Robocode, Java, and Trigonometry Tutorial

Jacob Cole

Table of Contents:

Note: To view this tutorial best, you should hide the spelling/grammar "errors" in this document.
Windows: go to Tools > Options, click on the Spelling & Grammar tab and check the Hide spelling errors in this document and Hide grammar errors in this document checkboxes.
Macintosh: go to Word > Preferences, click on Spelling and Grammar, and then check the Hide spelling errors in this document and Hide grammar errors in this document checkboxes.

QUICKSTART

What is Robocode?

Robocode is a virtual tank battling game, in which you write the AI for the tanks and send them out to fight. When you make a good robot, you can submit it to the Eternal Rumble at http://robowiki.net, in which you battle against people from around the world. Everything is in Java. Robocode has its own special libraries.

Getting and Using Robocode

Downloading Robocode

Go to http://robowiki.net/cgi-bin/robowiki?Robocode and click on (on the top bar) the download link.

Installation

Double-click on the setup file (will be .jar) and follow the instructions.
Note for Windows: You should probably install robocode in Program Files, even though it defaults to the C drive (just replace C:\robocode with C:\Program Files\robocode when it asks you during the installation). If there is a problem, see the Beginners' FAQ. Don't add robocode to your Start menu when it asks, it sometimes messes up.

Running Robocode

Windows: double-click on robocode.bat
Macintosh: double-click on robocode.jar

How Robots Work

See Appendix E: Robocode Physics and Mechanics (IMPORTANT!)

Battling

To start a new battle, go to the Battle menu > New and then double-click on robots or select them and click add robot.
To see the radar beams of the robots during a battle, go to Options > Preferences and then check the View Scan Arc box.

Editing/Creating Robots

To go to the robot editor, go to the Robot menu > Editor. To make a new robot, go to File > New > Robot and follow the instructions.

Viewing Documentation

To see the documentation for robocode, go to the help menu and click on Robocode API. Once in the API, click on Robot or AdvancedRobot in left side bar and scroll down. Those are the only things that are important for new robocoders.

Note: **API** means Application Programming Interface. It is basically the documentation of a program. In robocode, the API shows all the special **methods**, sections of code that can be run on a command (see below), that you need to program your robot (like `ahead(`*`distance`*`)`) to move your robot forward).

Advice for Beginners

Have fun. Don't give up too quickly. Robocode takes determination and persistence. Don't despair at the length of the tutorial, you don't have to read it all at once. Just read up to Anatomy of Robot Source Code to get started. Also, check out the sample robot included in this tutorial.

Also, read the Beginners' FAQ, the other FAQ, and the Game Physics page. Note that a robot's gun spins with its body, and its radar spins with its gun.

Challenge for Beginners

To start out, experiment and try to make a robot that acts like the sample walls bot. First learn how to make the robot go forward, and then work on turning `onHitWall`. HINT: the easy way to make a walls bot involves the `getBearing()` method (explained in Headings versus Bearings).

Places to Go and Things to See

Go to http://robowiki.net for EVERYTHING ROBOCODE. It teaches targeting, moving, strategies, and much more. Robocode is BIG (thousands of people).
Go to http://robowiki.net/cgi-bin/robowiki?BeginnersFAQ for a beginner FAQ
Go to http://robocode.sourceforge.net/help/robocode.faq.txt for another FAQ
Go to http://robowiki.net/cgi-bin/robowiki?GamePhysics for the physics of robocode.
Go to http://java.sun.com/j2se/1.5.0/docs/api/ for info on Java (the API).
Go to http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Math.html for the Math class API (scroll down to method summary).
Go to http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html for the String class API (scroll down to method summary).
Go to http://java.sun.com/docs/books/tutorial/ for an in-depth Java tutorial.
Go to http://www.robocoderepository.com/ to download other people's robots.
Go to http://www.robocoderepository.com/BotSearch.jsp to search for other people's bots to download.

Downloading Robots

You can download and try other people's robots (and if they are open source, use their source code). Simply save the robots' .jar files in the robots directory in the robocode folder (this is wherever you installed robocode). If robocode is already open, hit F5 in the new battle dialog box after you downloaded the robot. Watch them battle for inspiration. To see the source code of open source robots, go to the robot editor, and click on File > Extract downloaded robot for editing, then select the robot. See the Historical Robots section for advice on which bots to get.

INTRO TO JAVA

Key Information

There are a few basic things that you must understand about Java before you start building ownage robots.

Syntax is the punctuation of a program, the format that the **compiler** (the thingy that turns the stuff you type into stuff the computer can read) can understand. In Java, **source code** (the stuff that you type) is stored in .java files, and compiled programs are stored in .class files. In Java, a main component of the syntax is the semicolon (`;`). Semicolons are needed after every complete statement.

Comments

In Java, there are 3 types of **comments**. Comments are things you add to code to explain what it does. Comments do not affect your program in any way.

```
//comment (1 line)

/*
multiple lines
of comment
*/

/**
multiple lines of comment (double star just means special comment that
can be used for documentation)
*/
```

I will mostly use the `//` style comments. When I put stuff in *italics* it means to input a name for the value in real life (e.g. if your initials were "bc," in real life code you would substitute `bc` for *`yourInitials`*).

Variables

Variable Declaration

In Java, you declare variables like this:

`variableType variableName``;`

For example

```
int myInteger;
String myString;
char myChar;
```

Variable types and names are cAsE sEnSiTiVe, just as everything is in Java. Note: variable names must start with letters or underscores. Also note that variables cannot have names that are Java keywords. If you are having strange compile errors, try changing the names variables that seem suspicious.

Variable Types

Primitive ("basic") Variable types (common ones):

`int` – holds from -2,147,483,648 (same as -2^{31}) to 2,147,483,647 (same as $2^{31}-1$)
`long` – holds integers from -9,223,372,036,854,775,808 (same as -2^{63}) to 9,223,372,036,854,775,807 (same as $2^{63}-1$)
`float` – holds numbers with decimal points up to 3.4028234663852885981170418348452 ∗ 10^{38} Note: ∗ means multiply
`double` – holds numbers with decimal points up to 1.7976931348623157081452742373170 ∗ 10^{308} Note: ∗ means multiply
`char` – holds single characters. You must use single quotes (`myChar='?';`)
`boolean` – holds `true` or `false`

One Non-Primitive Variable Type:
`String` – holds text. You must use double quotes for the text (`myString="Hi";`). To add to the end of a String (called **concatenation**), use the `+` sign: `myString=myString+"5";` The String would now say `"hi5"` Note: the capitalization of the word `String` matters. Again, the proper capitalization of variable types (and everything in Java) is vital.

To set a value to a variable, use the assignment operator, the equals sign
```
myInteger = 5;
```
This sets the value of `myInteger` to `5`

You can both initialize and declare variables in the same statement like this
```
int myInteger=5;
```

Printing Output to Screen/Basic Math
To print out stuff for debugging, use:
```
System.out.println(valueToPrint);
```
In robocode, you must start a battle and click on the button on the right-hand side with the name of your robot on it to view the output of this.

The value in `System.out.println` can be a complex expression like:
```
System.out.println(2*5+3);
```
Note on math: in Java, the basic math operators are `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `(` and `)` (parentheses). Also, order of operations is standard.
If the value is something that you do not want Java to try to evaluate, make it a `String` by putting quotes around it:
```
System.out.println("This is just text");
```
You can print out Strings and values that must be evaluated at the same time by concatenating them to a String:
```
System.out.println("The value of 2*5+3 is: " + 2*5+3);
```
This prints out: `The value of 2*5+3 is: 13`

If you don't want a new line to be created after the value is printed, use `System.out.print(something);` . In the opposite direction, you can manually make a new line by putting `\n` into any String.

Variable Scope

Curly braces, `{` and `}`, enclose all blocks of code in Java. Variables only exist within the curly braces, they are declared in and are visible in all sub-curly braces. Where a variable exists is called its **scope**. In Java, variables of more local scope override variables of more global scope:

```
boolean testVal=false;
{
      boolean testVal=true;
      System.out.println(testVal);
}
System.out.println(testVal);
```

Output:

```
true
false
```

Methods

Definition and Basic Syntax

Methods, called functions or subroutines in other languages, are reusable segments of code. In Java and robocode, other people's methods are vital (you use them for everything as you will see). First, however, you will learn to make your own. Very basic methods are declared like this:

```
void methodName() {
      thingsThatYouWantToDoRepeatedlyWithoutCopyingAndPastingCode;
}
```

These let's say you wanted to print out a bunch of variables at different points in your code and didn't want to copy the `System.out.println()` a zillion times. These variables would have to be declared in the same scope as method or more global, otherwise the method could not see them. Later, you will learn how to pass variables from any scope into a method. Example:

```
int a=1;
int b=2;
int c=3;
int d=4;

void printVars() {
      System.out.println(a);
      System.out.println(b);
      System.out.println(c);
      System.out.println(d);
}


//note: make a basic method execute (called calling a method),
//you simply type methodName();

printVars();
a=5+b;
b=3;
c=8-2;
d=55;
printVars();
```

This will print out the values of `a`, `b`, `c`, and `d` before and after they are modified. The `printVars()` method saves you from having to type 4 `System.out.println()`'s each time and also makes it so that you can edit the method and then not have to update everything in your entire source code.

Method Arguments

To make methods more versatile, you can pass in parameters, or **arguments**. They allow the methods to make use of the values of variables that are declared in a more local scope than the method is. To make a method have arguments, declare what types you want them to be and the names under which they can be accessed in the parentheses after the method's name. Arguments are simply variables that are set when you call the method (you shall see how). Note: arguments are local to their method; they cannot be accessed outside it. Syntax for basic methods with arguments:

```
void methodName(argType argVarName, argType argVarName, . . .) {
      thingsThatYouWantToDoRepeatedlyWithoutCopyingAndPastingCode;
}
```

Example:

```
void printSum(double num1, double num2) {
      System.out.println(num1+num2);
}
```

Methods that use arguments are called like this:

```
methodName(valForArg1,valForArg2, . . .);
```

Example:

```
printSum(52,3);
```

This will print out `55`.
When you call `printSum(52,3);` it sets the value of `num1` equal to `52`, and `num2` equal to `3`. Then, it adds `num1` and `num2`, and prints them which yields `55`
Note: the arguments you pass in must be in the same order and of the same type as the ones in the method declaration. You couldn't call `printSum` like this:
`printSum("52",3);` because Java doesn't know how to add `String`s, which are text, to `double`s, which are numbers. This would be like trying to add `3` to the word `"blah"`.

You can, as you can everywhere in Java, use variables to represent the numbers you pass in. If the type of the variable is the type that the method asked for or a subset of the type the method asked for, the program works.

```
int x=1;
double y=8
printSum(x,y);
```

Since every `double` can hold integer values, this works. If you want to do the reverse (use a `double` where an `int` is asked for), you must use type casting or conversion methods, which are not explained in this section.

The `return` Statement

Finally, the true power of methods shall be revealed. Let's say that you had some complex piece of code that you didn't want to retype a zillion times but that didn't actually print anything. For example, let's say you had some groundbreakingly complex encryption algorithm (note: an algorithm is a procedure for doing something. An example of an algorithm you are probably familiar with is the Division Algorithm, which is commonly known as long division), like adding 1 to the number inputted as an argument. However, what if you didn't want it to only be able to print the encrypted version of the number, but rather wanted to make the method useful for a variety of applications. To do this, you can make methods **return** stuff to be used later in your program. The syntax for methods that return stuff is:

```
returnType methodName(argType argVarName, argType argVarName, . . .) {
	thingsThatYouWantToDoRepeatedlyWithoutCopyingAndPastingCode;
	return returnValue;
}
```

Example:

```
double add(double num1, double num2) {
	return num1 + num2;
}
```

The `return` statement must always return a value the type *`returnType`*, otherwise, the program won't compile. Also, don't forget to put `void` as the return type of methods that return nothing. Also note that `return` IMMEDIATELY ends the method, doing nothing else after. If you want to end a method with return type `void`, simply say `return;` with nothing between the word `return` and the semicolon.

When you call `add()`, you get the sum of 2 numbers inputted. However, unlike with `printSum()`, it is up to you what you do with the output of the method.
Maybe you would do this:

```
double cookiesEatenToday=32.5;
double cookiesEatenYesterday=88;
double totalNumberEaten=add(cookiesEatenToday, cookiesEatenYesterday);
```

<u>Method Overloading</u>
You can define two methods with the same name as long as they have different **signatures**. A method's signature consists of its name and its arguments' types and their order. For example, the signature of the `add` method from above is:
`add(double, double)`
The signature consists of everything that can be used to tell what method is to be called when you call a method. This leads to an interesting possibility: methods with the same name but different signatures. The creating of such methods is called **method overloading**. For example, you could overload the `add(double, double)` method and create an `add(double, double, double)` method as well. To call this method, you would simply say `add(`*`num1`*`, `*`num2`*`, `*`num3`*`)`. In this way, you can make a more intuitive, versatile method structure.

Conventions

In Java, as there are in many languages, there are a few conventions. It is important to be able to distinguish necessities of programming from these conventions.

Variable/method names – it is standard for each variable name to start with a lowerCaseLetter and have each consecutive word capitalized. Note: variable TYPES must be capitalized as shown (all the ones you've seen are all lower case except for `String`). Also, note that when referring to variables, you must use the same cApItAlIZaTiOn in their names as when you declared them.

Indentation – it is also standard to indent stuff inside a layer of curly braces with 5 spaces (or for we lazy people, a tab). This is by no means necessary, but it is recommended.

Where to put curly braces – You can either put curly braces on the same line as the name and arguments or below it (white space doesn't matter at all).

ANATOMY OF ROBOT SOURCE CODE

Below is a generic skeleton of a robocode robot. In real life, there might be prefabricated comments in the code (you can leave those, they don't matter).

Package and Import

```
package yourInitials; //shows who the bot belongs to

import robocode.*; //makes special robocode methods available
```

Robot Name and Type

```
//following line makes a robot with the name BotName and type BotType.
//BotType can be Robot (which is easier but more basic; useful for
//beginners), AdvancedRobot (which is more powerful but harder to use),
//or TeamRobot (which is for making teams; it is also AdvancedRobot)

public class BotName extends BotType {
```

Global Variable Declarations

```
      //here, you declare global variables, variables that are visible
      //throughout the program

      variableType variableName;
      anotherVariableType anothervariableName;
       . . .
```

The `Run` Method

```
      //the following method, the run() method, is called by robocode
      //when the battle begins. This is where the program starts.

      public void run() {
```

Startup Code

```
            //here, put stuff that you only want to do once at the
            //beginning of the battle
            //e.g. initialize variables
            //Note: if you want to initialize global variables with
            //stuff that requires method calls, you must declare the
            //variables above but initialize them here. Above, nothing
            //can actually be evaluated, it is outside the part of the
            //program that runs

            somethingThatYouWantToDoOnlyAtBeginning;
            somethingElseThatYouWantToDoOnlyAtBeginning;
            . . .
```

Main Loop

```
            //the following construct is called a while loop, it
            //executes the stuff in its curly braces as long the
            //statement in the parentheses is true. Since true is
            //always true, this loops forever, or in this case,
            //through the duration of the battle. Put stuff you want to
            //do throughout the duration of the battle here (like move)

            while (true) {
                  somethingThatYouWantToDoThroughoutBattle;
                  somethingElseThatYouWantToDoThroughoutBattle;
                  . . .
            }
      }
```

Event Listeners

```
//below are the event listener methods. They are called whenever
//the event specified in their name occurs. In the robocode, all
/of them have a name beginning with "on" (like onHitWall or
//onScannedRobot). Note: in AdvancedRobots and TeamRobots, the
//events might fire a little bit late (you must correct for
//this). For example, if your turret is spinning (with your
//radar), and the enemy robot is scanned, and you say to fire,
//your turret may have passed the enemy slightly. You can correct
//for this by using the information in the argument; the event
//object. An event object is sort of a supervariable which
//contains variables and methods of its own (technically called a
//class). When the event is run, an Event is passed in (e.g. a
//ScannedRobotEvent). You specify the name you want it to have
//(as with all methods). Often, it is simply named e to save on
//typing (shown below). To get information from the event object,
//say in the event listener: EventTypeEvent.methodName(). To see
//what methods the event objects have, find the event listeners
//in Robot section of the API and click on the links to each
//listener's event type

public void onEventType(EventTypeEvent e) {
      somethingThatYouWantToDoOnThisEvent;
      somethingElseThatYouWantToDoOnThisEvent;
      . . .
}

public void onAnotherEventType(AnotherEventTypeEvent e) {
      somethingThatYouWantToDoOThisnEvent;
      somethingElseThatYouWantToDoOnThisEvent;
      . . .
}
. . .

//here is a sample event listener:
/*
public void onScannedRobot(ScannedRobotEvent e) {
      //whenever we scan the enemy, turn the robot towards him
      //note: getBearing() returns the number of degrees your
      //robot must turn right to face the enemy see Headings
      versus Bearings
      turnRight(e.getBearing());

      //then CHARGE!
      ahead(e.getDistance());
```

```
        }
        */
}
```

MORE JAVA PROGRAMMING

To make good robots, there are some essential things about Java you must learn.

if Statements

Definition and Syntax

`if` statements are used to test if a condition is true and act accordingly
An `if` statement executes the stuff in the curly braces only if the statement in the parenthesis is true.
This is the syntax for an `if` statement:

```
if(statementHereIsTrue) {
      doTrueThing;
}
```

If you have no curly braces, only stuff up to the first semicolon executes:

```
if(statementHereIsTrue)
      doTrueThing;
butNotThis;
```

To do something if the `if` statement is false, use the else statement:

```
if(statementHereIsTrue) {
      doTrueThing;
}
else {
      doFalseThing;
}
```

Nested `if` Statements

When you put an `if` statement into another if statement (or into the `if`'s `else` statement), it is called nesting.

```
if(statementHereIsTrue) {
      doTrueThing;

      if(statementHereIsTrue) {
            doAnotherTrueThing;
      }
}
else {
      doFalseThing;
}
```

`else if` Statements

Here is a useful way of nesting (called `else if` statements):

```
if(statementHereIsTrue) {
      doTrueThing;
}
else if(statementHereIsTrue)  {
      doOtherTrueThing;
}
else if(statementHereIsTrue)  {
      doOtherTrueThing;
}
else {
	doFalseThing;
}
```

Conditions and Boolean Operators

Statements to replace `statementHereIsTrue` are called **conditional statements** or simply **conditions**. The operators used in conditional statements (called **Boolean operators**) are: `<` (less than), `>` (greater than), `==` (equal to, for testing equality. There are 2 equals signs. MAKE SURE YOU USE TWO! Note: to compare `Strings`, you must say: `string1Name.equals(string2Name)`), `<=` (less than or equal to), `>=` (greater than or equal to), and `!=` (not equal to). You can group them with parentheses.

Example 1:

```
int integer1=3;
int integer2=5;

//tests if integer1 is less than integer2
if(integer1 < integer2) {
	System.out.println("Hi");
}
```

This will print out `Hi`

Example 2:

```
String firstString="hello";
String secondString="hello";

//tests if firstString is the same as secondString
if(firstString.equals(secondString)) {
	System.out.println("Yay!");
}
```

This will print out `Yay!`

Since `boolean` variables are equal to `true` or `false`, to test them, you can just say

```
boolean testBool=true;

if(testBool)
{
	System.out.println("Success");
}
```

This prints out `Success` since the statement within the parentheses is equal to true.

In some cases, instead of nesting or using multiple `if` statements, you can use **Boolean logic operators**. The basic logic operators are parentheses, ! (this means NOT), `&&` (this means AND), and || (this mean OR, and you get these by pressing shift-backslash). The order of operations for logic operators is: parentheses, `&&`, ||, and finally !.

Instead of saying:

```
if(integer1>0) {
	if(integer2>0) {
		System.out.println("Both integers are greater than 0");
	}
}
```

You can say:

```
if(integer1>0 && integer2>0) {
      System.out.println("Both integers are greater than 0");
}
```

You can group these logic operators using parentheses.
Example:

```
int integer1=3;
int integer2=4;
//tests if both are less than 5 and their sum is above 6
if((integer1<5 || integer2<5) && (integer1+integer2>6))
{
      System.out.println("Yay");
}
```

Output: `true`

<u>The Ternary Operator</u>
There is also a shorthand `if` statement that is highly useful. It is called the **ternary operator**:

```
condition ? doThisIfTrue : doThisIfFalse;
```

Example:

```
int x=4;
System.out.println(x>0 ? "x is positive" : "x isn't positive");
```

This will print out `x is positive`

Using Built-In Methods
To see a list of useful methods, see the Useful Methods Quick Reference Sheet.

Calling `static` Methods

To call methods documented in the Java API, you have to say where you want to look for them are. Otherwise the program can't find them. Most of the methods you will be using for now can be called like this:

```
Class.methodName(args)
```

Methods that can be called like this are called `static` methods. They take all of their input through their arguments.
When looking at the Java API, there will be a list of classes (for now, you can think of classes as ways of holding and arranging methods) on the left-hand frame. Click on them and scroll down to see their methods. Look for the `static` keyword next to a method you are trying to call this way (it is to on the left-hand side of the method summary).
One very useful class is the Math class, which contains methods for mathematical operations. For example, if you wanted to take the sin of an angle, you would say:

```
double myAngle=0;
double angleSin=Math.sin(myAngle);
```

The output would be `0`

Note: ALL JAVA TRIG FUNCTIONS USE RADIANS! To get π, type `Math.PI`

```
myAngle=Math.PI/2;
angleSin=Math.sin(myAngle);
```

The output would be `1`

Calling non-`static` Methods

Non-`static` methods are methods that operate on whatever called them. These methods do not have the `static` keyword in front of them. You call them with the following syntax:

```
objectName.methodName(args);
```

To demonstrate, I will use the `length()` method that every `String` has (it returns the `String`'s length):

```
String testString="testing";
System.out.println(testString.length());
```

Output: `7`

Type Casting and Conversion Methods

Definition:
Type casting and *conversion methods* are ways of turning one type of variable another. For example, what if you had a `String` that held `"39"` and you wanted to add `5` to it? Since you can't add `Strings` together (since they can hold non-numeric values), you

must convert them a numeric type of variable first. To do this, you would use conversion methods.

`String` to Number Conversions:
To go from a `String` to a primitive type of variable, use the `static` parse methods of the primitive type **wrapper classes** (these contain methods pertaining to each primitive type and are a capitalized version of the full name of the variable. e.g. the wrapper class of `double` is `Double`, and the wrapper class of `int` is `Integer`). The general form for parse methods is:
`WrapperClassName``.parse`*`VarType`*`(`*`StringToConvert`*`)`
They return a *`VarType`* representation of the `String`.
Example 1:
```
String testNumString="39";
int testNum=Integer.parseInt(testNumString);
```
Now, `testNum` is equal to `39`.

Example 2:
```
String testNumString="39.5";
double testNum=Double.parseDouble(testNumString);
```
Now, `testNum` is equal to `39.5`.

`String` to `char` Conversions:
To get `char`s out of a String, use the non-`static` *`nameOfString`*`.charAt(`*`index`*`)` method (*`index`* is the position of the `char` you want in the `String`, starting from 0).
Example:
```
int n=2;
String s="testing";
char myChar=s.charAt(n);
```
Now, `myChar` is equal to `'s'`

You can also convert a `String` to a `char` array using the the non-`static` *`nameOfString`*`.toCharArray()` method (it returns a `char` array representing the `String`).

Primitive Type to `String` Conversions:
To go from a primitive type to a `String`, the easiest way is to use the `static` `String.valueOf(`*`valueToConvert`*`)` method of the class `String`.
Example:
```
String testNumString="39";
int testNum=Integer.parseInt(testNumString);
testNum=testNum+5;
testNumString=String.valueOf(testNum);
```
Now, `testNumString` is equal to `"44"`.

Converting between Primitive Types:
If a certain type is a subtype of another (`int` is a subtype of `double`, every `double` can hold `int` values), there is no need for conversion. You could legally say:
```
int someInteger=10;
```

```
double someDouble=intVal;
```

To convert the other way, from a complicated type to a simple one, you must use **Type Casting**. To type cast, use the following syntax:
`(`*`typeToConvertTo`*`)`*`thingToConvert`*
When casting from decimal-pointed number types to non-decimal-pointed number types, the decimal portion is **truncated**, or cut off (they are NOT rounded). Also, you must be sure that the numbers you are trying to send are within the range of the type you are trying to send them to. For example if you have a `long` that you are trying to cast to an `int`, the number contained by the long could be too large for the `int` to hold.

Example:
```
double someDouble=39.8;
int someInt=(int)someDouble;
```
Now, someInt contains the number `39`

Note: since `char`s are stored as ASCII (American Standard Code for Information Interchange) values, you can convert from `int` to `char` and back. A type cast from `int` to `char` will return the character with the ASCII value of the `int`. The other way returns the ASCII value of the `char`.

Java math:
To add, use `+`
To multiply use `*`
To subtract, use `-`
To divide, use `/`
To group operations, use parentheses
Order of operations is just like in normal math (parentheses, `*` and `/`, `+` and `-`).
To raise to powers use `Math.pow(`*`base`*`,`*`exponent`*`)`

Note: when you divide an integer by another integer, it truncates the decimal portion.
`5/2` is equal to `2`
If you didn't want stuff to truncate make at least one of numbers have a .0 after them
`5.0/2` is equal to `2.5`

Modular Arithmetic
In Java, the `%` sign means modulus. **Modulus** means the remainder when divided.
Ex. `11%5` is `1`
This would be said in speech: "11 mod 5 is 1" or more formally: "11 is congruent to 1 when expressed in the modulo 5" There is something slightly different about modular arithmetic in Java than you might expect. If you say 11.3 mod 5, you get 1.3. What Java does is find out the greatest multiple of 5 that is less than 11.3 (10 it turns out to be) and finds the difference of that and 11.3 (which would be 1.3). Note: the modulus operator (`%`), has the same precedence (place in the order of operations) as `*` and `/`.

Shortcuts
There are a few ways to save on typing. You will encounter these in code a ton.
You can declare and initialize multiple variables of the same type in 1 line:
```
double x=1,y=9;
```
Integers default to 0.
Here are some other short cuts:
`x+=y` is equivalent to `x=x+y`
`x-=y` is equivalent to `x=x-y`
`x*=y` is equivalent to `x=x*y`
`x/=y` is equivalent to `x=x/y`
`x%=y` is equivalent to `x=x%y`
`x++` is equivalent to `x=x+1`
`x--` is equivalent to `x=x-1`

Loops:
Loops are ways to repeatedly execute blocks of code. Like many other languages, **Java has 3 types of loops: `while` loops, `do-while` loops, and `for` loops**. All loops run until a given condition is false.

This is the syntax for a **`while` loop**, the simplest kind of loop:

```
while(thisExpressionIsTrue)
{
	somethingThatIWantToRepeat;
}
```

When you make a loop you must be careful that it does something that will make it stop. Otherwise it will go on forever and ever and ever and ever and ever . A way to make an infinite loop is by saying `while(true)`. This is what you see in the `run()` method of a robocode robot, where you want the robot to repeat its actions forever. Since the expression in the parentheses is always true, the loop will run forever.
A variable that changes itself to stop a `while` loop is called a **sentinel**. Programmers call the statements that update them **update statments**. They are often implemented as below:

```
int i=0;
while(i<10)
{
	System.out.println(i);
	i++;
}
```

This loop will print the numbers `0` to `9`. The variable `i` is used as the sentinel as well as the thing to print out. Sentinels are often used this way. It is always wiser to inequality checkers like <, >, <=, and >= than `!=` in the condition. If the above loop used `!=` instead of <, if there was somehow a weird glitch and `i` got over 10, the program would end up stuck in an infinite loop. This sometimes happens with `float`s and `doubles` because the computer sometimes stores them as approximations (i.e. `1` might really be `1.00000000000000000000000000001` with `doubles`). Also, note that the loop actually runs 10 times, not 9 times (count how many numbers there are 0 to 9). Often, people make the mistake of using <= where a < should be used and vise versa. This results in an **off-by-one error**. Be careful of these when programming.

`do-while` loops are VERY similar to `while` loops; the only difference is that `do-while` loops run one iteration (an iteration is one cycle of a loop) before checking the condition. Their syntax is:

```
do
{
	somthingThatIWantToRepeatButDoOnceBeforeChecking;
}
while(thisExpressionIsTrue);
```

Note that there is a semicolon in the `while(thisExpressionIsTrue);` part.

`for` loops are a quick, easy way to make a loop with a sentinel. They are declared like so:

```
for(sentinel=initialValue;conditionToTest;updateStatment)
{
      somethingThatIWantToRepeat;
}
```

Example:

```
int theSentinel;
int topNum=10;
for(theSentinel=0; theSentinel<=topNum; theSentinel++)
{
      System.out.println(theSentinel);
}
```

This loop will print out the integers from 0 to 10. Note that because there is a <= in the condition, it really runs 11 times (count to 10 starting from 0 and you will see).
Note about `for` loops: you can declare the sentinel variable in the *`sentinel=initialValue`* part if you have no need of them outside the loop. You would say:
`for(int theSentinel=0; theSentinel<=topNum; theSentinel++)`
Another note about `for` loops: you can have multiple sentinels, conditions, and update statements. You just have to separate them with commas.
Ex. `for(int i=0, int j=2; i<=100, j>5*i; i++,j++)`

Note about loops: there is one other type of loop with the syntax

```
for(Object someName:anotherName)
{
      stuff;
}
```

but you will probably not see it and it is used only for some advanced Java stuff.

Arrays:
Arrays are simply a way of quickly and easily making and accessing a bunch of variables of the same type. In Java, array variables (variables that can hold arrays) are declared like this:
```
dataTypeYouWantArrayToHold[] arrayName;
```
An example of an array declaration is:
```
double[] numbersToFindAverageOf;
```

Now, this array doesn't hold anything yet (in fact it can't yet as you shall soon see). All that we said with the above line of code was that we want to make something named `numbersToFindAverageOf` that can hold an array of `doubles`. Arrays in Java are non-dynamic, you can't change their size once they are created. To make dynamic arrays, arrays that can change their size, you must use Vectors or ArrayLists, which are fairly complicated. To make our array variable actually hold a new array, we have to add a statement like this:
```
arrayName=new dataTypeYouWantArrayToHold[arrayLengthYouWant];
```

An example for the `numbersToFindAverageOf` array is:

```
//note: don't make your variable names this
//long or your fingers will soon be sore
int numberOfNumbersToAvg=11;
double[] numbersToFindAverageOf;
numbersToFindAverageOf=new double[numberOfNumbersToAvg];
```
Just as you can declare and initialize variables in the same statement, you can both declare and create new arrays in the same statement. You just have to put the `new` statement on the same line as the declaration. In the above example, you would do this:

```
int numberOfNumbersToAvg=11;
double[] numbersToFindAverageOf=new double[numberOfNumbersToAvg];
```

Or more compactly:
```
double[] numbersToFindAverageOf=new double[11];
```

To access elements in arrays, use the following syntax:
```
//note: index means which element number that you want
//It starts from zero; first element is 0, second is 1, and so on
arrayName[index]
```

Accessing an element in an array simply gives you the *`index`*th variable stored in the array (starting from zero). An array's maximum element number is one less than its length (if you declare `int[] myArray=new int[2]`, the 2 legal indices (plural of index) will be 0 and 1). You can set or modify array elements accessed this way just as if they were normal variables.
This example demonstrates the power of arrays and `for` loops combined:

```
//makes a new array
double[] someArray=new double[3];
```

```
//sets the values of the elements
someArray[0]=3.14159;
someArray[1]=2;
someArray[2]=-4;

//the length variable of an array holds the number of
//elements in the array. In this case, it is 3.
//To access it, say arrayName.length
for(int i=0;i<someArray.length;i++) {
      //this prints out the element corresponding to index i
      //since i starts at 0, increases by one each
      //time the loop runs, and the greatest value that it
      //reaches is the array’s length-1 (the condition uses <),
      //this loop gets the entire array printed
      //out without going out of bounds
      System.out.println(someArray[i]);
}
```

The output of this code is:

```
3.14159
2.0
-4.0
```

Note: since we are using `double`s, Java puts a .0 after numbers that don’t have decimals (same for `float`s).

If arrays are confusing, you can think of expressions of the form `arrayName[index]` as an easier, more powerful, way to do something like this:

```
int num0=0;
int num1=1;
int num2=2;
System.out.println(num0*2+1);
System.out.println(num1*2+1);
System.out.println(num2*2+1);
```

The declarations and assignations can be replaced with:

```
int[] numbers=new int[3];
numbers[0]=0;
numbers[1]=1;
numbers[2]=2;
```

Or more compactly (shorthand way to do the same thing):

```
int[] numbers={0,1,2};
```

To access the first number, you would say `numbers[0]` instead of saying `num0`
For example, you would replace `num0=0;` with `numbers[0]=0;`

The following line of code:

```
System.out.println(num0);
```

Is equivalent to this:

```
System.out.println(numbers[0]);
```

You could replace the repetitive set of `System.out.println` statements with a `for` loop like this:

```
for(int i=0;i<numbers.length;i++) {
      System.out.println(numbers[i]*2+1);
}
```

The output is:

```
0
1
2
```

Break and Continue statements:
Sometimes you want to escape from loops before they are done or skip all or part of an iteration of a loop. To do this, use the `break` or `continue` statements. `break` ends the loop entirely and `continue` skips after the `continue` statement and goes to the next iteration. They work in any type of loop (a `while` loop is shown below, though you could do this in a `for` loop or a `do-while` loop as well).

```
while(conditionIsTrue)
{
      //you don't usually use both break and continue in the same loop
      //and you certainly don't have to
      //this is just for demonstration purposes

      //in the following code, if something were true, the
      //break statement would be invoked and it would
      //IMMEDIATELY end this while loop and continue with the program
      //as if the loop's condition had been false.
      //normallyDoThis (which occurs in the loop AFTER the break
      //statement) would not be executed

      if(something)
            break;

      //in the following code, if somethinElse were true, the
      //continue statement would be invoked and it would
      //IMMEDIATELY end the current ITERATION of the loop, however,
      //if the loop's condition remained true, the loop would continue
      //beginning with the next iteration. normallyDoThis would not be
      //executed on the current iteration, but it would be executed
      //in later iterations, if somethingElse were false

      if(somethingElse)
            continue;

      normallyDoThis;
}
```

`break`'s usefulness is demonstrated below:

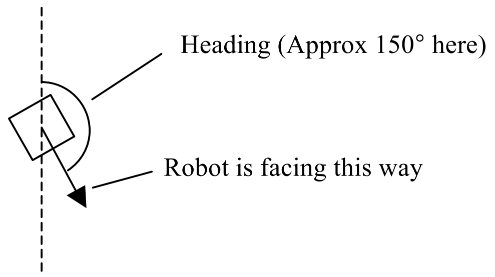
```
//the following code checks to see if the
//array is sorted least to greatest

//creates and initializes arrayToCheck
int[] arrayToCheck={2,4,5,22,56,42,82,99,101};

//the following says if the array is sorted or not. It is initialized
//to true and will remain that way unless it is shown to be unsorted
boolean sorted=true;

//the following scans through the array. Note: the -1 is there so i+1
//is always a legal index (for example, if the array's length were 3,
//the greatest index would be 2. If the -1 were not there, i would only
//get to 2, but i+1 would get to 3, which is illegal)
```

```
for(int i=0;i<arrayToCheck.length-1;i++) {
      if(arrayToCheck[i] > arrayToCheck[i+1]) {
            sorted=false;

            //if one element is unsorted, that means the array isn't
            //completely sorted, and we have no need to waste our time
            //checking any more elements, so we can break
            break;
      }
}

System.out.println(sorted);
```

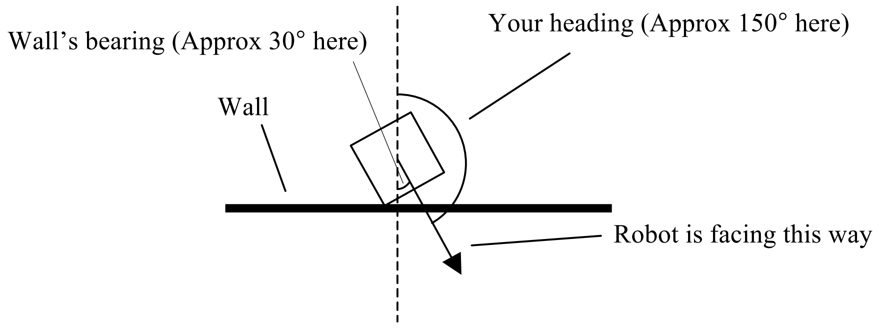
Output:
```
false
```

Appendix A: Headings Versus Bearings:

When robocoding, you will come upon `getHeading()` and `getBearing()` methods. The heading of something is the ABSOLUTE direction that it is facing, counted clockwise from vertical. You can use the `getHeading()` method on your robot, your gun, your radar, and enemy robots.



In robocode, bearings are how far your need to turn YOUR ROBOT RIGHT, to FACE something. This will be a number from -180 to 180. If you call a turn right method with a negative value, it will turn you left. When you call an object's `getBearing()` it will simply return the number of degrees that will most efficiently turn your robot to face it. This angle is your bearing relative to the object, or the object's relative bearing.


Wall's bearing (Approx 30° here)
Your heading (Approx 150° here)
Wall
Robot is facing this way

If your code said:

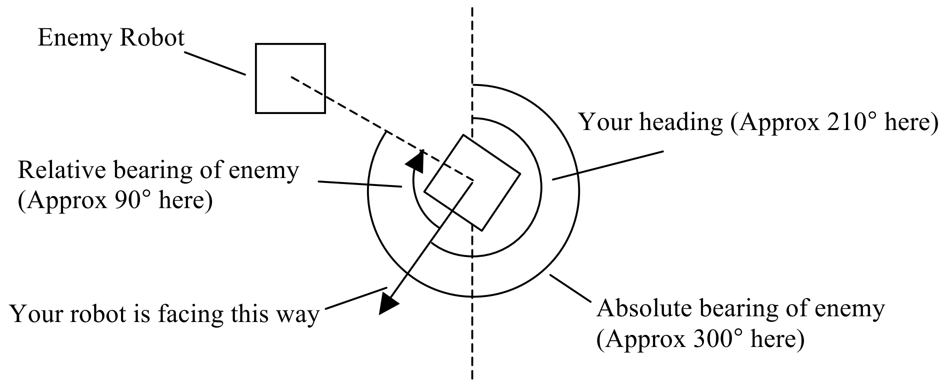
```
public void onHitWall(HitWallEvent wall) {
      System.out.println(wall.getBearing());
}
```

It would print out a number around `30` in the above situation.

Note: there are `getHeadingRadians()` and `getBearingRadians()` methods as well to make life easier for people dealing with radians.

Absolute Bearings:

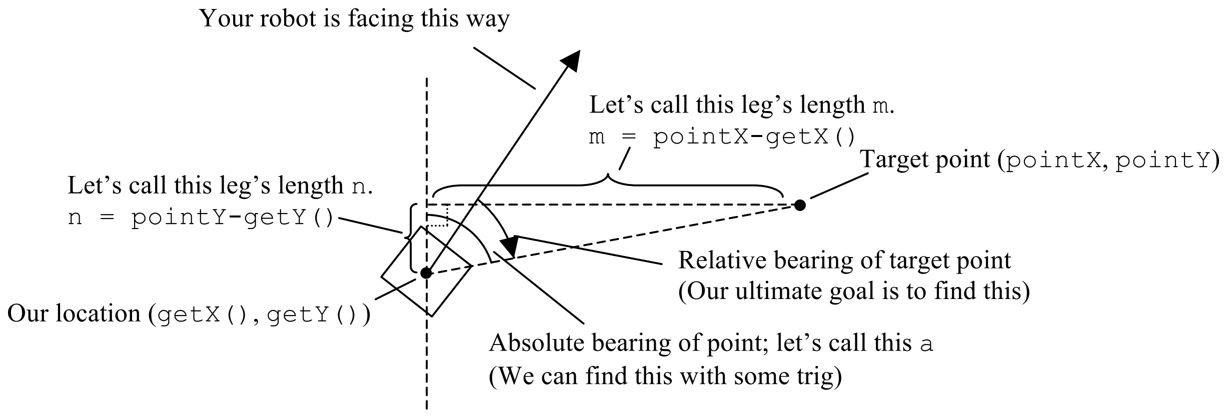
Though these aren't built into robocode, absolute bearings are used so often that they are worth mentioning here. The absolute bearing of something is the actual direction that you would have to be facing to point towards something. It is equal to the sum of your heading and an object's relative bearing (the thingy returned by the `getBearing()` method).


Enemy Robot
Your heading (Approx 210° here)
Relative bearing of enemy (Approx 90° here)
Your robot is facing this way
Absolute bearing of enemy (Approx 300° here)

Absolute bearings, with some simple trigonometry, can be used to derive incredibly useful methods. Below is a walkthrough of making a method that turns you towards a given point.

How to Turn (a part of the robot) Towards a Target Point:
Note: You should be familiar with the trigonometry in the Trigonometry Tutorial before reading this.

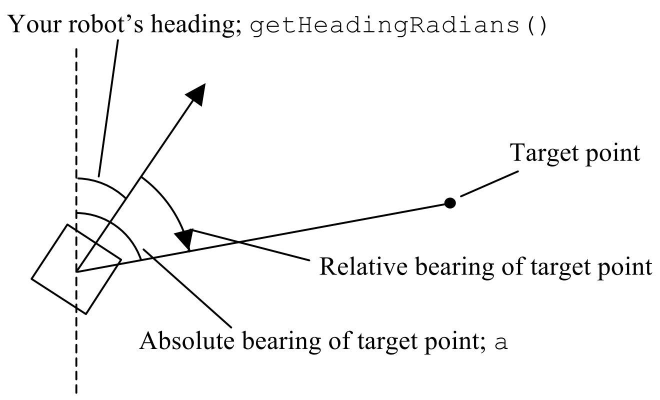
Writing the Absolute Bearing Method
We are trying to find how far we must turn right to face the target point (its relative bearing). We know our x and y coordinates (courtesy of the `getX()` and `getY()` methods), our heading (courtesy of the `getHeadingRadians()` method), and the coordinates of the target point (any point you choose).


Your robot is facing this way
Let's call this leg's length m.
m = pointX-getX()
Target point (pointX, pointY)
Let's call this leg's length n.
n = pointY-getY()
Relative bearing of target point (Our ultimate goal is to find this)
Our location (getX(), getY())
Absolute bearing of point; let's call this a (We can find this with some trig)

Now, since we know the 2 legs of the right triangle, we can determine the tangent of `a`. Tangent is equal to opposite over adjacent, so the tangent of `a` is equal to `m/n`. Since we know the tangent of the angle, we can find the actual angle with the arctangent function and some algebra:

```
tan(a) = m/n
atan(tan(a)) = atan(m/n)
a = atan(m/n)
```

We now know the angle `a`.


Your robot's heading; getHeadingRadians()
Target point
Relative bearing of target point
Absolute bearing of target point; a

Since we know `a`, and we know our own heading, it is trivial to find the relative bearing of the target point. The relative bearing is equal to `a-getHeadingRadians()`. This is obvious if you look at the part of the absolute bearing that is NOT covered by your robot's heading.

All this can easily be transformed into 1 line of code by replacing `a` with what it represents, `atan(m/n)`. We can further simplify this by replacing `m` and `n` with what they represent: `m=pointX-getX()` and `n=pointY-getY()`. Our code turns out as follows:

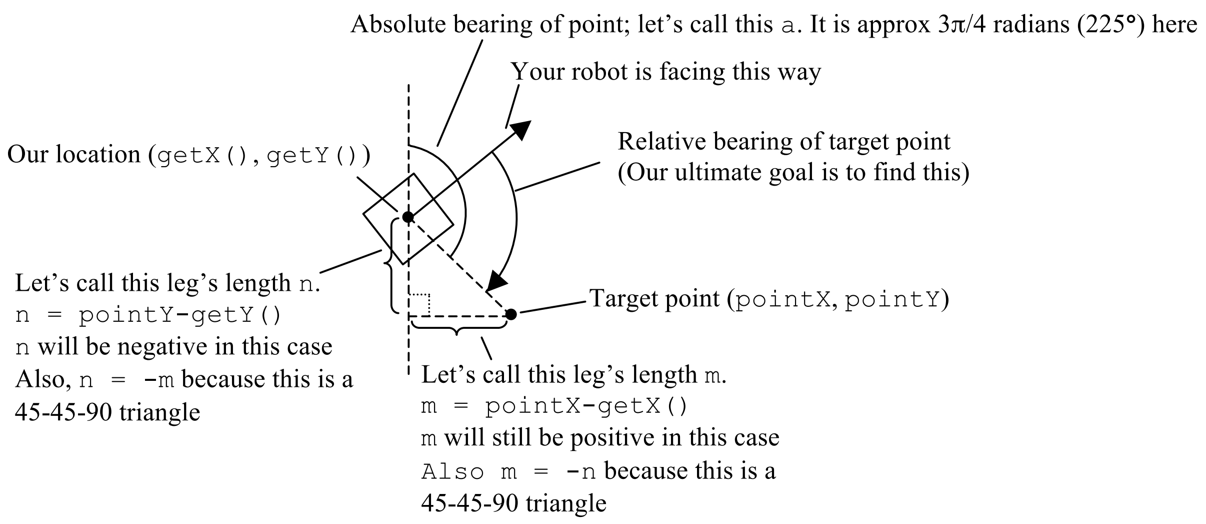
```
//returns the absolute bearing of
//the point (pointX,pointY)
//note: this will not always work (explained below)
double absBearing(double pointX,double pointY)
{
      return Math.atan((pointX-getX())/(pointY-getY()));
}
```

In your code, if you wanted to make your radar or something face some point, you could use the logic above:

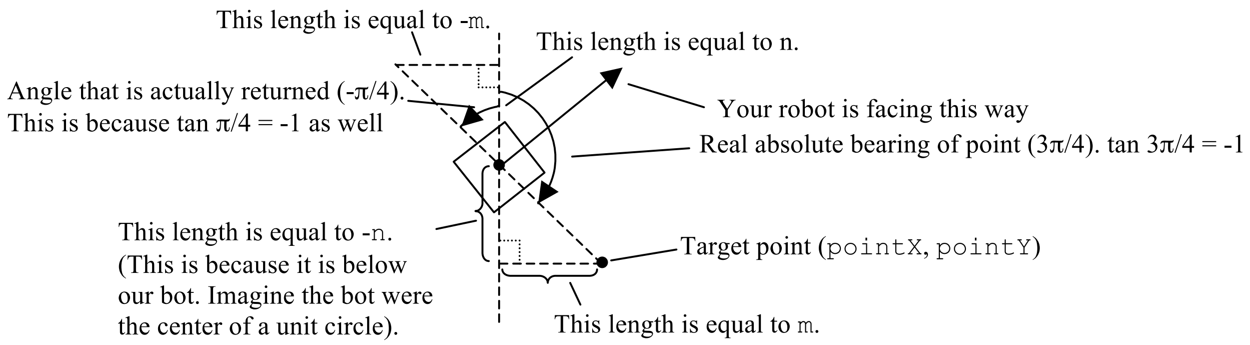
```
double x=50;
double y=100;

//see note
turnRadarRightRadians(robocode.util.Utils.normalRelativeAngle(
absBearing(x,y)-getRadarHeadingRadians());
```

Note: the `robocode.util.Utils.normalRelativeAngle method` ensures that you are most efficiently turning to face the point (turning the short way around). See Normalizing Angles in the Useful Methods Quick Reference Sheet for an explanation.

Before you get too excited, you should test this method. You may be surprised to see that it does not work when your robot is in certain quadrants of the board (relative to the target point). We did nothing to correct for the sign problem! (See trigonometry tutorial.) If your robot is above the target point, it will move in the direction EXACTLY OPPOSITE the one you want! It is demonstrated how this happens below.


Absolute bearing of point; let's call this a. It is approx 3π/4 radians (225°) here
Your robot is facing this way
Our location (getX(), getY())
Relative bearing of target point
(Our ultimate goal is to find this)
Let's call this leg's length n.
n = pointY-getY()
n will be negative in this case
Also, n = -m because this is a
45-45-90 triangle
Target point (pointX, pointY)
Let's call this leg's length m.
m = pointX-getX()
m will still be positive in this case
Also m = -n because this is a
45-45-90 triangle

In the above example, the m/n is -1 since m=-n.When we plug this into Math.atan, we get an angle whose tangent is -1 (namely -π/2 radians, or -45°). We get this instead of 3π/4 radians (225°), which we want because of the convention with inverse trig functions that returns the angle closer to 0°.


This length is equal to -m.
This length is equal to n.
Angle that is actually returned (-π/4).
This is because tan π/4 = -1 as well
Your robot is facing this way
Real absolute bearing of point (3π/4). tan 3π/4 = -1
This length is equal to -n.
(This is because it is below
our bot. Imagine the bot were
the center of a unit circle).
Target point (pointX, pointY)
This length is equal to m.

There is an easy way to correct for this problem. Since the error only occurs in the when the point is below your bot, or its y coordinate is negative relative to you, we have an easy way to figure out if the problem will occur. (The arctangent function will always return a value in the first or fourth quadrant, and in robocode, the quadrants are calculated clockwise from vertical, so the first and fourth quadrants are the top 2.) Since the angle returned when the point is below our bot is always exactly opposite the angle that we want we can correct for it by adding π (180°) to the result of the arcsine function. (This is because tan = sin/cos, and the angle opposite of another has the same sin and cos, except for the sign. The sign is going to be the opposite for both sin and cos. Therefore, the tangents of an angle and the angle opposite it are the same.)
Our code would be as follows:

```
//absBearing - returns the absolute bearing of the point
//at the location (pointX,pointY)
//note: THIS WILL ALWAYS WORK
double absBearing(double pointX,double pointY)
{
      //if the point is below us, add pi to the angle
      if(pointY-getY()<0)
```

```
            return Math.PI+Math.atan((pointX-getX())/(pointY-getY()));
      //otherwise, just return the angle
      else
            return Math.atan((pointX-getX())/(pointY-getY()));

      //note: we could simplify this code into one line with the
      //ternary operator (the shorthand if statement) as follows
      /*
      return ((pointY-getY)<0 ? Math.PI : 0) +
      Math.atan((pointX-getX())/(pointY-getY()));
      */
      //This says says to add 0 if the point is above you and
      //to add pi if it is below you
}
```

To turn our radar to face a point, we would say the same thing as above. To make it face the point (x,y) we could now safely say:

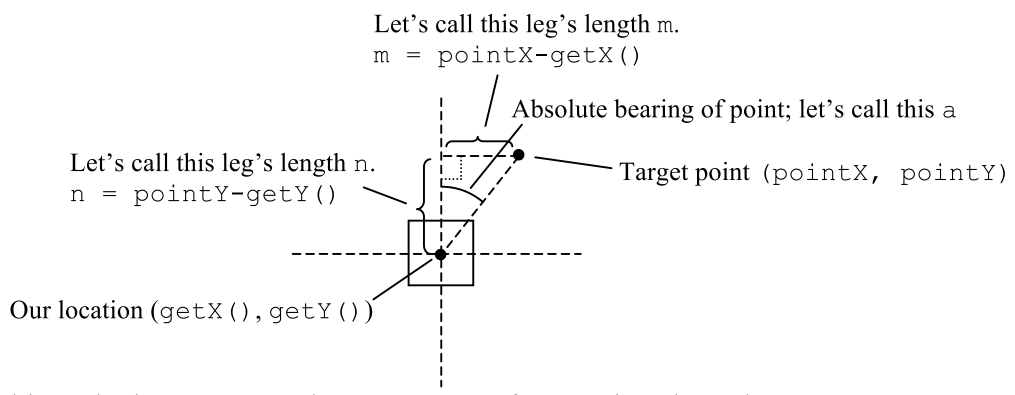
```
double x=someValue;
double y=someValue;
turnRadarRightRadians(robocode.util.Utils.normalRelativeAngle(
absBearing(x,y)-getRadarHeadingRadians());
```

The `atan2` Method

This should work fine. However, we reinvented the wheel by writing it. `absBearing` is virtually identical to the method `Math.atan2(yCoord,xCoord)`, which returns the θ portion of the polar representation of the Cartesian point `(yCoord,xCoord)`. However the following code will NOT work (even though `atan2` corrects for the sign problem):

```
//THIS WILL NOT ALWAYS WORK:
double pointX=someValue;
double pointY=someValue;

//note: in the below line of code, the getY() and the getX()
//are just to make it as if your robot were at the origin so
//that the polar coordinates work correctly (remember, polar
//coordinates are measured from the origin)

turnRadarRightRadians(robocode.util.Utils.normalRelativeAngle(
Math.atan2(pointY-getY(),pointX-getX())-getRadarHeadingRadians());
```

The `Math.atan2` method returns the polar representation of a point (standard trig), not the robocode representation. However, we can fool it into giving us what we want. In the below diagram, since tangent is equal to opposite over adjacent, the tangent of `a` is equal to `m/n`, which is the relative x coordinate over the relative y coordinate. However, the `atan2` method assumes that tangent of the desired angle is going to be equal to `y/x` (where `y` and `x` are given as arguments) because that is what the tangent of an angle in standard trig. This is not standard trig. Here, the tangent of angle `a` is the target point's x coordinate relative to your bot over its y coordinate relative to your bot (`m/n`). Therefore, if we called the `atan2` method with reversed arguments, it would give us the proper robocode angle.

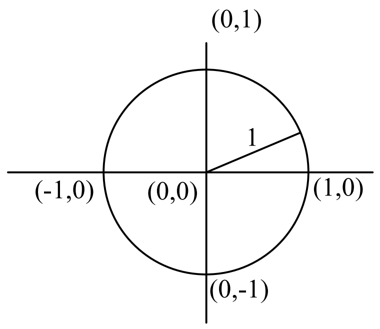

Let's call this leg's length m.
m = pointX-getX()
Absolute bearing of point; let's call this a
Let's call this leg's length n.
n = pointY-getY()
Target point (pointX, pointY)
Our location (getX(),getY())

This code demonstrates the proper use of `atan2` in robocode:

```
//THIS WILL ALWAYS WORK:
double pointX=someValue;
double pointY=someValue;
//the getY() and the getX() are just to make it as if your
//bot were at the origin so that the polar coordinates
//work correctly
//here, we input the x coordinate where y is expected and
//the y coordinate where x is expected to "trick" the
//function into giving us the robocode angle
turnRadarRightRadians(robocode.util.Utils.normalRelativeAngle(
Math.atan2(pointX-getX(),pointY-getY())-getRadarHeadingRadians());
```

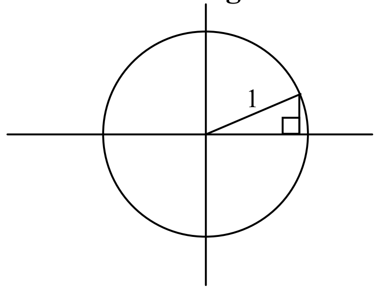
Appendix B: Trigonometry Tutorial:

Here is some basic yet important trigonometry. Trigonometry is a BIG word for something very simple. It all starts with the **unit circle**. A unit circle is a circle with the radius of 1 unit.

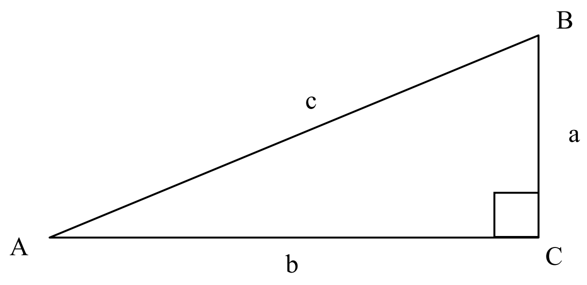


Above is a circle with its center at the **origin**, the center of the coordinate plane. It contains the points (1,0), (-1,0), (0,1), (0,-1).

This circle is highly useful. Let's say that you were given the hypotenuse and an angle of a right triangle. Since 2 angles are given (the right angle is 90 degrees by def of right triangles), and the side is given you are given an **A**ngle, an **A**ngle, and a **S**ide.
This is enough to determine a right triangle (AAS Thm, since sum of degrees in triangle=180, if you know 2 angles, you know the third, and then you can just use ASA). This means that if you know an angle and the hypotenuse of a right triangle, you know the sides. Since all triangles with the same angles have no differences except for their size (they are just bigger versions of the small triangles), the ratios of their sides are constant. These ratios are known as the **Trigonometric Ratios!!!**



Trigonometric Ratios:
Let's drop an altitude from the end of the radius to the x axis in our unit circle. We now have a right triangle. The unit circle is good for representing trigonometric ratios because it makes the hypotenuse of the right triangle 1 (you will see how this helps in a second). The ratios are illustrated below (distances are represented by lower case letters):



Sine (abbreviated sin) = opposite/hypotenuse. sin A=a/c
Cosine (cos): adjacent/hypotenuse. cos A=b/c
Tangent (tan): opposite/adjacent. tan A=a/b

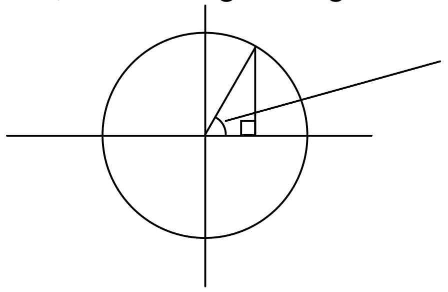
Sin B=b/c
Cos B=a/c
Tan B=b/a

Note: I will denote the measure of angle just by its letter for convenience.

There are other trigonometric ratios defined, but these 3 are the standard ones. Anyway, the others all are easily derivable from the above 3.

The reason a unit circle is useful is because in a unit circle, c=1.
If c=1, then the sines and cosines of angles are simply equal to the lengths the legs of the right triangle! This makes it easier to imagine the ratios.

In a unit circle, we call the given angle the Greek letter theta. Theta is written like this: θ.

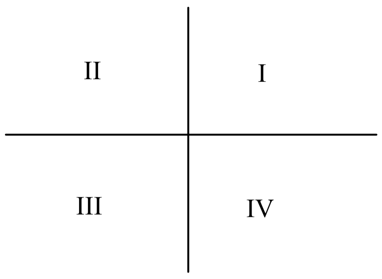


This is θ. It is counted counterclockwise from the positive x axis. A negative θ goes clockwise. Note: In robocode, angles measures are counted clockwise from vertical

In a unit circle, because the hypotenuse is equal to one unit, the y coordinate of the end of the radius is equal to the sine of θ and the x coordinate equal to the cosine. The tangent is equal to the y coordinate over the x coordinate. Though there can't be negative distances, the sines and cosines of angles can be negative if the y or x coordinate of the end of the radius is negative (sin if y, cos if x). The tangent is negative if x or y (not both) are negative. (This is because the tangent is equal to the sine over the cosine. If sin is the y coordinate, and cos is the x, and tan is y/x, then tan is sin/cos) Even though they don't make sense with distances, negative trigonometric ratios actually are logical. You will see as you use them.

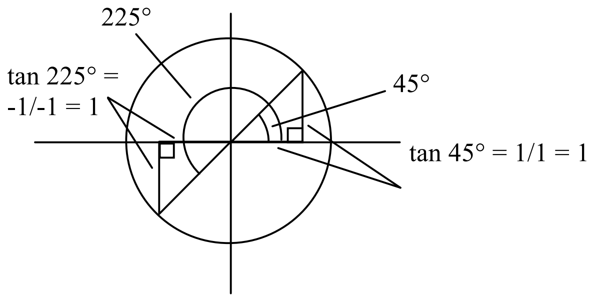
Inverses of trigonometric functions:
The inverse of a trigonometric function is called the arc*functionName*. The opposite of sine is arcsine (asin), cosine is arccosine (acos), and tangent is arctangent (atan). The arcsine of the sine of an angle is equal to the angle, the arccosine of the cosine of an angle is the angle, and so on. Note: the inverse of a function is denoted by a superscript -1 after the functions name. The inverse of `f(x)` would be $f^{-1}(x)$.

The Sign Problem:
When you do the inverse function of a normal trigonometric function of an angle, you might not get the result you expect. The coordinate plane is divided into 4 quadrants based on the signs of the x and y coordinates. They are counted, like θ, counterclockwise from the x axis.



In quadrant I, the sin, cos, and tan are all positive, but this is not so in the other quadrants:
Quad I: sin = +,cos = +, tan = +
Quad II: sin = +, cos=-, tan = -
Quad III: sin = -, cos=-, tan = +
Quad IV: sin = -, cos = +, tan = -

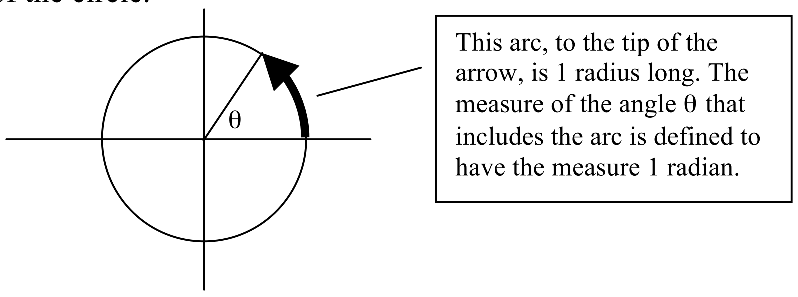
Let's take the tangent of 45°. Since a 45-45-90 right triangle's two legs are the same, tan 45°=1. However, tan 225° is also 1.



Because of this, a computer or calculator cannot know which angle to return. To make it possible to evaluate inverse trig functions with a computer, people came up with a convention. Computers and calculators always return the angle closer to 0°**.** If you use your calculator or computer to evaluate atan(1), you will get 45, not 225. If you asked for atan(-1), you would get -45, not -225 (-45° is equiv to 315° and -225° is equiv to 135°)

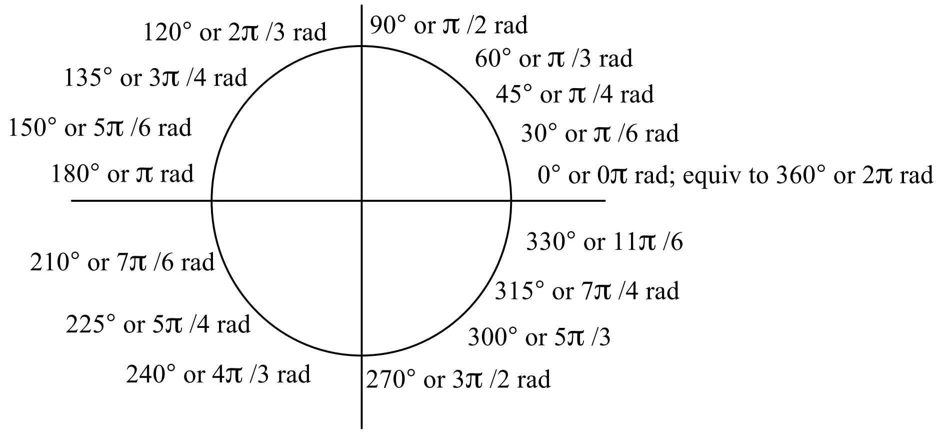
Note about trig functions: Only a few obvious ones are doable by hand, most require calculus to derive and their values are irrational. That is why we use computers and calculators to figure them out.

Radians:
Degrees are arbitrary. It makes no sense to divide a circle into 360 segments. The 360 is just what somebody decided a long time ago.
There is a better system, radians. Radians are based on the radius of the circle. The circumference of a circle is equals π times the diameter (πD) or 2π times the radius (2πr).
Since C=2πr, C/r=2π (C is circumference)
Therefore, the radius of a circle goes into its circumference 2π times, NO MATTER WHAT. **The number of radii (plural of radius) that go into a circle is constant**.
Because of this, we can make a new, more logical way of measuring angles. Imagine you start at the point (1,0) and travel counterclockwise around the circumference of the circle. Suppose you measure the distance you've traveled in units equal to the length of the radius of the circle.


θ
This arc, to the tip of the arrow, is 1 radius long. The measure of the angle θ that includes the arc is defined to have the measure 1 radian.

Radian to Degree Conversions:
Since the radius goes into any circle 2π times, there are always 2π radians in a circle. In the degree system, there are 360 degrees in a circle. Therefore 2π radians is equivalent to 360 degrees.
We can derive the following identity from this:
r = number radians
d = number degrees
r/2π=d/360
This simplifies to r/π=d/180
From this we can convert radians to degrees.
In English, the identity r/2π=d/360 becomes obvious. You are just saying "What fraction of the whole circle equals what fraction of the whole circle?" The only difference is the unit.

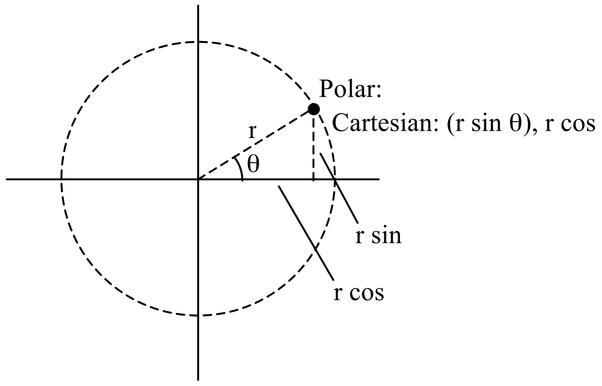
Following are some useful angle measures in both radians and degrees:



Radians are VERY important for programming because in Java, the trigonometric functions return in radians. If you printed out atan(1), you would not get 45 degrees but rather 0.7853981635, or π/4. I make mistakes by mixing up my radian and degree measures all the time.

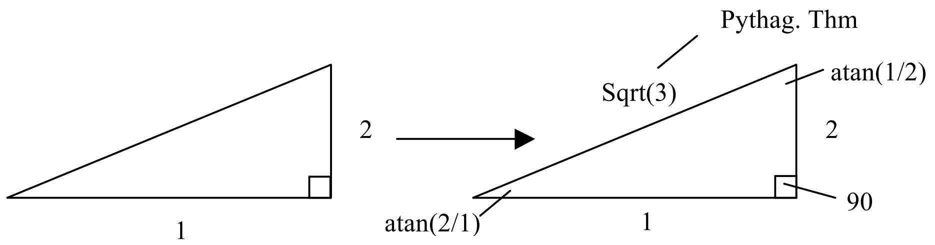
Polar coordinates
Just as you can express points on a coordinate plane with normal coordinates (called Cartesian or rectangular coordinates), you can express points using circular coordinates (I am not sure if people actually call them this), or polar coordinates. Polar coordinates contain 2 values: r and θ. They are written (r, θ). r is the radius of an imaginary circle whose center is at the origin and that includes your point, and θ is the angle that the point represents when it is expressed on the circle.

Here is a point expressed in 2 different ways. In polar coordinates, it is expressed with r, the length of the radius of the imaginary circle, and θ, the number of degrees around the imaginary circle. To convert these to Cartesian coordinates, you multiply r by the sine and cosine of θ (remember that the x coordinate on a unit circle is the cosine and the y coordinate is the sine)



The only reason that polar coordinates are worth mentioning is that Java provides a VERY easy way to go from Cartesian to polar coordinates, the `atan2` method. This method is explained above.

Solving oblique (non-right) triangles
Solving a triangle is defined as finding all angles and sides from given measures. This is easy if you have a right triangle, you can simply use the trigonometric functions and/or the Pythagorean theorem (in right triangle ABC (above), $a^2 + b^2 = c^2$)

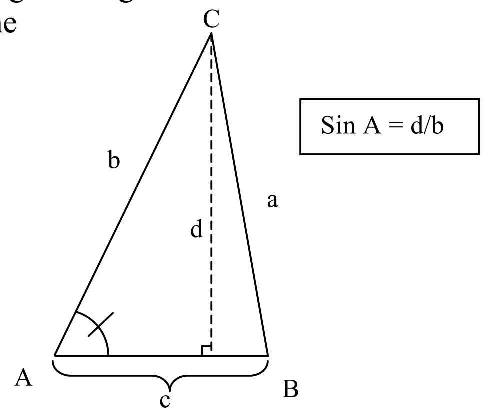


However, what do you do if you are given a non-right, or oblique triangle?

The law of sines:
For any triangle ABC (lowercase letters correspond to lengths of sides opposite angles):
a/(sin A)=b/(sin B)=c(sin C)
Use this if you are given angle, angle, and side (AAS) or ASA. Also, you can use the law of sines find some attributes of the triangle if you are given SSA.
Remember that if you know 2 angles, you know all 3.
Go to http://en.wikipedia.org/wiki/Law_of_sines for a proof

The law of cosines:
For any triangle ABC (lowercase letters correspond to lengths of sides opposite angles):
$a^2=b^2 + c^2 – 2bc*\cos(A)$
$b^2=a^2 + c^2 – 2ac*\cos(B)$
$c^2=a^2 + b^2 – 2ab*\cos(C)$
Use this if you are given SAS or SSS.
FYI: ∗ (or *) means multiply on computers.
Go to http://en.wikipedia.org/wiki/Law_of_cosines for a proof

Although angles A, B, and C are not parts of a right triangle, they still have sines, cosines, and tangents (this is obvious but when I learned this, I didn't get it at first). The trigonometric functions are defined for an angle no matter what triangle it is in because, to represent the functions, you just have to make a unit right triangle (one with hypotenuse length of 1) with given angles. It doesn't matter where the angle is, its trig functions are always the same



Appendix C: A Sample Robot

This is a robot called AheadBot. It bounces off walls, moves with its gun perpendicular to its body, spins its gun at the end of each motion (each time it hits wall), and fires at whatever is in front of it. Copy this into your folder in the robots directory, change the package to yours, and compile this robot. Try fighting it against Corners (in the sample package).

```
package jc;
//your package should be your initials
//this corresponds to the folder is where your robots are stored

import robocode.*;

/**
 * AheadBot - a robot to demonstrate robocode
 */
public class AheadBot extends Robot
{
      //1 for forward, -1 for backwards
      int direction=1;

      /**
       * run: AheadBot's default behavior
       */

      //this is where the program starts
      public void run() {
            //note: code here only runs once

            //makes gun face sideways
            turnGunRight(90);

            //the following variable holds a distance as long as the
            //robot will ever have to travel. This is the diagonal from
            //one corner to another
            //this works because of the Pythagorean Theorem
            //(a^2 + b^2 = c^2  ->  sqrt(a^2 + b^2) = c)
            //n^2 means n squared in computer notation
            //therefore, the distance from corner to corner of
            //the battlefield is sqrt(fieldwidth^2 + fieldheight^2)
            //in code, it comes out as below

            double maxBattleFieldDimension =
            Math.sqrt(getBattleFieldWidth() * getBattleFieldWidth() +
            getBattleFieldHeight() * getBattleFieldHeight());

            //note: gun turns with robot
            //radar turns with gun (to see radar, go to Options >
            //Preferences, then click visible scan arcs)
            //the setAdjustGunForRobotTurn(),
            //setAdjustRadarForRobotTurn(), and
            //setAdjustRadarForGunTurn() make the robot part mentioned
            //in the method name independent of the other robot part
            //mentioned in the method name
```

```
		while(true) {
			//note: code in here repeats throughout the battle

			//this moves you ahead maxBattleFieldDimension units
			//if direction is 1, otherwise, it moves you
			//ahead -maxBattleFieldDimension units (backwards)
			//the reason we move maxBattleFieldDimension pixels
			//is so that the robot doesn't pause after it has
			//finished its move
			//This way, it will always hit a wall before its
			//motion is done

			ahead(maxBattleFieldDimension*direction);

			//spins the gun around after it reaches the
			//end of its motion
			turnGunRight(360);
		}
	}

	/**
	 * onScannedRobot: What to do when you see another robot
	 */
	public void onScannedRobot(ScannedRobotEvent e) {
		//fires a bullet of power 1
		fire(1);

		//note: since the radar moves with the gun, whenever we
		//scan an enemy, our gun is facing him
	}

	/**
	 * onHitByBullet: What to do when you're hit by a bullet
	 */
	public void onHitByBullet(HitByBulletEvent e) {
		//this moves you perpendicular to the bullet
		turnRight(e.getBearing() + 90);

		//e.getBearing() here returns the number of degrees you
		//must turn right to face the bullet, so turning 90 past
		//that turns you perpendicular to it
	}

	public void onHitWall(HitWallEvent e) {

		//toggles the movement direction when you hit a wall
		if(direction==1)
			direction=-1;
		else
			direction=1;

		//note: this could be shortened with the ternary operator.
		//all of this could be replaced with:
		//direction = direction==1 ? -1 : 1;
	}

}
```

Appendix D: Historical Robots

SandboxDT by Paul Evans
This is probably the most famous robot ever in the history of robocode. Was deemed "unbeatable" for a long time.
Wiki Page:
http://robowiki.net/cgi-bin/robowiki?SandboxDT
Download the latest version (version 3.02 as of 12/23/06; non-melee. Is this a bug?) by clicking on the Download from Repository link on the wiki page.
Download the melee version (latest is 2.71m as of 12/23/06) at:
http://www.robocoderepository.com/BotDetail.jsp?id=2008

Dookius by Voidious
1v1 only champion and highest ranked bot as of 12/23/06
Wiki Page:
http://robowiki.net/cgi-bin/robowiki?action=browse&id=Dookious&oldid=Dookius
Download by clicking on the link underneath "Great, I want to try it. Where can I download it?"

Aleph by rozu
Melee 2nd place as of 12/23/06 (longtime champion however)
Wiki Page:
http://robowiki.net/cgi-bin/robowiki?Aleph
Download by clicking on the link underneath "Great, I want to try it. Where can I download it?"

Shadow by ABC
The best overall robot, melee and 1v1 combined (as of 12/23/06). Also, melee champion as of 12/23/06.
Wiki Page:
http://robowiki.net/cgi-bin/robowiki?Shadow
Download the best version of Shadow (as of 12/23/06) at:
http://robocode.aclsi.pt/abc.tron3.Shadow_3.66b.jar

FloodMini by Kawigi
Very famous MiniBot (bot with codesize rated < 1500, see http://robowiki.net/cgi-bin/robowiki?WeightClass for the "weight" classes)
Wiki Page:
http://robowiki.net/cgi-bin/robowiki?FloodMini
Download by clicking on the link underneath "Great, I want to try it. Where can I download it?" (scroll down)

Appendix E: Robocode Physics and Mechanics

Note: + means add, - means subtract, * means multiply, / means divide

Robocode Time/Space:
Time in robocode is divided into discreet "ticks" and space is divided into units which are sized approximately 1 pixel, depending on how much they are scaled (units scaled to fit on screen). Angles in robocode are calculated clockwise from vertical. After you have been inactive for "Inactivity Time" number of ticks (specified in the rules tab when you start a battle, default is 450 ticks) robots begin to lose health if they are "inactive". Inactivity in robocode is defined by not losing energy (you lose energy when you fire or collide with enemy robots, and AdvancedRobots lose energy when they hit a wall).

Robot Structure:
Robots consist of 3 parts, a body, a gun turret, and radar device. The turret is mounted on the body and the radar is mounted on the gun (each spins with the parts below it). Your radar can only see enemy robots (not flying bullets). Also, you cannot get the direction the enemy's radar or gun is facing, just the way its body is. You can also get an enemy's velocity and energy.
Robots are treated as non-rotating squares.
Robot Size: 36 x 36 units^2

Movement Physics:
Max Acceleration: 1 unit per tick per tick
Max Deceleration: 2 units per tick per tick
Max Velocity (negative means moving backwards): 8 and -8
Max Body Rotation: `(10 - 0.75 * Math.abs(velocity))` degrees per tick
Max Turret Rotation: `20` degrees per tick (plus effect of body rotation)
Max Radar Rotation: `45` degrees per tick (plus effect of body/turret rotation)

Note: robots start at random locations on the battlefield, facing random directions at the beginning of each round

Energy (Bullets and Health):
In robocode, energy serves both as ammunition and health. When you fire a bullet (legal power is 0.1 to 3), your robot loses energy depending on the bullet's strength (higher is more loss). When a bullet hits, the enemy loses energy bullet and you gain energy, based on the strength of the bullet. Also, every time you fire, your gun becomes hot based on the strength of the bullet. You can specify the gun cooling rate under rules when you start the battle.
Legal bullet powers: `0.1` to `3`
Bullet Speed: `20-3*power`
Bullet Damage: `4*power. If power > 1, it does an additional 2*(power-1)`
Energy lost when fired: same as `power`
Energy gained on hit: `3*power`

Gun Heat Generated: 1 + `power/5`. You cannot fire if `gunHeat > 0`. Guns start with heat 3.0 at the beginning of each round.
Gun Cooling Rate: Specified by user under rules at the beginning of each battle. Default is .1 heat units per tick.

Collisions:
Robot-Robot: `.6` damage. If a robot is moving away from the collision, it will not be stopped.

Robot-Wall (AdvancedRobots only): `Math.abs(velocity)*0.5 - 1` If this value is less than `0`, it is set to `0` (so no negative damage)

Robot-Bullet: Self-explanatory. Damage calculated by formula under Energy (see above)

Bullet-Bullet Collisions: These are very complicated. See http://robowiki.net/cgi-bin/robowiki?BulletShielding (scroll down)

Quick list of Robocode/Java stuff:
http://robowiki.net/cgi-bin/robowiki for everything robocode.
http://robowiki.net/cgi-bin/robowiki?GamePhysics for the physics of robocode
http://java.sun.com/j2se/1.5.0/docs/api/ for info on java (the API)
http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Math.html for the Math class API
http://java.sun.com/docs/books/tutorial/ for an in-depth Java tutorial
For the robocode API, go to the help menu in Robocode and click on robocode API

`if` Statements

```
if(condition) {
        doThisStuffIfTrue;
}
else {
        doThisStuffIfFalse;
}
```

Ternary Operator (shorthand if statement):
`condition ? doThisIfTrue : doThisIfFalse`

Variables
`varType varName;`
Valid types include (capitalization matters):
`int`, `long`, `float`, `double`, `char`, `String`, and `boolean`
Example:
`int someInteger;`

Boolean Operators:
< (less than), > (greater than), == (equal to for testing equality), <= (less than or equal to), >= (greater than or equal to), != (not equal to).
Boolean Logic Operators (ordered by precedence): `&&` (AND), `||` (OR), `!` (NOT)
You can group these operators with parentheses.
To compare `String`s, use
`string1Name.equals(string2Name)`

Method Declarations

```
returnType methodName(argument,argument . . .)
{
        stuffToDoInMethod;
        return valueOfTypeSpecifiedAbove;
}
```

If the method returns nothing don't write a return statement and replace *returnType* with `void`
Arguments are declared like variables and are local to the method

Loops
`while` loops:

```
while(condition)
{
        stuffToRepeat;
}
```

do-`while` loops

```
do
{
        stuffToRepeat;
}
while(condition);
```

`for` loops:

```
for(sentinel=initialValue;condition;updateStatment)
{
        stuffToRepeat;
}
```

Java Math:
To add stuff, use `+`
To multiply use `*`
To subtract, use `-`
To divide, use `/`
To do modular arithmetic, use `%`
To raise to powers use
`Math.pow(base,exponent)`
Use parentheses to group terms
When you divide an integer by another integer, it truncates; `5/2` is equal to 2
To prevent this, put a .0 after one of the numbers
`5.0/2` is equal to 2.5
Order of operations: parentheses, `*` and `/` and `%`, `+` and `-`

Java Trig: See Useful Methods Quick Reference Sheet

Java Shorthand: `x+=y` means `x=x+y`, `x-=y` means `x=x-y`, `x*=y` means `x=x*y`, `x/=y` means `x=x/y`, `x%=y` means `x=x%y`, `x++` means `x=x+1`, `x--` means `x=x-1`

Type Casting and Conversion Methods:
Type Casting (between primitive types):
`(typeToConvertTo)thingToConvert`

Conversion Methods: See Useful Methods Quick Reference Sheet

Arrays
To declare an array variable: `varTypeToHold[] arrayName;`
To make a new array: `arrayName=new varTypeToHold[arrayLength];`
To access array elements (index is the number of the element that you want starting from 0): `arrayName[index]`
To get an array's length: `arrayName.length`

Useful Methods Quick Reference Sheet

See http://java.sun.com/j2se/1.5.0/docs/api/ for all Java methods. Note: the Math and String classes contain many useful methods.

Java Trig Functions:

Note: In robocode (not Java in general), angles are counted clockwise from vertical and **coordinates are counted from the lower left** corner of the screen (always positive). Also, **all Java trig functions use radians.**

Normal Trig Functions:

```
Math.sin(angleInRads) //returns the sine of given angle
Math.cos(angleInRads) //returns the cosine of given angle
Math.tan(angleInRads) //returns the tangent of given angle
```

Inverse Trig Functions:

`Math.asin` returns the arcsine of a given angle. Note: always returns a radian value from π/2 to -π/2. This means it ALWAYS returns angles in the first or fourth quadrant.

`Math.asin(sinOfAngle)`

`Math.acos` returns the arccosine of given angle (returns 0 to π).

`Math.acos(cosOfAngle)`

`Math.atan` returns the arctangent of given angle (returns π/2 to -π/2).

`Math.atan(tanOfAngle)`

There is one more critical method: `Math.atan2`. It returns the θ portion of the polar representation of the Cartesian point (x,y). Note the order of the arguments, first y, then x.

`Math.atan2(y,x)`

Normalizing Angles:

The following method converts any angle to an equivalent angle between -π and π radians (equiv -180° and 180°) Use it to compute the most efficient way to turn. For example, if you put in 3π/2 (same as 270°), it would return -π/2 (same as -90°) because if you were turning right 3π/2 radians, you might as well turn left -π/2 radians which would be more efficient.

`robocode.util.Utils.normalRelativeAngle(angleInRads)`

The following method returns 0 to 2π radians (equiv 0° to 360°) to turn angles greater than 2π or less than 0 into "normal" angles. For example, if you put in -π/2, you would get 3π/2, which is the same direction but expressed in "normal" terms.

`robocode.util.Utils.normalAbsoluteAngle(angleInRads)`

Radian/Degree Conversions:

```
Math.toDegrees(radianVal)
Math.toRadians(degreeVal)
```

Useful String Methods (not including conversion methods):

`nameOfString.length()` Returns length of String

`String1Name.equals(String2Name)` Returns `true` if the `Strings` are the same, else `false`

`nameOfString.toUpperCase()` Returns an upper case version of the `String`

`nameOfString.toLowerCase()` Returns a lower case version of the `String`

`nameOfString.substring(startIndex)` Returns a substring of the `String` from `startIndex` to the end.

`nameOfString.substring(startIndex, endIndex)` Returns a substring of the `String` from `startIndex` to `endIndex-1` inclusive

Conversion Methods:

`String` to Number:

`WrapperClassName.parseVarType(StringToConvert)`

Wrapper classes' names are the capitalized and non-abbreviated versions of their primitive types (e. g. `double`'s wrapper class is `Double`, `int`'s is `Integer`, etc.)

`String to char:`

`nameOfString.charAt(index) or`
`nameOfString.charToCharArray()`

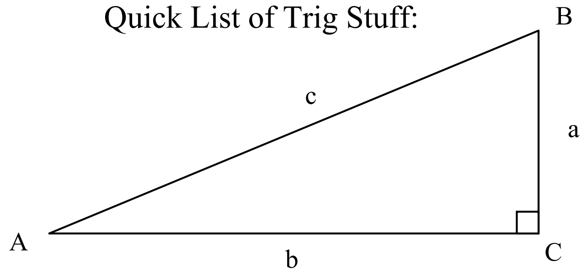
Primitive Type to `String`:

`String.valueOf(valueToConvert)`

Primitive Type to Primitive Type: See Robocode/Java Quick Reference Sheet

NOTE: in ROBOCODE, not Java in general, angles are counted CLOCKWISE FROM VERTICAL instead of counterclockwise from horizontal.

Quick List of Trig Stuff:



Trigonometric Functions:
Sine A=a/c abbrev. sin
Cosine A=b/c abbrev. cos
Tangent A=a/b abbrev. tan
sin B=b/c
cos B=a/c
tan B=b/a

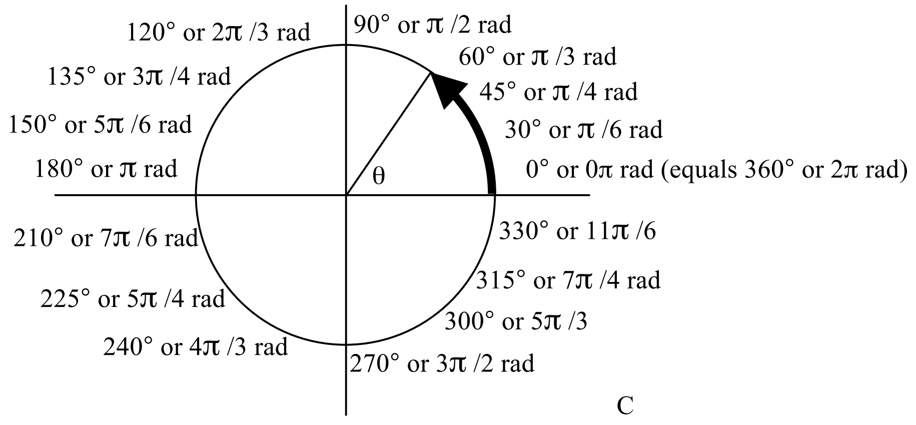
Inverses of Trigonometric Functions:
$\sin^{-1}$ (a/c) = arcsine (a/c) = A abbrev asin
sos^{-1} (b/c) = arccosine (b/c) = A abbrev acos
$\tan^{-1}$ (a/b) = arcsine (a/b) = A abbrev atan
asin (b/c) = B
acos (a/c) = B
atan (b/a) = B

Sign Problem: Inverse trig functions could return 2 possible values, so there is a convention that says to return only the angle closer to 0°
Pythagorean Theorem:
In right triangle ABC, $a^2 + b^2 = c^2$

Common/Useful Trig Ratios:

θ°	θ rad	Sin	Cos	Tan
0	0	0	1	0
30	π/6	1/2	√3/2	√3/3
45	π/4	√2/2	√2/2	1
60	π/3	√3/2	1/2	√3
90	π/2	1	0	undef

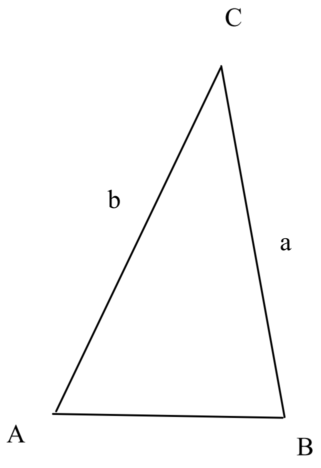
Radians:
Radians are the number radius lengths it takes to cut off a given angle θ by going around a circle.
Since C=πD=2πr, C/r=2π. This means that there are 2π radius lengths in the circumference of any circle, or that it takes 2π radians to go all the way around a circle.
Radians/degree conversions:
r/π = d/180
Radians to Degrees:
d = 180r/π
Degrees to Radians:
r = πd/180



Laws of Sines/Cosines
Use when you are given an oblique (non-right) triangle to solve

Law of Sines:
For any triangle ABC,
a/(sin A)=b/(sin B)=c(sin C)
Use if you are given AAS, ASA, or SSA (you can only find some parts with SSA).
Always remember that if you know 2 angles, you know all 3.

Law of Cosines:
For any triangle ABC,
$a^2=b^2 + c^2 - 2bc*\cos(A)$
$b^2=a^2 + c^2 - 2ac*\cos(B)$
$c^2=a^2 + b^2 - 2ab*\cos(C)$
Use if you are given SAS or SSS.
Always remember that if you know 2 angles, you know all 3.



Polar coordinates: Coordinates of a desired point, expressed (r, θ). r is the radius of an imaginary circle centered on the origin, θ is the number of radians around the circle the point you want is.