Arrays and **`while`** Loops

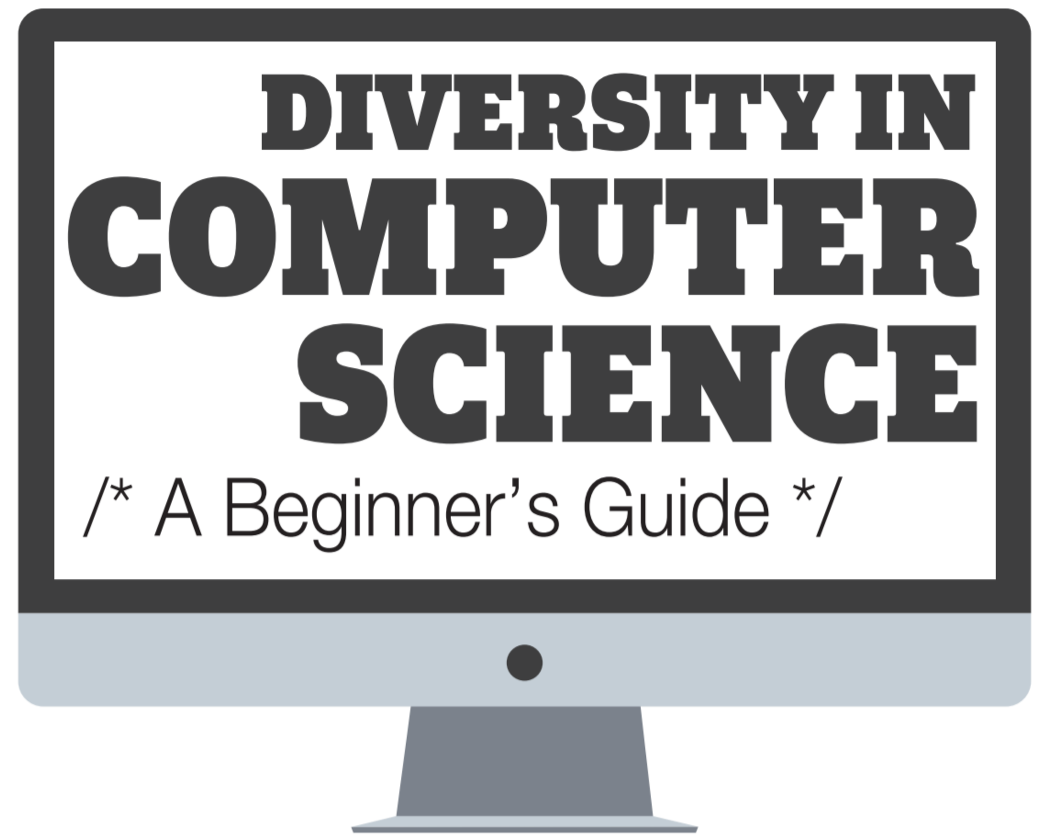
Lecture 15 - COMP110 - Spring 2018

Announcements

- PS04 - March Sadness - Due Sunday 3/25 at 11:59pm
- Review Session - Tomorrow at 5pm in SN014
- Tutoring – Friday from 12pm - 4pm in SN115

Diversity in CS Event

- March 28, 2018 - Next Wednesday - 6pm
- We are hosting a panel discussion around the experiences of women and people of color in tech.
- Panel of UNC faculty, students, and alumni
- Where? Sitterson 011
- Add yourself to the Facebook Event "UNC Computer Science Diversity Panel"



HACK110 – Friday, April 6th at 7pm

- Considering / definitely taking COMP401 in the Fall?
  OR just want to spend more time working on a project idea of yours?

- Come to the COMP110 Hackathon! TEAM110 will be there helping you realize your dreams.
  - Tech Talks
    - What's next in COMP401
    - How-to make games
    - How-to make web apps
  - Food, Snacks, Red Bull
  - Sunrise Yoga

- Two variations of final projects:
  - Standard 110 - A normal problem set assigned by default.
  - Hack 110 - Those who attend can choose their own final project and get a big start on it at the Hackathon.

- To opt-in, RSVP here! - http://bit.ly/2pshkk5

1. Warm-up: What is the output?

```
let xs = listify("a", "b", "c");
let s = reduce(xs, (memo, x) => x + memo + x, "");
print(s);
```

2. Warm-up: What is the output?

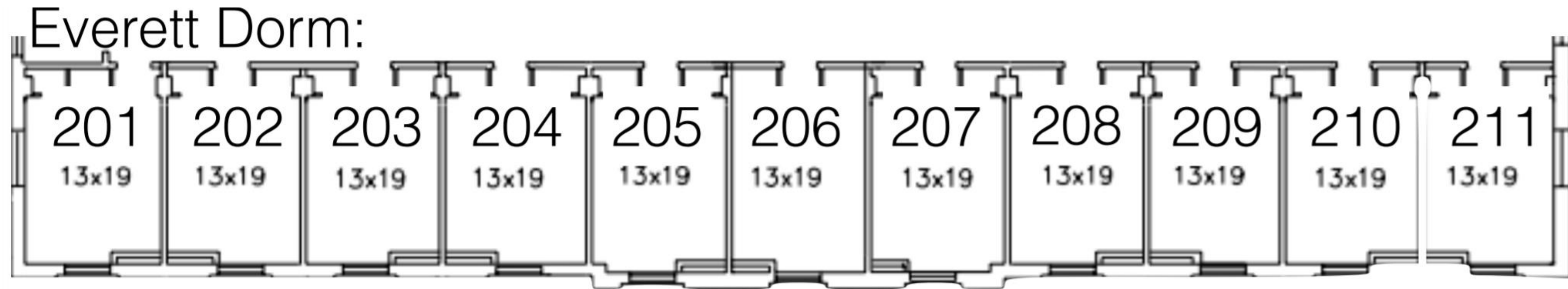
```
let xs = listify(0, 1, 2, 3);
print(first(rest(rest(xs))));
```

Let's Talk Dorms

Dorm Room Naming Proposal

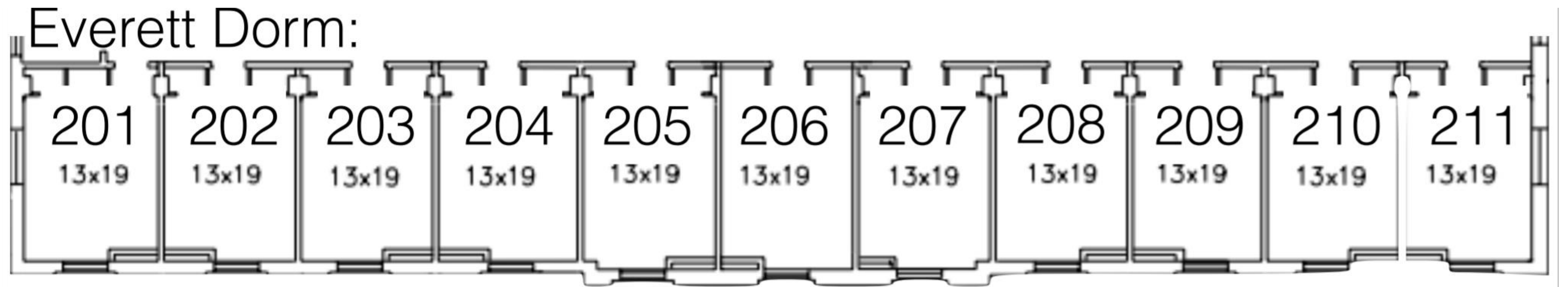
- What if dorms were not numbered? What if they were *named*?
- Some dorm room *name* ideas:
  - Erudite Elephant
  - Silly Snake
  - Loud Liger
  - Ornery Onyx
  - Fuzzy Frog
  - Healthy Hippo
  - Petite Pig
- *What benefits do dorm room numbers provide?*
  - If you can reason through this, you can reason through arrays.

What are the benefits of a Dorm **Name + Number** address scheme?



- Naming is difficult. Numbering is very easy.
  - With a dorm room you only need to remember *one* name.
- Assigning numbers in order allows you to quickly locate a room.
- You can talk about entire *ranges* of rooms.
  - "Ok, I'll check rooms 100-110, you take rooms 111-120."

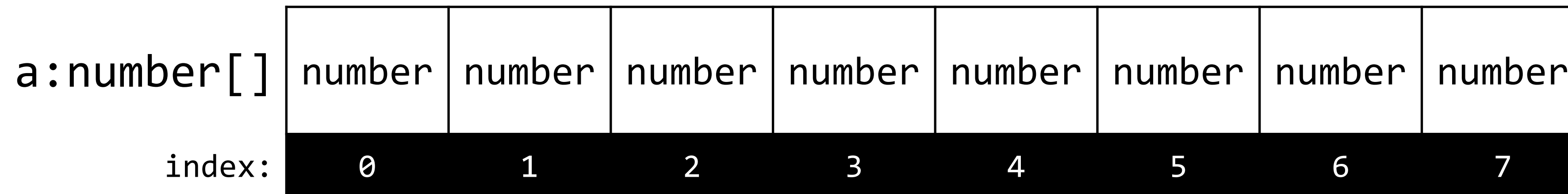
Arrays are like Dorms for Data



a:number[]	number	number	number	number	number	number	number	number
	0	1	2	3	4	5	6	7

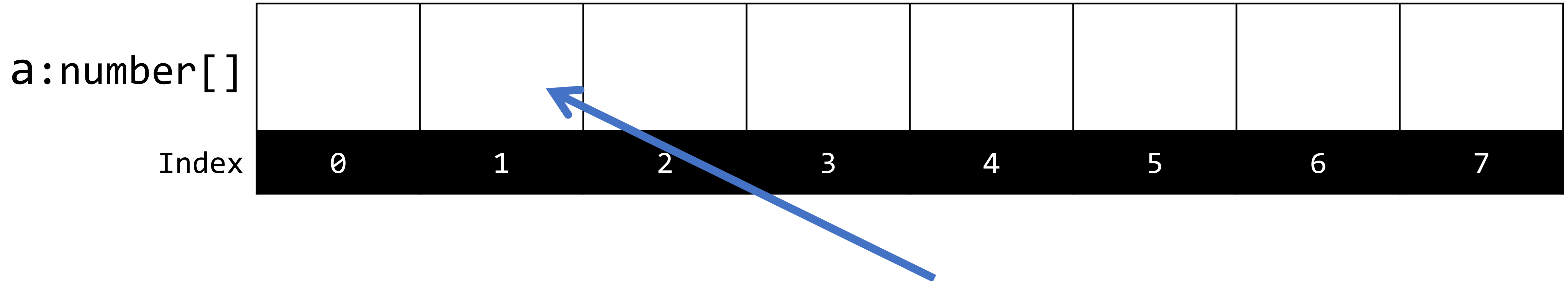
An **array** is a variable, with a **name**, that holds *many* values addressed by an **index** "room" number.

Arrays provide uniform housing for **many values of the same type**.



1. Each item in an array is called an **element**
2. An element is a single value **addressed by its index** ("Room #")
3. All elements in an array are of the **same *type***
   - An array of `numbers, strings, objects, etc`

Elements are addressed by the array variable's **name** and **index**



1. Notation: **`arrayName[index]`**, i.e. **`a[1]`**

2. **Indexing starts at [0]** (not [1])**!!!**
   - First index *always* 0
   - Last index *always* length of array – 1
   - This is a convention shared by most programming languages

Declaring and Constructing **new** Arrays

1. You can **declare an array** of *any type* by placing an empty pair of square braces after the type:

```
let <name>: <type>[];
```
– array of <type>

```
let ages: number[];
```
– array of int values

```
let words: string[];
```
– array of Strings

2. You **construct** an empty array by writing the empty square brackets

3. These two tasks are usually done at the same time:

```
let words: string[] = [];
```

Follow-along: Array Operations

- Open lec15-arrays-and-while-loops / 00-arrays-app.ts
- We'll work through each of the commented lines together…

```
// 1. Declare an array of strings
let words: string[];

// 2. Initialize the array
words = [];

// 3. Assign values to elements by index
words[0] = "Hello";
words[1] = "World";

// 4. Print values
print(words[0]);
print(words[1]);

// 5. Print array
print(words);

// 6. Print array length
print("words length is " + words.length);

// 7. Declare and initialize an array of numbers
let primes: number[] = [2, 3, 5];
print(primes);

// 8. Declare a sum variable and add up the elements of primes
let sum: number = primes[0] + primes[1] + primes[2];
print(sum);
```

Array Operations Reference

Operation	Form	Example
Declaration	let <name>: <type>[];	let scores: number[];
Construction (Empty)	<name> = [];	scores = [];
Construction (Non-empty)	<name> = [<values>];	scores = [12, 0, 9];
# of Elements	<name>.length	scores.length
Access Element	<name>[<index>]	scores[0]
Assign Element	<name>[<index>] = <expression>;	scores[1] = 12;

Lists

- Ordered, linked nodes of data*
- Accessing the *nth* node requires *n* steps…
  - OK when processing an entire List
  - Expensive when you need one thing
- Why did we start with Lists in 110?
  - Forces the good practice of thinking only one element at a time.
  - Lists pair beautifully with learning recursion.

Arrays

- Indexed elements of data taking up a contiguous block of memory
- Accessing the *nth* node requires only 1 step
- In some programming languages (C, C++, Java) arrays must be constructed with a *fixed* size that can't change.
  - TypeScript's arrays *can* grow in size.

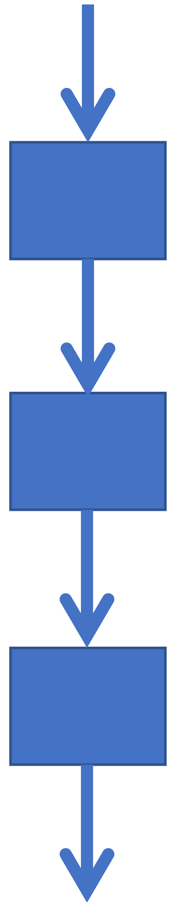
* The concept of the data structure "List" has different specific meanings in different programming languages. Our "List" is technically a singly-linked list that is often seen in functional programming languages like LISP, Scheme, Racket, and clojure. In Java, C++, and others, a List is often thought of as an array you can append to.

Hands-on: Print "I love you" 1,000 times

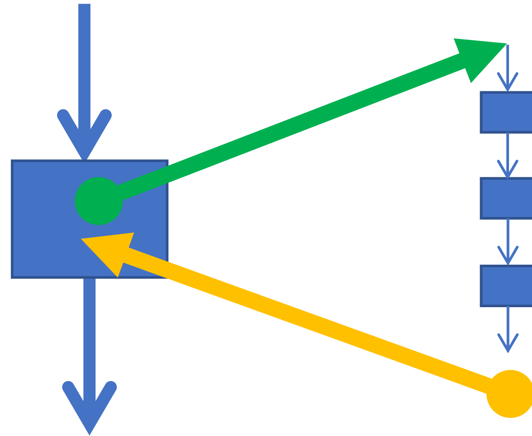
- Open lec15 / 01-while-loop-app.ts
- Goal: Write a program that prints "I love you!! <number>" 1,000 times
- We've done the first 3 numbers for you
- At TODO #1 - Print the message 997 more times.
- Check-in on PollEv.com/compunc when complete

The Moves

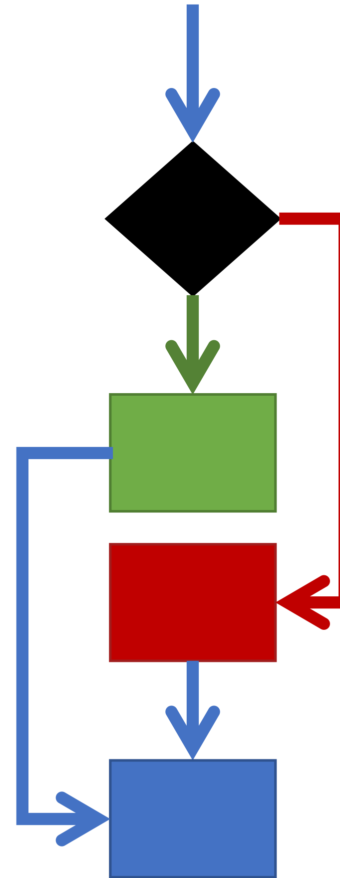
Next Line



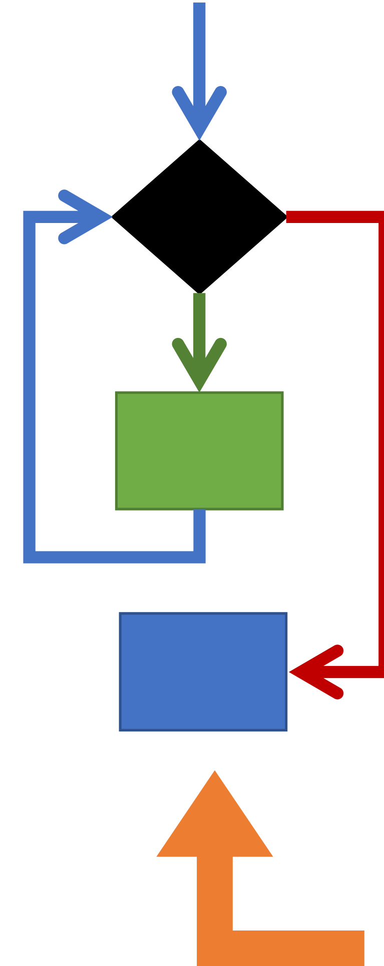
Calls (to Functions)



If-Then-Else



Loops



Today!

Follow-Along: Our first `while` loop

- Open lec15 / 01-while-loop-app.ts
- We'll write a while loop that runs a specific number of times.

```
import { print } from "introcs";

export let main = async () => {

    print("Do you want to know a secret?");

    // Declare a counter variable named i, initialize to 0
    let i = 0;

    // Write a while loop that runs 1000 times and prints "I LOVE YOU"
    // *** DON'T FORGET TO ADD 1 TO i INSIDE OF THE LOOP! ***
    while (i < 1000) {
        print("I LOVE YOU!" + i);
        i = i + 1;
    }

    print(";)");

};
```

Introducing: `while` Loops

- General form of a **`while`** loop statement:

```
while (<boolean expression "test">) {
    <repeat block – statements in braces run when test is true>
}
```

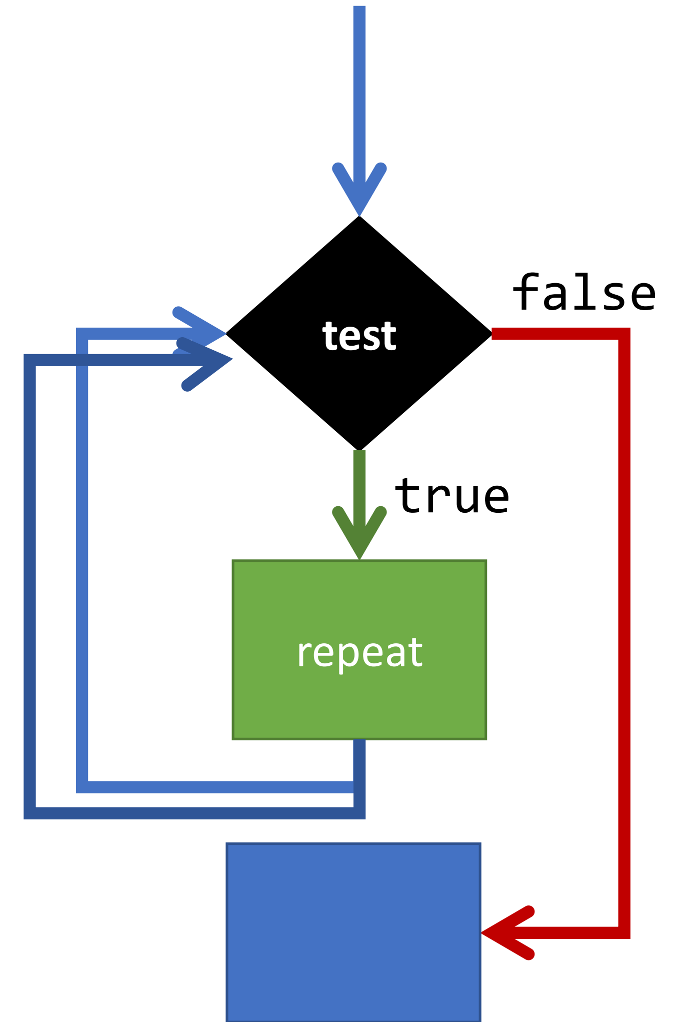
- A **`while`** loop statement can be used *anywhere* you can write a statement.

- ***Like*** an if-then statement:
  - the test you place in the parenthesis must be a `boolean` expression
  - if the test evaluates to **`true`**, the computer will move to the first line of code in the repeat block
  - If the test evaluates to **`false`**, the computer will *jump* over the repeat block

- ***Important! Unlike*** an if-then, **after the last statement in the repeat block** completes, the computer will next ***jump backwards*** **up to the test** and start afresh.

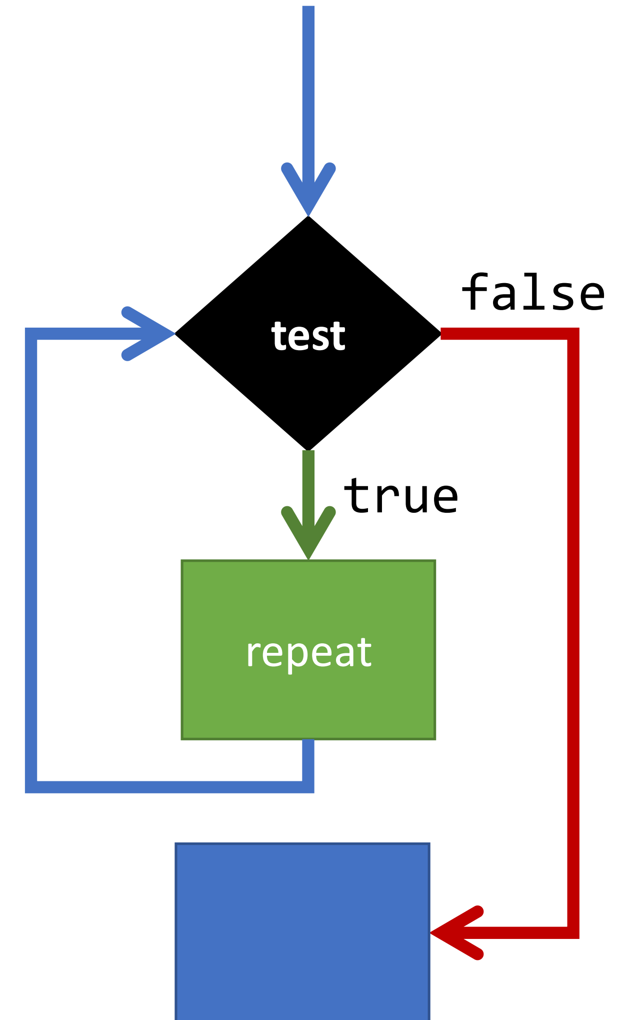
while loop Statement Flow

1. When a while statement is encountered, first its test expression will be evaluated
2. If the test is true, then the processor will continue into the repeat block.
3. After the repeat block, the processor will return to step 1.
4. If the test is still true, it will repeat steps 2 and 3.
5. At some point, the test *should* evaluate to false and the processor will skip past the repeat block.



while loop Statement Notes

- If the test is ***not true*** the first time the while loop is encountered, the computer will jump past the repeat block.
- If the test is ***never false***, the loop is called an ***infinite loop***.
- The only way to *stop* an *infinite* loop is to force quit/close your browser.
- We will learn to write loops that are not infinite with some clever uses of variables and `boolean` expressions.



How do we **avoid infinite loops**?

- How did we avoid infinite recursion?
  - By changing an argument that brought the recursion closer to the base case.
  - Lists – we recurred on the *rest* of the list
  - Numbers – we recurred on the number plus/minus 1

- A **loop will complete** when its **test** expression is **`false`**

- ***Something must change*** **each time a loop runs to bring the test expression closer to being `false`.**

MICHAEL
JACKSON
OFF
THE
WALL
"while loops…
they don't stop 'til
they get enough"
~ Michael Jackson

Writing a **`while`** loop that repeats a specific number of times.

- Repeating a task a specific number of times is a **very** common task in computing.
- You will see this all semester.
- Three keys:
  1) Declare a counter variable and initialize it to 0.
  2) The loops test will check that the counter variable is less than the # of times you want to repeat
  3) **Don't forget!** The last step inside of the repeat block is incrementing your counter variable.

```
1
let i = 0;

while (i < __2__) {

   // Do Something Useful

   i = i + 1; 3

}
```

Loops, Arrays, and Looping Algorithms

1. Arrays allow us to refer to values *by index number from 0…length-1*
   - ex: `a[0] … a[a.length – 1]`

2. Loops allow us to easily *increment a number variable by 1 repeatedly*

- Combining these enables us to process arrays using *looping algorithms*

- Let's try printing out all the values of an array individually…

Hands-on # 2: Printing all elements of an array

- Open lec15 / 02-loop-through-array-app.ts

1. Declare a counter variable named i and initialize it to 0.
2. Write a while statement that loops while **i** is less than **a.length**
   1. Inside of the loop, print the value stored at a[i]
   2. DON'T FORGET TO INCREMENT i BY 1 INSIDE THE REPEAT BLOCK!

```
while (<boolean expression>) {
    <repeat block>
}
```

- Check-in on pollev.com/compunc when complete

```
import { print } from "introcs";

export let main = async () => {

    let a = [2, 3, 5, 7, 11, 13];

    let i = 0;
    while (i < a.length) {
        print(a[i]);
        i = i + 1;
    }

}

main();
```

Increment Operator (++)

- Adding one to a variable is so common when looping there is a special operator for it…

- We often write: **`i = i + 1;`**

- We can instead write: **`i++;`**

- These two statements have the exact same impact of incrementing **`i`**'s value by **`1`**.

Sum Algorithm Intuition

1. Declare a variable to hold the total.

2. Work through each element of the array, index-by-index.

3. Take the number stored at each index and add it to the total.

4. Once the processor has moved through every index, total stores the sum.

Hands-on #3: Summing a number array

- How can we add up all the numbers in a number array?
- Open Lec 15 / 03-sum-array-app.ts
- Try to follow the TODO instructions. Refer to the previous example for help. NOTE: rather than printing each element, you are trying to increase the total variable by that element's value.
- Check-in on PollEv.com/compunc when complete
- Done? Try changing the numbers in your original array in the main function and convincing yourself the algorithm is correct.

```
let sum = (a: number[]): number => {
    let total = 0;

    // TODO: Write a while loop
    // It should:
    // 1. Loop while i is less than parameter a's length
    // 2. Inside the loop:
    // 2.1. Increase the result variable by a[i]
    // 2.2. Increment i by 1
    let i = 0;
    while (i < a.length) {
        total = total + a[i];
        i = i + 1;
    }

    return result;
};
```

Modulo Operator – "Remainder" – %

- The modulo operator (%) can be used to find an integer division remainder:
  - 3 % 2 is "the remainder of 3 divided by 2" … 1!
  - 4 % 2 is 0
  - 5 % 2 is 1
- A simple way of remembering modulo is forming teams… if you have 5 people and you are forming teams of 3, how many people are not on a team? 5 % 3

Recursion

- More generally powerful
- Less computationally efficient*
  - Time: function call jumps require more cpu steps than loop jumps
  - Space: each function has its own "frame" of memory for its parameters and local variables
- Require you to write small, single-purposed functions which leads to elegant, composable code.

Loops

- Often quicker to think up a looping solution vs. recursive
- More computationally efficient
- Warning: Ability to write single, long functions with lots of convoluted loops *can* lead to fragile, monolithic code.
- Theoretically: Any program you can write recursively you can also write with loops. However, some recursive algorithms require fancy trickery to express as loops.

* Some language engines employ a technique called tail-call optimization (TCO) which make recursive function solutions just as efficient as looping solutions by converting a recursive function into a loop behind the scenes. TCO gives you the best of both worlds!