LOOPS

Introduction to Loops

In the programming world, we hate repeating ourselves. There are two reasons for this:

- Writing the same code over and over is time-consuming.
- Having less code means having less to debug.

But we often need to do the same task more than once. Fortunately, computers are really good (and fast) at doing repetitive tasks. And in Java, we can use loops.

A *loop* is a programming tool that allows developers to repeat the same block of code until some condition is met.

First, a starting condition is evaluated. If the starting condition is true, then the loop body is executed. When the last line of the loop body is executed, the condition is *re-evaluated*. This process continues until the condition is false (if the condition never becomes false, we can actually end up with an **infinite loop**!). If the starting condition is false, the loop never gets executed.

We employ loops to easily scale programs - saving time and minimizing mistakes.

We'll go over three types of loops that we'll see everywhere:

- while loops
- for loops
- for-each loops

While We're Here

A while loop looks a bit like an if statement:

```
while (silliness > 10) {
  // code to run

}
```

Like an `if` statement, the code inside a `while` loop will only run if the condition is `true`. However, a `while` loop will continue running the code over and over until the condition evaluates to `false`. So the code block will repeat until `silliness` is less than or equal to 10.

```
// set attempts to 0
int attempts = 0;

// enter loop if condition is true
while (passcode != 0524 && attempts < 4) {

  System.out.println("Try again.");
  passcode = getNewPasscode();
  attempts += 1;

  // is condition still true?
  // if so, repeat code block
}
// exit when condition is not true
```

`while` loops are extremely useful when you want to run some code until a specific change happens. However, if you aren’t certain that change will occur, beware the infinite loop!

Infinite loops occur when the condition will never evaluate to `false`. This can cause your entire program to crash.

```
int hedgehogs = 5;

// This will cause an infinite loop:
while (hedgehogs < 6) {
  System.out.println("Not enough hedgehogs!");

}
```

In the example above, hedgehogs remains equal to 5, which is less than 6. So we would get an infinite loop.

Instructions

1. Take a look at **LuckyFive.java**. We've set up a random number generator that allows you to simulate the roll of a die.

Create a while loop that will continue to loop while dieRoll is NOT 5.

Do NOT run your code yet — you will get an infinite loop here because the value of dieRoll is never changed.

Inside the loop, reset dieRoll with a new random value between 1 and 6.

Now you can run the code.

2. Inside the while loop, above the line where you reset dieRoll, print out dieRoll to the terminal.

```
// Importing the Random library
import java.util.Random;

class LuckyFive {

  public static void main(String[] args) {

    // Creating a random number generator
    Random randomGenerator = new Random();

    // Generate a number between 1 and 6
    int dieRoll = randomGenerator.nextInt(6) + 1;

    // Repeat while roll isn't 5
  while(dieRoll != 5){

    System.out.println(dieRoll);
     dieRoll = randomGenerator.nextInt(6) + 1;
  }
  }}
```

```
2
4
2
3
```

Incrementing While Loops

When looping through code, it's common to use a counter variable. A *counter* (also known as an *iterator*) is a variable used in the conditional logic of the loop and (usually) incremented in value during each iteration through the code. For example:

```
// counter is initialized
int wishes = 0;

// conditional logic uses counter
while (wishes < 3) {

  System.out.println("Wish granted.");
  // counter is incremented
  wishes++;
}
```

In the above example, the counter `wishes` gets initialized before the loop with a value of `0`, then the program will keep printing `"Wish granted."` and adding `1` to `wishes` as long as `wishes` has a value of less than `3`. Once `wishes` reaches a value of `3` or more, the program will exit the loop.

So the output would look like:

```
Wish granted.
Wish granted.
Wish granted.
```

We can also decrement counters like this:

```
int pushupsToDo = 10;

while (pushupsToDo > 0) {

  doPushup();
  pushupsToDo--;

}
```

In the code above, the counter, `pushupsToDo`, starts at 10, and increments down one at a time. When it hits 0, the loop exits.

Instructions

1. In **Coffee.java**, initialize an `int` variable called `cupsOfCoffee` with a value of `1`.

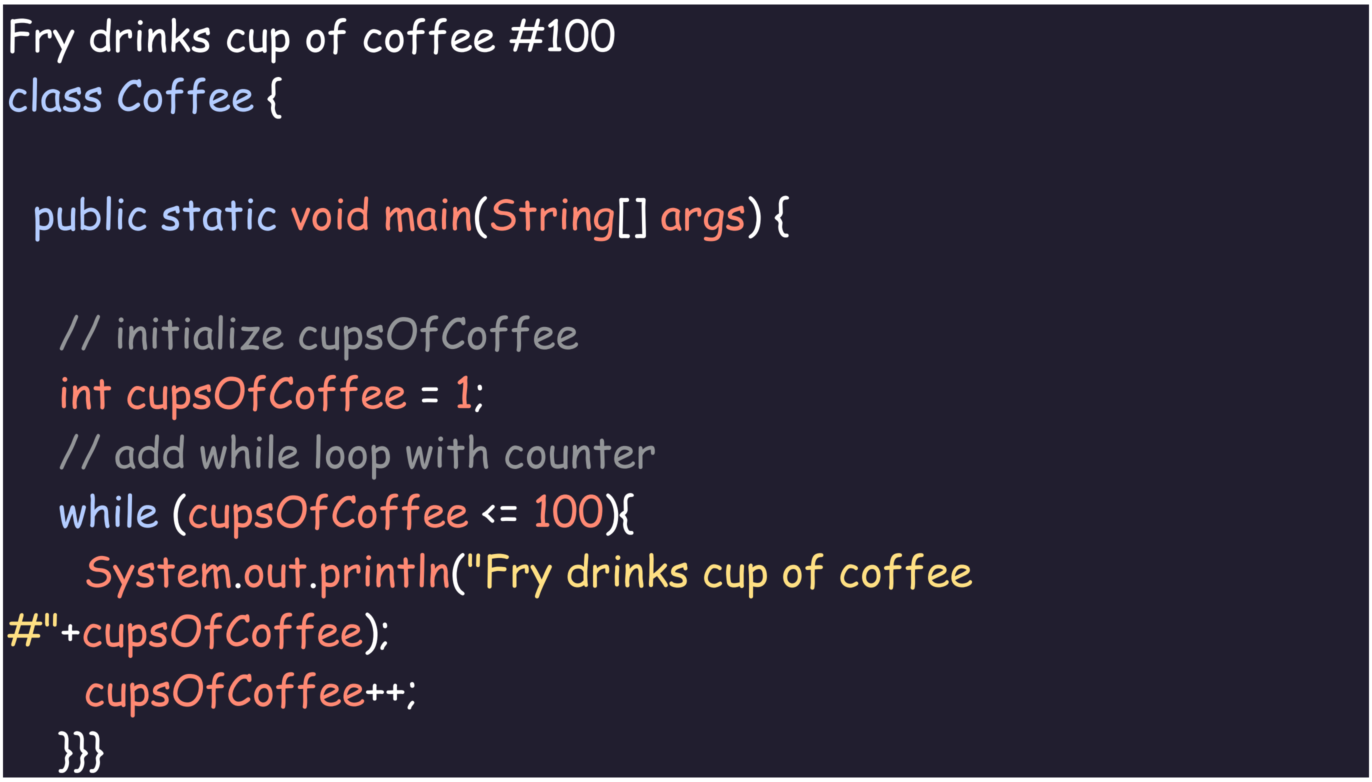
2. Create a `while` loop that runs as long as `cupsOfCoffee` is less than or equal to `100`.

Important: Inside the `while` loop, increment `cupsOfCoffee` by `1` to prevent an infinite loop.

3. Inside the `while` loop above where you incremented `cupsOfCoffee` print the following:

```
Fry drinks cup of coffee #1
```

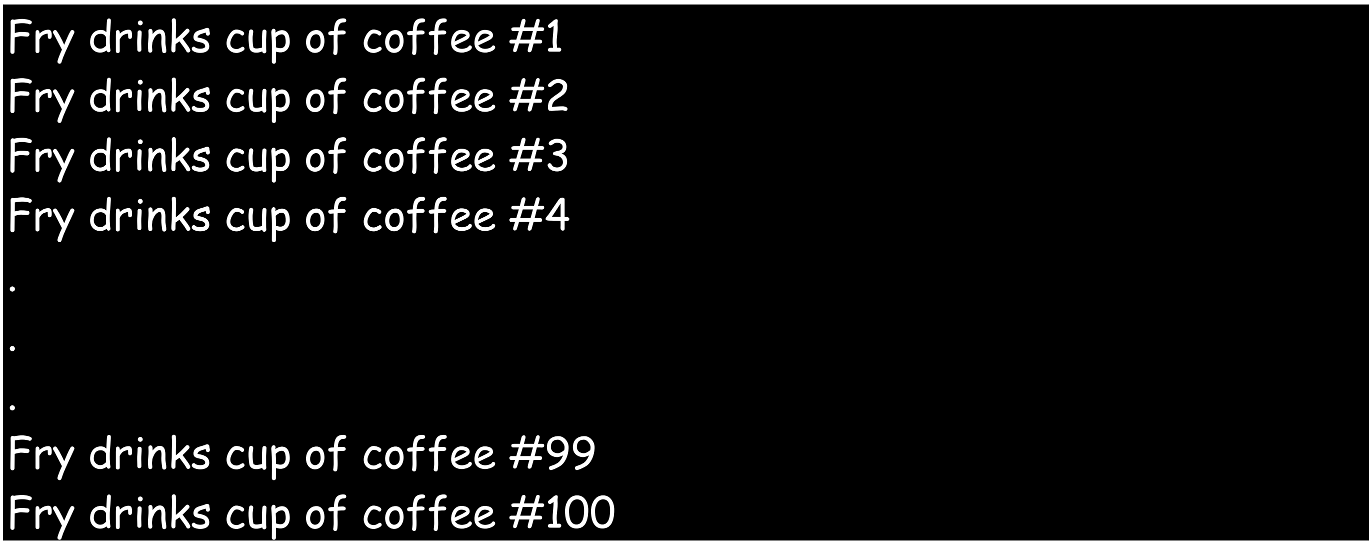
The `1` in this statement should correspond with the current value of `cupsOfCoffee`. When `cupsOfCoffee` is `100`, this should be printed:



```
Fry drinks cup of coffee #100
class Coffee {

  public static void main(String[] args) {

    // initialize cupsOfCoffee
    int cupsOfCoffee = 1;
    // add while loop with counter
    while (cupsOfCoffee <= 100){
      System.out.println("Fry drinks cup of coffee
#"+cupsOfCoffee);
      cupsOfCoffee++;
    }}}
```



```
Fry drinks cup of coffee #1
Fry drinks cup of coffee #2
Fry drinks cup of coffee #3
Fry drinks cup of coffee #4
.
.
.
Fry drinks cup of coffee #99
Fry drinks cup of coffee #100
```

For Loops

Incrementing with loops is actually so common in programming that Java (like many other programming languages) includes syntax specifically to address this pattern: `for` loops.
A `for` *loop* header is made up of the following three parts, each separated by a semicolon:

1. The initialization of the loop control variable.
2. A `boolean` expression.
3. An increment or decrement statement.

The opening line might look like this:

```
for (int i = 0; i < 5; i++) {

  // code that will run

}
```

In a `for` loop, an initialization statement is run once in order to initialize the loop control variable. This variable is modified in every iteration, can be referenced in the loop body, and used to test the boolean condition. In the example above, `i` is the loop control variable.

Let's breakdown the above example:

1. `i = 0`: `i` is initialized to `0`
2. `i < 5`: the loop is given a `boolean` condition that relies on the value of `i`. The loop will continue to execute until `i < 5` is `false`.
3. `i++`: `i` will increment at the end of each loop and before the condition is re-evaluated.

So the code will run through the loop a total of five times.

We'll also hear the term "iteration" in reference to loops. When we *iterate*, it just means that we are repeating the same block of code.

Instructions

Review the syntax of for loops and click Next when you're ready to build some yourself!

Using For Loops

Even though we can write while loops that accomplish the same task, for loops are useful because they help us remember to increment our counter — something that is easy to forget when we increment with a while loop.

Leaving out that line of code would cause an infinite loop — yikes!

Fortunately, equipped with our new understanding of for loops, we can help prevent infinite loops in our own code.

It's important to be aware that, if we don't create the correct for loop header, we can cause the iteration to loop one too many or one too few times; this occurrence is known as an "off by one" error.

For example, imagine we wanted to find the sum of the first ten numbers and wrote the following code:

```
int sum = 0;
for (int i = 0; i < 10; i++) {
  sum += i
}
```

This code would produce an incorrect value of 45. We skipped adding 10 to sum because our loop control variable started with a value of 0 and stopped the iteration after it had a value of 9. We were off by one! We could fix this by changing the condition of our loop to be i <= 10; or i < 11;.

These errors can be tricky because, while they do not always produce an error in the terminal, they can cause some miscalculations in our code. These are called logical errors — the code runs fine, but it didn't do what you expected it to do.

Instructions

1. We've provided a while loop that loops from 1 to 100 outputting a string on each iteration. Refactor (rewrite) this code as a for loop. Remember, the syntax of a for loop looks like:

```
for (int i = 0; i < 5; i++) {

  // code that will run

}
class Coffee {

  public static void main(String[] args) {

    int cupsOfCoffee = 1;

      for(int i = 1; i <=100; i++){
cupsOfCoffee = i;
System.out.println("Fry drinks cup of coffee #" +cupsOfCoffee );
      }}}
```

```
Fry drinks cup of coffee #1
Fry drinks cup of coffee #2
Fry drinks cup of coffee #3
Fry drinks cup of coffee #4
.
.
.
Fry drinks cup of coffee #99
Fry drinks cup of coffee #100
```

Iterating Over Arrays and ArrayLists

One common pattern we'll encounter as a programmer is *traversing*, or looping, through a list of data and doing something with each item. In Java, that list would be an array or `ArrayList` and the loop could be a `for` loop. But wait, how does this work?

In order to traverse an array or `ArrayList` using a loop, we must find a way to access each element via its index. We may recall that `for` loops are created with a counter variable. We can use that counter to track the index of the current element as we iterate over the list of data.

Because the first index in an array or `ArrayList` is `0`, the counter would begin with a value of `0` and increment until the end of the list. So we can increment through the array or `ArrayList` using its indices.

For example, if we wanted to add `1` to every `int` item in an array `secretCode`, we could do this:

```
for (int i = 0; i < secretCode.length; i++) {
  // Increase value of element value by 1
  secretCode[i] += 1;
}
```

Notice that our condition in this example is `i < secretCode.length`. Because array indices start at 0, the length of `secretCode` is 1 larger than its final index. A loop should stop its traversal before its counter variable is equal to the length of the list.

To give a concrete example, if the length of an array is `5`, the last index we want to access is `4`. If we were to try to access index `5`, we would get an `ArrayIndexOutOfBoundsException` error! This is a very common mistake when first starting to traverse arrays.

Traversing an ArrayList looks very similar:

```
for (int i = 0; i < secretCode.size(); i++) {
  // Increase value of element value by 1
  int num = secretCode.get(i);
  secretCode.set(i, num + 1);
}
```

We can also use while loops to traverse through arrays and ArrayLists. If we use a while loop, we need to create our own counter variable to access individual elements. We'll also set our condition to continue looping until our counter variable equals the list length.

For example, let's use a while loop to traverse through an array:

```
int i = 0; // initialize counter

while (i < secretCode.length) {
  secretCode[i] += 1;
  i++; // increment the while loop
}
```

Traversing through an ArrayList with a while loop would look like this:

```
int i = 0; // initialize counter

while (i < secretCode.size()) {
  int num = secretCode.get(i);
  secretCode.set(i, num + 1);
  i++; // increment the while loop
}
```

Instructions

1.Let's use a for loop to iterate over expenses and sum up the total of all items.

Start with the skeleton of a for loop:

- Initialize a counter i with a value of 0.
- The loop should run while i is less than the size() of expenses.
- Increment i.

You can leave the body empty for now.

2.Inside the for loop, add the item's value to total.

```
import java.util.ArrayList;

class CalculateTotal {

  public static void main(String[] args) {

    ArrayList<Double> expenses = new ArrayList<Double>();
    expenses.add(74.46);
    expenses.add(63.99);
    expenses.add(10.57);
    expenses.add(81.37);

    double total = 0;

    // Iterate over expenses
    for(int i =0; i < expenses.size(); i++){
     total +=  expenses.get(i);

    }

    System.out.println(total);

  }
```

```
230.39
```

break and continue

If we ever want to exit a loop before it finishes all its iterations or want to skip one of the iterations, we can use the break and continue keywords.

The break keyword is used to exit, or break, a loop. Once break is executed, the loop will stop iterating. For example:

```
for (int i = 0; i < 10; i++) {
  System.out.println(i);
  if (i == 4) {
    break;
  }
}
```

Even though the loop was set to iterate until the condition i < 10 is false, the above code will output the following because we used break:

```
0
1
2
3
4
```

The continue keyword can be placed inside of a loop if we want to skip an iteration. If continue is executed, the current loop iteration will immediately end, and the next iteration will begin. We can use the continue keyword to skip any even valued iteration:

```
int[] numbers = {1, 2, 3, 4, 5};

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

This program would output the following:

```
1
3
5
```

In this case, if a number is even, we hit a `continue` statement, which skips the rest of that iteration, so the print statement is skipped. As a result, we only see odd numbers print.

Keep Reading: AP Computer Science A Students

Loops can exist all throughout our code - including inside a method. If the `return` keyword was executed inside a loop contained in a method, then the loop iteration would be stopped and the method/constructor would be exited.

For example, we have a method called `checkForJacket()` that takes in an array of `String`s. If any of the elements are equivalent to the `String` value `"jacket"`, the method will return `true`:

```
public static boolean checkForJacket(String[] lst) {
  for (int i = 0; i < lst.length; i++) {
    System.out.println(lst[i]);
    if (lst[i] == "jacket") {
      return true;
    }
  }
  return false;
}

public static void main(String[] args) {
  String[] suitcase = {"shirt", "jacket", "pants", "socks"};
  System.out.println(checkForJacket(suitcase));
}
```

As soon as an element equals "jacket", return true; is executed. This causes the loop to stop and the compiler to exit checkForJacket(). Running this code would output the following:

```
shirt
jacket
true
```

Instructions

1. Take a look at the for loop in the code editor. It starts its iteration at 0 and continues to iterate until i < 100 is false.

Inside the loop, create a condition that checks if i is **not** divisible by 5. If the condition is true, skip the iteration. Outside the condition statement, print i. The final solution **should not** contain an else statement.

The only numbers that should be printed are those that are divisible by 5!

```
class Numbers {
  public static void main(String[] args) {
    for (int i = 0; i <= 100; i++) {
      // Add your code below
    if(i % 5 !=0){


      continue;
    }
     System.out.println(i);

 }}}
```

0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
90
95
100

For-Each Loops

Sometimes we need access to the elements' indices or we only want to iterate through a portion of a list. If that's the case, a regular `for` loop or `while` loop is a great choice.

For example, we can use a `for` loop to print out each element in an array called `inventoryItems`:

```
for (int inventoryItem = 0; inventoryItem < inventoryItems.length;
inventoryItem++) {
  // Print element at current index
  System.out.println(inventoryItems[inventoryItem]);
}
```

But sometimes we couldn't care less about the indices; we only care about the element itself.

At times like this, for-each loops come in handy.

For-each loops, which are also referred to as *enhanced loops*, allow us to directly loop through each item in a list of items (like an array or ArrayList) and perform some action with each item.

If we want to use a for-each loop to rewrite our program above, the syntax looks like this:

```
for (String inventoryItem : inventoryItems) {
  // Print element value
  System.out.println(inventoryItem);

}
```

Our enhanced loop contains two items: an enhanced `for` loop variable (`inventoryItem`) and a list to traverse through (`inventoryItems`).

We can read the `:` as "in" like this: for each `inventoryItem` (which should be a `String`) in `inventoryItems`, print `inventoryItem`.

If we try to assign a new value to the enhanced for loop variable, the value stored in the array or ArrayList will not change. This is because, for every iteration in the enhanced loop, the loop variable is assigned a copy of the list element.

Note: We can name the enhanced for loop variable whatever we want; using the singular of a plural is just a convention. We may also encounter conventions like String word : sentence.

Instructions

1.Let's use a for-each loop to find the priciest item in expenses.

Build a for-each loop that iterates through
each expense in expenses. For now, leave the body of the loop empty.

2.Inside the for-each loop, check if expense is greater than mostExpensive.
If it is, set mostExpensive equal to expense.

```
import java.util.ArrayList;

class MostExpensive {

  public static void main(String[] args) {

    ArrayList<Double> expenses = new ArrayList<Double>();
    expenses.add(74.46);
    expenses.add(63.99);
    expenses.add(10.57);
    expenses.add(81.37);

    double mostExpensive = 0;

    // Iterate over expenses
    for(double expense : expenses){
      if (expense > mostExpensive){
        mostExpensive = expense;
      }
    }
  System.out.println(mostExpensive);
  }
}
```

```
81.37
```

Removing Elements During Traversal

If we want to remove elements from an ArrayList while traversing through one, we can easily run into an error if we aren't careful. When an element is removed from an ArrayList, all the items that appear after the removed element will have their index value shift by negative one — it's like all elements shifted to the left! We'll have to be very careful with how we use our counter variable to avoid skipping elements.

Removing An Element Using while

When using a while loop and removing elements from an ArrayList, we should **not** increment the while loop's counter whenever we remove an element. We don't need to increase the counter because all of the other elements have now shifted to the left.

For example, if we removed the element at index 3, then the element that was at index 4 will be moved to index 3. If we increase our counter to 4, we'll skip that element!

Take a look at this block of code that will remove all odd numbers from an ArrayList. Think about what the value of i is, when we're increasing the value of i, and when i < lst.size() becomes False.

```
int i = 0; // initialize counter

while (i < lst.size()) {
  // if value is odd, remove value
  if (lst.get(i) % 2 != 0){
    lst.remove(i);
  } else {
    // if value is even, increment counter
    i++;
  }
}
```

Removing An Element Using `for`

We can use a similar strategy when removing elements using a `for` loop. When using a `while` loop, we decided to not increase our loop control variable whenever we removed an element. This ensured that we would not skip an element when all of the other elements shifted to the left.

When using a `for` loop, we, unfortunately, *must* increase our loop control variable — the loop control variable will always change when we reach the end of the loop (and it will usually change by `1` because we often use something like `i++`.) Since we can’t avoid increasing our loop control variable, we can take matters into our own hands and decrease the loop control variable whenever we remove an item.

For example:

```
for (int i = 0; i < lst.size(); i++) {
  if (lst.get(i) == "value to remove"){
    // remove value from ArrayList
    lst.remove(lst.get(i));
    // Decrease loop control variable by 1
    i--;
  }
}
```

Now whenever we remove an item, we'll decrease i by 1. Then when we reach the end of the loop, i will increase by 1. It will be like i never changed!

Note: Avoid manipulating the size of an ArrayList when using an enhanced for loop. Actions like adding or removing elements from an ArrayList when using a for each loop can cause a ConcurrentModificationException error.

Instructions :

1. Take a look at the code placed in the main() method of the Lunch class.

Inside the method removeAnts(), use a for loop or a while loop to iterate through lunchBox and remove any element that has the value "ant".

Outside the loop, return the value of lunchBox.

```
import java.util.ArrayList;

class Lunch {

  public static ArrayList<String> removeAnts(ArrayList<String>
lunchBox) {
    // Add your code below
   for(int i=0;i<lunchBox.size();i++){
  if(lunchBox.get(i) == "ant"){

  lunchBox.remove(lunchBox.get(i));
  i--;
}
   }
   return lunchBox;
  }

  public static void main(String[] args) {
    ArrayList<String> lunchContainer = new ArrayList<String>();
    lunchContainer.add("apple");
    lunchContainer.add("ant");
    lunchContainer.add("ant");
    lunchContainer.add("sandwich");
    lunchContainer.add("ant");
    lunchContainer = removeAnts(lunchContainer);
    System.out.println(lunchContainer);

  }
}
```

```
[apple, sandwich]
```

Review

Nice work! Let's iterate over what you've just learned about loops:

- `while` loops: These are useful to repeat a code block an unknown number of times until some condition is met. For example:

```
int wishes = 0;

while (wishes < 3) {
  // code that will run
  wishes++;


}
```

- `for` loops: These are ideal for when you are incrementing or decrementing with a counter variable. For example:

```
for (int i = 0; i < 5; i++) {

  // code that will run
}
```

- For-each loops: These make it simple to do something with each item in a list. For example:

```
for (String inventoryItem : inventoryItems) {

  // do something with each inventoryItem

}
import java.util.ArrayList;
import java.util.Arrays;

class Playground {

  public static void main(String[] args) {

    for (int i = 0; i < 5; i++) {
      System.out.println("Congrats on finishing Java loops!");
    }
  }
}
```

```
Congrats on finishing Java loops!
Congrats on finishing Java loops!
Congrats on finishing Java loops!
Congrats on finishing Java loops!
```