CODING DOJO

Skills Assessment Cheat Sheet

DIRECTIONS

To help you prepare for the skills assessment, read through this document before you begin the placement assessment. Then reference it when you get to the Problem-Solving sections in the assessment.

Variables

In programming, we often need to create and pass around information. To do this, we need to create containers to hold the information. These containers are called **variables** in programming. Variables provide flexibility so that we can capture and hold data in our computer's memory, update it, and do stuff with it.

To create a variable, we use the keyword var and then give it any name we'd like. Like this:

```
var daysUntilBootcamp = 32;
```

Here, we've **assigned** the number 32 to the **variable** daysUntilBootcamp. It's like we put the number 32 into the envelope labeled daysUntilBootcamp.

The output

So, here's the trick in programming software: even though the code *is* doing something, we *can't see* it. Okay, yes, we can see how much is in the variable, but when the code is running, we can't see it. So, developers created a thing called a console log, which spits out information when a program is running. All we have to do is add the keyword console.log and then add what we want it to spit out in parentheses. Like this:

```
console.log(daysUntilBootcamp);
```

In this case, the console log would say "32". Make sense?

Conditionals

What if we have some code that we only want to execute under certain conditions? In computer programming, this is called a *conditional statement*. Like this:

```
if (bankBalance > 100) {
      console.log("I have a lot of money.");
}
else if (bankBalance > 100) {
      console.log("Time to make more money.");
}
```

Note the pattern of an if/else chain. The moment the program hits a condition that evaluates as true, our code won't check any of the other conditions unless it hits an "if" again.

Logic Operators

Ah, math. For those who’ve forgotten, here is a table of the most common ways you can compare two values.

Operator	Description	Examples
==	equal	1 == 1
!=	not equal	1 != 2
>	greater than	2 > 1
<	less than	1 < 2
>=	greater than or equal to	2 >= 2
<=	less than or equal to	1 <= 2

Arrays

So we've learned about capturing data with variables. That's great - we know how to capture data! But what if we need a collection of data? For example, if we are managing office data and we want to store data about an employee in our application, the following code is *just okay*. We ideally want to store these all together though, because they are related!

```
var favoriteMovie = “Titanic”;
var anotherFavoriteMovie = “Schlaf”;
var yetAnotherFavoriteMovie = “Pacific Rim”;
```

Instead, we may want to use a data structure called an **array**, which is like a collection of similar data. It would look like this:

```
var favoriteMovies = [“Titanic”, “Big”, “Pacific Rim”];
```

Access/Update:

Arrays are *indexed*. Meaning that every item in the array is in an indexed position, starting with 0. To access or update values in an array we access the name of the array and the item we want from within it.

```
console.log(favoriteMovies[0]);
```

The console would print out "Titanic" because it's the first in the index. Remember, indexes start at 0.

Length:

Arrays also have a length property, which tells us how many values are contained in the array:

```
console.log(favoriteMovies.length);
```

The console would print out 3 because that's how long the array is.

Functions

Remember algebra? Where you had x = y+3 and you were supposed to figure out what x was if you were given what y was? Here's how that would look in JavaScript:

```
function figureOutX(y) {
      y + 3
}
```

We named the above function figureOutX. And we're promising to give it a value called y.
The second line is what happens when the function runs: it will take the information we gave it - in this case y - and does something with it - in this case, it adds 3 to the value of y.

But what is the value of y? That's what we need to tell it! You tell it, but running the function and giving it the information it needs. To run this function, we have to call it by name and give it the information it needs - in this case, the value of y. Here's how that looks in code:

```
figureOutX(2)
```

Notice a pattern? The 2 in when we called the function and the y in the function's first line are both in parentheses. When you type out the function figureOutX, you're telling the computer, "hey, run this function" and you're giving it the information it needs. In this case, it needs a value for y, which we're telling it is 2.

Loops

Sometimes in code, we'll want to do one thing several times. Let's say I was programming a machine, whose job it is to fill a jar with 100 jellybeans, but it could only add one at a time. I would need to program it so that it would put one bean in and count it as 1. Then, I would need to program it so it would add another bean and count that as 2, because it's 1 + the original. Then I'd need to program it to add another one and count that as 3, as that's 1+ the other 2. Here's how it would look:

```
var firstBean = 1;
console.log(firstBean);
var secondBean = firstBean+1;
console.log(secondBean);
var thirdBean = secondBean+1;
console.log(thirdBean);
```

See how tedious this would be? But, what if I could program a machine to just keep the newest value and just add to it? That's a loop. Here's how it would look:

```
var beanNumber = 0;
while (beanNumber <= 100){
      console.log(beanNumber);
      beanNumber = beanNumber+1;
}
```

The loop would start with counting beanNumber as zero. Then it would add 1. It would circle back up and look at the condition in the while statement: is the number less than or equal to 100? Yes, then loop through again. It would continue doing this until the beanNumber reached 100.

There are 3 key things we need for our loop to properly run:

1. Where to **start**
2. When to **stop**
3. How to **change** our variable during each loop

Types of Loops

While Loops

"Continue to loop **while** the following condition is true."

A while loop has a condition that it checks if it has met each time it loops. Until that condition is met it will continue running our code within the loop:

```
var beanNumber = 0;
while (beanNumber <= 100){
      console.log(beanNumber);
      beanNumber = beanNumber+1;
}
```

This condition will be our stopping point, or what we want to go until we reach. Our starting point is declared before we enter the loop, and we need to change our variable each time within our loop to make sure we aren't stuck in a never ending cycle. If we were missing any one of our three key components, our loop would no longer function.

For Loops

“For the following condition, loop through until I tell you to stop.”

Until while loops that keep going until a condition is met, for loops continue to loop until we tell it to stop. Here’s what the jelly bean counter looks like as a for loop:

```
var beanNumber = 0;
for (var beanNumber = 0; beanNumber <= 100; beanNumber+1){
      console.log(beanNumber);
}
```

Compare it to the while loop. All the same info is there, but it’s just placed in the condition statement of the for loop. Here’s the order the sections of the for loop get executed in:

1. The first piece, var = beanNumber, gets executed just once
2. The second piece, beanNumber <= 100; gets evaluated. If true, the loop happens. If false, the loop stops.
    a. If true:
        i. Whatever is inside the for loop's code block gets executed. In our case the console log would log beanNumber, which would be 0 this first loop.
        ii. Then, the last part of the for loop instructions gets executed. In our case, we would add +1 to the beanNumber that was just logged. So, we would end up with 0+1, which is 1.
        iii. Then it loops back to Step 2: is beanNumber still <=100? If true, repeat the “If true”
    b. If false, loop stops.