3

Programming

// Basic Operators

Often when you are programming you will need to do simple (and sometimes not so simple) mathematical operations. The signs used to do this vary from very simple to confusing if you've never seen them before. Below is a table of definitions as well as some examples:

Arithmetic Operators

Arithmetic operators are your standard mathematical signs

+ (addition)
- (subtraction)
* (multiplication)
/ (division)
% (modulus)
= (assignment)

Relational Operators

Relational operators are used to compare values and variables

== (equality)
!= (inequality)
> (greater-than)
< (less-than)
>= (greater than or equal to)
<= (less than or equal to)

Pay attention to = and ==.
= is used to assign variable values,
== to compare values.

Logical Operators

Logical operators are used to join two or more conditional statements together

! (NOT)
&& (AND)
|| (OR)

Relational Operator Example

```
if (x!=7){
//loop body code here
}
```

Compares x to the number 7,
executes code inside body loop
if the value of x **does not equal 7**

Logical Operator Example:

```
if ((x==7)||(x==9)){
//loop body code here
}
```

Compares x to the number 7 and 9,
executes code inside body loop
if the value of x equals 7 **or** 9

// Comments

As you use code other people have written you will notice //, /* and */ symbols. These are used to "comment" lines out so they do not affect the code. This way people who write code can add comments to help you understand what the code does. Good code has comments that explain what each block of code (functions, classes, etc.) does but does not explain simpler portions of the code as this would be a waste of time. Commenting lines out is also a very useful tool when you are writing code yourself. If you have a section of code you are working on, but isn't quite finished or doesn't work, you can comment it out so it does not effect the rest of your code when you compile or upload it.

//

This is used to comment out a single line

//commented out line

/*

This is used to start a section of commented lines

/*comments start here

*/

This is used to end or close a section of commented lines

comments end here*/

// Vocabulary: Variable, Boolean, Integer, Character, Value

Variables are one of the most important concepts in computer programming. But what exactly are **variables**? **Variables** are like baskets that hold pieces of information. There are a couple different kinds of **variables** depending on what kind of information you need to keep track of. You have probably already heard of most of the different kinds of **variables**. Here are the definitions of three different kinds of **variables**. There are more types of variables, but, let's start with these.

- **Boolean variable:** A boolean variable can be true or false (one or zero).
- **Integer variable:** An integer variable can be any whole number between −32768 and 32767.
- **Character variable:** A character variable can be any one letter (or punctuation or symbol).

Below is a robot, answer the questions to the right of the robot and be as silly as you want. Then write the type of **variable** you would use to store this information. For a **boolean** write "boolean", for an **integer** write "int" and for a **character** write "char".
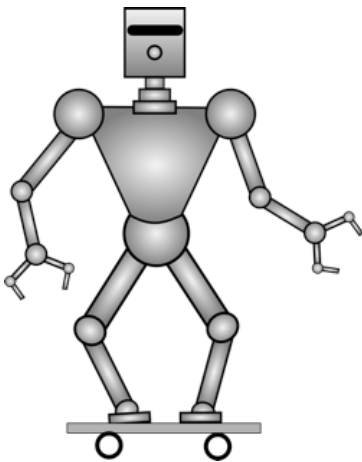
Is this robot good at skateboarding? _____ Variable type:_____

How old is this robot?_____ Variable type:_____

What is the first letter of this robot's name?_____ Variable type:_____

How many years has it been skateboarding? _____ Variable type:_____

Is it wearing pants? _____ Variable type:_____

What is the first letter of the robot's dog's name? _____ Variable type:_____

Is the robot going to crash? _____ Variable type:_____

How many feet of air has this robot gotten? _____ Variable type:_____

The number, or character, you put into a **variable** is called its **value**. Once you have created a **variable** you can change the **value** whenever you need to. For example, if we decided the robot is 1000 years old, in a year we need to be able to change its age to 1001. First we need to create a **variable** to keep track of its age. We can name the **variable** whatever we want, but "age" makes sense so we'll go with that. Then we need to put a **value** into the **variable**. The first **value** was 1000, but a year later we delete that **value** and replace it with the new **value**, 1001. Pretty easy, huh? If we wanted to keep track of how old the robot used to be when we met it we could create a new **variable** called "ageWeMet". That way when we have to change the "age" **variable** we can keep track of how old the robot was when we met it in the other **variable** "ageWeMet". You may have noticed that there are no spaces in the name of this second **variable**. That is because **variable** names can't have any spaces.

Circle the **variable** in the sentences below and put a box around the **value**.

The robot's favorite letter is Q. The robot's height is 100 ft.

The robot's power is on.

// Vocabulary: Boolean, Declare, Assign

OK! You're ready to start programming your first **boolean** variable. Anytime you see *italics like this* it is an example of how you would write something in the Arduino language.

• A **Boolean** variable is the simplest kind of variable, it is either true or false.
• True is represented by a one or HIGH and false is represented by a zero or LOW.
• HIGH can be used as true, but it means there is electricity flowing through a circuit.
• LOW can be used as false, but it means there is no electricity flowing through a circuit.
• To create a **Boolean** variable you type the following: *boolean variableName;*
• Creating a variable is called "**declaring**" a variable.
• The variableName can be anything you like, but it should make sense to you.

For example you could **declare** a **Boolean** variable named *dayLight(boolean dayLight;)* that represents whether it is daytime or not. Once you have **declared** your variable it is not equal to anything, it is empty and waiting for you to set it equal to true or false. To do this you type the following: *dayLight = true;* or *dayLight = 1;*. (Don't forget the ; at the end, it's very important! It is called a semicolon and it tells the computer that you are finished doing something.)

This means that dayLight is true, and you can see the sun. Setting a variable equal to a value is called "**assigning**". **Declare** three **Boolean** variables about the robot on this page in the spaces below and then **assign** them values of true or false (or one or zero). Remember, you can name the variables whatever you want! They're your variables, it's up to you. Look at the example above if you are unsure of how to **declare** and **assign**. (Don't forget the semicolons at the end of each line, they're important!)

Declare:

Assign:

List three of the silliest things you can think of that you might keep track of with a **boolean** variable. Examples: Do I have peanut butter in my ear? Are penguins good to use as dodgeballs?

_____

_____

_____

_____

Now pick one of the silly ideas above. In the space below **declare** your silly variable and then **assign** it a value. For example: *boolean peanutButter; peanutButter = true;* This means that I do have peanut butter in my ear... maybe I am saving it for lunch.

_____

_____

_____

_____

// Vocabulary: Integer, Declare, Assign

Wow! You're ready to start programming your first **integer** variable. Anytime you see *italics* it is an example of how you would write something in Arduino language.

• An **Integer** variable is a number (no fractions or decimals) between -32768 and 32767.
• To create an **Integer** variable you type the following: *int variableName;*
• This is called "**declaring**" a variable.
• The variableName can be anything you like, but it should make sense to you.
• To **assign** an **Integer** variable the value 120 type the following: *variableName = 120;*

For example you could **declare** an **Integer** variable named clouds (*int clouds;*) that represents the number of clouds in the sky. Once you have **declared** your variable it is not equal to anything, it is empty and waiting for you to set it equal to a number between −32768 and 32767. To do this you type the following: *clouds = 8;*. (Don't forget the ; at the end. This is called a semicolon and it's how the computer knows you are finished doing something.)

This means that you can see eight clouds in the sky. Setting a variable equal to a value is called "**assigning**". **Declare** three **Integer** variables about the picture on this page in the spaces below and then **assign** them values between −32768 and 32767. Include at least one variable with a negative value and one variable with a value greater than ten. Feel free to make up variables and values that you can't actually see in the picture. Try to keep it making sense. Look at the example above if you are unsure of how to **declare** and **assign**. (Don't forget the semicolons at the end of each line!)

Declare:

Assign:

List three of the silliest things you can think of that you might keep track of with an **integer** variable. Example: How many pieces of ham do I have in my pocket? How many bugs could you fit in a rocket?

_____

_____

Now pick one of the ideas above. In the space below declare your variable and **assign** it a value. For example: *int ham; ham = 1073;* I either have big pockets or small pieces of ham.

_____

_____

// Vocabulary: Character, Declare, Assign

OK! You're ready to start programming your first **character** variable. Anytime you see *italics* it is an example of how you would write something in the Arduino language.

• A **Character** variable is a single letter, symbol or number.
• To create a **Character** variable you type the following: *char variableName;*
• This is called "**declaring**" a variable.
• The variableName can be anything you like, but it should make sense to you.
• To **assign** a **Character** variable the value "Q" you type the following: *variableName = 'Q';*

For example you can **declare** a **character** variable named weather (*char weather;*) that uses a letter to represents the weather. You can use the letter R to mean it is raining, S for snow, and C for clear. Once you have **declared** your variable it is not equal to anything, it is empty and waiting for you to set it equal to a **character**.

To do this you type the following: *weather = 'C';*. (Don't forget the ; at the end. This is called a semicolon and it's how the computer knows you are finished doing something.) Also, there are many different **character** types other than a letter: !?*%$&@ are all valid **character**s.

For example, *weather = 'C';* means that the sky is clear, but that's just because you decided it means that. C could mean whatever you need to keep track of. For example C could mean that it is cold out, if that's what you decided. Setting a variable equal to a value is called "**assigning**". **Declare** three **Character** variables about the picture on this page in the spaces below and then **assign** them **character** values that make sense. Check the example when you are **assigning** a value, this can get tricky. Make sure the variable names describe the object you want to keep track of. Look at the example above if you are unsure of how to **declare** and **assign**. (Don't forget the quotation marks and semicolons at the end of each line!)

Declare:

Assign:
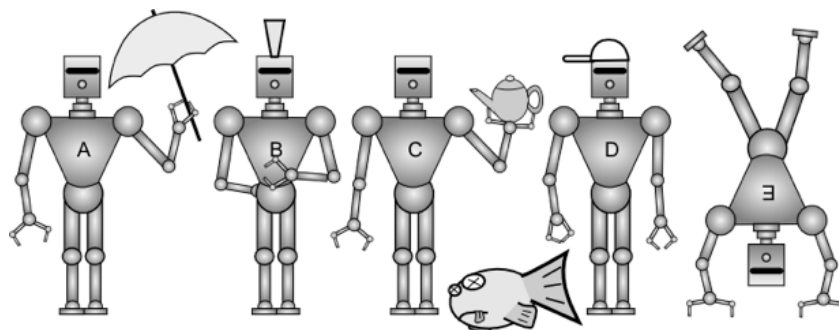
List three of the silliest things you can think of that you might keep track of with a **Character** variable. Example: What color lollipops do robots eat? What's a pirate's favorite letter?

_____

_____

Purpose: Group activity teaching how to declare and assign the variable types Boolean, Integer and Character. Text in italics denotes actual Arduino code.

Materials: None

Vocabulary to be explained prior to activity:
Variable: A way to store a piece of information that may change.
Value: Piece of information assigned to a variable.
Declaration: Creating a variable, when you declare a variable it has no value.
Assignment: Sets or resets the value of a variable.

Types of variables:
Boolean: This variable type has only two values. True or false, which can also be represented as one and zero or HIGH and LOW. Arduino syntax: *boolean*
Integer: This variable type is used to store whole numbers. Because RedBoard uses two bytes to store integers it can only store numbers from −32768 to 32767. Arduino syntax: *int*

Character: This variable is used to store any character you can type on a keyboard (and some you can't). It is basically an integer, but it is used for letters and characters. It is mainly used to print messages or send messages when human interaction is needed. Arduino syntax: *char*

Declaring variables:
Boolean: *boolean variableName;*
variableName can be anything as long as it makes sense and has no spaces in it.

Example: *boolean pamHappy;* This variable could be used to indicate if Pam is happy or not. Remember the semicolon, it's important!

Integer: *int variableName;*
variableName can be anything as long as it makes sense and has no spaces in it.
Example: *int pamAge;* This variable could be used to indicate how old Pam is. Remember the semicolon, it's important!

Character: char *variableName;*
variableName can be anything as long as it makes sense and has no spaces in it.
Example: *char pamShirtColor;* This variable could be used to indicate the color of Pam's shirt. Remember the semicolon, it's important!

Assigning variables:
Assigning variables is really easy! No matter what type of variable you simply type the variable name followed by a single equals sign and then the value you are assigning to your variable followed by a semicolon. Example: *pamShirtColor = 'p';* Values have certain requirements depending on their types. A boolean needs to be true or false (or one or zero), an integer should be a number between -32768 and 32767 and a character should be a single character with single quotation marks around it. **Finally, remember the semicolon, it's important!**

Activity

Activity:

Students should have completed the introduction to variables worksheet that comes with this activity. Examples of variable types, declarations and assignments can be posted somewhere visible in the classroom to help students who are not completely comfortable with the concepts yet.

Students go around in a circle declaring variables that apply to themselves and other students. For example, if they wish to declare a variable about their age they would need to declare an integer variable with a name that makes sense. It is up to the students how specific they want to get, they can declare an integer variable named age, or they could go so far as to declare a variable named pamAge. The difference is that the variable age can apply to anyone, the variable pamAge is specific to a person named Pam. A boolean variable can be used for any quality that is either yes or no. For example, a student might declare pamHappy as a boolean variable to indicate whether Pam is happy or not. Character variables can be used to keep track of anything that does not fit nicely into either integer or boolean. For example, a student may create a variable called pamShirtColor. Declaration of variables should be in the syntax used in Arduino, for examples see previous page.

Once each student has declared a variable go around the circle and have each student assign a value to their variable. Assignment of variables should be in the syntax used in Arduino, for examples see previous page.

Additional activities:

Students can declare their variables on pieces of construction paper. Each variable type should have a distinct color or shape (or some other way to identify the variable type other than the declaration). Students can write their variable declaration and assignment for display and personalize the construction paper so it makes sense with their variable name. Throughout the unit students should be encouraged to reassign the value assigned to their variable if it changes. Obviously you will probably want to have a designated time for variable reassignment to avoid classroom disruption. For example, Pam may declare char pamShirtColor; on a shirt shaped piece of yellow construction paper (yellow to designate it a character variable). Pam can then tape a piece of paper with the letter 'B' (don't forget the single quotation marks) to indicate she is wearing a blue shirt. The next day Pam may then replace the letter 'B' with a 'P' to indicate that today she is wearing a purple shirt. You may want to limit reassignment to once a week if your class has a tendency to be overzealous about activities like this.

If your students are having difficulty with the concept of variable types try this activity: Create three different shaped holes in a board, designate one hole for each of the three variable types. Label each hole with the corresponding variable type and definition. Create or buy a bunch of objects that can only fit through one of the holes and label the objects with values that correspond to the variable type. Give the objects out to students and explain that each object can only be one of the three different type of variables and the students need to match up the objects with the variable types by putting them in the corresponding holes.

Activity

// Vocabulary: If, Parenthesis, Curly Brackets

The *If statement* is one of the most basic building blocks in computer programming. The easiest way to understand a computer language If statement is to look at real life *If statements* first. *If statements* have two different parts, the question and what happens if the answer to the question is yes. Below are a bunch of real life if statements. On the left are the questions or "if" portions of the *If statements*. On the right are the actions that happen when the answer to the questions are true. Unfortunately only the first *If statement* is connected to the correct action, the rest are up to you.

Draw a line between the two that make the most sense together.
The first one is done for you:

If you play around with electronics	Then you can build some cool stuff.
If you run over a porcupine with your bike	Then your feet will smell funny.
If you are an alien	Then you pollute less.
If you do push ups and pull ups	Then you say Arrrrr a lot.
If you put peanut butter in your sock	Then you have feathers and don't like cats.
If you eat too much candy	Then you might catch a fish or fall in.
If you bike everywhere you go	Then you might have six arms and one eye.
If you go fishing in a canoe	Then someone might sing Happy Birthday.
If you are a pirate	Then you get stronger.
If you today is your Birthday	Then you get a flat tire.
If you are a parakeet	Then you get sick.

In computer programming the *If statement* works the same way as real life. There is a question and something that happens if the answer to the question is "yes". The question is written inside of the parenthesis ( ) and whatever happens if the question is true is written inside of the curly brackets { }.

Here are a couple examples of pseudo-code versions of *If statements:*

If (you play around with electronics){then you can build some cool stuff}

If (you remember parenthesis and curly brackets){then *If statements* are easy}

If (you understand *If statements*){then you are on your way to learning programming}

Just remember: If (the answer to this question is yes) {then do this}

Example of an *If statement*:
```
if ( val == HIGH ) {
digitalWrite ( ledPin, LOW );
}
```

All *If statements* start with "if" followed by the question in parenthesis. In this example the question is; does the variable "val" equal HIGH? (HIGH is a boolean value that is the same as true. HIGH means there is electricity present and LOW means there is not.) If "val" does equal HIGH then Arduino does whatever is inside of the two curly brackets { }. In this case it tells ledPin it should not conduct electricity. Here is a pseudo-code of the same If statement:

If (the variable "val" has electricity running through it) {then tell (the pin ledPin, to turn off) }

If parts of this last example don't make sense, don't worry, the important thing is to understand what an If statement is. So... If (the last example didn't make sense) {don't worry}.

Write three of the funniest, or most interesting, *If statements* you can think of in the space below. Don't worry about putting them inside of parenthesis and curly brackets, we'll get to that later.

Example 1: If dinosaurs were still alive then we would have to run a lot more.

———————————————————————

———————————————————————

———————————————————————

Now write your *If statements* the way they would look with the parenthesis. Don't forget the difference between the two different kinds of parenthesis!

Example 1: If (dinosaurs were still alive) {then we would have to run a lot more.}

———————————————————————

———————————————————————

———————————————————————

But what if there are two or more things that could happen if the question is true?

Example 2: If dinosaurs were still alive then we would have to run when we were outside, but if they were our pets we could walk and we would need really big litter boxes.

Is this really just one *If statement?* No, it's actually two, and one of the *If statements* is inside the other. Don't worry! This is ok, in fact it happens all the time. Here is how it looks in pseudo-code:

Example 2:
If (dinosaurs were still alive){
then we would have to run a lot more, but
If (they were our pets) {
we could walk and we would need really big litter boxes} }

It may look complicated but it's just one *If statement* inside of another. There is no limit to how many *Ifs* you can put inside of another *If statement*. Go ahead and write one *If statement* with another *If statement* inside of it in plain English below. Make sure you use the word "if" twice.

———————————————————————

———————————————————————

———————————————————————

Now you're going to take that sentence and turn it into pseudo-code. Pay attention to where the parentheses and curly brackets are and how many there are. Start with writing the first question, put a curly bracket just after the question like this { and then put a curly bracket at the very end of the lines like this }. Now put what happens when the question is true and the second *If statement* inside of your first two curly brackets. If (you're confused) {look at example number two.}

———————————————————————

———————————————————————

———————————————————————
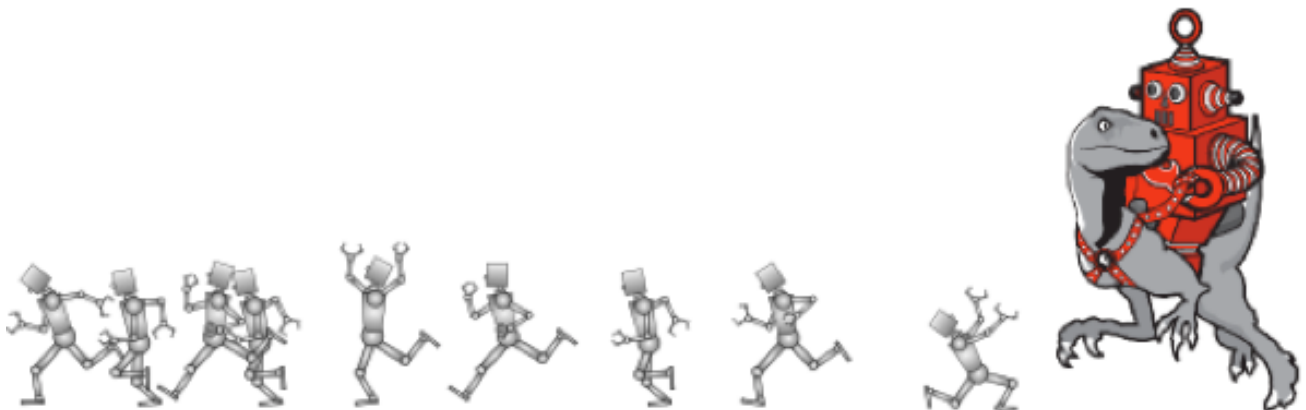
Now that you understand the basics of *If statements* you're going to practice filling in various parts of some *If statements*. These *If statements* are not written in code, but you should be getting comfortable with what goes where as well as the parenthesis and curly brackets. Remember, you will only do what is in the curly brackets if the question is true. Fill in the blanks and if you feel like it make them funny.

(you build a robot)

If (_____)   { _____ }
{ then you can fly. }

(your dog runs away)   If (you build an electronic drum set)
{ then you need to go looking for your dog. }   then you can practice quietly.

If (you are hungry)   If (you are an elephant)
{ _____ }   { _____ }

If (_____)   (you make pancakes)
{ then you burp. }   { _____ }

If (you want to become an astronaut)   If (_____)
{ _____ }   { you should hit the pinata. }

                                (you want pizza)
If (_____)   { _____ }
{ _____ }

Purpose: Group activity teaching the concept of *If statements* and their syntax.

Materials: Cut up sheet of silly conditionals and actions.

Vocabulary to be explained prior to activity:

If statement:
These simple statements exist in real life as well as in computer programming. They are simple statements that indicate if something is true or has occurred, then a resulting action takes place.
If statement pseudo-code: If (conditional) { action }

if:
The word that always starts an *If statement*. It's never capitalized.

Parenthesis ( ) :
Indicates and bookends the conditional portion of an *If statement*.

Conditional:
The question or condition that if true initiates the action of the If statement.

Curly brackets { } :
Indicates and bookends the action portion of an *If statement*.

Action:
Portion of code that occurs when the conditional is true. This can be anything including another *If statement*.

Activity:

Preparation:
Cut up the conditional and action portions of the silly *If statements* included with this activity, or you can write your own and cut those up.

Activity:
First mix and then distribute the slips of paper among your students. Explain the concept of an *If statement* to your students and then have them try to match up all the conditionals with the resulting actions. It is possible to mismatch the conditionals and actions, but this portion of the activity is mainly to have fun and establish the idea of a conditional and a resulting action, so don't worry if the kids mismatch some, just make sure you get some laughter out of this portion of the activity.

Second have seven students stand up to model portions of the *If statement*. The first student is the "If", the second student is the first parenthesis, the third student is the conditional, the fourth student is the closing parenthesis, the fifth student is the first curly bracket, the sixth student is the resulting action and the final student is the closing curly bracket. Students then model one of the silly *If statements* they have matched up. Each student reads or says aloud the portion of the *If statement* they represent. Once the seven students have gone through the *If statement*, the last student sits down, all the standing students move one space over to the right and a new student stands up to join the group as the "If" portion. Students should cycle through this way until either everyone has had a turn to be each part of the *If statement*, or all the silly *If statements* have been used up. Encourage students who are representing the parenthesis and curly brackets to make parenthesis and curly brackets with their arms to demonstrate which are opening parenthesis and curly brackets and which are closing parenthesis and curly brackets.
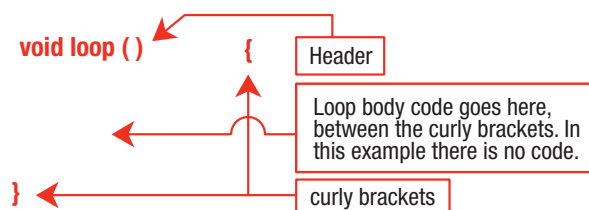
Once the *If statements* and position of the parenthesis and brackets have been established in your classroom you can use the semantics where ever you see fit. For example, If (we line up quickly and quietly) { then we will have more recess time. }

// Vocabulary: repetition, header, loop body, curly brackets

In computer programming **repetition** means repeating a portion of code. This can happen in a bunch of different ways, but the most important thing is to first understand how it happens, not all the different ways it can happen. There are really only two portions to any **repetition**, the **header** and the **loop body**. The **header** usually looks about the same, but the **loop body** can contain any kind of code depending on what you are programming. The **loop body** can even contain another **repetition**!

Repetition with the header, loop body, semicolons and curly brackets labeled:

loop ( ):

void loop ( )   {   Header

Loop body code goes here, between the curly brackets. In this example there is no code.

}   curly brackets

while ( ):

This is not a part of the while loop, it just sets variableN equal to zero before the while loop happens.

variableN = 0;

while (variableN < 10)   {   Header

variableN = variableN + 1;

Loop body code goes here, between the curly brackets. In this example it changes variableN so you're not stuck in the while loop forever.

}   curly brackets

for ( ):

for (int X = 0; X < 100; x = x + 1)   Header   semicolons seperate three sections of the header

{

Loop body code goes here, between the curly brackets. In this example there is no code.

}   curly brackets

Just so we're clear on the important concepts that we will use when we talk about each different kind of repetition, please fill in definitions or explanations of the terms below.

Repetition: _____

Header: _____

Loop body: _____

Curly brackets: _____

Name:
Date:

// Vocabulary: loop ( )

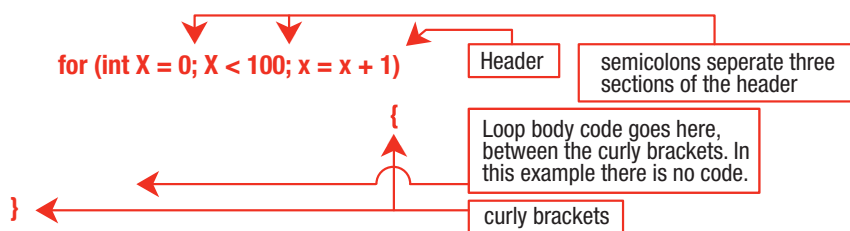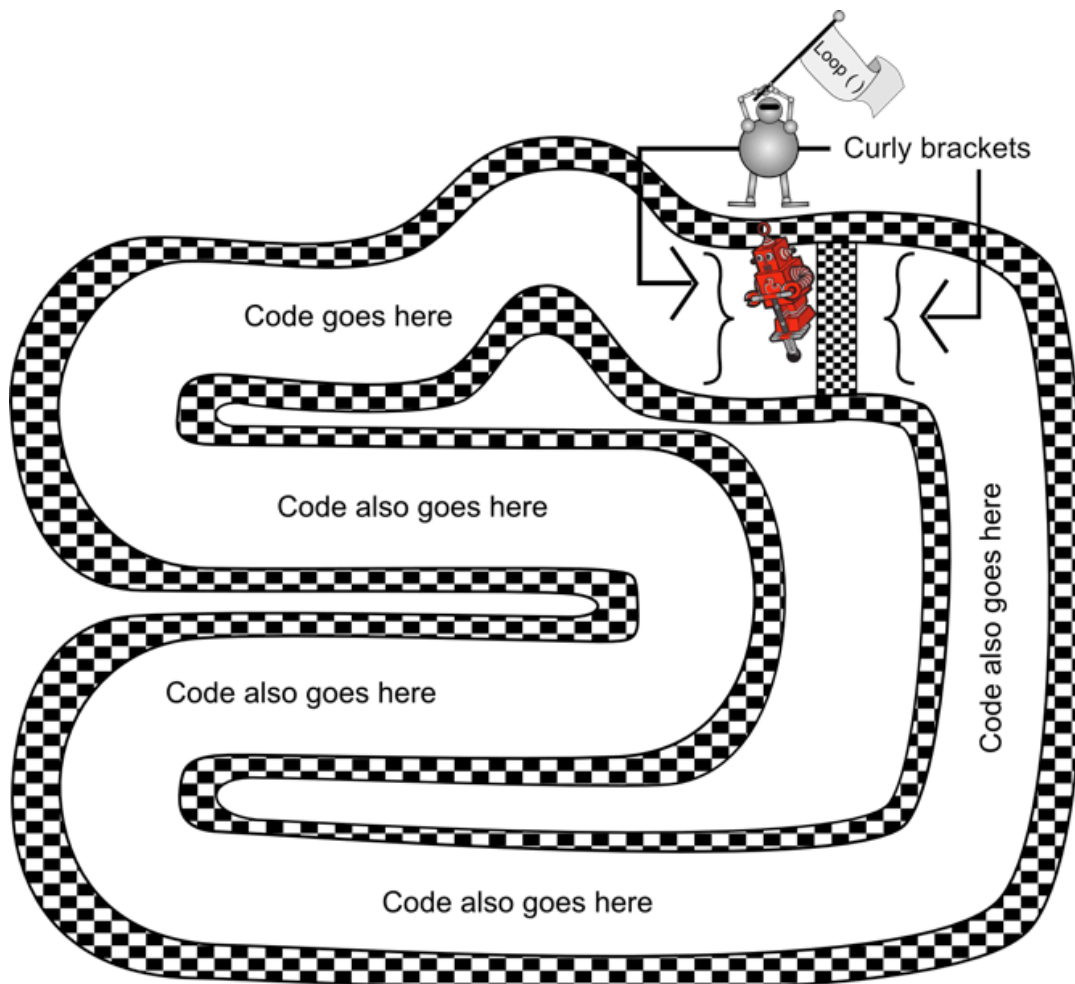The most common form of iteration in Arduino is called the loop( ) function. It exists in all Arduino sketches and its whole purpose is to do all the code written inside of it once, then start over back at the beginning of the loop( ) function and do it all again. Pretty simple, right? The most important things to remember about the loop( ) function are that it is present in every single Arduino sketch, can only be used once per sketch, and it never ends. You will not find a single Arduino sketch that does not have a loop( )function in it and whenever anything happens in your sketch it is because of code inside the loop( ) function.

The loop( ) function looks like this:

```
void loop( ){
 // Lots (or just a little) of loop body code here between curly
brackets.
}
```

Pay attention to the header and the curly brackets which are at the beginning and end of the loop body code. The header is just void loop( ). Think of the loop( ) function as a racetrack. The loop( ) header portion is the flag that starts the computer going around the racetrack and the curly brackets are the beginning and end of the racetrack. Now imagine your computer, Arduino, or robot running around and around the racetrack. It's up to you, the programmer, to put If statements, variables and other code along the way around the racetrack.
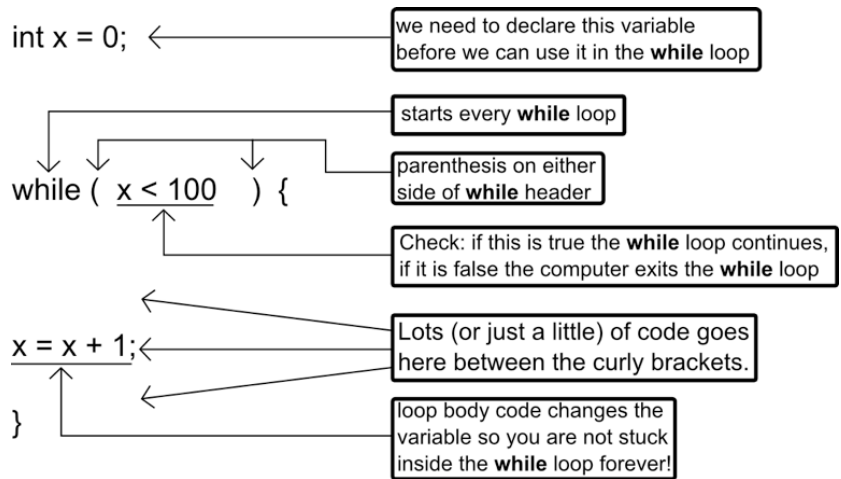

Loop ( )

Curly brackets

Code goes here

Code also goes here

Code also goes here

Code also goes here

Code also goes here

// Vocabulary: while, loop ( )

So, you just learned about loop( ), which is the simplest form of repetition, but there are many other forms of repetition in Arduino. Another very common form of repetition is the while loop. A while loop is used when you want the computer or Arduino to do some code while a statement is true. The while loop is usually found inside of the loop( ) function. The code of a while loop has two parts, the header and the loop body code. The header is the most important part to learn and always has the same structure. The code in the curly brackets below the header can be anything, it just depends on what you want to happen each time the computer goes around your while loop.

The header of a while loop has the word while and a statement inside of parenthesis. The while loop checks to see if the statement inside of the parenthesis is true and will repeat as long as that statement remains true. Pretty simple, right?

While loop example with variable declaration.   Explanation of the while loop example.

```
int x = 0;

while (  x < 100   ) {

x = x + 1;

}
```

we need to declare this variable before we can use it in the **while** loop

starts every **while** loop

parenthesis on either side of **while** header

Check: if this is true the **while** loop continues, if it is false the computer exits the **while** loop

Lots (or just a little) of code goes here between the curly brackets.

loop body code changes the variable so you are not stuck inside the **while** loop forever!

What happens during the while loop above using our robot racetrack as an example:

At the beginning:	Later on, after 100 laps:

Before the while loop:
int x = 0;

Check:
If x is smaller than 100 the racing robot should run around the track.

x = x + 1;

"X equals 0 right now, so I'm going to race around this track!"

while ( )

This time around the track X equals 100.

Check:
If x is smaller than 100 the racing robot should run around the track.

x = x + 1;

"X equals 100 right now, so I'm NOT going to race around this track anymore!"
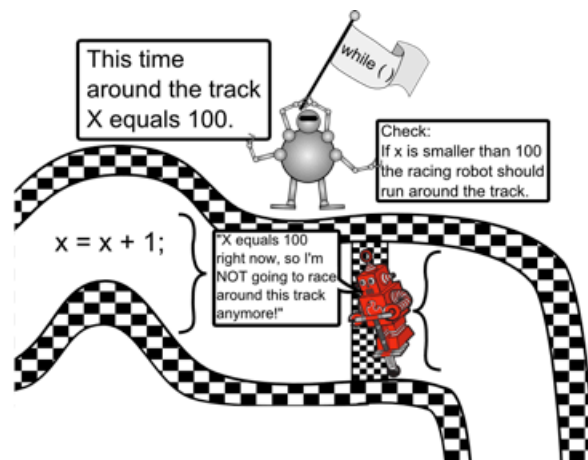
while ( )

// Vocabulary: for, loop ( )

So, you just learned about while, which is a simple form of repetition, but there are many other loop functions. Another very common form of repetition is the for loop. A for loop is used when you want the computer or Arduino to change a variable each time through the loop and do code which often uses that variable. For loops are usually found inside of the loop( ) function. The code of a for loop has two parts, the header and the code inside the loop. The header is the most important part to learn and always looks about the same. The loop body code in the curly brackets below the header can be anything, it just depends on what you want to happen each time the computer goes around your for loop.
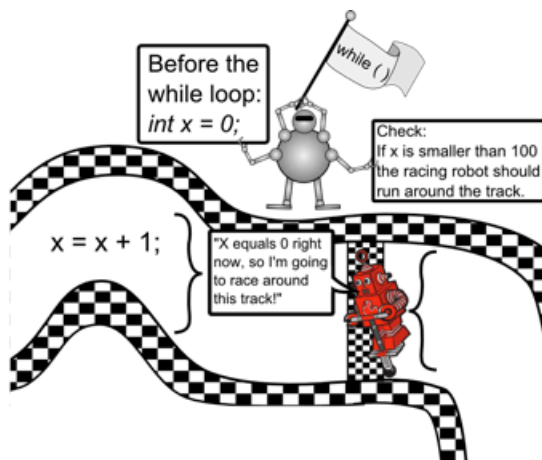
The header of a for loop has the word for and in parenthesis three parts called start, check and change. Each of these parts have semicolons between them so you can tell them apart. These three parts (circled in gray below) are the most important parts to understand, they are the three simple parts you need to make a for loop work.
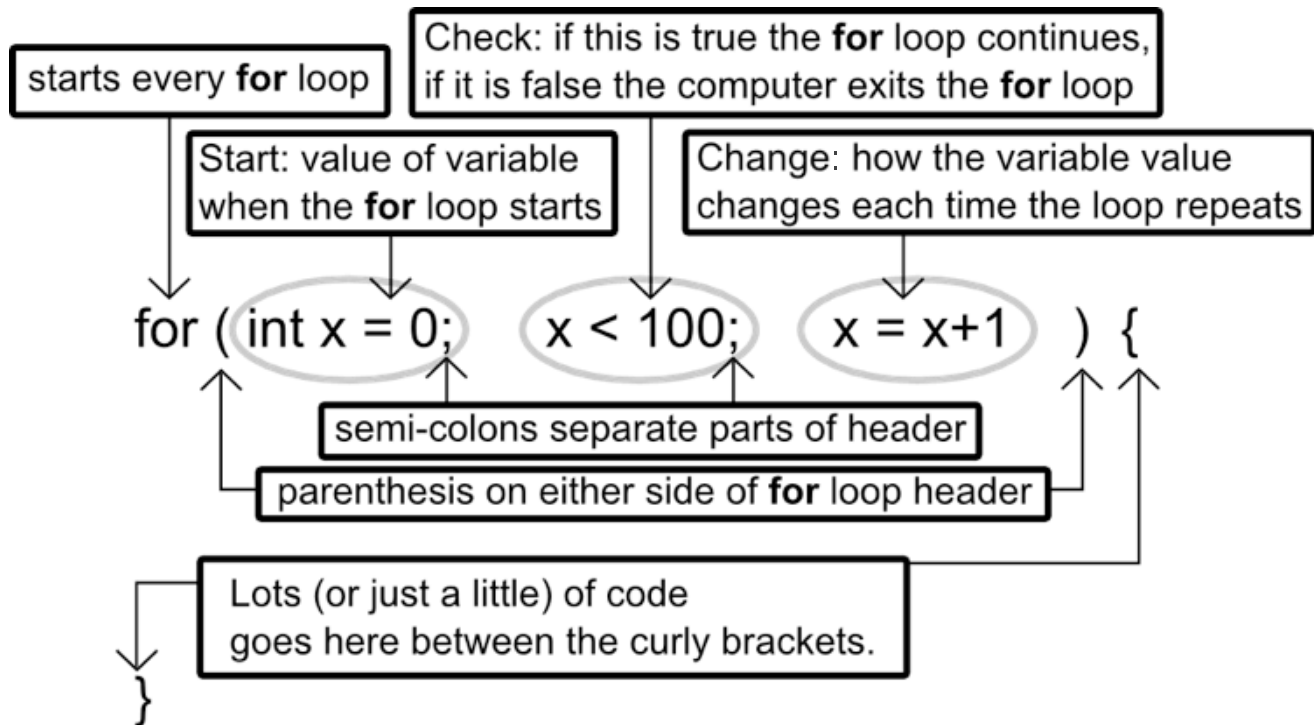
starts every **for** loop

Check: if this is true the **for** loop continues, if it is false the computer exits the **for** loop

Start: value of variable when the **for** loop starts

Change: how the variable value changes each time the loop repeats

for (int x = 0;   x < 100;   x = x+1) {

semi-colons separate parts of header

parenthesis on either side of **for** loop header

Lots (or just a little) of code goes here between the curly brackets.

}

Start:
The first circled part is start, this happens before anything else, it's sort of like putting on running shoes before starting to run around the track. It is a simple declaration and assignment of a variable, in this case the variable is an integer named x.

Check:
The second circled part is check. Every time the computer gets to the end of the for loop the computer will check to see if this part is true. The first time the for loop above checks,

x is equal to zero, so the for loop continues, does change and then the code inside the curly brackets. It's kind of like checking how many laps a racer has completed to see if the racer has finished the race.

Change:
The third circled part is change, after the variable is checked it changes so that it is closer to making the check statement false so the for loop stops. For a racer this part of the for loop is like adding to (or updating) the number of laps or miles the racer has completed so far in the race.

// Vocabulary: for, loop ( )

Here is an example of what happens when the for loop on the previous page begins using our robot racetrack as an example:
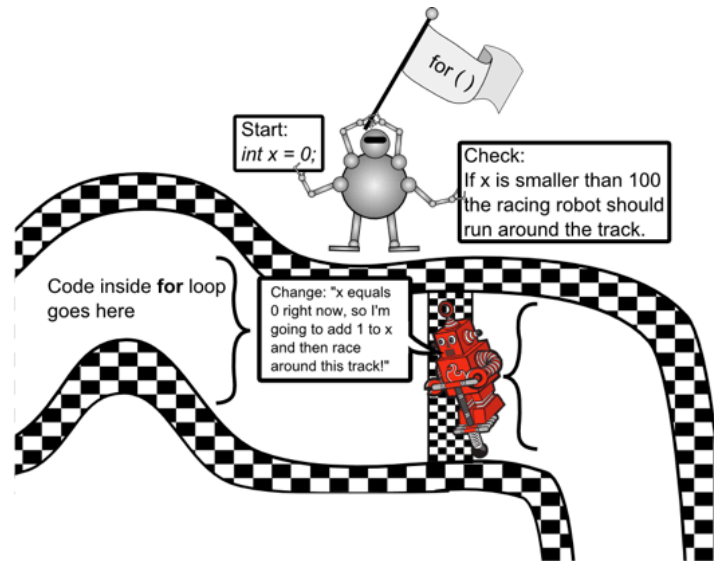
The next time the racing robot makes its way around the track to the starting line it has to check again. It doesn't have to start again, but it does need to check to see if the race is over. The first time around the track, x will equal one and the check that x is smaller than 100 is still true. The robot changes the variable by adding one to x again (x now equals two) and then the robot runs around the track executing the loop body code between the curly brackets. The robot will continue to run around the racetrack until x equals 100 at which point the computer exits the for loop.

// Vocabulary: Nested, Repetition

Now that you know about repetition we can talk about ways to put code inside of other code, which is called nesting, and in fact most loops are nested loops since they are inside of the original loop( ) function. It's easy, all you do is put your loop inside the curly brackets of another loop. Nested if statements work exactly the same way as nested loops.

Example of nested loop:
```
void loop ( ) {
int x = 0;
while (x < 10) {
x  =  x + 1;
}
}
```

Example of nested if statement:
```
if (int x < 10) {
if ( x == 5 ) {
//code here happens if x < 10 & x = 5
}
//code here happens if x < 10
}
```

Imagine your loop( ) racetrack with another for loop racetrack attached to it. This way each time the robot runs (or drives or whatever) around the racetrack it must stop when it reaches a new while loop, run around that race track until that while loop is over and then it can continue running around the larger loop( ) racetrack.

The robot has to run through the whole while loop before it can continue running around the larger loop racetrack. But let's break it down a little more; x starts as zero, if x is less than ten the robot continues running around the while loop until x is not less than ten. If the robot is adding one to x each time it checks the while loop then the robot must run around the while loop a total of ten times. The robot then exits the while loop and continues around the loop racetrack. Next time around the racetrack the variable x will be set to zero again just before the while loop. So, you don't have to worry about the while loop not working due to x being more than or equal to ten.

You can nest as many loops inside of other loops as you like, just make sure you don't get stuck inside of a loop. One way to do this is to misplace curly brackets, so make sure they're in the right spot. If this happens your computer or Arduino will just freeze and you won't really be able to tell why.

Nesting works for code other than just loops! You can nest if statements, loops and many other code structures. All you need to remember is that nesting is a complicated way to say "put code inside of other code" and that the computer eventually needs to get out of the nested statements and back to the loop( ) function so everything can start over again.

Activity

Name:
Date:

Purpose: Group activity teaching the concept of repetition as used in Arduino programming. Text like this denotes actual Arduino code.

Materials: Cones, large boards to display loop headers and pseudo-code, equipment for physical activities, and a field or gym.

Vocabulary to be explained prior to activity:

loop or repetition:
A section of code that repeats.

repetition header:
The line at the very beginning of a loop that tells the computer how the code inside the loop will repeat. This section is different for each different type of loop.

Conditional or question:
This is the statement that is checked to see if the loop is completed. Conditionals are present in loop headers and often look like this: $x < 10$. This indicates that the loop will continue until $x < 10$ is false.

Increment:
The section of code (may be in the header or may be in the loop body code) used to change the variable that is checked in the conditional. Using the example above, $x = x + 1$, one is added to x getting it a little closer to being larger than or equal to 10.

Nested repetition:
A loop inside of a loop. This concept is key for any type of even slightly advanced programming.

Types of loops:

• loop:
This loop is the most basic of all loops (that's why it's called loop) and is present in all Arduino sketches. loop( ) repeats as long as there is power to the Arduino. Inside this form of repetition is where you will find all other forms of repetition.

Header:
loop( )

Increment:
N/A

Conditional:
Power must be on.

• while:
This loop repeats as long as the conditional listed inside the parenthesis is true. This loop's conditional is incremented in the body code or through an Arduino input.

Header:
while( )

Increment:
In body code or Arduino input

Conditional:
Inside header parenthesis.

• for:
This loop repeats as long as the conditional listed inside the parenthesis is true. The for loop header declares a variable, checks a conditional and increments the conditional variable all inside the parenthesis...

Header:
for (int x = 0; x < 10; x = x + 1)

Increment:
Inside header parenthesis, in this example $x = x + 1$.

Conditional:
Inside header parenthesis, in this example $x < 10$.

Preparation:

This activity is a physical activity and you will need to set up an obstacle loop or course that reflects the repetitions you have decided to include in this activity. You may wish to work with a gym teacher in order to set this activity up.

The examples in this activity require three different stations. These include a "loop" station at the beginning of the obstacle course with a teacher or student helper, a "while" station with jump ropes and an area for spinning in circles, and a "for" station with an area for doing jumping jacks and shooting basketballs.

Each station will need a poster displaying the pseudo-code that students need to follow in order to complete the obstacle loop. The poster materials are included with the rest of the

activity materials in the folder programming in the file called LoopActivityMaterials.

You will also need a field for kids to run around or cones to set up an area for kids to run around inside a gym.

Also- this is a really big activity. It takes a lot of prep and will probably be chaos the first time you try it, but it is easily customizable to age or skill level and should be lots of fun if you stick with it.

Activity:

Students should have completed the introduction to repetition worksheets that come with this activity. Students should also be familiar with variables and if statements.

What your loop activity might look like:
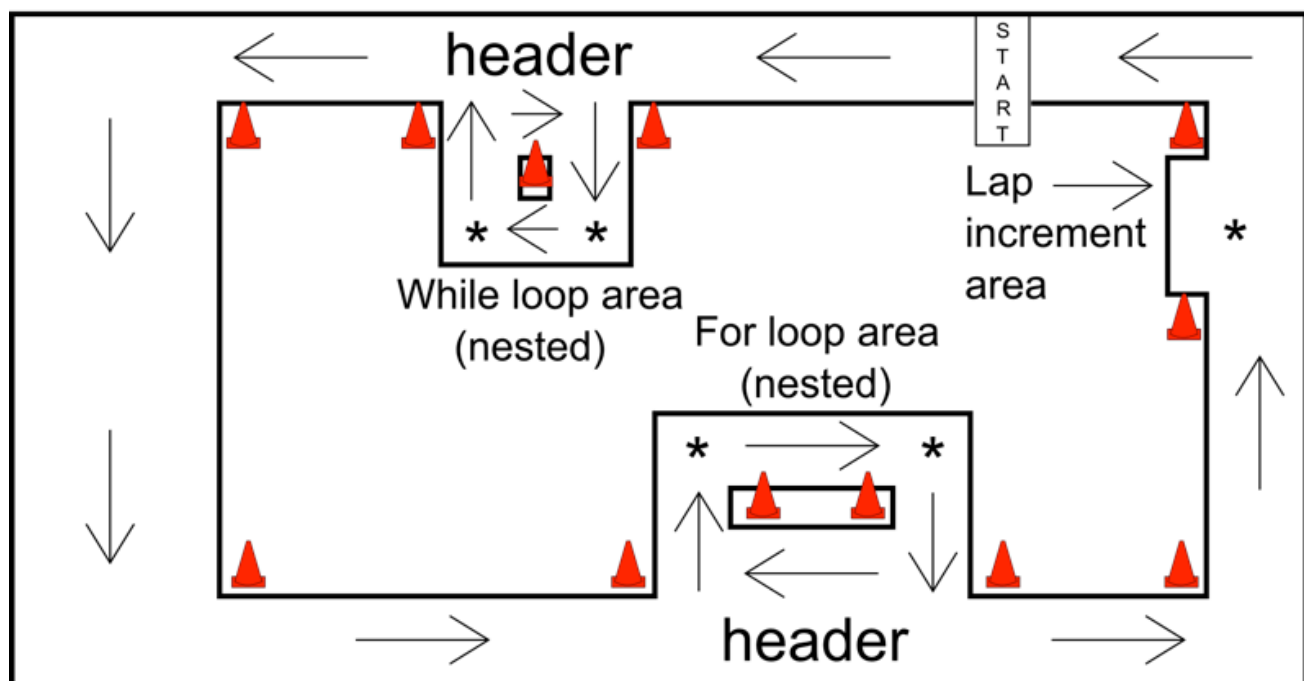

Cones kids have to run around

Direction kids should be running

* Areas with physical activities that kids complete

header Instructions in pseudo-code form for above activities

Name:
Date:

The idea is that the complete obstacle course from the start position back to the start position represents the loop ( ) function. Inside this loop ( ) function are two nested loops, a while ( ) loop and a for ( ) loop.

At the beginning of the obstacle course each student needs to declare an integer variable called lapNumber or something similar. This variable will be used in each of the loop activity areas and the lap increment area. The lapNumber variable can also be used to end the obstacle course if you do not wish to have students run the obstacle course until the end of the period. When students start the obstacle course lapNumber should equal zero since they have not run any laps yet.

Nested loop activity areas: These areas are nested loops where students will perform a certain number of tasks depending on what your loop headers say. You can have as many or as few activity areas as you like. You may also tailor the number of physical tasks inside these nested loops to make your obstacle course more fun for your students.

These activity areas should look like little loops that the students can run around completing tasks. The headers should follow the format of the loop type it represents. For examples see the end of the activity. Once inside the nested loop activity area students must complete the physical activities according to the pseudo-code posted inside the nested loop activity area. Once students are done with the first repetition of the physical activities inside the nested loop activity areas they should look at the header again and decide if they have completed the nested loop represented by the header. With younger students you may want to have someone helping them with this step. (This can be fun, the observer can yell out error in a friendly voice if students exit the loop too quickly) Once students have completed the nested loop activity area they continue around the obstacle course to the next activity.

Header examples:
If a student's lapNumber is equal to three and the pseudo-code header reads:

```
while (lapNumber > basketsMade) {
do (lapNumber * 2) jump ropes at jump rope station
shoot lapNumber basketball baskets
}
```

This time around the obstacle course, the student would run through the nested loop activity area once, jumping rope six times and shooting three baskets along the way.

If a student's lapNumber is equal to three and the pseudo-code header reads:

```
for (int x = 0; x < lapNumber * 2; x = x + 1) {
do (x * 2) jump ropes at jump rope station
shoot lapNumber basketball baskets
}
```

The student would run through this nested loop activity six times jumping rope a different amount and shooting three baskets each time for a total of thirty six jump ropes and eighteen baskets.

There is a lot of room for personalization in this activity; it's an opportunity to really solidify the loop concept as well as getting your kids some exercise.

Name:
Date:

Lap increment area:
The lap increment area is where students will add one to their lapNumber variable to keep track of how many times they have run the obstacle course. You can also set up the headers and nested loop activities to use the lapNumber variable. The lap increment area is where you might insert an if statement to end the obstacle course after students complete a certain number of laps.

Additional thoughts:
Definitely call the obstacle course a loop( ) instead of an obstacle course in order to really get kids comfortable with the concepts. You may also wish to include your students in the planning of the obstacle course. Planning the obstacle course is another opportunity to talk about the loop concept and it gives them a stake in the learning exercise. Lastly, not that this needs pointing out, but this is a great activity just prior to computer lab time. Instead of having kids bouncing off the monitors they will be calmer and ready to sit still applying the concepts they just solidified through physical activity. This is great for kinesthetic learners in particular.