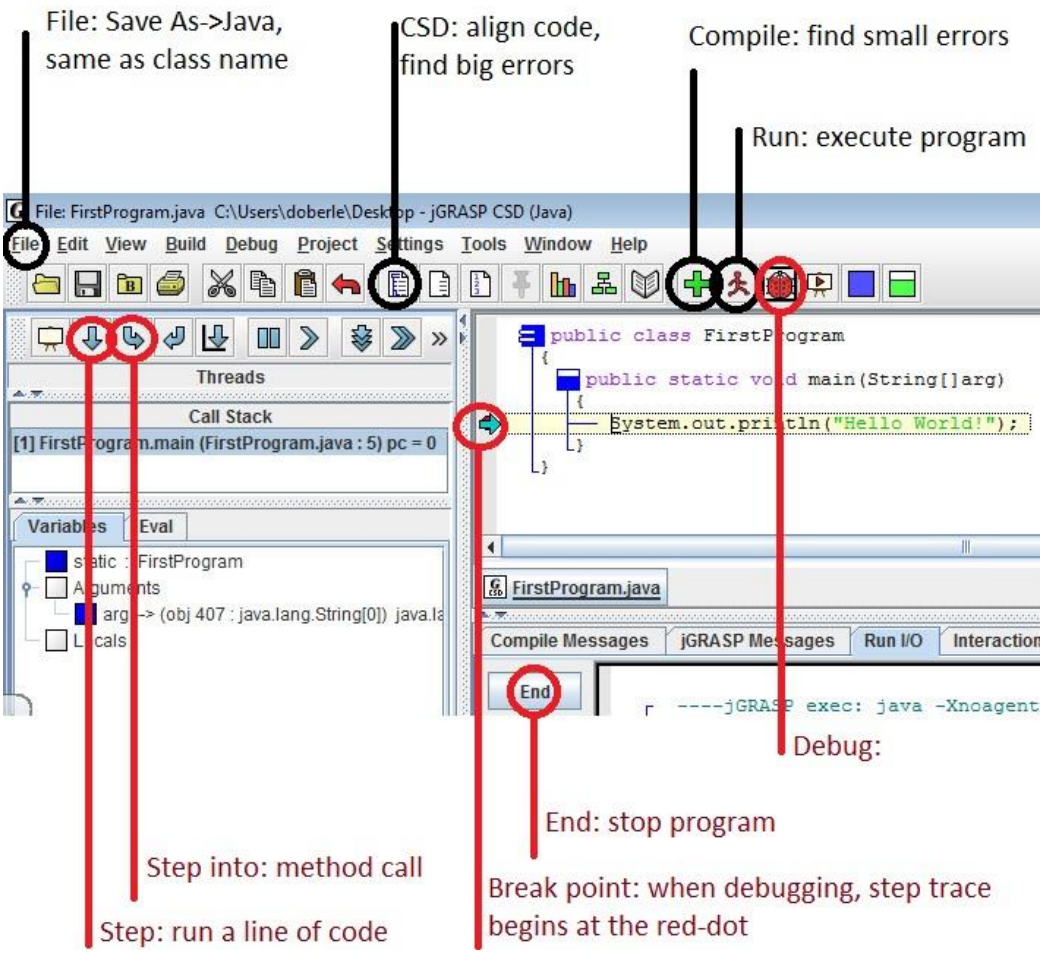
- Go to: http://www.jgrasp.org/index.html and download jGrasp for your operating system.
- Install jGrasp.
- If the link is inactive, search for "jGrasp", or use any other Java IDE, like BlueJay, Eclipse or Netbeans.

Test your installation on a standard Java program.

- Double-click on jGrasp.
- In the upper-left hand corner, click on File->new->Java.
- A large, blank editing window will open in the center-right side of the program. In that editing window, type in the following:

```
public class HelloWorld
{
     public static void main(String[] arg)
     {
          System.out.println("Hello World");
     }
}
```

- Now go to File->Save As. If the program is typed in correctly, it will save as HelloWorld.java.
- Click on the Compile button (the button with the +).
- If there are no errors, click on the run button (next to compile).
- Avoiding common errors:
    - The file must be the same as the title on the first line after public class, including case sensitivity.
    - If the program is defined within public class HelloWorld, the file must be saved as HelloWorld.java.
    - The vertically alligned brackets are the “squiggly-brackets”, to the right of the ‘p’ key on the keyboard.
    - The parenthesis after main and println are the round-parenthesis SHIFT-9 and SHIFT-0 keys.
    - The System.out.println call is followed by a semi-colon, to the right of the ‘L’ key.



File: save as a Java file with the same name as the public class name.

CSD: finds big errors, like unmatched parenthesis, braces and brackets. Aligns your code to make it easy to read.

Compile: finds small errors, like missing semicolon, spelling and case-sensitivity issues.

Run: executes last successful compile.

Debug: if your program compiles but does not execute correctly, you can run the program one line at a time and watch the variables change with the debugger. Set up a break point at the part of the program you want to debug, hit the Debug button and step through the code to find the logical error.

Break Point: along the left border of the code window, toggle red-dot for a break point where you want to start debugging.

Step: when debugging, step will execute a single line of code

Step into: when debugging, step into will enter a method call to allow you to debug within.

End: halt a program if you have an infinite loop.

Java Basics

Java programs must be titled with a class name that is exactly the same as its file name, with case sensitivity. When you run a java program, it executes the code found in the main function.

```
public class FirstProgram
{
   public static void main(String [] arg)
   {
      System.out.println("Hello World.");
   }
}
```

- This must be in a file called FirstProgram.java, since that is the name of the public class.

- The main function always appears as: public static void main(String [] arg),
and the code within the braces contains the commands that you want to run.

- System.out.println will write text and information out to the screen. Upon running the program, it will write the message **Hello World**. to the screen.

Java is an object oriented programming language, which means you can define and use objects.

An **Object** is an entity that can store multiple pieces of data (data fields) as well as perform actions (methods). Consider that you might have a Tank object called sherman that stores information, like its location, speed, and amount of ammo, but can also perform actions like sherman .move(5), which might move forward 5 meters, or sherman .shoot(), which fires a shell and decreases the ammo count by one.

Primitive variables are simple data types that can store a single value. There are many different types, but we will primarily use the following:

- int stores a single whole number (integer value)
- double stores a single real number (may have a decimal)
- boolean stores the value true or false (used for conditions)

And we will also use a common object called the String:

- String stores a collection of characters (a word or sentence)

Declaring a variable requires assigning a type, and giving it a name to identify it. Identifier names need only follow these simple rules:

- only comprised of letters, numbers and the underscore
- must start with a letter, and should be a name that makes sense for its use
- can't be a reserved word, which is already taken by Java (public, static, void, String, int, etc).

```
int num = 5;                    //creates an integer called num that stores the state 5.
double accel = 9.81;            //creates a real number called accel that stores the state 9.81
boolean done = false;           //creates a boolean called done that stores the state false
String word = "hello";          //creates a String called word that stores the state hello.
```

Note that literal strings are placed inside of quotes "".

```
//in SecondProgram.java
public class SecondProgram
{
   public static void main(String [] arg)
   {
      int num = 5;
      double x = 4.5;
      System.out.println("Our integer is " + num + " and our real is " + x );
   }
}
```

The program will output the following when run: **Our integer is 5 and our real is 4.5**

Literal text is placed within quotes: "Our integer is ". We can write the state of a variable by adding the variable name to the literal text. Upon reaching the num, it writes the state of the variable which is 5.

Consider that our println command looked like the following:
System.out.println("Our integer is num and our real is x");
In this case, the output would be **Our integer is num and our real is x**, because the entire contents would be considered literal text.

Any text after a double slash // will be ignored by the compiler and is used to comment code.
This is called an end-of-line comment, because everything after it on that line is ignored.
You can comment out several lines of code by placing comments between /* and */.

```
//in SecondProgramB.java
public class SecondProgramB
{
   public static void main(String [] arg)
   {
      /*
         NONE OF THIS TEXT WILL AFFECT THE CODE.
         It is just a comment block, and used to communicate information about the program.
         You will learn what is worthy to comment later.
      */
      int num = 5;
      double x = 4.5;

      //this is also a comment and will be ignored by the compiler

      System.out.println("Our integer is " + num + " and our real is " + x );
   }
}
```

Basic math operators include:
+ (addition), - (subtraction), * (multiplication), / (divide), / (div), and % (modulous)

/ (division), when given at least one number with a decimal.

```
System.out.println(3 / 6.0);      //would output 0.5
System.out.println(3.0 / 6);      //would output 0.5
System.out.println(3.0 / 6.0);    //would output 0.5
int x = 3;
double y = 6.0;
System.out.println(x / y);        //would output 0.5
```

/ (div), when given two integers, performs whole number division.

```
System.out.println(3 / 6);        //would output 0, because 6 goes into 3 zero times
int a = 3;
int b = 6;
System.out.println(a / b);        //would output 0
a = 7;
b = 3;
System.out.println(a / b);        //would output 2, because 3 goes into 7 two times
```

% (mod), when given two integers, mod returns the remainder after whole number division.

```
System.out.println(3 % 6);        //would output 3, because 6 goes into 3 zero times
                                  //with a  remainder of three
a = 7;
b = 3;
System.out.println(a % b);        //would output 1, because 3 goes into 7 two times
                                  //with a remainder of one
System.out.println(6 % 3);        //would output 0, because 3 goes into 6 two times
                                  //with a  remainder of zero
```

The assignment operator = is used to assign a value to a variable.
Only the variable to the left of the assignment operator will change.

```
        a = 7;                    //a is assigned the state 7
/*      7 = a;                    //WILL NOT COMPILE              */
        int num = 5;              //an integer num is assigned the state 5
        num = num + 10;           //num is assigned to its old state + 10, so it changes to 15
        System.out.println("Num is " + num);     //will output Num is 15
        int value = 3;            //value is assigned to 3
        value = value + num;      //value is assigned to its old state + num's state
                                  //note:  num does not change here
        System.out.println("Value is " + value);  //will output Value is 18
```

Arithmetic shortcuts:

The increment operators ++ and -- will add or subtract 1 to the previous state of any numeric variable.

```
a = 8;
a++;                    //x now stores the state 9.  Equivalent to a = a + 1;
num = 12;
num--;                  //num now stores the state 11.  Equivalent to num = num - 1;
```

The operators +=, -=, *=, /= and %= will perform an operation on the previous state of a variable with a value.

```
a = 3;
a += 10;        //equivalent to a = a + 10;
                //a is assigned to its old state 3 plus 10, so now a is 13.
b = 3;
b *= 2;         //equivalent to b = b * 2;
                //b is assigned its old state 3 times 2, so now it is 6.

num = 7;
num %= 2;       //equivalent to num = num % 2;
                //num is assigned to its old state 7 % 2, so now it is 1.
                //2 goes into 7 three times with a remainder of 1.
```

Getting input from the keyboard:

```
//in ThirdProgram.java
import java.util.*;       //the import statement makes useful tools available for your program
import java.io.*;
public class ThirdProgram
{
   public static void main(String [] arg)
   {
        Scanner input = new Scanner(System.in);        //used to get input from the keyboard
        double x, y, ans;                              //3 primitive variables of type double (real #)
        System.out.println("Enter a number");
        x = input.nextDouble();                        //waits for input and stores it in x
        System.out.println("Enter another number");
        y = input.nextDouble();                        //waits for input and stores it in y
        ans = x * y;
        System.out.println(x + " times " + y + " is " + ans);
   }
}
```

If the user enters in the value 2 for x, then enters 4 for y. The program will output: **2 times 4 is 8**

Given a Scanner object called input, the methods used for reading values include:

```
input.nextInt();        //for reading in whole numbers
input.nextDouble();     //for reading in numbers that might have a decimal
input.next();           //for reading in a String (any characters up to the first space or enter key)
input.nextLine();       //for reading in an entire line of text (a String that might include spaces)
```

We will get to the String objects later on...

```
import java.util.*;                                    //makes useful tools available
import java.io.*;
public class ThirdProgramB                             //in ThirdProgramB.java
{
   public static void main(String [] arg)
   {
      Scanner input = new Scanner(System.in);          //used to get input
      String name;
      int age;
      System.out.println("Enter your 1st name");
      name = input.next ();                            //waits for input, stores it in name
      System.out.println("Enter your age");
      age = input.nextInt();                           //waits for input, stores it in age
      System.out.println(“Little " + name + " is already " + age + " years old!" );
   }
}
```

If the user enters in the value Doug for name, then enters 45 for age.
The program will output: **Little Doug is already 45 years old!**

Writing a program:
1) define any needed variables (one for each piece of input, one for each solution that must be found)
2) ask for the input and read it in (use System.out.println and the Scanner input object)
3) find the answer using assignment statements
4) show the answer (using System.out.println statements)

Consider that we want a program that will compute the mpg rating for a car after we take a trip. The user will need to enter the number of miles they drove and the number of gallons they used. That will require two variables of type double, since they might have decimals. The program will then take the user input and compute the mpg rating for the car, which will require another variable, also of type double.

So we need to define 3 variables of type double, ask for and read in the miles driven and gallons used, compute the solution and then show the solution to the user:

```
//in a file called FourthProgramA.java
import java.util.*;
import java.io.*;
public class FourthProgramA
{
   public static void main(String [] arg)
   {
      //1 - define any needed variables
      Scanner input = new Scanner(System.in);
      double miles, gallons, mpg;

      //2 - ask for the input and read it in
      System.out.println("How many miles driven?");
      miles = input.nextDouble();
      System.out.println("How many gallons used?");
      gallons = input.nextDouble();

      //3 - find the answer
      mpg = miles / gallons;

      //4 - show the answer
      System.out.println("Your car gets " + mpg + " miles per gallon.");
   }
}
```

When the program runs, if the user enters 250 for miles and 15 for gallons, the output would be
Your car gets 17.2 miles per gallon.

```
//in FourthProgramB.java, which will find the distance between two points
import java.util.*;
import java.io.*;
public class FourthProgramB
{
   public static void main(String [] arg)
   {     //1 - define any needed variables
         Scanner input = new Scanner(System.in);
         double x1, y1, x2, y2, dist;

         //2 - ask for the input and read it in
         System.out.println("Enter the first point x-coordinate");
         x1 = input.nextDouble();
         System.out.println("Enter the first point y-coordinate");
         y1 = input.nextDouble();
         System.out.println("Enter the second point x-coordinate");
         x2 = input.nextDouble();
         System.out.println("Enter the second point y-coordinate");
         y2 = input.nextDouble();

         //3 - find the answer
         dist = Math.sqrt(((x2-x1)*(x2-x1)) + ((y2-y1)*(y2-y1)));

         //4 - show the answer
         System.out.println("the distance is " + dist);
   }
}
```

The Math library has many useful functions, like:

```
Math.sqrt(x);          //returns the square root of x
Math.abs(x);           //returns the absolute value of x
Math.random();         //returns a random double between 0 inclusive and 1 exclusive
Math.sin(x);           //returns the sine of x (assumed that x is in radians)
Math.cos(x);
Math.tan(x);
Math.PI                //a constant value that returns PI (or a number really close to PI)
Math.min(x,y);         //returns the smaller of the two between x and y
Math.max(x,y);         //returns the larger of the two between x and y
```

Just like with a calculator, you will want to try to avoid the square root of a negative or dividing by zero.

Equality and inequality operators:

==	checks equality between two primitives
!=	checks to see if two primitives are not equal
<	checks to see if one primitive is less than another
>	
<=	checks to see if one primitive is less than or equal to another
>=	

Control structures - The if statement:

Used to make a block of code execute if a certain condition is true

```
if( /* a condition is true */)
{
   //execute the code here
}
```

The if-else statement:

Used to run one of two blocks of code depending on if a condition is true

```
if(/* a condition is true */)
{
   //runs the code here when the condition is true
}
else
{
   //runs the code here when the condition is false
}
```

The nested if statement - Used to run one of many blocks of code depending on many conditions

```
if(/* condition 1 is true */)
{
   //runs the code here when the first condition is true
}
else
   if(/* condition 2 is true */)
   {
      //runs the code here when the second condition is true
   }
   else
      if(/* condition 3 is true */)
      {
         //runs the code here when the third condition is true
      }
      else
      {
         //runs the code here when none of the conditions are true
      }
```

Note that if there is not a last-else statement at the end, there is a possibility that no code will run in the event that all the conditions are false.

```
//in FifthProgram.java, which will find the distance between two points
import java.util.*;
import java.io.*;
public class FifthProgram
{
   public static void main(String [] arg)
   {      //1 - define any needed variables
          Scanner input = new Scanner(System.in);
          double temp;

          //2 - ask for the input and read it in
          System.out.println("Enter the temperature in Fahrenheit");
          temp = input.nextDouble();

          //3 - find the answer (and in this case, that includes showing it at the same time)
          if(temp >= 90)
             System.out.println("Dress light - it is hot");
          else
              if(temp >= 70)
                  System.out.println("Dress light - it is warm");
              else
                  if(temp >= 50)
                     System.out.println("Dress regular");
                  else
                      System.out.println("Dress in layers - it is cold");
   }
}
```

If the user enters in the value **95**, the first condition will be true, and the program will output:

Dress light - it is hot

after which, it will skip the else (which contains the other if statements) and the program will end

If the user enters in the value **60**, the first and second conditions will be false, but the third condition will be true, and the program will output: **Dress regular**
after which, it will skip the else (which contains the last statement) and the program will end

If the user enters 28, all of the conditions will be false, and the program will default to the last else statement and output: **Dress in layers - it is cold**

Again, note that if the program did not have the last else statement and the user entered 28, the program would not have any output.

Commonly, begin and end braces are required to mark which block of code is to be executed for the if or else part of the structure. But if the code only consists of a single statement, the braces are not required. The code above could have been written as:

```
if(temp >= 90)
{
   System.out.println("Dress light - it is hot");
}
```

Control structures - the for loop:

A for loop is used to repeat a block of code a known number of times. A loop control variable is used to count how many times the loop repeats until a loop condition is no longer true.

```
for(int i=0; i < 5; i++)              //loop will repeat the loop body five times
{
   System.out.print("*");             //the System.out.print command will keep output on the same line
}
```

The variable i starts at zero, then it checks the condition. Since zero is < five, the loop body executes. Then i increments to the value one, and it checks the condition again. The loop will end once the condition is false. If i starts at zero, continues while i is < five and increases one at a time, then the loop body will repeat five times, and the output will be *********.

```
import java.util.*;
import java.io.*;
public class SixthProgram
{
   public static void main(String [] arg)
   {
         Scanner input = new Scanner(System.in);
         int num;

         System.out.println("Enter the number of stars you want");
         num = input.nextInt();

         for(int i=0; i<num; i++)
          {
            System.out.print("*");
         }
   }
}
```

If the user enters the value 8 when prompted, the program will output ************.

A for loop can go either direction, and the variable can increment in any way such that the condition eventually becomes false:

```
for(int i=0; i<=5; i++)      //will repeat 6 times because it starts at zero and ends at (and including) five.

for(int i=10; i > 4; i--)    //will repeat 6 times because it starts at ten and ends before reaching four.

for(int i=1; i<14; i+=2)     //will repeat 7 times because it starts at 1, ends before fourteen
                             //and i increases 2 at a time.

for(int i=0; i < 10; i--)    //an infinite loop, because i starts at zero, continues while i is < 10 and
                             //decreases by one each time, which will ALWASY be < 10.

for(int i=0; i > 10; i--)    //a dead loop, because i starts at zero and zero is not greater than ten.
```

Control structures - the while loop:
The while loop is a loop that is used to repeat a number of times that is unknown at compile time. It repeats a block of code while a condition is true.

```
while(/* condition is true */)
{
   //repeat the body of code here
}
```

```
int count = 0;
int num = 7842;
while(num > 0)          //while num has digits
{
   num /= 10;           //effectively knocks the least significant digit off of num, i.e. num = num / 10
   count++;             //add one to count
}
```

Since num starts at 7842, and 3842 is > 0, the loop body will execute.
num changes to 784 and count goes to 1.
Since 384 is > 0, the loop body executes again.
num changes to 78 and count goes to 2.
Since 38 is > 0, the loop body executes again.
num changes to 7 and count increases to 3.
Since 3 > 0, the loop body will execute one more time.
num changes to 0 and count increases to 4.
Zero is not > zero, so the loop ends.
count stores the number of digits in num.

```
public class SeventhProgram                    //assume import statements are added above
{
   public static void main(String [] arg)
   {
        Scanner input = new Scanner(System.in);
        double x, y, ans;
        int  again = 1;
        while (again == 1)                     //while the user wants to run the program again...
        {
            System.out.println("Enter a number");
            x = input.nextDouble();
            System.out.println("Enter another number");
            y = input.nextDouble();
            ans = x * y;
            System.out.println(x + " times " + y + " is " + ans);
            System.out.println("Enter 1 to run again, or 0 to quit");
            again = input.nextInt();           //allow the user to quit the program
        }
   }
}
```

Any for loop can be written as a while loop, but the compact nature of the for loop makes it more efficient when you know how many times a loop should repeat:

```
for(int i=0; i < 5; i++)          //first initialize i to its starting value (zero)
{                                 //while the condition is true (i < 5)
   System.out.print("*");         //complete the loop body and increment i (i++)
}
```

this loop is equivalent to...

```
int i = 0;                        //first initialize i to its starting value (zero)
while(i < 5)                      //while the condition is true (i < 5)
{
   System.out.println("*");       //complete the loop body
   i++;                           //increment i
}
```

Loops can also be called from inside of another loop. These are called nested loops.
The outside loop will repeat an inside loop a certain number of times.

```
for(int r=0; r<3; r++)            //the outside loop repeats the following three times...
{
   for(int c=0; c<4; c++)         //this loop writes four stars on the same line
   {
      System.out.print("*");
   }
   System.out.println();          //then goes down to the next line
}
```

A total of twelve stars will be written (4 stars on a line, repeated three times).
The output will be:

```
****
****
****
```

```
for(int r=3; r>0; r--)            //the outside loop repeats the following three times...
{
   for(int c=0; c<r; c++)         //this loop writes as many stars as is the state of the variable r
   {
      System.out.print("*");
   }
   System.out.println();          //then goes down to the next line
}
```

A total of six stars will be written (3 stars on a line, then 2 stars, then 1 star).
The output will be:

```
***
**
*
```

Nested loops are used to execute a lot of instructions with relatively little code.

Boolean operators: && (AND), || (OR) and ! (NOT)
Conditions can be composed using the three logical operators, &&, || and !.

The && operator (AND) only yields true when all conditions are true.
```
if(x >= 0 && y >= 0)
    System.out.println("DONE");
```
This will only output **DONE** in the case that x is positive AND y is positive.
If either or both conditions are false, the whole condition is false.

The || operator (OR) yields true if any condition is true.
```
if(x >= 0 || y >= 0)
    System.out.println("DONE");
```
This will output **DONE** in the case that x is positive or y is positive or both are positive.
Only if both conditions are false, the whole condition is false.

The ! operator (NOT) makes any boolean condition its opposite.
```
if(!word.equals("yes"))
    System.out.println("DONE");
```
This reads as: if it is **not** true that word equals "yes".
It will output **DONE** in the event that word is equal to anything except "yes".

&& has higher order of operation precedence over ||.
(A || B && C) is equivalent to (A || (B && C)).

Java will short circuit a boolean expression by ending a check once the result is known:

Given (A && B && C), it examines the conditions from left to right.
If A is false, it stops checking and returns false (because false && anything will yield false).
It will only examine the condition B if A is true.
If B is false, it stops checking and returns false.
It will only examine condition C if A is true and B is true. The result will then depend on C.

Given (A || B || C), it examines the conditions from left to right.
If A is true, it stops checking and returns true (because true || anything will yield true).
It will only examine the condition B if A is false.
If B is true, it stops checking and returns true.
It will only examine condition C if A is false and B is false. The result will then depend on C.

Demorgan's Theorem:
!(A && B) == !A || !B, !(A || B) == !A && !B
Think of this like distribution of a negative into a polynomial: every term becomes its opposite, but with Demorgan's theorem, an && will switch to an ||, or the || will switch to an &&.

!(x > 0 && y<=14) is equivalent to (x<=0 || y>14).
Both conditions become their opposite, and the && operator changed to an ||.
Remember that the opposite of (greater-than) is (less-than-or-equal-to), not (less-than).

!(num >= 0 || word.equals("hello")) is equivalent to (num < 0 && !word.equals("hello"))

Error checking user input:
You can use a while loop to trap invalid user input until it is received as valid.
Consider that invalid input would be nonsensical values, like a negative age or a height of zero, or values that could cause the program to throw an exception, like division by zero or square root of a negative.

The model is:
1) ask for and read in user input
2) while(the input is bad)
    {
      tell the user the input is bad
      ask for and read in user input again
    }

```
public class EighthProgram                          //assume import statements are added above
{
   public static void main(String [] arg)
   {
         Scanner input = new Scanner(System.in);
         double x, ans;
         int  again = 1;
         while (again == 1)                         //while the user wants to run the program again...
         {
            System.out.println("Enter a number to find the square root of:");
            x = input.nextDouble();
            while( x < 0)                           //we will stay in the loop until the input is valid
            {
               System.out.println("Negative values are invalid.");
               System.out.println("Enter a number to find the square root of:");
               x = input.nextDouble();
            }
            ans = Math.sqrt(x);
            System.out.println("The square root of " + x + " is " + ans);
            System.out.println("Enter 1 to run again, or 0 to quit");
            again = input.nextInt();                //allow the user to quit the program
         }
   }
}
```

Consider you are asking the user to pick one of many options:

```
int option = 0;
System.out.println("Pick an option (1, 2, 3, 4, or 5):");
option = input.nextInt();
while(option < 1 || option > 5)             //we will stay in the loop until the input is valid
{
   System.out.println("That option is invalid.");
   System.out.println("Pick an option (1, 2, 3, 4, or 5):");
   option = input.nextInt();
}
```

Methodizing:
When a complex task can be broken down into logical subtasks, they are written as methods that can streamline the code, be called many times, or reused in multiple applications. Static methods are subtasks in which there is only one version that does not require creating an instance of an object. They may return a value, like the way square root returns a number with a decimal. Or they may be a void method, which performs a task, but does not return a value. They may require parameters (arguments) to work , like the way square root requires that you send it a number, or they may not need any arguments sent.

```
public class NinthProgram                    //assume import statements are added above
{
   public static void showMenu()             //this method only shows options on the screen
   {                                         //so it does not need arguments or return a value
      System.out.println("type 1 to find the area of a right triangle");
      System.out.println("type 2 to find the perimeter of a right triangle");
      System.out.println("type 3 to find the area of a rectangle");
      System.out.println("type 4 to find the perimeter of a rectangle");
   }

   public static double findTriangleArea(double base, double height)
   {                                         //this method finds the area of a right triangle
      return 0.5*base*height;                //so it needs information to do its job (parameters)
   }                                         //and returns a value as a number with a decimal

   //assume similar methods called findTrianlgePerim, findRectangleArea and others are defined here
   public static void main(String [] arg)
   {
      Scanner input = new Scanner(System.in);
      double base, height, ans=0;
      int option;
      System.out.println("Enter the base:");
      base = input.nextDouble();             //assume error checking is done for the base here
      System.out.println("Enter the height:");
      height = input.nextDouble();           //assume similar error checking is done for the height
      showMenu();                            //calls the showMenu method defined above
      option = input.nextInt();              //assume error checking is done for the option here
      if(option == 1)
         ans = findTriangleArea(base, height);  //calls the methods defined above, and the returned
      else if(option == 2)                      //value is stored in the variable ans.
         ans = findTrianglePerim(base, height);
      else if(option == 3)
         ans = findRectangleArea(base, height);
      else if(option == 4)
         ans = findRectanglePerim(base, height);
      System.out.println("The solution you seek is: " + ans);
   }
}
```

Exercise: complete this as a working program with all methods and error checking defined, and test it.

Let's look more closely at some of these methods:

```
public static void showMenu()                          //this line is called the method header
{
   System.out.println("type 1 to find the area of a right triangle");
   System.out.println("type 2 to find the perimeter of a right triangle");
   System.out.println("type 3 to find the area of a rectangle");
}
```

This method merely writes information out to the screen. Therefore, it does not require that we send it any information as arguments in the parenthesis.
The result of performing the commands to show the user the available options, there is no part of the process that requires finding a particular solution, so it is defined as a void method. The term void means that the method will not return a value (like the main function).

If the main objective of a subtask is to find a solution of some kind, then it will be a return method. In the method header, the return type is declared before the method name.

```
public static double findTriangleArea(double base, double height)        //method header
{
   return 0.5*base*height;
}
```

Note that in order to find the area of a right triangle, we need to know some information to complete the task: the dimensions of the triangle. So we have parameters declared in the parenthesis to state what information is required to complete the task. The base and height might have decimals, so they are declared as type double (seen to the right of the method name). The solution that is returned will also have a decimal, so the return type is also declared as double (seen to the left of the method name).

Calling a void method only requires stating the method name, and sending it any needed arguments. Since showMenu does not require any arguments, all we have to do is type:

```
showMenu();
```

Calling a return method should usually be done in such a way that we use or can retrieve the value that is returned. Send it any needed arguments, and call it in an assignment statement, a println command (to write out the returned value) or in a condition.

For the method findTriangleArea, we need to send it two numbers in order for it to complete its task. Here are different ways of calling it such that we use the returned value in some way:

```
//calling a return method in a println
System.out.println( "Area is " + findTriangleArea(3, 4));        //will output Area is 6.0

//calling a return method in an assignment statement
double ans = findTriangleArea(3, 4);                             //returned value is saved in ans

//calling a return method in a condition
if (findTriangleArea(3, 4) < 10)
   System.out.println("Small triangle");
else
   System.out.println("Big triangle");                          //will output Small Triangle
```

Variable scope:

Variables only exist within the construct in which they are defined. If a variable is defined inside of a method, then it only exists within the method. If a variable is defined inside of a loop or if statement, then the variable no longer exists once the body of code for that construct ends. Consider this:

```
//within a method...
int x = 5;
/* int x = 3;          //WILL NOT COMPILE, because x is already defined!       */

//BUT, now consider the following:
if(/* condition is true */)
{
   int x = 5;                        //x begins its existence here
   System.out.println(x);
}                                    //x no longer exists when the if-body ends
else
{
   int x = 3;                        //we can define another variable called x,
   System.out.println(x);            //because the old one is gone
}                                    //x no longer exists when the else-body ends
```

It is because of variable scope that you can have disjoint for loops with the same loop variable name:

```
for(int r=0; r<3; r++)               //r begins to exist here
{
/*  for(int r=0; r<4; r++)           //WILL NOT COMPILE - r is already defined for the loop body*/
   {
      //some code here
   }
}
//BUT, the following would work
for(int r=0; r<3; r++)               //variable r begins to exist here
{
   //loop body
}                                    //variable r ends existing here
for(int r=0; r<4; r++)               //a new variable r begins existing here
{
   //loop body
}                                    //the new variable r ends existing here
```

If you define multiple variables in different methods with the same name, it is important to note that they are not the same variable.

```
public static void method1(int x)
{
   String word = "hello";
}
public static void method2(int x)          //this x is entirely different from the x in method1
{
   String word = "world";                  //this word has nothing to do with the word in method1
}
```

The String object:
The String is a data type that can store a collection of characters, like a word or sentence, and can perform actions that either reveal information about the string or return new strings.

Creating a string and assigning it to a literal value requires quotation marks.

```
String word = "hello";
System.out.println(word);        //will output hello
```

The Scanner object can read in a string in one of two ways:

```
Scanner input = new Scanner(System.in);
System.out.println("Enter a word:");
String word = input.next();
```

The input.next() method will wait for the user to enter text and press enter. It will return a string comprised of the characters the user typed in up to the first occurrence of a space or carriage return.

```
System.out.println("Enter a sentence:");
String word = input.nextLine();
```

There is another Scanner method called input.nextLine(), which will return a String that can include spaces within it. This is what you would use if you wanted the user to type in a sentence, or someone's full name.

String methods:

```
int length()                          //returns the number of characters in a String
int indexOf(String part)              //returns position of part in the string, returns -1 if not found
String substring(int start)           //returns a new String from index start to the end of the String
String substring(int start, int end)  //returns a new String from index start to index (end-1)
String toUpperCase()                  //returns a new String of all upper-case characters
String toLowerCase()                  //returns a new String of all lower-case characters
```

Calling a method from a string requires using dot-notation. It is important to note that once a string has been created, it can't be changed unless you assign it to a different String. For example:

```
String word = "hello";
System.out.println(word.toUpperCase());      //will output HELLO
System.out.println(word);                    //will output hello
```

The method .toUpperCase() merely returns a new String that is then sent to a println statement. The original String word is unchanged. If you wanted to change word to its upper-case version, you would need to reassign it to a new String:

```
String word = "hello";
word = word.toUpperCase();                   //word changes to its upper-case version
System.out.println(word);                    //will output HELLO
```

.indexOf(String part) //returns the position of part within the string, returns -1 if not found

Each character in a String is stored at a unique location called an index. The first character is stored at index 0 and the last character is stored at the String's length - 1.

```
        String word = "hello";
index:  0    1    2    3    4
word:   h    e    l    l    o
```

word.indexOf("h") returns 0, since the "h" is the first character in the String
word.indexOf("e") returns 1, since the "e" is the second character in the String
word.indexOf("o") returns 4, since "o" is the fifth character in the String
word.indexOf("z") returns -1, since "z" is not found in the String
word.indexOf("H") returns -1, since there is no capital "H" in the String (case sensitivity)
word.indexOf("hell") returns 0 because the substring "hell" is found starting at index 0 of the String
word.indexOf("llo") returns 2 because the substring "llo" is found starting at index 2 of the String

The indexOf method can be used as a search engine for a String, since it returns -1 if the argument is not found.

.substring(int start) //returns a new String from index start to the end of the String
.substring(int start, int end) //returns a new String from index start to index (end-1)

Consider the following String and each character's index

```
        String name = "Jefferson";
index:  0    1    2    3    4    5    6    7    8
name:   J    e    f    f    e    r    s    o    n
```

name.substring(3) returns "ferson" - the substring from index 3 all the way to the end
name.substring(3,7) returns "fers" -the substring from index 3 to index 6 (7-1)
name.substring(6) returns "son" -the substring from index 6 all the way to the end
name.substring(6,8) returns "so" -the substring from index 6 to index 7 (8-1)
name.substring(0) returns "Jefferson" -the substring from index 0 all the way to the end
name.substring(0,4) returns "Jeff" -the substring from index 0 to index 3 (4-1)

Note that for the two argument version of substring, we are going from index start (inclusive) to index end (exclusive), so effectively from index (start) to index (end-1).

name.substring(0, word.length() / 2) returns "Jeff" because name.length() is 9, and 9/2 is 4.
So we take the substring from index 0 to index 3.

name.substring(3, word.length() - 3) returns "fer", since name.length() is 9, and 9-3 is 6.
So we will have the substring from index 3 to index 5 (6-1).

int index = name.indexOf("er"); //"er" is found at index 4
name.substring(index) returns "erson" because index will store the state 4.
So we take the substring from index 4 all the way to the end of the String

name.substring(0, index) returns "Jeff", the substring from index 0 to index 3 (4-1).

Strings can be added together using the + operator. This is called concatenation.

```
String word1 = "ABC";
String word2 = "123";
String combined = word1 + word2;
System.out.println(combined);                    //will output ABC123
combined = combined.toLowerCase() + "!!!";       //change combined to its lower case version plus "!!!"
System.out.println(combined);                    //will output abc123!!!
```

```
//here is a program that will ask the user to enter their name in the format <first-name last-name>
//and output it in the form <last-name, first-name>
import java.util.*;
import java.io.*;
public class TenthProgram
{
   //pre:  name is in the format <first-name last-name>
   //post: returns the name in the format <last-name, first-name>
    public static String formatName(String name)
    {
       int index = name.indexOf(" ");                 //search for position of the space within the name
       String first = name.substring(0, index);   //from the 1st character to the last one before the space
       String last = name.substring(index+1);    //from the character after the space all the way to the end
       return last + "," + first;
    }

    public static void main(String [] arg)
    {
          Scanner input = new Scanner(System.in);
          String name, ans;
          System.out.println("Enter your first and last name separated by a space");
          name = input.nextLine();
          while (name.indexOf(" ") == -1)  //can't find a space in the name, so error check
          {
             System.out.println("Invalid input - no space found:");
             System.out.println("Enter your first and last name separated by a space");
             name = input.nextLine();
          }
          ans = formatName(name);
          System.out.println(ans);           //shows the name in the format of <last-name, first-name>
    }
}
```

A common error with the Scanner object occurs when you call the nextLine() method call after reading in a number with nextInt() or nextDouble(): it would appear to skip the nextLine() command. If this happens, just add an extra nextLine() command right before the nextLine() that is being skipped. It will clear the keyboard buffer and allow the String to be read in.

Casting primitives:
You can take most primitive types and return them in a different form, such as dropping the decimal from a double to get the whole number part, or forcing an int to be a double so that you can use a division operator as opposed to div (whole number division):

```
double num = 4.8;
System.out.println( (int)(num));          //casts 4.8 into an integer, which will output 4
```

Note that the example above did not round up:
Casting a double into an int will truncate the number, which effectively knocks off the decimal part.

```
int value = 9;
System.out.println( (double)(value));     //casts 9 into a double, which will output 9.0
```

```
int x = 3, y=6;
System.out.println( x / y);               //will output 0, because 6 goes into 3 zero times
System.out.println( (double)(x) / y);     //casts x into 3.0, so now this will divide and output 0.5
```

Using Math.random() :
Math.random() returns a double between 0 inclusive and 1 exclusive. So it can be as low as zero and as large as almost one, or 0.999999999

You can easily make a program flip a coin by checking to see what Math.random returns.

```
if(Math.random() < .5)                    //this will be true 50% of the time
   System.out.println("HEADS");
else
   System.out.println("TAILS");
```

You can shape a random value to get an integer between a range of values, say from min to max:
The algorithm is:

1) get a random number between 0 and almost 1.
2) multiply it by the range of values, or the number of random values you want.
   this will be (max - min + 1)
3) cast the resulting product into an integer to drop the decimal
4) add the min value to the result

This will give you a random integer between min and max inclusive.
As a single assignment statement, it would be:

```
int ran = (int)(Math.random() * (max - min + 1)) + min;
```

Examples:
the result of a single die roll would be: (int)(Math.random()*6) + 1
1 is our min and 6 is our max.

a random number between 0 and 9 would be: (int)(Math.random()*10)

a random number between 15 and 20 would be: (int)(Math.random()*6) + 15
Note that there are six numbers between 15 and 20 inclusive (20 - 15 + 1).

Arrays:
an array is a single entity that can store many values, each at its own unique index.
We have seen this before - a String object contains an array of characters.
Creating an array requires stating the type of values it will store, giving it a valid identifier name and stating how many elements we want to store.

```
int[] numbers = new int[5];       //creates an array called numbers that can store five integers.
double[]vals = new double[8];     //an array called vals that can store eight real numbers.
String[]words = new String[50];   //an array called words that can store 50 String objects.
```

It would be expected that we fill the arrays somewhere after assigning it.

You can also create an array and immediately assign values to it.

```
int[] nums = {5, 3, 7, 0, 9};          //nums stores six integers:  5, 3, 7, 0 and 9
String[] names = {"Bob", "Otto", "Anna"};
                                       //names stores three String objects:  Bob, Otto and Anna
```

Each element of the array can be accessed or changed by its index. Like a String, the first element is stored at index zero, and the last element is stored at the array's length - 1.

Each array has a data field that stores the number of elements called length.

```
int[] nums = new int[5];
nums[0] = 5;
nums[1] = 3;
nums[2] = 7;
nums[3] = 0;                           // note that the array could have been created this way:
nums[4] = 9                            // int[] nums = {5, 3, 7, 0, 9};
```

The array is visualized as:

index:	0	1	2	3	4
value:	5	3	7	0	9

```
System.out.println(nums[2]);               //would output 7, because the seven is stored at index two
System.out.println(nums[2] - nums[1]);     //would output 4, because 7 - 3 is four.
System.out.println(nums.length);           //would output 5, because there are five elements in the array
```

Note that calling the length data field for an array does not need parenthesis afterwards, where it does for a String object. The length for an array is a data field, where the length() of a String calls a method.

Traversing through every element of an array can be easily done with a for loop to access every index.

```
for(int i=0; i < nums.length; i++)         //i will traverse through each valid index of nums
   System.out.print(nums[i] + " ");        //will output 5 3 7 0 9
```

The print statement differs from the println statement in the following way:
A print statement will keep the cursor on the same line after it has completed its command, so multiple consecutive calls to the print statement will all be output on the same line.
A println statement will send the cursor to the next line after it has completed its command, so multiple consecutive calls to the println statement will result in each output being printed on its own line.

```
//here is a modified version of TenthProgram that will work for five names
public class TenthProgramB                          //assume needed import statements are included
{
   //pre:  name is formatted  <first-name last-name>
   //post: returns the name in the format <last-name, first-name>
    public static String formatName(String name)
    {
       int index = name.indexOf(" ");               //search for position of the space within the name
       String first = name.substring(0, index);   //from the 1st character to the last one before the space
       String last = name.substring(index+1);    //from the character after the space all the way to the end
       return last + "," + first;
    }

    //post:  fills array with user chosen names in the format <first-name last-name>
    public static void fillArray(String [] names)
    {
       Scanner input = new Scanner(System.in);
       for(int i=0; i<names.length; i++)              //note that this will work with an array of any size
       {
           System.out.println("Enter your first and last name separated by a space");
          names[i] = input.nextLine();
          while (names[i].indexOf(" ") == -1)         //can't find a space in the name, so error check
          {
              System.out.println("Invalid input - no space found:");
              System.out.println("Enter your first and last name separated by a space");
              names[i] = input.nextLine();
          }
       }
    }

    //post:  displays each element of array, one element per line
    public static void showArray(String [] array)
    {
       for(int i=0; i<array.length; i++)              //note that this method will work with an array of
           System.out.println(array[i]);              //any size, not just the one defined in the main function
    }

    public static void main(String [] arg)
    {
          String [] names = new String[5];           //An array of five Strings.
          fillArray(names);                          //Calls the method above to fill with user input.
          for(int i=0; i<names.length; i++)          //Traverse through each index of the names array and
              names[i] = formatName(names[i]);       //have each name change to its formatted version.
          showArray(names);                          //Calls the method above to show all array elements.
    }
}
//NOTE: if we wanted to change the program to work for an array of 8 names, we would only have to
//change one character in the main function (in defining the array size).
```

Two Dimensional Arrays:
You can store data in a row-column oriented chart using a two-dimensional array. Creating one is similar to a regular array, but you must specify the number of rows and number of columns.

```
int[][] nums = new int[3][5];          //creates a 3 x 5 array called nums that can store 15 integers.
double[][]vals = new double[8][8];     //vals  can store 64 real numbers in 8 rows and 8 columns.
String[][]words = new String[10][5];   //words that can store 50 Strings in 10 rows and 5 columns.
```

Accessing any 2-D array element requires first specifying the row index, then the column index.
Given a 2-D array called chart, the number of rows is specified by chart.length and the number of columns is returned by chart[0].length. Why? Because compositionally, a 2-D array is built as an array of arrays.

```
int [][] chart = new int[2][3];          //6 integers arranged in 2 rows and 3 columns
chart[0][0] = 5;
chart[0][1] = 9;                         //chart at row 0, col 1 is assigned the state nine
chart[0][2] = 4;
chart[1][0] = 7;                         //chart at row 1, col 0 is assigned the state seven
chart[1][1] = 0;
chart[1][2] = 6;
for(int r = 0; r < chart.length; r++)    //r traverses through each row index (chart.length will be 2)
{
   for(int c = 0; c < chart[0].length; c++) //c traverses through each column index (chart[0].length is 3)
   {
      System.out.print(chart[r][c] + " ");
   }                                     //would output:      5 9 4
   System.out.println();                 //                   7 0 6
}
```

Remember that the print statement keeps the output on the same line, so each row element will be displayed on a single line. After the inner loop completes showing every row element, the println statement is used to drop the cursor to the next line for the next row.

Note that we use a nested for loop to traverse through each row and each column.
Let's say we wanted to find the sum of all of the values in chart:

```
int sum = 0;
for(int r = 0; r < chart.length; r++)    //r traverses through each row index
{
   for(int c = 0; c < chart[0].length; c++) //c traverses through each column index
   {
      sum = sum + chart[r][c];           //add each array element to the previous state of sum
   }
}
System.out.println(sum);                 //would output 31, the sum of 5+9+4+7+0+6
```

Notice that the nested for loop used to show all of the array elements is the same as the one used to find the sum. The only difference is how each array element is handled.

Wrapper Classes:
An object, like the String, is an entity that can store information and perform methods that access or change the information they store. Again, consider that the String stores information as a collection of characters, but can perform methods that return information or even new Strings, like toUpperCase().

For each primitive type, there is an object version called a wrapper class. It gets its name because there is an object that "wraps around" a single primitive.

Integer is the object version of the primitive int.
Double (with a capital 'D') is the object version of a primitive double.
Boolean (with a capital 'B') is the object version of the primitive boolean.

the Integer object has the following methods and data fields:

```
int intValue()          //returns the int that is stored within the object
Integer.MIN_VALUE       // returns the minimum value represented by an int or Integer
Integer.MAX_VALUE       // maximum value represented by an int or Integer
```

the Double object has similar methods, like:

```
double doubleValue()    //returns the double that is stored within the object
```

You can create an instance of a wrapper object the following way:

```
Integer num1 = new Integer(3);          //creates an integer object called num1 that stores 3
Double num2 = new Double(3.5);          //creates an object called num2 that stores 3.5
System.out.println(num1. intValue()  + num2. doubleValue());     //will output 6.5
```

The ArrayList:
A standard array must be assigned a size when it is created.

There are occasions when you need a container to store multiple items, but the number of items you need is not known until run time. The user of the program might be in charge of adding and removing items, or an algorithm needs to store information as it finds it, but does not know how many items it may find until it is done.

In these situations, the ArrayList is a good option. The ArrayList is a single entity that can store multiple objects that will size itself as you add and remove items. You do not need to specify its size, and you do not need to worry about running out of room.

An ArrayList can't store primitive types: it can only store objects. So if you need to have an ArrayList of whole numbers or real numbers, just use the Integer object or the Double object. Here is how you can create different kinds of ArrayLists:

```
ArrayList <Integer> nums = new ArrayList();       //nums will be a collection of Integer objects.
ArrayList <String> words = new ArrayList();       //words can store a collection of String objects.
ArrayList <Double> values = new ArrayList();      //values can store a bunch of real numbers
```

The ArrayList object has the following methods available:
Consider that AnyType is whatever type you want your ArrayList to store, as defined.

```
int size()                      //returns the number of elements stored in the ArrayList
boolean add(AnyType x)          //adds x to the end of the list, size increases by one
void add(int i, AnyType x)      //adds x at index i in the list, size increases by one
AnyType remove(int i)           //removes and returns the element at index i, size decreases
AnyType get(int i)              //returns the element at index i, list is unchanged
AnyType set(int i, AnyType x)   //change the element at index i to x and return the old value
```

```
//here is a modified version of TenthProgram that will work for an unknown number of names
public class TenthProgramC                          //assume needed import statements are included
{
   //pre:  name is formatted  <first-name last-name>
   //post: returns the name in the format <last-name, first-name>
    public static String formatName(String name)
    {
       int index = name.indexOf(" ");               //search for position of the space within the name
       String first = name.substring(0, index);   //from the 1st character to the last one before the space
       String last = name.substring(index+1);    //from the character after the space all the way to the end
       return last + "," + first;
    }

    //post:  fills names with user chosen names in the format <first-name last-name>
    public static void fillArray(ArrayList <String> names)
    {
       Scanner input = new Scanner(System.in);
       String temp = "";                            //will store user input
       while (!temp.equals("0"))                    //end adding names when user inputs "0"
       {
           System.out.println("Enter your first and last name separated by a space, or 0 to stop");
          temp = input.nextLine();
           if(!temp.equals("0"))                    //only proceed if user did not just type "0" to stop
           {
              while (temp.indexOf(" ") == -1)       //can't find a space in the name, so error check
              {
                 System.out.println("Invalid input - no space found:");
                 System.out.println("Enter your first and last name separated by a space");
                 temp = input.nextLine();
              }
              names.add(temp);                      //add the name to the end of the ArrayList
          }
       }
    }

    public static void main(String [] arg)
    {
          ArrayList <String> names = new ArrayList();
          fillArray(names);                         //Calls the method above to fill with user input.
          for(int i=0; i<names.size(); i++)
             names.set(i, formatName(names.get(i)));
          System.out.println(names);                //shows all the elements of the ArrayList
    }
}
//NOTE  the line of code that says: names.set(i, formatName(names.get(i)));
//it is a bit cryptic, but start in the innermost parenthesis and work your way out:
//get the element in names at index i.  Then send it to the formatName method.
//now set the element at index i to the resulting String that is returned from formatName.
```

The Enhanced For Loop, aka the for-each loop:
Java supports a compact loop structure for traversing through an array or ArrayList called the enhanced for loop. It should only be used when you intend to traverse though each element of a container starting at the first element, and you do not intend on changing any elements in the list. With a for-each loop, you do not have direct access to the index. So if you require knowledge of the index of any element, it is best to use a traditional for loop. If you need to traverse in another direction, change list elements or start at an index other than zero, use a traditional for loop.

```
String [] names = new String[10];
//assume code is here to fill the array
//the following loop will show all the names in the array, one per line
for(String x : names)              //for each String element x within the array called names
    System.out.println(x);         //show each array element
```

As you can see, you do not have direct access to the index. There is no loop control variable to represent the index like we have with a traditional for loop. The String x that is defined must be the same data type as is stored in the array names, and will represent every array element from the first to the last.

```
ArrayList<Integer> nums = new ArrayList<Integer>();
//assume code is here to fill the list
//the following code will find the sum of all list elements
int sum = 0;
for(Integer x : nums)              //for each Integer element x within the array called nums
  sum = sum + x.intValue();        //add each array element to the sum
```

Note here that x is defined as an Integer object because we want to traverse through every element of a list of Integers. Compare the code above to its traditional for loop version:

```
int sum = 0;
for(int i=0; i < nums.size(); i++)
    sum = sum + nums.get(i).intValue();
```

You can see that for this case, and likely most cases where the enhanced for loop is appropriate, our enhanced for loop is more compact and easier on the eyes.

Let's say we want to search in an ArrayList for a particular value and report its index if found. In that case, a traditional for loop will be better because it has direct access to each element's index.

Let's say we want to traverse through an array of Strings and remove any element that has a particular value. Again, in this case we should use a traditional for loop because we intend on altering the array.

If we want to traverse though an array backwards, or only access elements at an odd index, the traditional for loop is the one to use.

But often times, we want to start at the beginning and go through the entire list until the end. If we do not need to change the array elements, the enhanced for loop is very convenient.

Creating objects:
When you want to define a new kind of object, you need to consider what information you want it to store (data fields) and what abilities you want it to have (methods). Consider that we want a Car object that stores the car's name and price tag:

```
//in Car.java
public class Car
{
   private String name;                    //data fields
   private double price;

   public Car(String n, double p)          //METHOD: constructor
   {
      name = n;
      price = p;
   }

   public String toString()                //METHOD
   {
      return "MODEL:" + name + "     PRICE TAG: $" + price;
   }

   public String getName()                 //METHOD: accessor
   {
      return name;
   }

   public double getPrice()                //METHOD: accessor
   {
      return price;
   }

   public void setPrice(double p)          //METHOD: mutator
   {
      price = p;
   }
}
```

The visibility modifier called **private** means that the data fields name and price are only directly accessible from within Car.java. No other program has access to them.

The **constructor** method is called when we create an instance of a Car, and it assigns starting values to the data fields. For example, in some other program within the same folder, we might do this:

```
Car coup = new Car("Civic", 18000);
```

This will call the constructor, sending the String "Civic" in for the name, and the number 18000 in for the data field price.

The **toString** method is used to return information we want to see about the Car object as a String. It is called automatically when we send an instance of a Car to a print or println statement.

```
System.out.println(coup);          //will output MODEL: Civic     PRICE TAG: $18000
```

Accessor methods are used to return a data field that is private in the object's definition. The methods getPrice() and getName() are accessor methods for the Car object. If we want to see a Car's name or price from some driver program outside of Car.java, we can't access them directly outside of the class definition. But we can call an accessor method that returns the value we want:

```
        Car suv = new Car("CRV", 24000);
        System.out.println(suv);                    //will output MODEL: CRV     PRICE TAG: $24000
/*      System.out.println(suv.price);    WILL NOT COMPILE, because price is private to Car.java */
        System.out.println(suv.getPrice());         //will output 24000
```

Mutator methods are used to change a data field that is private for the object. The method setPrice(x) is a mutator method for the Car object. Let's say we want to put a car on sale and drop its price.

```
        System.out.println(suv);                    //will output MODEL: CRV     PRICE TAG: $24000
/*      suv.price = 22800;                WILL NOT COMPILE, because price is private to Car.java */
        suv.setPrice(22800);
        System.out.println(suv);                    //will output MODEL: CRV     PRICE TAG: $22800
```

Note that there is not a mutator method for the name of the car. Why? It is a simple design decision: it seems more likely that we might change the price of a car after it has been created, but certainly not likely that we would change its name.

```
public class EleventhProgram     //assume this is in the same folder as Car.java
{
   public static void main(String [] arg)
   {
        //create three instances of Car objects
        Car pickup = new Car("F150", 26000);
        Car suv = new Car("CRV", 24000);
        Car coup = new Car("Civic", 18000);
                                                    //will output
        System.out.println("INVENTORY:");           //INVENTORY
        System.out.println(pickup);                 //MODEL: F150   PRICE TAG: $26000
        System.out.println(suv);                    //MODEL: CRV    PRICE TAG: $24000
        System.out.println(coup);                   //MODEL: Civic  PRICE TAG: $18000

        System.out.println("All SUVs price drop 5%");   //All SUVs price drop 5%
        suv.setPrice(suv.getPrice() * 0.95);
        System.out.println(suv);                    //MODEL: CRV    PRICE TAG: $22800
   }
}
```

Notice the line that says: suv.setPrice(suv.getPrice() * 0.95);
We go to the inner most parenthesis first and call suv.getPrice(). This returns 24000, which is then multiplied by 0.95. The result which is 22800 is then sent to the setPrice method to change the price of the suv.

Subclasses and inheritance:
From any base class, you may define a subclass that inherits all data fields and concrete methods from the base class by using the reserved word **extends**. The only items that are not inherited from a base class are constructors and any abstract methods (which we will get to later).

```
//in the same folder as Car.java
public class RentalCar extends Car          //a RentalCar now has all features and abilities of a Car
{
   private double pricePerDay;              //a new data field in addition to the ones inherited

   public RentalCar(String n, double p, double ppd)
   {
      super(n, p);                          //calls the constructor for Car and sets the name and price
      pricePerDay = ppd;                    //initializes our extra data field
   }

   public String toString()                 //override the toString that is inherited with a new version
   {
      return super.toString()+ "     RENTAL FEE: $" + pricePerDay;
   }

   public double getPricePerDay()           //a new accessor method in addition to the ones inherited
   {
      return pricePerDay;
   }

   public void setPricePerDay(double ppd)
   {                                        //a new mutator method in addition to the ones inherited
      pricePerDay = ppd;
   }
}
```

When we say that a RentalCar extends Car, it means that a RentalCar is a Car. It now inherits the data fields name and price, as well as the methods getPrice, setPrice, getName and toString (which the RentalCar will override with its own version).

In the constructor for the RentalCar, you see the reserved word **super**, which is a call to the base class Car. When invoked in the RentalCar's constructor, it calls the constructor for the Car and sets its name and price data fields to the values of the first two arguments sent into the RentalCar's constructor.

We also see the term super used in the RentalCar's version of toString. When a subclass has a method with the exact same header as one that it inherits from the base class, the subclass version will **override** the one that it inherits. The command super.toString() calls the toString method from the base class Car, which returns the Cars name and price. Then we add onto that String the cost of our rental car's price per day.

We do not inherit constructors. It is expected that when you create a new type of object that you define a constructor for it.

If we wanted, we could make a subclass of RentalCar, which would inherit all properties and abilities from the RentalCar and the Car.

```
public class EleventhProgramB   //assume this is in the same folder as Car.java and RentalCar.java
{
   public static void main(String [] arg)
   {
         Car suv = new Car("CRV", 24000);
         RentalCar coup = new RentalCar("Civic", 18000, 24.5);
                                          //will output
         System.out.println(suv);         //MODEL: F150    PRICE TAG: $26000
         System.out.println(coup);        //MODEL: Civic   PRICE TAG: $18000   RENTAL FEE: $24.5

          System.out.println(suv.getPrice());              //will output 24000
/*       System.out.println(suv.getPricePerDay());         WILL NOT COMPILE:  suv is not a RentalCar */
         System.out.println(coup.getPrice());             //will output 18000
         System.out.println(coup.getPricePerDay());       //will output 24.5
   }
}
```

Note that the coup is a Car AND a RentalCar, but the suv is just a Car object. So we may call the RentalCar method getPricePerDay() from the coup, but not from the suv.

Interfaces and **Comparable**:
An interface is a collection of method headers for which there is no body of code. When an object definition **implements** a particular interface, it is expected that the object will then define all of the interface methods. When unrelated objects implement the same interface, then there is a nice consistency between the objects, in that you can expect that they have similarly named methods.

One of the more common Java interfaces is the Comparable interface, which looks like the following:

```
public interface Comparable
{
   //returns a positive number if this is greater than x
   //returns zero if this is equal to x
   //returns a negative number if this is less than x
   public int compareTo(Object x);
}
```

For any object that implements the Comparable interface, you must define and give a body of code to the method compareTo. Consider that different types of objects might compare to one another in different ways: Cars might compare by price, people might compare by age, airplanes might compare by top speed. Each object type can define compareTo so that it makes sense for that kind of object.

Any argument sent to a method can be declared as an interface type, meaning that it can be any type of object that implements that interface. Consider the following:

```
public static void sort(Comparable [] array)
```

For the sort method, it is expecting an array of any kind of object, as long as it implements Comparable.

```
//here is a new version of the Car object
public class Car implements Comparable
{                                                   //we now must define the method compareTo
   private String name;                             //data fields
   private double price;

   public Car(String n, double p)                   //METHOD: constructor
   {
      name = n;
      price = p;
   }
   public String toString()                         //METHOD
   {
      return "MODEL:" + name + "     PRICE TAG: $" + price;
   }
   public String getName()                          //METHOD: accessor
   {
      return name;
   }
   public double getPrice()                         //METHOD: accessor
   {
      return price;
   }
   public void setPrice(double p)                   //METHOD: mutator
   {
      price = p;
   }
   public int compareTo(Object x)                   //mandated by implementing Comparable
   {
      Car other = (Car)(x);
      if(price < other.getPrice())                  //the object calling compareTo is less than x
         return -1;
      if(price > other.getPrice())                  //the object calling compareTo is greater than x
         return 1;
      return 0;                                     //the object calling compareTo and x are equal
   }
}
```

Note that the first line in compareTo creates an object of type Car called other, which gets the state of x, but cast into a Car object. This is required in that the argument x must be of type **Object**, but we want to make sure that it is, in fact, a Car and will have a getPrice method.

What is the class **Object**? Consider that every object created in java is understood to be a subclass of the highest base class called Object. It contains methods for the following:

```
String toString()
boolean equals(Object x)
```

So every class definition that you create already has a toString and equals method. The idea is that you override those methods to make more sense for your particular class if you need them.

Once the base class Car is Comparable, now any subclass of Car will inherit that ability automatically, so the RentalCar is now also Comparable (unless we want to override the method so that RentalCars compare by rental rate instead of price).

```
public class EleventhProgramC              //assume in the same folder as the new Car and RentalCar
{
   public static void main(String[] arg)
   {
      Car sedan = new Car("Town Car", 45000);
      RentalCar minivan = new RentalCar("Caravan", 38000, 42);

      if(sedan.compareTo(minivan) > 0)                    //if sedan is greater than minivan
         System.out.println(sedan.getName() + " is more expensive than " +minivan.getName());
      else
         if(sedan.compareTo(minivan) < 0)                 //if sedan is less than minivan
            System.out.println(minivan.getName() + " is more expensive than " + sedan.getName());
         else
            System.out.println(sedan.getName() + " and " + minivan.getName() + " are the same price");
   }
}                                          //will output Town Car is more expensive than Caravan
```

Consider the following method:

```
//post:  returns the index of the largest element in the array
public static int findGreatest(Comparable [] array)
{
   int maxIndex = 0;
   for(int i = 0; i < array.length;  i++)
      if(array[i].compareTo(array[maxIndex]) > 0)
         maxIndex = i;
   return i;
}
```

Consider that we now have a Car object that is Comparable. The String object is also Comparable, where Strings are compared to one another by their alphabetic order. The wrapper class Integer is also Comparable, where Integers compare to one another by their numeric values. The findGreatest method will work with an array of Cars, an array of Strings, an array of Integers or an array of ANY kind of object that implements Comparable.

```
Car [] fleet = {new Car("A", 90000), new Car("B", 50000), new Car("C", 20000)};
Integer [] nums = {new Integer(3), new Integer(5), new Integer(7)};
String [] words = {"hello", "world", "exclamation mark"};

System.out.println(findGreatest(fleet));     //will output 0, because the most expensive car is at index 0
System.out.println(findGreatest(nums));    //will output 2, because the largest value is at index 2
System.out.println(findGreatest(words));  //will output 1, because the word furthest down the alphabet
                                           //is at index 1
```

You can now see the consistency between Cars, Strings and Integers: they are all Comparable, and we need only one version of the method findGreatest to work with an array of any kind of Comparables.

Abstract classes:
There may be a case when a base class needs to mandate that all subclasses have a certain method to define, but that method is too vague to give a body of code in the base class. When you define a method header for which there is no body of code, it is called an **abstract method**.

Consider a base class for a SchoolEmployee, which has data fields for name and salary. Concrete methods would include constructors, accessor and mutator methods and toString. It should be true that every SchoolEmployee also has a method called work, since that is what an employee is expected to do.

Now if we define subclasses for Teacher and Administrator, we could easily imagine what the work method might do for each: when a Teacher works, they write lesson plans, grade papers and lecture. When an Administrator works, they conduct meetings, make phone calls and fill out paperwork. So each subclass could define the work method concretely.

But what should the work method do for the base class SchoolEmployee?
The answer is: "I don't know". It is true that every SchoolEmployee should be able to work, but the contents of the method is too ambiguous for the base class. So it will be an abstract method in the base class, which means that it will have a method header with no code body.

Once that is done, the SchoolEmployee will be an abstract class. The rules are:
* You can't create an instance of an abstract object. It is there to define the common data fields and methods for the subclasses to inherit from.
* For every subclass of an abstract base class, you inherit all data fields and concrete methods (except constructors), and you MUST define any method that is abstract in the base class, giving it a code body.

abstract SchoolEmployee

data fields:	name, salary
concrete methods:	toString(), getName(), setName(x) getSalary(), setSalary(x)
abstract methods:	public abstract void work();

subclass Administrator

inherits: name, salary, toString(),
getName(), setName(x),
getSalary(), setSalary(x)
must define: public void work()

subclass Teacher

data fields: department
inherits: name, salary, toString(),
getName(), setName(x),
getSalary(), setSalary(x)
must define: public void work()

```
public abstract class SchoolEmployee                //in SchoolEmployee.java
{
    private String name;
    private double salary;

    public SchoolEmployee(String n, double s)
    {
        name = n;
        salary = s;
    }

    public abstract void work();

    public String toString()
    {
        return "NAME:" + name + "     SALARY: $" + salary;
    }
    public String getName()
    {
        return name;
    }
    public void setName(String n)
    {
        name = n;
    }
    public double getSalary()
    {
        return salary;
    }
    public void setSalary(double s)
    {
        salary = s;
    }
}

public class Administrator extends SchoolEmployee        //in Staff.java, same folder as SchoolEmployee
{
    public Administrator(String n, double s)
    {
        super(n, s);                                      //calls constructor from SchoolEmployee
    }

    public void work()
    {
        System.out.println(getName() + ": Meeting, phone calls, paperwork, repeat...");
    }
}
```

```
public class Teacher extends SchoolEmployee     //in Teacher.java, in same folder as SchoolEmployee
{
   private String department;

   public Teacher(String n, double s, String d)
   {
      super(n, s);                                  //calls constructor from SchoolEmployee
      department = d;
   }

   public void work()
   {
      System.out.println(getName() + ": Plan, lecture, grade, repeat...");
   }

   public String toString()
   {                                                //calls toString from SchoolEmployee
      return super.toString() + "     DEPARTMENT: " + department;
   }

   public String getDepartment()
   {
      return department;
   }
}
```

The Teacher and Administrator both inherit the data fields name and salary. They also inherit the methods getName, setName, getSalary, setSalary and toString (but the Teacher overrides toString with its own version). The mandate is that both the Teacher and Administrator MUST define the work() method concretely, which they do but in different ways.

```
public class TwelfthProgram
{
   public static void main(String[]args)
   {
      SchoolEmployee one = new Administrator("Higgins", 90000);
      SchoolEmployee two = new Teacher("Geeves", 40000, "English");
      SchoolEmployee three = new Teacher("Coleman", 40000, "Math");
    /*SchoolEmployee four = new SchoolEmployee("Ruprect", 60000);    WILL NOT COMPILE!!! */

      one.work();         //will output    Higgins: Meeting, phone calls, paperwork, repeat...
      two.work();         //               Geeves: Plan, lecture, grade, repeat...
      three.work();       //               Coleman: Plan, lecture, grade, repeat...
   }
}
```

Note that we can't create an instance of a SchoolEmployee and call its constructor. Why? A SchoolEmployee is abstract, and you can't create an instance of an abstract object. What would happen if we could create an instance of a SchoolEmployee, and then directed them to work?

Polymorphism:
Consider that with the abstract base class SchoolEmployee and the subclasses Administrator and Teacher, we have two classes with a similarly named method (work) that does something different between the two subclasses: The Administrator's work method makes them go to meetings, make phone calls, do paperwork and repeat. The Teacher's work method makes them plan, lecture and grade.

Now if we were to know that an instance of an object was guaranteed to be some kind of SchoolEmployee, what would happen if we told that object to work? The answer is...

"I don't know: are they an Administrator or a Teacher?"

This is a working example of **polymorphism**: a method call that is ambiguous at compile time, and doesn't become concrete until run time. We need to know which subclass a SchoolEmployee is in order to know whether it is going to call meetings or do lesson plans.

```
public class TwelfthProgramB
{
   //post:  make all employees in the school work
   public static void schoolDay(SchoolEmployee [] array)
   {
      for(int i=0; i<array.length; i++)
         array[i].work();            //POLYMORPHISM:  which version of work is being called here?
   }
   public static void main(String[]args)
   {
      SchoolEmployee [] staff = new SchoolEmployee[3];
      staff[0] = new Administrator("Higgins", 90000);
      staff[1] = new Teacher("Geeves", 40000, "English");
      staff[2] = new Teacher("Coleman", 40000, "Math");
     schoolDay(staff);             //will output    Higgins: Meeting, phone calls, paperwork, repeat...
   }                               //               Geeves: Plan, lecture, grade, repeat...
}                                  //               Coleman: Plan, lecture, grade, repeat...
```

With the program above, look at the schoolDay method. It is sent an array of SchoolEmployees, of which some are Administrators and some are Teachers. When we have the line of code that says array[i].work(), there is no way of knowing which version of work is being called here until we know if that particular element of the array is an Administrator or a Teacher.

We can see from the main function that the array at index zero happens to be an Administrator. So back in the schoolDay method's for loop, if the variable i is storing the state zero, then the work method will hold meetings, make phone calls, etc. But in the main function, we see that index one and two in the array happen to be Teacher objects. So in the schoolDay method's for loop, if the variable i is storing the state one or two, then the work method will plan, lecture and grade.

So on its own, the line of code that says array[i].work() is completely ambiguous until you know which of the subclasses that element of the array has to be. The method call is ambiguous at compile time, and made concrete at run time.

Recursion:
There are certain algorithms that are best defined by referencing themselves in a more simple state. The technique of having a method call itself inside of itself is called recursion. While recursion achieves loop-like behavior, there are some tasks that can only be done recursively, or can best be done recursively. In most cases, methods that might be designed recursively can also be done with a loop.

Consider factorial, where 5 factorial equates to 5 * 4 * 3 * 2 * 1, and by definition, 0 factorial equates to 1. It can also be said that 5 factorial equates to 5 * (4 factorial), since 4 factorial is 4 * 3 * 2 * 1. So, we can define n factorial as n * (n - 1 factorial), where 0 factorial is 1.

5 factorial = 5 * (4 factorial). We cannot complete the arithmetic until we find 4 factorial...
4 factorial = 4 * (3 factorial). We can't multiply here until we find 3 factorial...
3 factorial = 3 * (2 factorial). Now we must find 2 factorial...
2 factorial = 2 * (1 factorial). On to find 1 factorial...
1 factorial = 1 * (0 factorial). Almost done...
0 factorial = 1, by definition. So, now we can compute 1 factorial...
1 factorial = 1 * (1) = 1. Now we can find 2 factorial...
2 factorial = 2 * (1) = 2. Now we can find 3 factorial...
3 factorial = 3 * (2) = 6. Now on to computer 4 factorial...
4 factorial = 4 * (6) = 24. And we can now finally find 5 factorial...
5 factorial = 5 * (24) = 120.

Consider the definition as a Java method:

```
//pre:   n >=0
//post: returns the factorial of n
public static long fact(int n)       //a long is like an int, but it can store larger values
{
   if(n == 0)                        //terminating case
      return 1;
   return n * fact(n - 1);           //recursive call
}
```

We first see what is called the **terminating case**: this is the condition that tells the recursion when to stop. Consider that it handles the most simple possible input that you can send to the method. Which number can you find the factorial of which would require the least amount of work? Zero, because by definition, the factorial of zero is one. If a recursive method does not have a terminating case, it will theoretically call itself forever, but realistically throw a stack overthrow exception when Java runs out of memory.
Next we see the **recursive call**. Note that the method fact calls itself inside of itself, but sends to itself the input that is one-step more simple than what was originally sent. If we try to find fact(5), consider what we could send to the method such that it is one step more simple than sending the method 5: The next easiest step would be fact(4). Now we just need to define fact(5) by calling fact(4): as we saw, fact(5) is 5 * fact(4). So, fact(n) is n * fact(n-1). Look familiar.
It is important to note that the recursive call must get us one step closer to the terminating case. If it does not, the method will call itself until Java runs out of memory with a stack overflow. So, if the method ends when we send it zero, then it would be safe to say that we will eventually get there by continually subtracting one from a positive number.

Recursive methods can come in the form of void methods as well. Consider the following:

```
public static void mystery(int n)
{
   if(n <= 0)                          //terminating case
      System.out.println("done");
   else
   {
      mystery(n-1);                    //recursive call
      System.out.print(n);
   }
}
```

Note that the method will end as soon as we send it a number that is zero or less. If we call the method recursively by sending it one less than its current value, the method will always eventually terminate, so we don't have to worry about a stack overthrow.

Let's assume we call mystery and send it the value 3. Replace all values of n with 3 and see what happens: mystery(3)

Since 3 is not the terminating case, we go to the else and have two commands to complete: mystery(2) and print(3). Remember that we must complete mystery(2) before we can print(3), but mystery(2) will require more work...

Replace all values of n with 2 and see what happens:

mystery(2)

Since 2 is not the terminating case, we go to the else and have to complete mystery(1) and print(2). Remember that we must do them in that order, and mystery (1) will require more work...

Replace all values of n with 1 and see what happens:

mystery(1)

Since 1 is not the terminating case, we go to the else and must complete mystery(0) and print(1).

mystery(0)

This is the terminating case, which will println("done") to the screen. Since we just completed the command mystery(0), we can now complete the print(1) command. That means we just completed mystery(1), which means we can execute the print(2) command. With that done, we completed mystery(2), and can now execute the print(3) command, thus finally completing mystery(3).

The method outputs **done123**

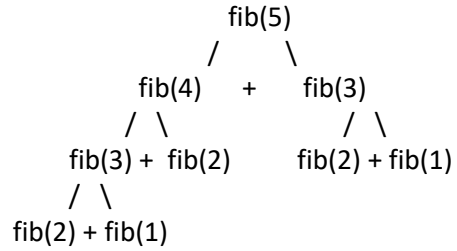
```
mystery(3)                                  //must finish mystery(2) before we print(3)
        mystery(2)                          //must finish mystery(1) before we print(2)
        |       mystery(1)                 //must finish mystery(0) before we print(1)
        |       |       mystery(0)         //terminating case
        |       |       |       print(done)  //first we write done -> mystery(0) is completed
        |       |       print(1)            //then we can write 1-> mystery(1) is completed
        |       print(2)                    //now we can write 2-> mystery(2) is completed
        print(3)                            //we can now write 3-> mystery(3) is completed
```

When there is code after a recursive call, the statements are saved in a stack, in that the last command called is the first one to complete. Imagine a stack of books where, when you add a book to the stack, you add it to the top. When you remove a book from the stack, you remove it from the top. So we added the commands in the order print(3), then print(2), then print(1), then print(done), but they are processed in the reverse order in which they were added: print(done), then print(1), then print(2), and finally print(3).

Recursion has its weaknesses: it tends to use more memory than equivalent methods done with loops, because of the recursive method calls that are saved in stack memory. It is also very easy to write a recursive algorithm that is horrifically inefficient: if a method calls itself more than once in the same case and the input sent recursively changes by addition or subtraction, then the method will have an efficiency that can be described as exponential. The number of times the method calls itself will grow exponentially as the input size grows. Consider a method that will find the nth term in the Fibonacci Sequence, where $f_1 = 1$, $f_2 = 1$, $f_3 = 2$, $f_4 = 3$, $f_5 = 5$, $f_6 = 8$, $f_7 = 13$, $f_8 = 21$,...,$f_n = f_{n-1} + f_{n-2}$. So if the first and second numbers in the sequence start at 1, then the n^{th} term in the sequence is the sum of the two terms before it.

```
//pre:   n>=1
//post: finds the nth term in the Fibonacci sequence
public static long fib(int n)        //a long is like an int, but it can store larger values
{
   if(n==1 || n==2)                  //terminating case:  1st and 2nd terms are 1
      return 1;
   return fib(n-1) + fib(n-2);       //recursive call
}
```

The method looks harmless, but consider that when the method is called, if it is not the terminating case it will call itself two more times...and each of those might call themselves two more times...etc.



Let's say you run a for loop to find the first 100 terms in the Fibonacci sequence:

```
for(int i=1; i<=100; i++)
   System.out.println("Term " + i + " is " + fib(i));
```

On most computers, it will find the first 20 terms in nanoseconds. But each next term will take twice as long as the one before it. Let's assume that going from the 40^{th} to the 41^{st} term takes a second. That means that the 42^{nd} term will take 2 seconds, and the 43^{rd} term will take 4 seconds. Extrapolate that single second for the 41^{st} term all the way to the 100^{th} term, and you have 2^{60} seconds, which is over 30 billion years. That is older than the age of the observable universe, and well past the live expectancy of our solar system...all for a three line method. This is to be avoided at all costs, so if you are considering similar logic for an algorithm, just use loops. A version that finds the n^{th} term in the Fibonacci sequence using a for loop will find the 100^{th} term in nanoseconds.

Steps to Object Orientation

Unit One - JKarel

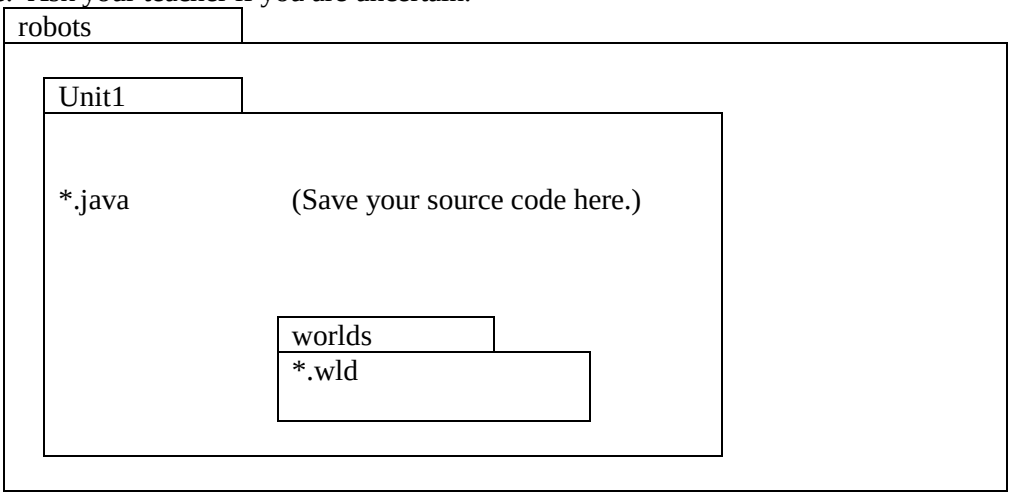
July 2016

Developed by Shane Torbert
and edited by Marion Billington
under the direction of Jerry Berry
with additions by Rev. Dr. Douglas Oberle

Thomas Jefferson High School for Science and Technology
Fairfax County Public Schools, Fairfax, Virginia

Discussion
File Management and Java basics

Java requires you to organize your files carefully. Your personal folder might be on a network drive, or a thumb drive. Ask your teacher if you are uncertain.



The programming language Java is a powerful one. It is a language that can be interpreted to work with just about every type of mainstream operating system and device you can find. That means the same program that is written on a PC can also work on a Mac with few to no changes.

Java is also a language that is object oriented. In older languages, most programs could manage and store information in primitive variables, like integers, real numbers, characters and Booleans (which can only store the value true or false). For complex programs, like a flight simulator, each airplane would be modeled by a large collection of primatives to keep track of the pitch, roll, yaw, airspeed, groundspeed, position in 3-d space, etc. Keeping track of and maintaining them was tedious.

Object orientation, a key feature of Java, allows a programmer to design an object, which is a single entity that can store several pieces of information (data fields) as well as perform actions that alter or access the information (methods). We will start with the Robot object, which store the data of its position (street and avenue, as integers), direction (NORTH, EAST, SOUTH or WEST) and the number of items they are holding (called beepers, also stored as an integer). A basic Robot has the ability to move (forward one block), turnLeft (90 degrees), pickBeeper (pick up a beeper) and putBeeper (drop a beeper). Your first lab, Lab00, will show you how to create your first program and what it looks like to create and command a Robot object.

JKarel is the Java version of Karel the Robot, originally designed by Richard Pattis as an educational tool for beginning programmers, like you. All of your JKarel programs will be written in Java, but the JKarel classes are not designed for commercial production. These classes exist solely to provide you, the student, with a gentle introduction to object-oriented programming.

Why "Karel"? According to Karel++: A Gentle Introduction to the Art of Object-Oriented Programming by Joseph Bergin, Mark Stehlik, Jim Roberts, and Richard Pattis, on page one:

> 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.

Setting up:

You will copy the **CS/Robots** folder onto your network drive:

1) On the desktop, double click on the folder called **CS/Quarter 1**

2) Single click on the folder called **Robots** to highlight it.

3) In the top menu bar, select **Organize**, then **Copy**.

4) On the desktop, double click on **My Computer**.

5) Under **Network Drives**, double click on the drive that contains your **student ID**.

6) When navigated in to your network drive, select **Organize**, then **Paste** to copy the **Robots** folder there. Right-Click on the folder, go to **Properties**, and un-check (disable) the **Read-Only** box.

Starting a project:

You will create a new Java file on your network drive with the name **Lab00.java**:

1) On the desktop, double click on the **jGrasp** icon.

2) Once **jGrasp** has loaded, go to the top menu bar and select **File** -> **New** -> **Java**.

3) In the editing window that pops up (called **[Grasp 1]**), type the following:

```
public class Lab00
{

}
```

4) In the top menu bar, select **File** -> **Save As**. Look in the **File Name:** field in the window that pops up. You should see that it already has the name **Lab00.java**. If not, change it to match the exact name that follows public class in the editor window (case sensitive).

5) In the **Look In:** field, navigate to your network folder, then double click into the **Robots** folder.

6) Click the **Save** button so that **Lab00.java** is saved on your network drive in your **Robots** folder.

Writing a program:

- Type in your program. For every brace, bracket and parenthesis that you type, make sure that its opposite directional twin is there.

- In the top icons, the blue-lined page next to the red arrow is **Generate CSD**: this will find major errors and line up the program to make it easier to read. It will only work if every begin parenthesis/ brace has a matching end parenthesis/brace.

- The green + icon is **Compile**: this will find minor errors and tell you what and where they are in the message window.
- The red-running-man icon is Run: this will execute your program if there are no errors as a result of compiling.

Lab00

Hello Robot

Objective
Classes, objects, and sending Robot commands. Program structure.

Background
A nice text editor for a beginning student in a Windows environment is JGrasp, which is free and available at http://www.jgrasp.org/.

Specification
Go to File, New, Java file. **Create** Robots\Lab00.java. Enter the source code shown below, then CSD, compile, and run.

```
//in Lab00.java
//author, name, date
public class Lab00
{
   public static void main(String[]arg)
   {
        World.readWorld("first");
        World.setSize(10, 10);
        World.setSpeed(6);

        //create an instance of a Robot

        Robot karel = new Robot();
        karel.move();
        karel.pickBeeper();
        karel.move();
        karel.turnLeft();
        karel.move();
        karel.putBeeper();
        karel.move();
        karel.turnLeft();
        karel.turnLeft();
   }
}
```

Line 1: For comments use //. Comments are ignored and do not affect the program.

The class and file name must match.

public class Lab00 MUST be in Lab00.java.

Applications have a main function. The statements execute in order.

The compiler ignores whitespace.

Our object named karel is an instance of the Robot class.

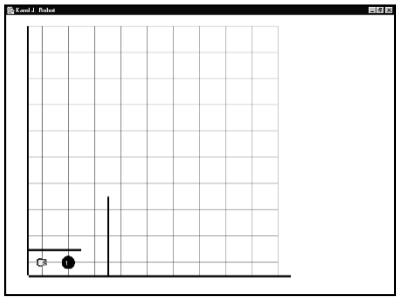
Call instance methods using an object's name—always a lowercase letter.

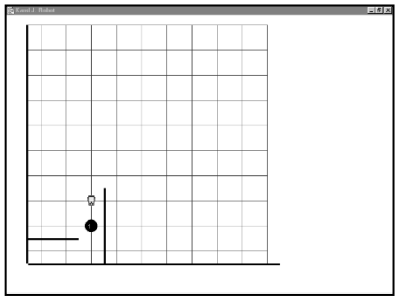
Method calls end with parenthesis.

Complete statements end with semi-colon ;

A command window and a graphics window open to display the program.

Sample Run

Start: 

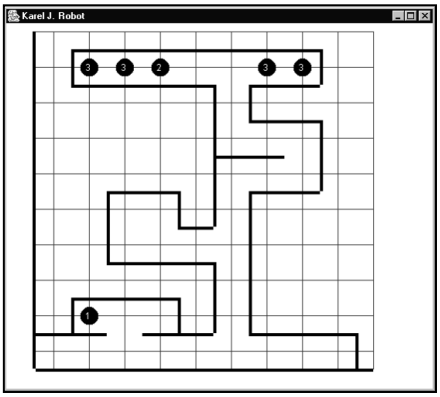
End: 

Lab01
Student and Teacher

Objective

Program structure. World commands. Robot commands.

Background



Worlds are divided into streets and avenues. Streets run west to east, or left to right. Avenues run south to north, or down to up.

Streets are numbered on the vertical axis starting with 1st Street, avenues on the horizontal axis starting with 1st Avenue. The corner at the bottom-left of the graphics window is (1st Avenue, 1st Street), or (1, 1).

Worlds create the context for solving robot problems. Pre-defined worlds are stored in the folder Robots\worlds.

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 must begin with a letter. 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.

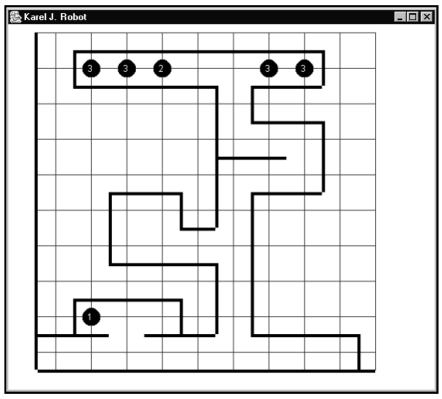
Once the `new` operator creates a robot object, that object can perform robot methods using *dot-notation*. For instance, a robot named pete will move forward with the command pete.move();. The identifier pete, before the dot, is a the name of an object and the identifier move, after the dot, is the name of a method.

Specification

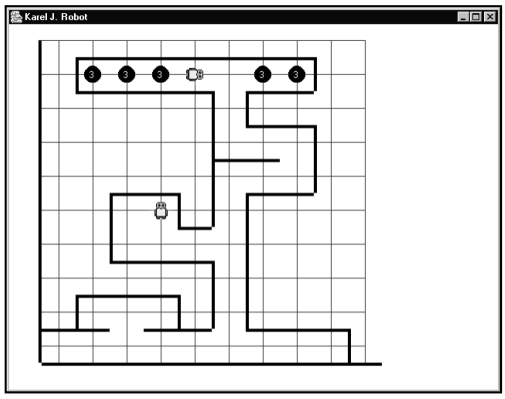
Create Robots\Lab01.java with the "school" world at size 10x10. Declare two Robot objects, one named lisa using the default constructor and the other named pete starting at (4, 5) facing south with zero beepers. Have lisa pick up the beeper from the math office and bring it to pete. Have pete take the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

Sample Run

start

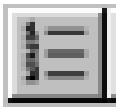


finish



Discussion

Errors and program format.



Number the Lines (pcGRASP)

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 message will tell you on which line your error occurs; using the line numbering feature in pcGRASP should make it easy to find out where you went wrong.)

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. jGrasp 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.

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. Also, every time you make a begin brace {, create its closing end brace } such that the CSD can still be generated.

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 red (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.

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 she pasted into the UNIVAC I logbook and used as a basis for the term "bug."

Java programs that you intend to run are often called *driver* programs. A driver program has a main function that includes the code that executes in order when you run the program. The main function always looks like this:

```
public class Lab99    //this would be in a file called Lab99.java
{
   public static void main(String[] arg)
   {
      //code that runs here
   }
}
```

Discussion – constructors and inheritance

Creating an instance of a Robot can be done in one of two ways. Calling the default constructor is done by sending no information into the constructor:

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

A default robot starts at (1, 1), facing east with no beepers.

A single equals (=) is the *assignment operator*. It assigns to the reference karel a new robot object.

You can also create an instance of a Robot by sending four pieces of information into the constructor, in the order of location, direction and number of beepers:

```
Robot hal = new Robot(6, 3, World.WEST, 10);
```

The robot hal will start at position (6, 3), facing west with 10 beepers. This kind of constructor can only be called by sending it 4 pieces of information in the expected order. Consider that the following robot instance will NOT compile:

```
Robot boo = new Robot(3, 4);    //DOES NOT COMPILE
```

You may have noticed that standard robots do not know how to turn right. You can easily accomplish the same by turning left three times. In the next lab (Lab02), you will create a new subclass of Robot that DOES know how to turn right. This will use a feature of object orientation called *inheritance*.

Let's say you wanted a new kind of robot that knows how to spin (turn 360 degrees). You can create a new class that is a Robot, but can do more than the standard Robot: we will call it a Spinner. The keyword *extends* will make the Spinner inherit all properties and abilities from the Robot. We will only need to define a constructor and our new ability, spin.

```
//in Spinner.java_____________________________________________________________________
public class Spinner extends Robot
{
   public Spinner()          //default constructor
   {
      super();                //this calls the constructor from the base class Robot, and starts a Spinner
   }                          // at (1,1) facing EAST with no beepers.

   public void spin()        //our new ability that will be available to every Spinner, but not Robots
   {
      turnLeft();    turnLeft();
      turnLeft();    turnLeft();
   }
}_____________________________________________________________________________________
```

This defines the DNA for what it means to be a Spinner. This is not a program that you would run, because there is no main function. Also note that for the turnLeft commands, you do not have to specify the name of an instance or use the dot-operator (like karel.turnLeft()). This is because the DNA for a Spinner has no idea of what the instance name will be. It might be karel, pete, lisa, or any identifier name that follows the rules of being alpha-numeric, starting with a letter and not reserved. Consider that in the main function of a driver program (like Lab00.java), we could now create an instance of a Spinner the same way we created an instance of a Robot:

```
Robot karel = new Robot();
Robot issac = new Spinner();
karel.move();
issac.move();    //issac can do everything a Robot can do, because it is a Robot an inherits its abilities
issac.spin();     //issac can also spin, which would make him turn 4 times.  Karel can not spin.
```

Lab02
Escape the Maze

Objective

Inheritance. Defining instance methods.

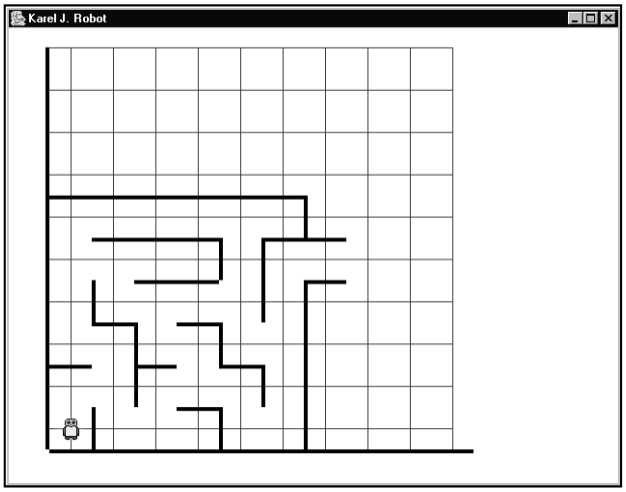
Background

Escaping the maze shown below involves both left turns and right turns. Since our robots only know how to turn left we will define another type of object (athlete) in a separate class (Athlete) that knows how to turn both left and right, and 180 degrees around. The structure of Athlete.java is shown.

```
//this already exists in Athlete.java
//so don't create it - just complete it.
public class Athlete extends Robot
{
 public Athlete()
 {
  super(1,1,World.NORTH,World.INFINITY);
 }
 public Athlete(int x,int y,int d,int b)
 {
  super(x,y,d,b);
 }

 public void turnAround()
 {
    //complete this method
 }

 public void turnRight()
 {
    //complete this method
 }
}
```



The keyword `extends` means that an athlete is a 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.

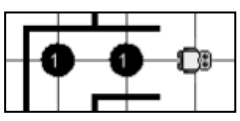
Athlete is not a program even though it is a class. No specific object is mentioned by name because you are creating a general resource that will be useful to a wide range of applications. Java uses the same keyword `class` to stand for both resources and applications.

Specification

Load Robots\Athlete.java. Complete 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 Robots\Lab02.java with "maze" world at size 8x8. Use one athlete object 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



Lab03
Take the Field

Objective

Passing an object as an argument. Class methods.

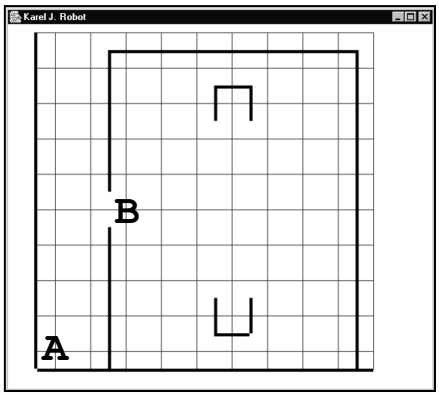
Background

An *instance method* is invoked on an object. A *class method* is invoked without an object.

Turning right is an ability that will be useful to a wide-range of applications. Therefore, turning right is appropriately defined as an instance method within the Athlete class. All the athletes on our sports team must go from the locker room (Point A) to the field (Point B), but athletes in general do not have to perform this task. Therefore, taking the field is not an appropriate instance method for the Athlete class.

Since we have to tell six different objects to take the field, we would like to write some kind of method so that we don't have to re-write the same code six times. The answer is a class method.

```
public class Lab03
{
  public static void takeTheField(Athlete arg)
  {
      arg.move();
      arg.move();
      arg.move();
      arg.move();
      arg.turnRight();
      arg.move();
      arg.move();
  }
  //You must define main.
}
```



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

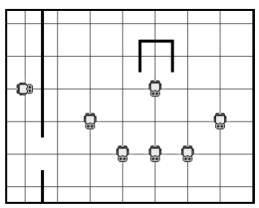
```
takeTheField(maria);
takeTheField(gary);
```

Notice that we do NOT say `maria.takeTheField()`. This is because `takeTheField()` is not part of an Athlete object. It is in the Lab03 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 Robots\Lab03.java with "arena" world at size 10x10. Create six athletes starting in the locker room; use the default constructor. Create one object outside the locker room to represent the coach. Have your team get positioned for the start of the game as shown below.

Sample Run



Discussion Methods

Methods that are defined as part of a class definition (a new data type like Athlete) become a part of every object defined of that type. You do NOT specify an instance of that class within the definition, because you are defining what EVERY Athlete can do – not just one with a specific name like karel. Ex //inside Athlete.java

```
public class Athlete
{
  …
  public void turnAround()
  {
    turnLeft();                    //notice that we don't say karel.turnLeft() because
    turnLeft();                    //there is no karel object defined here.  Instead, an
  }                                //object called karel can be defined as an Athlete
  …                                //in a main function somewhere.

}

//inside Lab02.java and within public class Lab02…
public static void main(String args[])
{
  …
  Athlete karel = new Athlete();
  Athlete chump = new Athlete();
  karel.turnAround();              //both karel and chump have their own turnLeft()
  chump.turnAround();              //and turnAround() methods built inside of them.
  …
}
```

What's the advantage of defining a method? It saves a lot of space. Instead of writing three turnLeft() commands every time we want to turn right, just define a turnRight() method that will do that for you.

In addition to that, methods allow you to break a large task into smaller and more logical sub tasks. If you need to write a program that reads in a list of numbers from a file and calculates the mean, median and mode, then you can write a method to get the input, methods for each of the three calculations and a method to show the results out on the screen.

Likewise, methods keep you from having to reinvent the wheel. There is no built in square root method in java. But so many people need and use it that it was written as a part of a library of math tools. Once you have the square root method available, all you have to do is call it:

```
answer = Math.sqrt(16);          //answer will store the value 4.0
```

Methods that are defined in the program with the main function are not designed as part of a new data type. They are just a definition of a sub task that can be called many times within the main function or other methods. They sometimes require that you send them input (parameters / arguments) in order to work. These methods are called **static** because there will only be one instance made of them (unlike an Athletes turnAround() which will have as many instances as Athlete objects that you define in the main function).

Discussion cont. Methods

Ex. //inside Lab03.java and within public class Lab03…

```
public static void takeTheField(Athlete arg)        //you do NOT need to send the
{                                                   //method an Athlete called "arg".
  arg.move();                                       //You just need to send an Athlete.
  arg.move();
  …                    //whatever the name of the Athlete that is sent in is the object
  arg.turnRight();     //that will be doing these actions.
  …
}

public static void main(String args[])
{
  Athlete karel = new Athlete();
  Athlete chump = new Athlete();
  …
  takeTheField(karel);      //karel goes to the method above and performs the tasks inside.
  takeTheField(chump);      //chump gets send to the method.  Wherever the method refers
  …                         //to "arg" it will be using chump this time.
}
```

Notice the difference in the way you call methods that are part of an object and methods that are **static**. Since every Athlete has turnRight() built into the class, we say karel.turnRight(). But takeTheField is NOT a part of the object, so we say takeTheField(karel).

If you have a bunch of commands that must be performed in a program and you can think of that set of commands as a sub-task of the greater design, then that sub-task should be written as a method.

If you want EVERY object of a certain type to have that ability built into it, then you define it as a part of the class definition
(like for an Athlete, **public void** turnRight()).

If that subtask is something that only needs to be done in a certain program, then you define it only for that program as a **static** method
(like **public static void** takeTheField(Athlete arg)).

Lab04 Climb Every Mountain

Objective Constructors.

Background

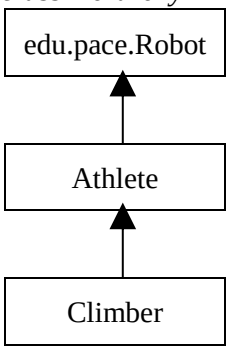
A *constructor* is a method that creates an object and initializes its private data.

One specialized type of athlete is the climber. A climber-robot explores mountains, has adventures, and, if lucky, finds lost treasure. Climbers always start on 1st Street facing north with one beeper, but they may begin on any avenue—depending on the terrain. Their beeper is used to mark the base camp so we'll know when we've made it home safely. Put the beeper down at the start and pick it up at the end.

In order to specify which avenue to start on, the Climber class must define a constructor. There is a standard format for defining a constructor for a class.

```
public class Climber extends Athlete
{
  public Climber(int x)
  {
    super(x, 1, World.NORTH, 1);
  }
  public void climbUpRight()
  {
    //pseudocode: tL, m, m, tR, m – (up 2 and over 1)
  }
  //You must define three other instance methods –
  //climbDownRight(), climbUpLeft() and climbDownLeft().
}
```

class hierarchy



A constructor is tagged `public` if we want people to be able to create objects of this class, which we do. A constructor is a special method and is NOT tagged with the keyword `void`. The name of a constructor always matches the class name. The Climber constructor takes one integer argument specifying the avenue to start on. The keyword `super` indicates that we are calling the superclass's constructor; in this case, that means we are calling Athlete's constructor. Note that we must pass four arguments to Athlete's constructor; three of these are pre-determined but x will vary.

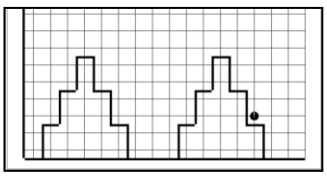
Specification

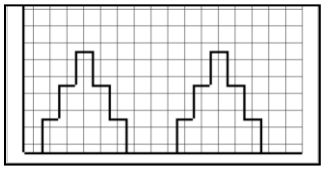
Consult the JKarel API for more information regarding the Climber class.

Create Robots\Climber.java. Implement the Climber class, then compile. Define the climbing methods assuming that the robot will start facing the direction that it is intending to climb up or down. You may also assume that it is directly in front of the mountain side it is going to climb up or down.

Create Robots\Lab04.java with "mountain" world at size 16 x 16. Create a climber starting at (8, 1). Find the treasure (beeper) and bring it back to (8, 1).

Test Data

Start: 

End: 

Discussion - Iteration

A process is made up of various statements. Any time you perform a single process repeatedly you are using iteration. In Java you can repeat a process using the `for`-loop.

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

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

```
for(int i=1; i<=3; i++)
  pete.pickBeeper();
```

This loop will cause `pete` to pick up a beeper three times (once when *i* has each of the values 1, 2, and 3).

Note: The braces are optional if there is only one task to be repeated.

This table shows the values that the `int`-type variable *i* takes on in the example on the left:

Value of *i*	*i* <= 6
1	true
2	true
3	true
4	true
5	true
6	true
7	false

The boolean-expression evaluates to false only when *i* reaches seven. The loop continues to repeat until this condition is false. Thus, the value of *i* is seven when the loop actually stops.

Watch for differences in conditional statements between <, >, <= and >=. Likewise, a loop that starts at a lower number and ends on a higher number should have a loop control variable that gets larger (i++, i=i+2, etc). A loop that starts at a higher number and ends on a lower number should have a variable that gets smaller (i--, i=i-2, i=i/2, etc).

Using the incorrect inequality signs with respect to the increment statement could lead to an infinite loop.

Note: for(int i=**2**; i**<=18**; i++) works because **2** is **<= 18** and i gets larger (i++).
for(int i=**10**; i**>0**; i--) works because **10** is **> 0** and i gets smaller (i--).
for(int i=1; i<=8; i--) is an infinite loop because i gets further away from 8.
for(int i=**1**; i**>10**; i++) is a loop that will not run at all because 1 is not > 10 .

Lab05
Shuttle Run

Objective

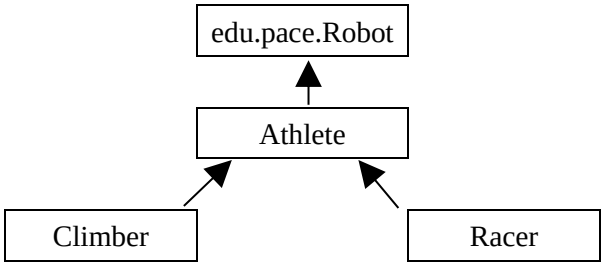
Control structure: for-loop.

Background

A *control structure* is a programming tool that alters the normal sequence of command execution.

In the class hierarchy, or inheritance hierarchy, shown to 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



Another name for superclass is *base class*. Another name for subclass is *derived 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.

Racers always start on 1st Avenue facing east with zero beepers, but they may begin on any street. We must define a constructor to specify the starting street (shown below).

Again, Racer's constructor calls Athlete's constructor, supplying all four arguments. Three of these arguments are pre-determined but y will vary.

Athlete's constructor will in turn call Robot's constructor. In order to create a racer object we must call the constructor of each of its superclasses. The *root* is the class at the top of any hierarchy; to create an object, each constructor all the way up to the root's must be invoked—but Java automatically goes up the hierarchy, invoking each as it goes.

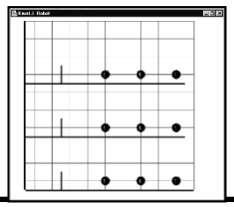
```
public class Racer extends Athlete
{
  public Racer(int y)
  {
    super(1, y, World.EAST, 0);
  }
  public void jumpRight()
  {
    //pseudocode: tL, m, tR, m, tR, m, tL
  }
  //Also define jumpLeft() and sprint(int n).
  //read the discussion below for sprint.
}
```

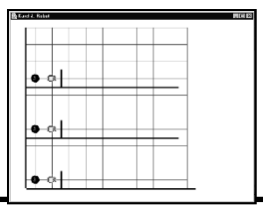
Specification

Consult the JKarel API for more information regarding the Racer class. **Create** Robots\Racer.java. Implement the Racer class, then compile.

Create Robots\Lab05.java with "shuttle" world at size 10x10. Create three racers to compete in a modified shuttle run hurdle race. Each beeper must be brought **individually** back to the racer's starting point and at the end each racer must step away from the pile to confirm that all three beepers have been retrieved.

Sample Run

Start: 

End: 

Hint Use a static class method.

Discussion

Formal argument versus actual arguments

The *formal argument* is the variable that appears in a method's header. The *actual argument* is the value passed when a method is called. The formal argument is sometimes called the *parameter* and the actual arguments are sometimes simply called the *arguments*.

Let's take a peek inside the World class:

```
package edu.pace;
public class World
{
  public static void setSize(int x, int y)
  {
    //The body consists of the definition.
  }
  //Other class methods are defined here.  There are no instance methods.
}
```

The formal arguments of the method setSize are the two integer variables x and y. The values of x and y depend upon the actual arguments.

Call to setSize	Value of x	Value of y
`World.setSize(8, 8);`	8	8
`World.setSize(10, 12);`	10	12

Java will pass the first argument to x and the second argument to y. The parameters take on whatever values are passed by the actual method call. The same is true of the method sprint().

```
public class Racer extends Robot
{
  //The constructor is defined here.
  public void sprint(int n)
  {
    for(int k = 1; k <= n; k++)
      move();
  }
  //Other instance methods are defined here.  There are no class methods.
}
```

Sprint uses iteration to repeat the move command n times. For instance, the following commands will cause a Racer named pete to move forward 10, 20, and 50 blocks, respectively:

Call to sprint	Value of n
`pete.sprint(10);`	10
`pete.sprint(20);`	20
`pete.sprint(50);`	50

The value of n affects the expression k <= n, thus determining the number of times the for-loop repeats.

Discussion
Loops and the If Statement

Any for-loop can be written as a while-loop. The for-loop is usually used for definite loops.

```
for(int i = 1; i <= n; i++)
{
  karel.turnLeft();
}
```

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

The command ***i++*** causes the value of the variable ***i*** to increase by one. If ***i*** was one, it is now two; if ***i*** was two, it is now three; and so on. Plus-plus (++), the *unary increment operator*, gave us the name C++.

The while-loop is used for indefinite loops, where you don't know beforehand exactly how many times to loop. This loop makes a robot pick up a pile of beepers, even if it doesn't know how many beepers are in the pile:

```
while(karel.onABeeper())
{
  karel.pickBeeper();
}
```

Similarly:

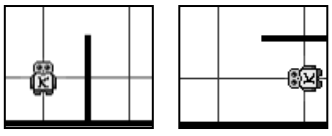
```
if(karel.onABeeper())
{
  karel.pickBeeper();
}
```

This if-statement will cause karel to pick up only one beeper and only if he is currently standing next to one. It is impossible for either of these segments to crash even if karel is on an empty corner.

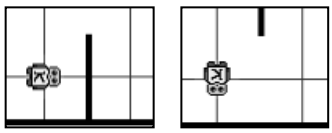
There are two types of methods, void types and return types. Void types take action and change the situation. Return types provide information about a robot's situation. Since there are different kinds of information, return types come in several varieties. One variety returns *boolean* values, either true or false. Think of booleans as answering yes-no questions. An example of a boolean method is onABeeper().

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



The exclamation mark (!) is the *negation operator* and follows the rules not-true is false and not-false is true. When you see an expression like !karel.frontIsClear(), say "not karel dot frontIsClear."

Warning

There is no such thing as a while-else structure. The commands that follow a while-loop execute whenever the loop ends (no else is required). Also, there is no such thing as an if-loop. An if-statement checks its condition only once.

Constructor Summary	
`Robot`()	`Robot`(int avenue,int street,int direction,int beepers)

Here are all the methods that come with every Robot – the bold return types show you which ones you must know and will use.

Method Summary	
-> boolean	`hasBeepers`() **//returns true or false as to whether or not the Robot is carrying any beepers**********
boolean	`areYouHere`(int street,int avenue) //returns true or false as to if a Robot is standing in a particular location**********
int	`avenue`() //returns the avenue number that a Robot is standing on**********
int	`direction`() //returns the direction that a Robot is currently facing**********
boolean	`facingEast`() //returns true or false as to whether or not the Robot is facing EAST**********
boolean	`facingNorth`()//returns true or false as to whether or not the Robot is facing NORTH**********
boolean	`facingSouth`()//returns true or false as to whether or not the Robot is facing SOUTH**********
boolean	`facingWest`()//returns true or false as to whether or not the Robot is facing WEST**********
-> boolean	`frontIsClear`()**//returns true or false as to whether or not the Robot has an open space in front of it (false if blocked)**
boolean	`leftIsClear`()//returns true or false as to whether or not the Robot has an open space to the left of it**********
->void	`move`() **//moves the Robot forward one space**********
-> boolean	`onABeeper`() **//returns true or false as to whether or not the Robot is standing on top of a beeper**********
-> boolean	`onARobot`()**//returns true or false as to whether or not the Robot is occupying the same space as another Robot****
-> **void**	`pickBeeper`() **//the Robot will pick up a beeper that it is standing on. If no beeper is there, an exception is thrown**
-> void	`putBeeper`() **//the Robot drops a beeper that it is carrying. If it has no beepers to drop, an exception is thrown*****
void	`restoreInitialState`() //the Robot resets to the state created by its constructor**********
-> boolean	`rightIsClear`()/**returns true or false as to whether or not the Robot has an open space to the right of it**********
int	`street`() //returns the street number that a Robot is standing on**********
String	`toString`()//allows for a String that contains the Robot's state to be written to the message window**********
-> void	`turnLeft`()**//the Robot turns in place to the left 90 degrees**********
INFO HERE->	Exclamation marks (!) will make a boolean method its opposite, as in !frontIsClear() means NOT frontIsClear()->front is blocked

Lab06
A Half-Dozen Tasks

Objective

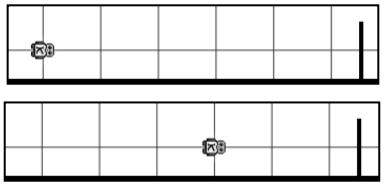
Control structures: while-loop and if-statement.

Background

The solution to task number three is:

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

Before



After

The name `temp` is short for temporary. This loop will repeat an unknown number of times, so long as temp's path is not blocked. The screen captures 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-worlds, all of which have the same basic structure. Thus, your program will not contain the command:

```
World.readWorld("tasks1");
```

because your program cannot be fully tested with only the tasks1 world. Instead you must import the java.io.* class and use the commands:

Row	Task
1	Go to 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.

```
String filename = JOptionPane.showInputDialog("What robot world?");
World.readWorld(filename);
```

When your program runs, an input dialog box will open prompting you for the name of the world to use. Run your program three times using the "tasks1", "tasks2", and "tasks3" worlds in turn. Your program does not work unless it runs successfully for all three of these worlds.

Specification

Consult the JKarel API for more information regarding boolean methods of the Robot class.

Load Robots\Lab06.java. Fill in the empty static void methods to accomplish all six tasks. The same code must work with all three worlds "tasks1", "tasks2", and "tasks3".

Extension

Create a valid robot-world for this lab using WorldBuilder.jar and test your program with that world.

You will find the WorldBuilder in the worlds folder. Save your world in the same folder with the extension .wld (i.e. Robots/worlds/tasks4.wld).

Lab07
Exploration

Objective

Polymorphism.

Background

Some other variations on creating an object:

`Robot karel = new Robot();`	Both reference and object created at once.
`Robot lisa;` `lisa = new Robot();`	The reference is created first and then the object is created.
`Robot pete = null;` `pete = new Robot();`	The reference is created and initialized to null. Then the object is created.
`Robot maria = null;` `maria = new Athlete();`	The reference is created and initialized to null. Then a subclass object is created.

The keyword null means the absence of an object, so null cannot execute any commands. If you try to call a method using a reference that points to null you will receive a null-pointer exception because no object exists to perform the method.

But why would we want a Robot pointer to point to an Athlete? Why would we want a superclass reference to a subclass object, i.e., *polymorphism*? Because it is powerful. We can write one, general code, but by passing subclass objects, which behave slightly differently, we can solve many different problems. You are going to define two subclasses of Climber, each designed to climb a different type of mountain. Each subclass will *override* the four instance methods defined in Climber.

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

Climber karel will execute the climbUpRight() command as it is defined in the HillClimber class. In the actual program, we will call explore(), passing karel as an argument.

Specification

Load Mountain.java and study its method explore().

Create Robots\HillClimber.java and Robots\StepClimber.java, both as subclasses of a Climber. Recall that a Climber will climb up or down two spaces and over one space (up 2 and over 1). Redefine the climbing methods for each subclass such that a StepClimber will climb up or down one space and over one space (up 1 and over 1). A HillClimber will climb up or down one space and over two spaces (up 1 and over 2).

Load Robots\Lab07.java. Set the size to 17x15. The program will prompt you to specify the world, the type of climber, and the starting position as follows:

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

Discussion
Polymorphism

The formal argument to the method Mountain.explore is a reference of type Climber. In Lab07, when an actual argument is passed to Mountain.explore, we can send it any kind of Climber (but not any kind of Robot).

```
public class Mountain    //in Mountain.java
{
  public static void explore(Climber arg)
  { //somewhere in here, arg is asked to climbUpRight() }
}_____________________________________________________________
public class Lab07       //in Lab07.java
{
  public static void main(String[] args)
  {
      explore( new Climber(y) );
  }
}
```

Consider that we have a few subclasses of Climber as well: the StepClimber and the HillClimber. As climbers, they all have the method climbUpRight, but they all perform that task in a different way. Climbers go up 2 and over 1, StepClimbers go up 1 and over 1 and HillClimbers go up 1 and over 2.

So in Lab07, can we do this?

```
public class Lab07
{
  public static void main(String[] args)
  {
      explore( new HillClimber(y) );
  }
}
```

Is a HillClimber a legal argument to the method explore? Yes, it is legal – HillClimbers ARE Climbers, as are StepClimbers. Thus we could also pass a StepClimber to explore, but we could not pass an Athlete. Since somewhere in explore, we ask arg to climbUpRight, any kind of Climber is fine because they ALL know how to climbUpRight. But regular Robots, Athletes and Racers do not.

Polymorphism is a frightening word that means something quite simple. Consider Mountain.java's explore method:

```
public static void explore(Climber arg)
{     //...some code here
      arg.climbUpRight();
      //…more code here
}
```

At the time this was coded, the programmer can't quite know whether arg will go up 2 and over 1, up 1 and over 1 or up 1 and over 2. Why? Because arg might be a Climber, might be a StepClimber or might be a HillClimber. The code arg.climbUpRight would do something different depending on what *kind* of climber was sent to the method. That is an example of polymorphism: code is ambiguous until you know what kind of object is sent to a method and the code is executed at run-time.

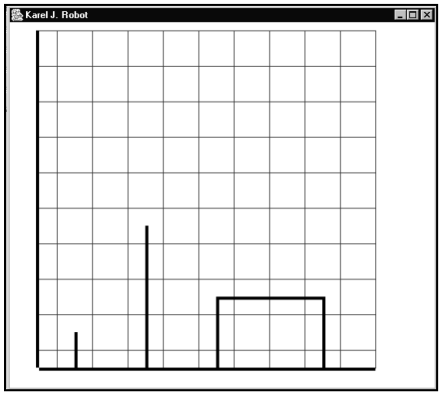
Lab08
Hurdle Racing

Objective

Polymorphism.

Background

Three types of hurdles:



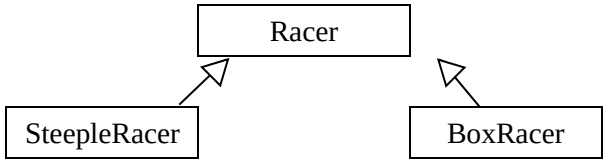
Imagine a hurdle race consisting entirely of regular, one-block tall hurdles with a beeper marking the finish line. An object of your Racer class can traverse that race-course using the algorithm:

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

If there's no hurdle move, otherwise jump over the hurdle. The else clause is executed when the condition frontIsClear is false.

Say the race-course is changed so that instead of one-block tall hurdles we must jump over hurdles of any height. Important: The algorithm shown above works for an object that jumps varying height hurdles. Our problem is not to change our end-application's algorithm. Rather, our problem is to create another kind of racer that knows how to jump hurdles of varying heights.

What about the hurdles of varying height and width? Again, our algorithm still works for a racer object that knows how to jump these hurdles. But how do we actually create these objects?



Each of the classes SteepleRacer and BoxRacer will extend Racer and override jumpRight.

Specification

A SteepleRacer must be able to jump over a hurdle of any height. A BoxRacer must jump over a box of any height and any width. Use while loops to accomplish these tasks.

Create Robots\SteepleRacer.java and Robots\BoxRacer.java. Implement both the SteepleRacer and BoxRacer classes, then compile.

Load Robots\Lab08.java. Set the size to 17x15. With worlds "hurdle1", "hurdle2", and "hurdle3", use an object of type Racer. With worlds "steeple1", "steeple2", and "steeple3", use an object SteepleRacer. With worlds "boxtop1", "boxtop2", and "boxtop3", use an object of BoxRacer.

Extension

Use the WorldBuilder to create additional valid race courses to test your two classes.

Lab09
Shifting Piles

Objective

Algorithms. Here's how we can count the number of steps we take before hitting a wall:

```
int count = 0;
while(karel.frontIsClear())
{
  karel.move();
  count++;
}
//count now stores the # of
//steps we just took.
```

Background

You'll get very few hints on how to solve this problem. Your robot is standing at (1, 1) facing east with zero beepers. Each of the next five blocks, (2, 1), (3, 1), (4, 1), (5, 1), and (6, 1), 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.

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.

Consider that in order to shift a pile of beepers, we have to know how many beepers we just picked up in order to know how many beepers we need to drop after we move one step. Given that, we can not pick up x beepers, move, then drop x beepers and repeat. Why? Because we will then pick up all the beepers that we just dropped plus all the beepers that are in the next pile, and therefore end up carrying every beeper to the last corner. Consider the following process:

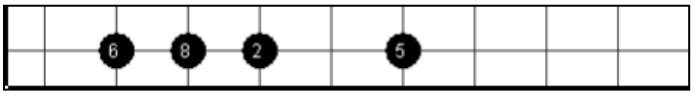
1) pick up all the beepers that are there and count them (store that value in some integer called ***pick***)
2) put down a number of beepers specified by some variable called ***put***
3) move
4) make the variable ***put*** now store the value in ***pick***

There will be no more than 8 piles to move, so repeat the steps above 8 times.

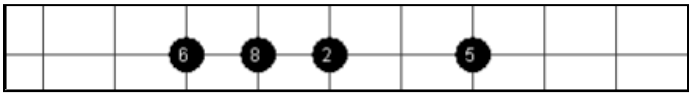
Specification

Create Robots\Lab09.java at size 10x10. Declare one robot at (1, 1) facing east with zero beepers. Shift each of the piles one block to the right. Test the program with worlds "pile1", "pile2", and "pile3".

Before:



After:



Extension

Use the WorldBuilder to create valid robot-worlds to test your program.

Discussion
Decision Structures

The simplest decision structure is the if-statement.

```
if(karel.onABeeper())
  karel.pickBeeper();
if(total != 0)
{
  karel.turnRight();
  karel.move();
}
```

We assume that karel is an Athlete and total is an integer. The if-statement is useful when we have one process that may or may not happen. When we have two processes, only one of which will happen, we use the if-else statement.

```
if(karel.onABeeper())
  karel.pickBeeper();
else
{
  karel.putBeeper();   //only happens if !karel.onABeeper()
  karel.move();
}
if(total != 0)
{
  karel.turnRight();
  karel.move();
}
else
{
  karel.turnLeft(); //only happens if total == 0
  karel.move();
}
```

In an if-else statement, exactly one of the processes 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, use an else-if ladder.

```
if(total != 0)
  karel.turnRight();
else
  if(total < 5)
    karel.turnLeft();
  else
    karel.turnAround();
karel.move();     //bottom factor move because we 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 processes 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. Your robot is in a maze of unknown size. A single beeper marks the end of the maze. You must escape the maze.

There is one rule in maze construction: no islands. There must be a continuous sequence of walls connecting the start of the maze to the end of the maze. Hint: The last sentence was a hint.

> Consider that you can find your way out of ANY maze by always keeping a wall on your right. Given that, you have to know how to handle each of three different situations:
>
> 1) If your right is clear, what should you do to get a wall on your right side? I say, go occupy that space to your right that you know is clear.
>
> 2) Otherwise, if your front is clear (and assuming your right is not), what should you do?
>
> 3) Otherwise, we can now assume that our front and right is blocked, so what do you do?

You can repeat the decision structure above until you are out of the maze, which is marked by a beeper.

Good luck.

Specification

Create Robots\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 the WorldBuilder to create valid robot-worlds to test your program.

Discussion
Abstract Methods

All classes until now have shown you both their interfaces and their implementations. The interface is what you see in the API. The interface tells you **what** methods are available. The implementation is the actual source code behind the class files that you use. The implementation tells you **how** the methods work.

An abstract method provides an interface with no implementation. The *signature* of a method is given. This tells us the name of the method and the number and type of arguments. We also know the return-type of the method (although the return-type is not technically part of the signature). No code is given.

Since no code is given, an object of this class would not know how to respond when the method is called. Thus, you are not permitted to create objects of a class containing any abstract methods. Such a class is called an *abstract class*. (The class could contain a thousand methods with implementation, but so long as there is even one abstract method, the class must be abstract.) Any class that does not contain an abstract method is called a *concrete class*. All the objects we actually create come from concrete classes.

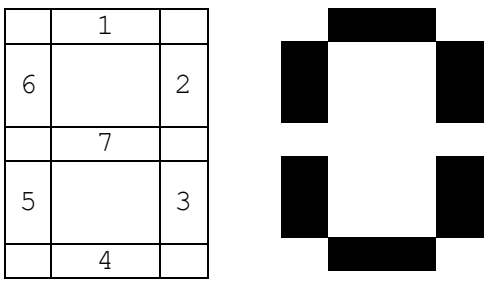
If you are extending an abstract class, you are making a contract to write the code for every abstract method. Classes derived from an abstract class are checked by the compiler to see if they provide the implementation code. Any subclass that does not override an abstract method will generate an error.

Digit is an abstract class because it has the abstract method display(). Notice that display() has no code, only a semicolon.

```
//in Digit.java
public abstract class Digit
{
  public abstract void display();

  public segment1_On()
  {
    //code
  }
  //more code for segment2, etc.
}
```

The class Zero extends the abstract class Digit and must implement the abstract method display. Digits are composed of seven segments each of which is either turned on or turned off. The Digit class provides methods (actual methods, not abstract methods) for turning each segment on or off. These methods must be called in numeric order from one to seven, as shown.



A primitive light emitting diode (LED) digit.

```
//in Zero.java
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();
   }
}
```

Lab11
Identification Numbers

Objective

Extending an abstract class.

Background

Each of you has a student ID number. You are going to display that student ID number in a robot-world using Digit-type objects. Mr. Torbert's student ID number is 564425. His program uses the code shown.

The first digit is a five, the second digit is a six, the third digit is a four, and so on. Notice the horizontal spacing of the digits. If your ID number has seven digits then you will have to create seven Digit-objects instead of six. Display your own student ID, not his.

Take a step back from the actual source code. What is your conception of a digit? Is this a higher form of abstraction than your conception of, say, a five? Is your conception of a five in turn a higher form of abstraction than your conception of this five—5. The various degrees of abstraction illustrated here relate precisely to the relationship between an abstract class, a concrete class, and an object.

```
Digit first = new Five(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 Five(31, 9);

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

Specification

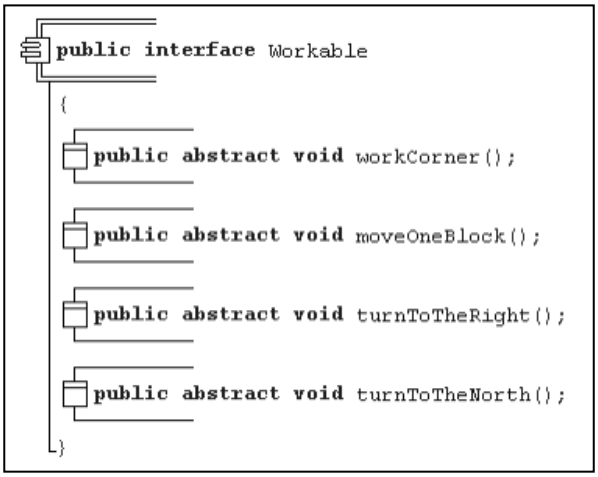
Create Robots\Lab11.java with default-world at size 36x32 or 42x37. Display your student ID number using appropriate Digit-type objects. For each different numeral, you will have to define a subclass. You will need to define Zero.java (see discussion above), One.java, Two.java, etc**.** Note that for each numeral, you will have to define the method void display() such that it turns on and off the appropriate segments to make the desired numeral (see discussion above). **Do not create Digit.java** – that is already defined in the Robots folder, and creating your own will override what is already built.

Extension

Since we only ever call one method for each object, we do not actually need named references. Instead of using `first.display();` we could use `new Five(1, 9).display();`. Whoa.

Discussion
Interfaces

An interface is like an abstract class that has only abstract methods. No method in an interface actually does anything. They are all abstract. For instance:

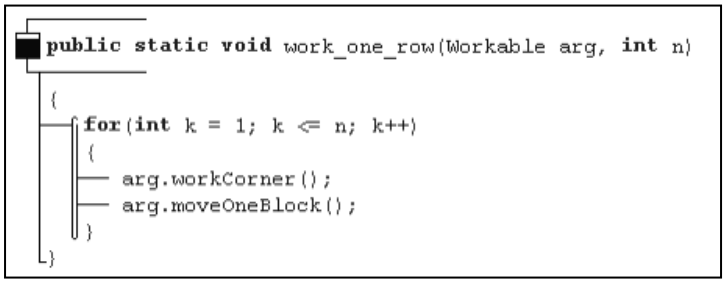


Since there are no methods that do anything, it doesn't make sense to extend an interface. There is nothing to extend. Thus we use a different keyword, implements, to indicate that our class is providing all the method definitions (all the implementations). For example,

```
public class Harvester extends Robot implements Workable
{
//constructors
//new 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 beauty of this is that the end-application does not care about the object's implementation. For instance:



We have no idea what this code will actually do. It depends. Any class with the Workable methods could be passed in here. So the behavior depends on how the methods workCorner and moveOneBlock have been defined in your class.

The process is as follows. You define a class that implements the interface Workable. Since Workable has four abstract methods, you must define all four methods. You then create an object of this class you've written and pass that object to the method shown. Someone else could have a completely different object from a completely different class, but if that class also implemented Workable then we would know it had these four methods and, thus, we could use work_one_row successfully.

Lab12
Spiral Square

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.

The command shown to the right will create either a harvester or a carpeter with equal probability. The method Math.random evaluates to a decimal number from zero to 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).

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

But what do harvesters and carpeters do? **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.

Warning. Some of the method definitions 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 implementing classes are also robots. **So if the interface asks you how to move forward one block, just use the robot's move command.** That's it.

Specification

Load Robots\Harvester.java. Implement the required methods. Then compile.

Load Robots\Carpeter.java. Implement the required methods. Then compile.

Load Robots\Lab12.java. 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.

Discussion: Methods that return values:

A void method merely executes a block of code when called. Non-void methods (methods that return values) will also execute the block of code defined for the method, and then send a value back to the line of code that called the method. Consider the square-root method. To operate the square-root method on the number 4 involves executing some mathematical code, but then the method returns the value 2.0 back to you. Examples:

```
public static int countBeepers(Robot arg)
{
    int count = 0;
    while(arg.onABeeper())
    {
        arg.pickBeeper();
        count++;
    }
    return count;
}
```

```
public static boolean rearIsClear(Athlete arg)
{
        boolean state = false;
        arg.turnAround();
        if(arg.frontIsClear())
                state = true;
        arg.turnAround();
        return state;
}
```

The method countBeepers will pick up all the beepers that a Robot is standing on and return back to you the number of beepers that it picked up.

```
Robot sam = new Robot();                //sam will turn left a number of times
int numBeepers = countBeepers(sam);     //depending on how many beepers it
for(int i=0; i<numBeepers; i++)         //picked up.
    sam.turnLeft();
```

The method rearIsClear will look to see if the space behind the Athlete is blocked by a wall.

```
Athlete jill = new Athlete(6,5,World.WEST, 0);
if(rearIsClear(jill))                   //note that rearIsClear is a static method
{                                       //and therefore NOT a part of an Athlete,
    jill.turnAround();                  //so we don't say jill.rearIsClear()
    jill.move();
}
```

A method that returns a value will end as soon as it hits the **return** statement, so any code following an executed **return** statement will not run.

Lab13
Treasure Hunt

Objective
Solve a problem.

Background
There are treasures hidden in various robot-worlds and maps (piles of beepers) to 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.
6. Otherwise, maintain your current heading.
7. Repeat as needed.

Be careful! You must pick up the pile after reading it. Keep all the beepers you encounter.

Consult the JKarel API for more information regarding the Pirate class.

Specification
Create Robots\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 you must implement so that Lab13 will work. You may want to use nested if-else statements for turnAppropriately.

Load Robots\Lab13.java at size 8x8. Use worlds "map1", "map2", and "map3". Compile and run.

Test Data
Look in the debugger 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 without using loops. Each method can be implemented recursively. See the appendix for more details on recursion.

Discussion
Methods that return values

As you have seen with the boolean methods like frontIsClear() and onABeeper(), not all methods are void methods. These non-void methods will not only perform a set of instructions, but they will also return a value back to the code that is calling them.

For example:

```
if (karel.hasBeepers())                    //hasBeepers will return true or false
    karel.putBeeper();
```

this can also be done the following way, even though it takes more code:

```
boolean ans;
ans = karel.hasBeepers();          //ans will store the value returned from the method
if (ans == true)
    karel.putBeeper();
```

There are also methods built into Robots that will return the street and avenue that they reside on:

```
int x = karel.avenue();          //public int avenue() returns its current avenue position
int y = karel.street();          //public int street() returns the street value that a Robot is on

//each value is now stored in the primitive variables x and y.
//We can now use those values to create a new Robot in the same location:
Robot baby = new Robot (x, y, World.NORTH, 0);
```

This can also be done in the following way:

```
Robot baby = new Robot(karel.avenue(), karel.street(), World.NORTH, 0);
//this is possible because the constructor is waiting for four int values.
//The methods avenue() and street() return int values.
```

All these are similar to other JAVA methods that return values, such as those in the Math object:

```
double x;                  //x will store a number with a decimal.
x = Math.random();         //x will be assigned a random number between [0, 1).
x = Math.sqrt(9); //x will be assigned the result of the square root of 9, or 3.0.
x = Math.cos(0);  //x will be assigned the result of the cosine of zero, or 1.
```

You can write methods that will **return** values by designating the return type where you use to have **void**.

```
public static int numBeepers (Robot arg)          //returns the number of beepers that a Robot is holding
{
    int count=0;                                  //initialize count to zero
    while(arg.hasBeepers())   //drop all beepers you have and count them
    {
        arg.putBeeper();
        count++;
    }
    for(int i=0; i<count; i++)                    //pick up the same number of beepers you just dropped
        arg.pickBeeper();
    return count;                    //end the method by having it return the number of beepers you have
}
```

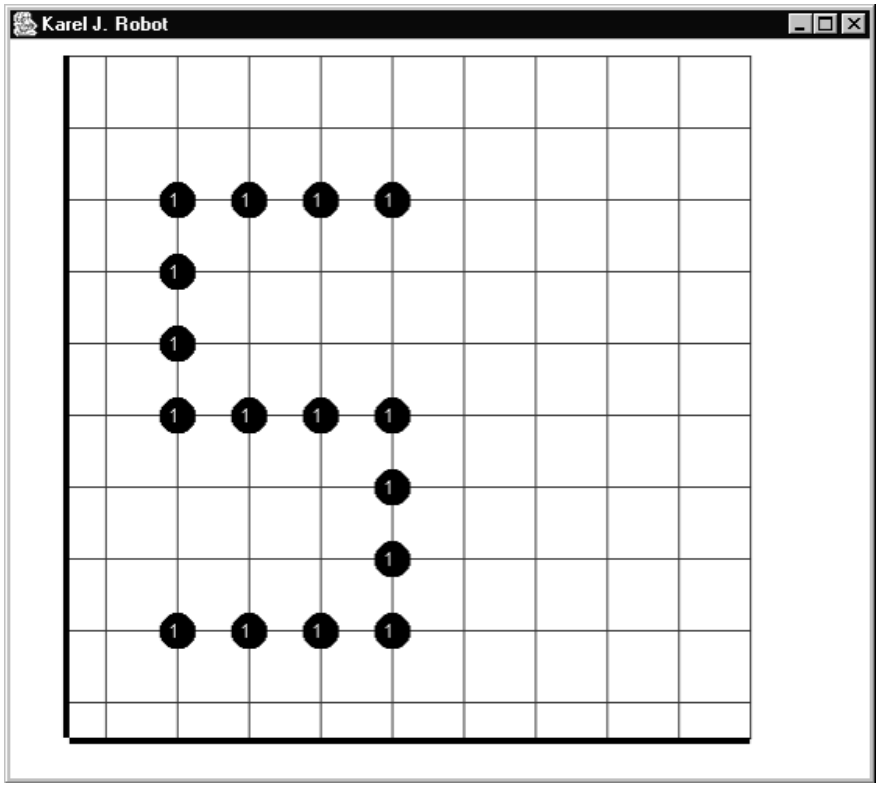
This method can now be used like the methods above.

```
int x = numBeepers(karel);
for (int i=0; i<x; i++)                           //karel will move the same number of spaces
   karel.move();                                  //as the number of beepers he is holding
```

Lab14
Yellow Brick Road

Objective
Solve a problem.

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. I recommend making a new subclass of Athlete – perhaps a PathFollower. Give this robot the ability to sense if there is a beeper in front of it, to the left of it and to the right of it. Like this:

```
public boolean beeperInFront()
{
  boolean ans = false;
  //code that changes ans to true if
  //there is a beeper in front and then
  //moves back to its starting position
  return ans;
}
```

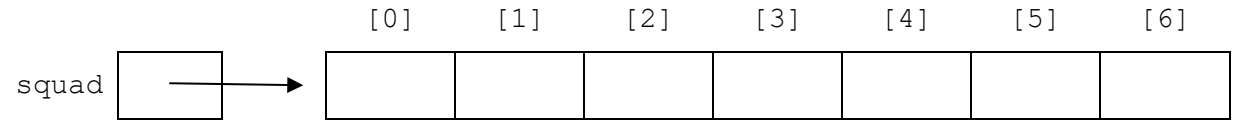
Specification
Create Robots\Lab14.java. Create an Athlete object. Use "path1", "path2", and "path3". Write an algorithm that will get your athlete from (2, 2) to the end of the path of beepers, wherever that may be.

Test Data
Use the WorldBuilder to create other valid robot-worlds to test your program.

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[] squad = 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 <squad.length; index++)
{
  squad[index] = new Robot(index + 1, 1, World.EAST, 0);
}
```

The first element of an array always lives at index 0. The last element of an array always lives at index that is the array's length - 1. 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 < squad.length; index++)
{
  squad[index].putBeeper();
}
```

If index traverses through every element of the array, the for loop tells every array element to drop a beeper.

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

```
for(int index = 0; index < squad.length; index++)
{
  for(int count = 1; count <= 5; count++)
  {
    squad[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 `squad`).

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.

Lab15
Take the Field, with arrays

Objective

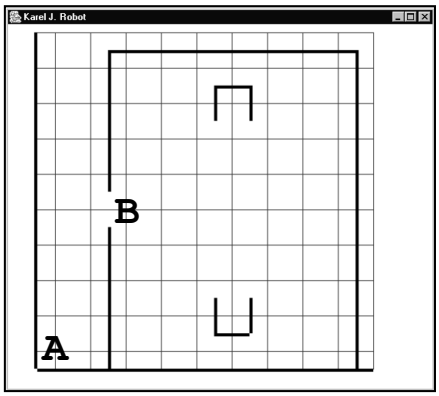
Modify Lab03 to use an array of Athletes.

Background

Recall that in Lab03, we had six different instances of Athletes. Since we had to tell six different objects to take the field, we wrote a static method so that we didn't have to re-write the same code six times. The answer was a static method called takeTheField.

```
public class Lab03
{
  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[]arg)
  {
     //read world, set size and speed
     Athlete a = new Athlete();
     Athlete b = new Athlete();
     //...four more defined here

     takeTheField(a);
     takeTheField(b);
     //...four more calls to the method here
  }
}
```

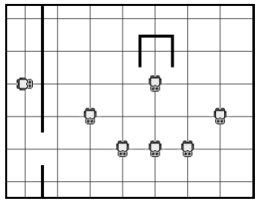


This could be done in a lot less code with the use of an array. With a modification like this, we could easily change the specifications to have a team of size 12, and the only required code change would be altering the size of the array that is defined.

Specification

Load Robots\Lab03.java and modify it in the following way: Change the name to Lab15, and save it as a new file. Rather than having six individual instances of Athlete, create one array of six Athletes. Then use a loop to send each element of the array to the static method takeTheField. You need only modify the main method. You should not have to modify the static method takeTheField. Have your team get positioned for the start of the game as shown below.

Sample Run



Lab16
Robot War

Objective

Process an array of Robots

Background

You are going to complete two methods that process an array of Robots by counting the number of array elements that meet specific requirements.

In lab15.java, a global variable is defined as: `public static final int SIZE = 10;`

The reserved word `final` means that our variable SIZE will not change from what it is initialized to throughout the program. It is initialized to 10, and will never be changed to a different value. This is useful, in that SIZE will govern the dimensions of the world that we create, as well as the number of elements in two arrays of Robots. It would be bad for the program if we were to change SIZE in the middle of execution. When size is initialized to 10, the world will be 11 x 11, and we will have two arrays of 10 elements. If we want to change the dimensions and array elements, we can alter the value of SIZE at the top of the program and everything will readjust to the value that SIZE is initialized to. Changing the initialization of SIZE to 15 will make the world 16 x 16 and we will have two Robot arrays, each with 15 elements.

The main function creates two Robot arrays, with each Robot starting on a unique location:

```
Robot [] squad = new Robot[SIZE];
for(int i=0; i<squad.length; i++)
   squad[i] = new Robot(1, i+2, World.EAST, 0, "pics/knight/karel");
```

Note that when i is 0, the first Robot named squad[0] starts at 1, 2 facing East.

When i changes to 1, the second Robot named squad[1] starts at 1, 3.

When i changes to 2, the third Robot named squad[2] starts at 1, 4.

Run the lab, and you will see two arrays of Robots created at opposite sides of the world. As they approach one another, a collision between Robots will result in one of them being declared the winner, who can continue to advance to the other side. The loser will turn south and move off of the field. The army that wins will be the one that successfully gets the most number of soldiers to the enemy base. With incomplete code, the program will not end and declare a winner.

Your task is to complete the methods that will be used to determine when the program should end, and which army wins the war.

```
public static int numDone(Robot[] a)
```

This method should traverse through the array a, and count the number of Robots who have reached a stopping point. This will be true if the Robot's front is blocked, or if the Robot's avenue is >= SIZE. Each Robot has a method called int avenue() that returns it's avenue position. The numDone method is used to tell the main function when all Robots have reached a stopping point.

Specification

Load Robots\Lab16.java. Complete the methods:

```
public static int numDone(Robot[] a)

public static int numWinners(Robot[] a)
```

If the methods are completed correctly, the program will declare which team wins, which is determined by which team has the most number of members that made it to the enemy base.

Discussion
Recursion

In Racer we defined a method sprint as follows:

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

This solution is iterative because it uses a for-loop. The following solution does not use a for-loop:

```
void sprint(int n)
{
  if(n > 0)
  {
    move();
    sprint(n - 1);
  }
}
```

This solution is recursive because the method sprint calls the method sprint—it calls itself. Imagine a call to sprint(2) was made. Since n = 2 is greater than zero the robot would move and then call sprint(1). In sprint(1) the value of n = 1 is also greater than zero, so the robot would move and call sprint(0). In sprint(0) the value of n = 0 is not greater than zero, so the robot would do nothing. The call to sprint(0) would end, then the call to sprint(1) would end, then the call to sprint(2) would end. In all, our robot moved twice.

Imagine a beeper is on the same street as your robot, some unknown distance due east of your robot's current location. Retrieve the beeper, then turn left and move north to drop-off the beeper. You must move the same number of blocks north as you had moved east. The following solution will work:

```
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();
```

Discussion
Recursion (continued)

It does not matter that we do not know the number of blocks to be traveled beforehand. We will keep track of how many blocks we travel east during the while-loop, then use that number to control our for-loop, which travels north. A more elegant solution involves the definition of a recursive method:

```
void recur()
{
  if(nextToABeeper())
  {
    pickBeeper();
    turnLeft();
  }
  else
  {
    move();
    recur();
    move();
  }
}
```

Assume that karel is a robot object of the class for which this method is defined. Then our solution consists of only two lines:

```
karel.recur();
karel.putBeeper();
```

The recursive method recur will cause karel to move east, locate the beeper, pick it up, turn left, and proceed north to the drop-off point. Then all we have to do is have karel put the beeper down.

So long as nextToABeeper is false, we move and call recur. Each call to recur will test nextToABeeper, only one block further east—after the last move. This is called the recursive case because the method calls itself.

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 having only the second move 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?

Lab17

R Re Rec Recu Recur Recurs Recursi Recursio Recursion

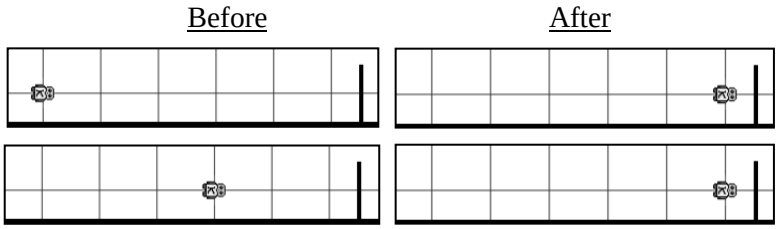
Objective

Getting looping behavior without loops – make methods call themselves.

Background

Consider the tasks from Lab06:

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



We can achieve similar results without a loop at all by having a method call itself inside of itself. It is a dangerous practice when done wrong, but can yield elegant code when done correctly.

```
public static void findTheWall(Robot temp)
{
   if(temp.frontIsClear())
   {
      temp.move();
      findTheWall(temp);
   }
}
```

Row	Task
1	Go to 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 end of the row of beepers and return back to the starting point.
6	Go to the beeper and return back to the start point.
7	Go to the wall and return back to the starting point.

Note that the method calls findTheWall within the method findTheWall. The important detail here is that temp moves before the method calls itself, thereby taking one step closer to the wall before repeating the process. If there eventually IS a wall somewhere in front of temp, this method will eventually end when he gets there. If there happens to be commands given after the recursive call, they will be remembered by being stored in a stack, and executed after the recursive call before it completes.

Specification

Consult the JKarel API for more information regarding boolean methods of the Robot class.

Load Robots\Lab17.java. Fill in the empty static void methods to accomplish all six tasks without using any loops. Each method should solve the task recursivel. Test the code using the world "tasks99".

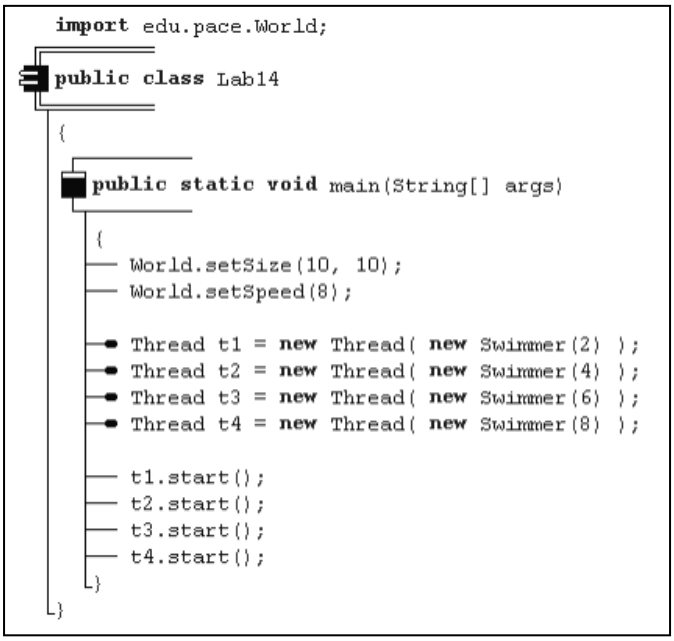
Extension

Create a valid robot-world for this lab using WorldBuilder.jar and test your program with that world.

You will find the WorldBuilder in the worlds folder. Save your world in the same folder with the extension .wld (i.e. robots/unit1/worlds/tasks99b.wld).

Discussion
Threads

After a first glance at Lab14.java (shown here to the right) you may be wondering if this is really a robot lab at all. On second glance, if you look carefully, four anonymous Swimmer objects are created and, as it turns out, Swimmer extends Robot.



```
import edu.pace.World;

public class Lab14
{
   public static void main(String[] args)
   {
      World.setSize(10, 10);
      World.setSpeed(8);

      Thread t1 = new Thread( new Swimmer(2) );
      Thread t2 = new Thread( new Swimmer(4) );
      Thread t3 = new Thread( new Swimmer(6) );
      Thread t4 = new Thread( new Swimmer(8) );

      t1.start();
      t2.start();
      t3.start();
      t4.start();
   }
}
```

We could have given our swimmers names:

```
Swimmer karel = new Swimmer(2);
Thread t1 = new Thread(karel);
```

The important point, though, is that each different robot is associated with its own thread. Those threads are then told to start. In fact, we didn't have to name the threads (as in the digit lab):

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

This is a perfectly valid command that works just as well as those shown in Lab14.java. Ultimately, though, convention dictates how to write the most readable code.

Thread is a built-in java class (java.lang.Thread) that allows us to create processes running on your computer separate from the main program's execution. In Lab14.java, four different processes each separate from the main program's process will all execute at the same time. The result will be parallel processing: different robots performing their own methods seemingly at the same time.

As an example, imagine you have a video clip of a person talking. The clip is really just a series of images that together give the appearance of continuous video. The video clip represents images, not sounds. If you played the video alone, you would have to lip-read in order to figure out what the person was trying to say. Imagine that you also have an audio clip of the same person talking. This audio corresponds with the video images so that if you played them both at the same time it would appear that the person in the video was saying the words from the audio. Now imagine you sold tickets for people to come and watch this person talking. When they arrive, you play the entire video clip, with no sound, followed by the entire audio clip, with no images. What if each clip was 45 minutes long? And the image and sound never once coincided? People would want a refund.

But this sequential procedure is exactly how every one of your previous JKarel programs has executed. In fact, this debacle could have been controlled by robot objects of the class Producer:

```
karel.playVideoClip();
karel.playAudioClip();
```

This would not work at all. We want our video and audio to execute together, at the same time. This synchronization is called *concurrency*. In order to cause independent processes to execute concurrently we will use *threads*, a built-in Java mechanism. Why don't we have to import the Thread class?

Lab18
Synchronized Swimming

Objective

Parallel programming with Java threads.

Background

Runnable specifies a single abstract method run. This is the method that the associated thread object will execute (in parallel) when your program runs.

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

In order to have our threads do anything meaningful when they execute, we must implement a Runnable class in order to create Runnable objects associated with each thread. The built-in Runnable interface (java.lang.Runnable) can be used. Each thread object expects to have an associated runnable object.

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

    }
}
```

Notice that the Swimmer class both extends Robot and implements Runnable. In fact, a class may implement as many interfaces as necessary, but it may only extend one class. Since interfaces don't actually define how methods work, it is impossible for two interfaces to have conflicting definitions of the same method. This is not the case for classes.

Why don't we have to import the Runnable interface?

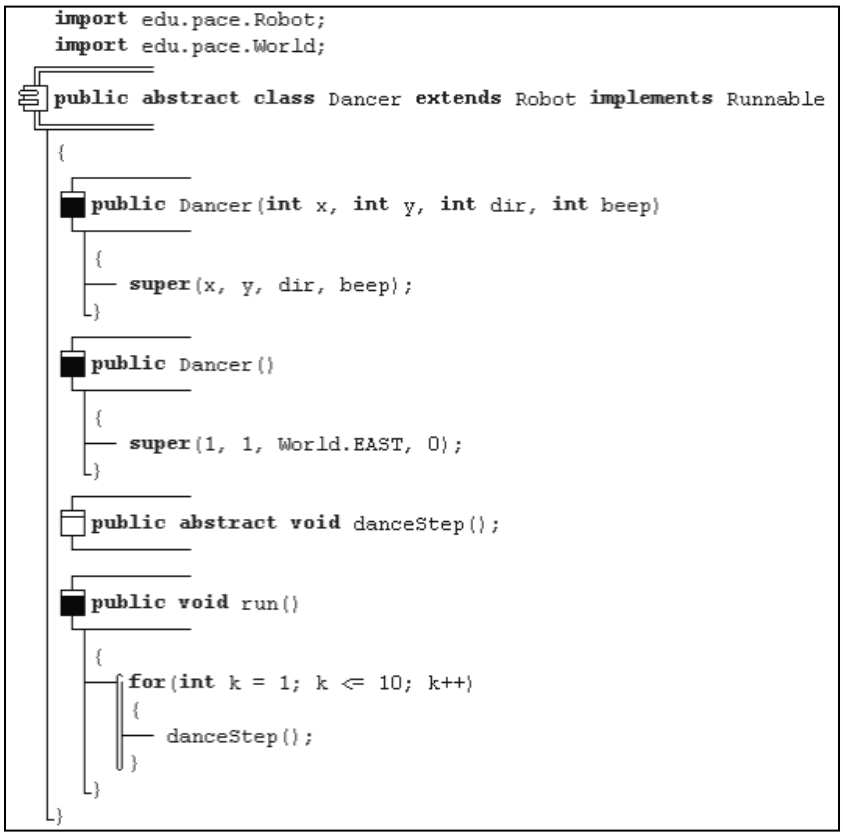
Specification

Load Robots\Swimmer.java. Implement the required run method (specified as abstract in the Runnable interface). The run method is the code that actually gets executed when an associated thread is started. Have your swimmers move forward twice, twirl all the way around, then move back to their starting position to prepare for the next iteration. Have each swimmer loop ten times. Compile.

Load Robots\Lab18.java. Compile and run.

Discussion
Concrete and Abstract Classes

It is illegal to create Dancer objects because the Dancer class has been tagged abstract. The Dancer class must be tagged abstract because it contains an abstract method danceStep.



```
import edu.pace.Robot;
import edu.pace.World;

public abstract class Dancer extends Robot implements Runnable
{
   public Dancer(int x, int y, int dir, int beep)
   {
      super(x, y, dir, beep);
   }
   public Dancer()
   {
      super(1, 1, World.EAST, 0);
   }
   public abstract void danceStep();

   public void run()
   {
      for(int k = 1; k <= 10; k++)
      {
         danceStep();
      }
   }
}
```

In order to create an object of a particular class, that class must have actual definitions for all of its specified methods. This includes methods listed in the class itself, methods inherited from its superclass, and methods in any interfaces it implements.

For instance, if Dancer did not implement the run method specified in Runnable, then the Dancer class would have two abstract methods that it hadn't defined. Each class is responsible not only for its own methods, but any methods specified above it in the hierarchy.

As another example, any subclass of Dancer must implement the method danceStep. Otherwise, that subclass will have to be tagged abstract. A class with no outstanding abstract methods to be defined is called a *concrete* class. Objects are always instantiated from concrete classes.

Anytime you see an argument whose type is an interface or an abstract class, or if you see a reference whose type is an interface or an abstract class, realize that the actual object cannot possibly have been created of that same type. The actual object must be from a subclass. At some point, the abstract methods must have been defined or the object could not have been created. For instance, one of the Thread constructors has the signature:

```
public Thread(Runnable arg)
```

Of course, the object that is passed when a Thread is created cannot be a direct instance of the Runnable interface. Rather, the object is an instance of a class that either implements Runnable itself or is derived (not necessarily directly) from a class that implements Runnable. Any of Dancer's concrete subclasses qualify as Runnable objects. Any of those subclass's subclasses qualify as Runnable objects, etc.

Lab19
Dancing Robots

Objective

Parallel programming with Java threads.

Background

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

```
import edu.pace.World;

public class BackAndForthDancer extends Dancer
{
   public BackAndForthDancer(int x, int y, int dir, int beep)
   {
      super(x, y, dir, beep);
   }
   public BackAndForthDancer()
   {
      super(1, 1, World.EAST, 0);
   }
   public void danceStep()
   {
      move();
      turnAround();
      move();
      turnAround();
   }
}
```

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 start back here for danceStep. The JVM always starts looking for a method's definition at the object on which it was invoked.

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, or do the waltz.

Create Robots\Lab19.java. Model your Lab19 main on the Lab18 main. Create objects from your three different dancer classes and have them "do their stuff" in parallel.

Lab20
Shifty Robots

Objective

Implementing multiple interfaces.

Background

Isn't it amazing that your Lab14 and Lab15 programs executed in parallel without your having done any of the real parallelization? Remember, only because the parallelization algorithms in the Thread class were written generically, for any Runnable object with any method run, was this possible. Oh, the power of object-oriented design.

One of the purposes of this lab is for you to see the statement:

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

Yes, the Shifter class implements both the Runnable and the Workable interfaces. This makes the Shifter class responsible for implementing all the abstract methods in both of these interfaces.

Another purpose of this lab is for you to see the statements:

```
private int myBeepers;
```

The Shifter class defines one field that stores an integer value representing the number of beepers a shifter needs to put down (like the variable ***put*** from lab09). This value must be initialized to zero in the constructor so it can be accessed later by the methods. Note carefully that each shifter object has its own copy of this variable. One shifter might be carrying three beepers that need to be put down while another has only one.

Specification

Load Robots\Shifter.java. Define all the required methods from the two interfaces that have been implemented. Use the field myBeepers to help manage the problem of shifting piles. 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 Robots\Lab20.java. Compile and run.

Note

Really you've been using one thread all along. At some point you may have had an error such as:

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

The main thread of any application exists by default—the commands that make up your main function are executed in this thread.

Discussion
Abstract Classes and Interfaces

So when do programmers create an abstract class and when do they create an interface?

```
public abstract class MazeEscaper extends Athlete
{
   public MazeEscaper()
   {
      super(1, 1, edu.pace.World.NORTH, 0); //no leaving a path
   }
   public abstract void walkDownCurrentSegment();

   public abstract void turnToTheNextSegment();
}
```

The MazeEscaper class specifies only abstract methods, so why not make it an interface? The answer to this question may be easier to understand if we look at the algorithm that uses MazeEscaper type objects.

```
import edu.pace.World;
public class Lab17
{
   public static void escape_the_maze(MazeEscaper arg)
   {
      arg.walkDownCurrentSegment(); //you are not at the end at the start
      while(!arg.anyBeepersInBeeperBag())
      {
         arg.turnToTheNextSegment();
         arg.walkDownCurrentSegment();
      }
   }
   public static void main(String[] args)
   {
      String filename = JOptionPane.showInputDialog("What robot world?");
      World.readWorld(filename);
      World.setSize(8, 8);
      World.setSpeed(10);
      escape_the_maze( new INSERT_ROBOT_TYPE_HERE() );
   }
}
```

The last command in main calls the method escape_the_maze passing an anonymous object of some MazeEscaper type class. Of course, it does not pass a MazeEscaper object because MazeEscaper is an abstract class. Rather, it passes an object of some concrete subclass of MazeEscaper.

Look carefully at the code for escape_the_maze. How do we know that arg must be a robot-type object, and thus why would an interface be insufficient? The rule of thumb is, if inheriting code is important, then use an abstract class (e.g. as we did with Digit). On the other hand, if two classes are completely unrelated (like Robot and Runnable), then use an interface.

Lab21
Follow Walls

Objective
Extending an abstract class.

Background
A maze is defined as a robot-world containing walls (no islands) with a beeper marking the goal. Using an algorithm known as the "follow walls right" strategy, your robot can successfully escape from any maze by constantly hugging a wall to its right-hand-side. It is only okay to move forward if there is a wall to your right AND your front is clear. Otherwise, you need to turn to the right or to the left (turning right takes precedence).

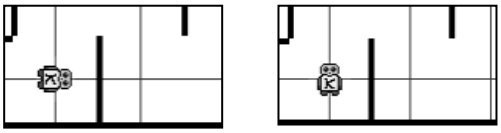
"Follow Walls Right" Algorithm

1. If you're next to a beeper, stop.
2. If it is "okay to go", move.
3. Otherwise, if right is clear, turn corner right and move.
4. Otherwise, turn corner left.
5. Repeat as needed.

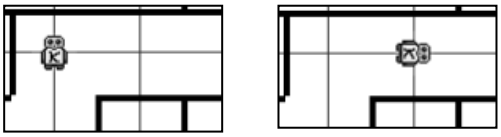
You do not need a special case for getting stuck in dead ends. The algorithm will simply repeat twice, making two left turns and thus effectively turning you around.

Of course, you could just as easily "follow walls left" to escape the maze.

Turning the corner to the left.



Turning the corner to the right.

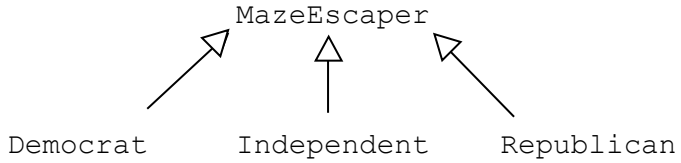


Specification
Create a concrete subclass of MazeEscaper. Create a new Java file that correctly implements both abstract methods specified in MazeEscaper. You may define a Republican (follow walls right), a Democrat (follow walls left), or an Independent (when encountering a T intersection where the right and left are clear, flip a coin each time to go either left or right. A conservative independent will follow a right wall until they reach a T intersection, and a liberal independent will follow left walls until they reach a T intersection.).

Load Robots\Lab21.java. Set size 8x8. Replace "INSERT_ROBOT_TYPE_HERE" with the actual name of the concrete class you've written. Use worlds "maze1", "maze2", and "maze3". Escape the maze!

Extension
Make the MazeEscaper a Runnable abstract class. Then make the tree subclasses of MazeEscaper (Democrat, Independent and Republican) simultaneously race to see who gets out of the maze first.

Class Hierarchy



Discussion
Return, Break and Continue

One way to approach the Lab21 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 Lab21). The return statement is a useful tool for handling special cases in an algorithm. Two other sometimes-useful tools are the break and continue statements, which work in loops.

```
while(karel.frontIsClear())
{
   if(karel.onARobot())
      break;
   if(karel.onABeeper())
   {
      karel.pickBeeper();
      continue;
   }
   karel.move();
}
```

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

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

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

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

Lab22--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. What obstacles must be considered?
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. Would any new classes or methods be helpful (or necessary) in solving the problem?
8. What existing JKarel class or classes may be appropriate to use in the solution?

Create a robot-world using the WorldBuilder. Use the JKarel packet and your correct solutions to previous assignments as a guide to writing up the problem. You may want to write your own solution before writing up the lab assignment.

Possible JKarel Problems:

1. Run a relay race using threads and Math.random().
2. Escape from a room.
3. Find a lost beeper.
4. Write a word in beepers.

Note

In the beginning of our JKarel unit the assignments were very specific to a particular robot-world. Later the assignments became more flexible and our programs were required to work correctly in more than one robot-world (though all of a similar type). Your project may be either fixed or flexible.

Since you are creating the problem you must also create the context in which the problem takes place. You must fashion the robot-world and the description of the problem so as to tell a story. Make it believable, but more importantly, make it interesting.

Lab23

Pacbot and Ghosts

Objective

Define subclasses with follower strategies.

Background

Here is a novel idea for a game: the player looks like a wheel of cheese with a mouth, and navigates around a maze eating pellets while being chased by ghosts. Naturally, if you eat a power-pellet, the ghosts will turn blue and the player can eat them as well…just like it happens in real life.

The ghost behavior should be as such: a RandomMover should make random decisions as to how to turn when it gets to an intersection. A SFollower is a ghost that will follow the player by street postion. They will move if their front is clear, but upon getting to an intersection, they will turn on the street that will move them towards the player. An AFollower is a ghost that will follow the player by avenue position. They will move if their front is clear, but upon reaching an intersection, they will turn on the avenue that will bring them closer to the player. Lastly, the ASFollower is a ghost that will follow the player across streets and avenues: they will move if their front is clear, but turn towards the player upon reaching any intersection.

Specification

Load Robots\PacbotDriver.java, Player.java, Ghost.java, RandomMover.java and SFollower.java. Look at how the object hierarchy is defined, and pay close attention to the SFollower and how it works. Use the SFollower as a model to create new subclasses of Ghost: the AFollower and the ASFollower.

Compile and run PacbotDriver.java

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

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

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

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`.

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`.

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

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

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

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`.

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.

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

Header: The first line of a method definition.

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

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

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

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

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).

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.

Robot methods that change the state of the Robot (position, direction, number of beepers):

.move()
.turnLeft()
.pickBeeper() .putBeeper()

Robot methods that returns true or false (used in conditions like an if statement or while loop):
.frontIsClear() .rightIsClear() .leftIsClear()
.onABeeper() .onARobot()
.hasBeepers()

To make code execute one time if a condition is true, use an if statement:

```
if(condition is true)
{
    //run this body of code
}
```

To make one of two blocks of code execute depending on a condition, use an if-else statement:

```
if(condition is true)
{
    //execute code A
}
else
{
    //execute code B
}
```

To make code continue to execute (in a loop) while a condition is true, use a while loop:

```
while(condition is true)
{
    //repeat this body of code
}
```

To make code repeat a known number of times, use a for loop

```
for(int i = 0; i < n; i++)
{
    //code here repeats n-times
}
```

Primative types:
int – an integer (a whole number – no decimals)
double – a real number (can have decimals)
boolean – can only store true or false
char – a single character from the keyboard

If you want a method to only be available for a certain driver program, define it as a static method, and send an argument for whatever Robot you want to complete the task.

If you want a method to be available for every instance of a particular subclass of Robot, define it within the subclass definition – it will not be static. Call the method using dot-notation.