type

variables in Java have a specified type

```
# okay in Python
foo = 7     # foo is an int
foo = False # now, foo is a boolean
```

```
// NOT okay in Java
int foo = 7; // foo is an int
foo = false; // Error: incompatible types:
             // boolean cannot be converted to int
```

in Java, declaring and initializing variables are separate things

-
  ```
  // Option A: two lines
  int foo; // declare a variable foo of type int
  foo = 7; // initialize foo to 7
  ```

-
  ```
  // Option B: one line
  int foo = 7; // declare int foo and initialize it to 7
  ```

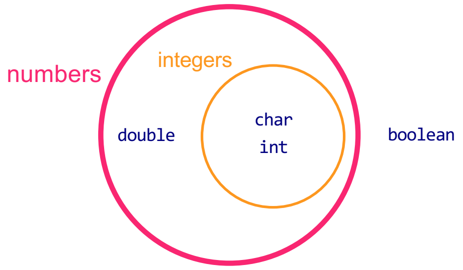
built-in data types

primitive data types

`boolean`, `char`, `double`, `int`

- a **boolean** stores a truth value
  - `true`, `false`
- a **char** stores a character
  - `'\0'`, `'a'`, `'Z'`, `'!'`, `'\n'`
- a **double** stores a floating point number
  - `0.0`, `-0.5`, `3.1415926`, `Double.NEGATIVE_INFINITY`
- an **int** stores an integer number
  - `0`, `-1`, `4`

primitive data type Venn diagram



`char` is an integer type

- a **`char`** is an integer type
  - each `char` has a corresponding integer, for example (`'a' == 97`)
  - the letters are in order (`'a' == 97`), (`'b' == 98`), (`'c' == 99`)...
  - the numbers are also in order (`'0' == 48`), (`'1' == 49`)...
  - you can do math with `char`'s
    - `char foo = 'a' + 2; // foo is 'c'`
    - `char bar = '0' + 7; // bar is '7'`
    - `int  baz = '6' - '0'; // baz is 6`

zero

- each primitive data type has its own notion of what it means to "be zero"
  - `int     zero =     0;`
  - `double  zero =   0.0;`
  - `boolean zero = false;`
  - `char    zero =  '\0'; // the "null character"`

String

`String` (1/2)

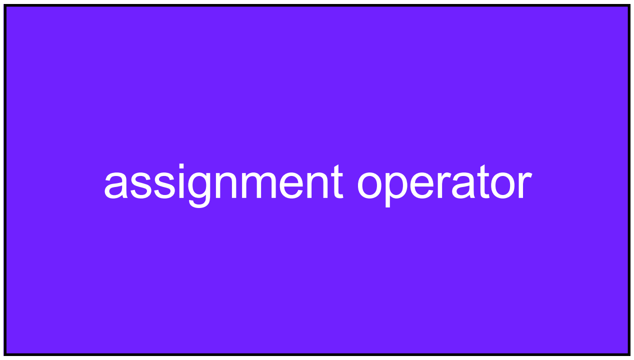
- a **`String`** is a sequence of characters
  - `"Hello World"`, `"Foo123"`

`String` (2/2)

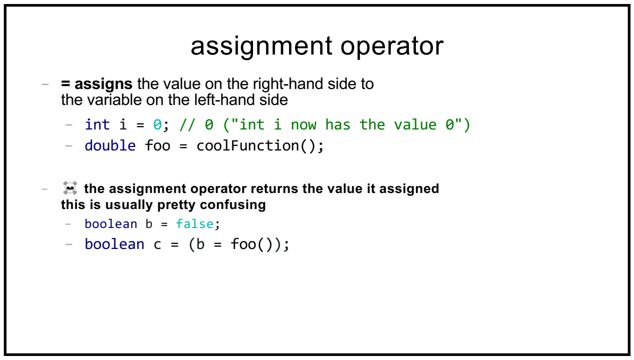
- `String`'s have several useful functions/methods
  (which are described in our Documentation)
  - `String string = "Hello"; // makes a new String`
  - `System.out.println(string.length()); // 5`
  - `System.out.println(string.charAt(1)); // 'e'`
  - `System.out.println(string.substring(0, 3)); // "Hel"`

operators

(except for bitwise operators, which we'll do later maybe)



assignment operator

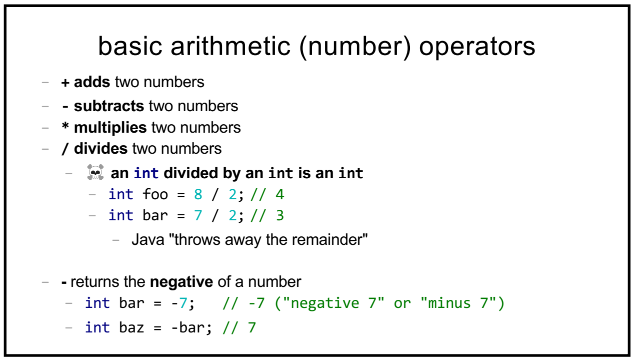


assignment operator

- **= assigns** the value on the right-hand side to the variable on the left-hand side
  - `int i = 0; // 0 ("int i now has the value 0")`
  - `double foo = coolFunction();`

- ☠️ **the assignment operator returns the value it assigned this is usually pretty confusing**
  - `boolean b = false;`
  - `boolean c = (b = foo());`

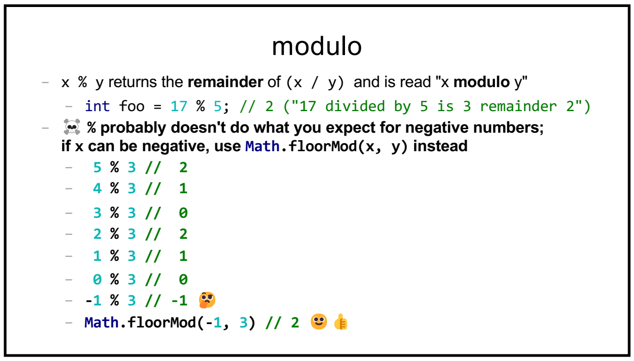
arithmetic operators



basic arithmetic (number) operators

- **+ adds** two numbers
- **- subtracts** two numbers
- *** multiplies** two numbers
- **/ divides** two numbers
  - ☠️ **an `int` divided by an `int` is an `int`**
    - `int foo = 8 / 2; // 4`
    - `int bar = 7 / 2; // 3`
      - Java "throws away the remainder"

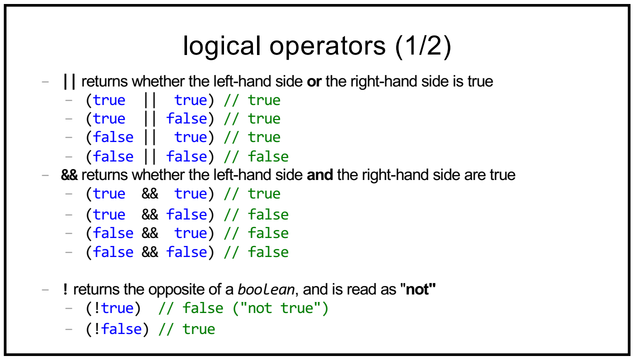
- **-** returns the **negative** of a number
  - `int bar = -7;   // -7 ("negative 7" or "minus 7")`
  - `int baz = -bar; // 7`



modulo

- `x % y` returns the **remainder** of `(x / y)` and is read "x **modulo** y"
  - `int foo = 17 % 5; // 2 ("17 divided by 5 is 3 remainder 2")`
- ☠️ **% probably doesn't do what you expect for negative numbers; if x can be negative, use `Math.floorMod(x, y)` instead**
  - ` 5 % 3 //  2`
  - ` 4 % 3 //  1`
  - ` 3 % 3 //  0`
  - ` 2 % 3 //  2`
  - ` 1 % 3 //  1`
  - ` 0 % 3 //  0`
  - `-1 % 3 // -1` 🤔
  - `Math.floorMod(-1, 3) // 2` 🙂👍

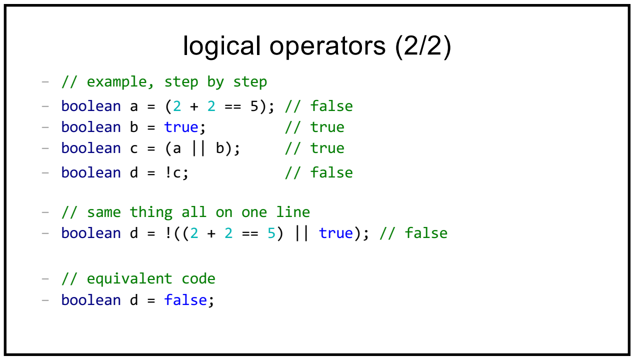
# logical operators



## logical operators (1/2)

- **||** returns whether the left-hand side **or** the right-hand side is true
  - `(true  ||  true) // true`
  - `(true  || false) // true`
  - `(false ||  true) // true`
  - `(false || false) // false`
- **&&** returns whether the left-hand side **and** the right-hand side are true
  - `(true  &&  true) // true`
  - `(true  && false) // false`
  - `(false &&  true) // false`
  - `(false && false) // false`

- **!** returns the opposite of a *`boolean`*, and is read as "**not**"
  - `(!true)  // false ("not true")`
  - `(!false) // true`

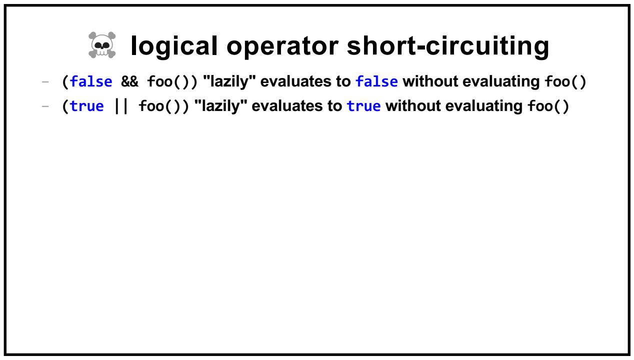


## logical operators (2/2)

- `// example, step by step`
- `boolean a = (2 + 2 == 5); // false`
- `boolean b = true;         // true`
- `boolean c = (a || b);     // true`
- `boolean d = !c;           // false`

- `// same thing all on one line`
- `boolean d = !((2 + 2 == 5) || true); // false`

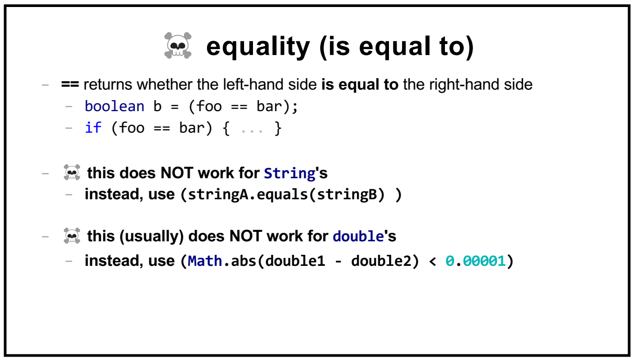
- `// equivalent code`
- `boolean d = false;`



## ☠ logical operator short-circuiting

- **`(false && foo())` "lazily" evaluates to `false` without evaluating `foo()`**
- **`(true || foo())` "lazily" evaluates to `true` without evaluating `foo()`**

comparison operators

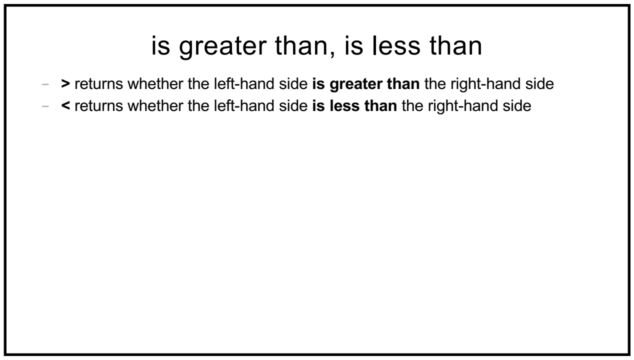


## ☠ equality (is equal to)

- **==** returns whether the left-hand side **is equal to** the right-hand side
  - `boolean b = (foo == bar);`
  - `if (foo == bar) { ... }`

- ☠ **this does NOT work for `String`'s**
  - **instead, use `(stringA.equals(stringB) )`**

- ☠ **this (usually) does NOT work for `double`'s**
  - **instead, use `(Math.abs(double1 - double2) < 0.00001)`**

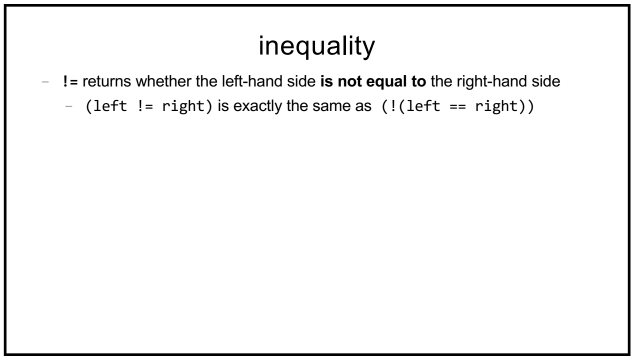


is greater than, is less than

- **>** returns whether the left-hand side **is greater than** the right-hand side
- **<** returns whether the left-hand side **is less than** the right-hand side

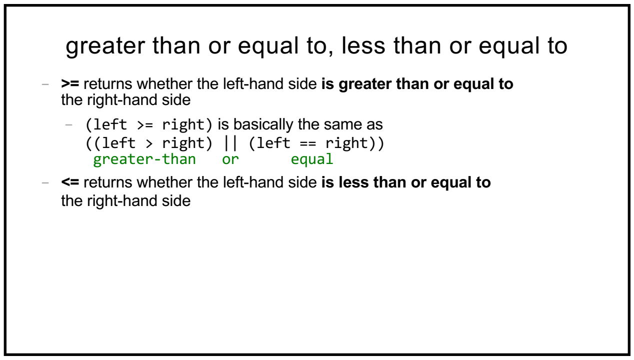
convenient operators

(feel free to ignore these for now)



inequality

- **!=** returns whether the left-hand side **is not equal to** the right-hand side
  - `(left != right)` is exactly the same as `(!(left == right))`

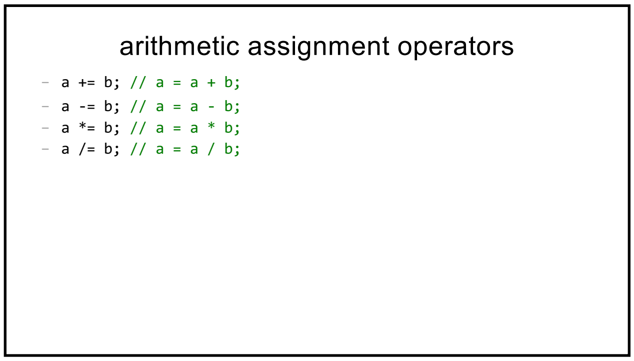


greater than or equal to, less than or equal to

- **>=** returns whether the left-hand side **is greater than or equal to** the right-hand side
  - `(left >= right)` is basically the same as

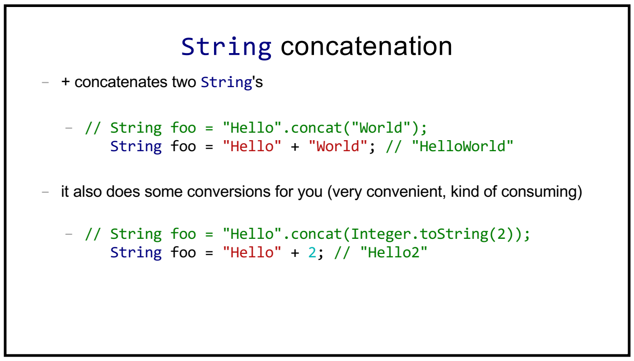
```
((left > right) || (left == right))
 greater-than   or      equal
```

- **<=** returns whether the left-hand side **is less than or equal to** the right-hand side



arithmetic assignment operators

```
a += b; // a = a + b;
a -= b; // a = a - b;
a *= b; // a = a * b;
a /= b; // a = a / b;
```



`String` concatenation

- + concatenates two `String`'s

```
// String foo = "Hello".concat("World");
String foo = "Hello" + "World"; // "HelloWorld"
```

- it also does some conversions for you (very convenient, kind of consuming)

```
// String foo = "Hello".concat(Integer.toString(2));
String foo = "Hello" + 2; // "Hello2"
```

increment operator

- to "**increment**" means to increase the value of a number by one
  - `i = i + 1;`
  - `i += 1;`

  - the **pre-increment** `++i` increments `i` and returns the new value of `i`
  - `j = ++i; // i = i + 1;`
  - `         // j = i;`

  - the **post-increment** `i++` increments `i` and returns the old value of `i`
  - `j = i++; // j = i;`
  - `         // i = i + 1;`

decrement operator

- to "**decrement**" means to decrease the value of a number by one
  - `i = i - 1;`
  - `i -= 1;`

  - the **pre-decrement** `--i` decrements `i` and returns the new value of `i`
  - `j = --i; // i = i - 1;`
  - `         // j = i;`

  - the **post-decrement** `i--` decrements `i` and returns the old value of `i`
  - `j = i--; // j = i;`
  - `         // i = i - 1;`

flow control

the execution of a Java program starts at the top of `main(...)` and flows down down down

let's step through this program in our minds, and then in DrJava's debugger

```
class Main {
    public static void main(String[] arguments) {
        double a = 3.0;
        double b = 4.0;

        double result = Math.sqrt(a * a + b * b);
        System.out.println(result);
    }
}
```

functions (preview)

functions (preview)

- when a **function** is called, control flow jumps to the top of the function, flows down through it, and then jumps back to right after the function call

```
class Main {
    static double pythagoreanTheorem(double a, double b) {
        return Math.sqrt(a * a + b * b);
    }

    public static void main(String[] arguments) {
        double hypotenuse = pythagoreanTheorem(3.0, 4.0);
        System.out.println(hypotenuse);
    }
}
```

## assert

`assert condition;` (1/2)

- an **assert statement** crashes the program if its condition is false
  - `assert false;` crashes the program no matter what

```
class Main {
  static double pythagoreanTheorem(double a, double b) {
        assert a > 0.0;
        assert b > 0.0;
        return Math.sqrt(a * a + b * b);
    }

    public static void main(String[] arguments) {
        double hypotenuse = pythagoreanTheorem(3.0, 4.0);
        System.out.println(hypotenuse);
    }
}
```

`assert condition;` (2/2)

- ☠️ **asserts are actually disabled by default in Java, and are ignored**
  - **DrJava enables them by default** 😎 👍

if and else

`if (condition) { body }`

- an **if statement** lets a program make a decision (instead of just stepping down down down forever down)

```
int choice = getIntFromUser();

if (choice == 0) {
    System.out.println("The user chose 0. What a fine choice");
}
```

`if (...) { if-body } else { else-body }`

- the body of an **else statement** is executed if its corresponding if statement's condition is false

```
int choice = getIntFromUser();

if (choice == 0) {
    System.out.println("The user chose 0. What a fine choice");
} else {
    System.out.println("The user did not choose 0.");
    System.out.println("How avant-garde!");
}
```

`if (...) { ... } else if (...) { ... } ... else { ... }`

- if and else statements can be chained together
  - this is great for decisions with many options

```
int choice = getIntFromUser();

if (choice == 0) {
    ...
} else if (choice == 1) {
    ...
} else if (choice == 2) {
    ...
} else {
    assert false;
}
```

`while` and `do...while`

`while (condition) { body }`

- a **while loop** repeats a block of code as long as the condition holds (if the condition is false the first time we see it, we never execute the body)
  - `while (true) { ... }` is an infinite loop

```
while (!gameOver) {
    ...

    if (player.health <= 0) {
        gameOver = true;
    }
}
```

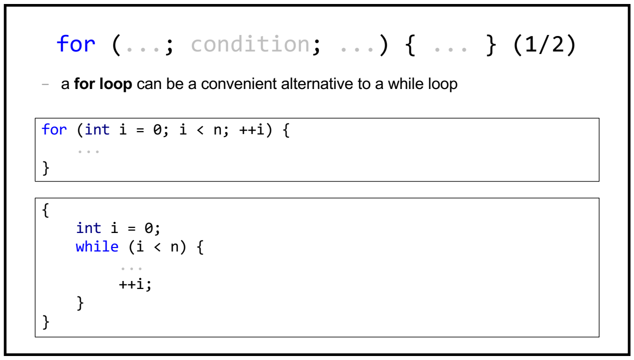
`do { ... } while (...);`

- a **do...while** loop is (sometimes) useful when you know you need to do something at least once (and then potentially a bunch more times)

```
int choice;
do {
    choice = getIntFromUser();
} while (!(0 <= choice && choice <= 2));
```

```
int choice = getIntFromUser();
while (!(0 <= choice && choice <= 2)) {
    choice = getIntFromUser();
}
```

`for`

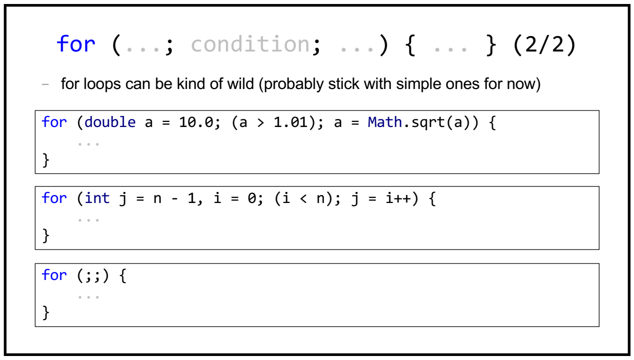


for (...; condition; ...) { ... } (1/2)

- a **for loop** can be a convenient alternative to a while loop

```
for (int i = 0; i < n; ++i) {
    ...
}
```

```
{
    int i = 0;
    while (i < n) {
        ...
        ++i;
    }
}
```



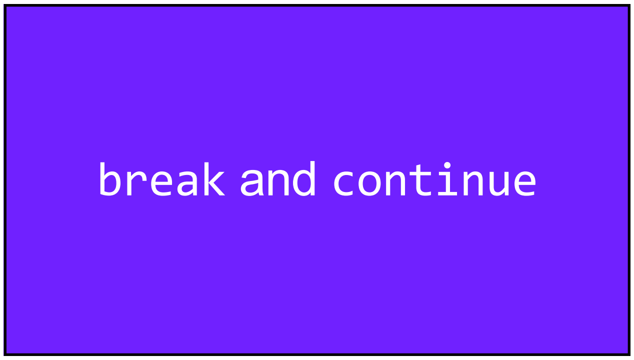
for (...; condition; ...) { ... } (2/2)

- for loops can be kind of wild (probably stick with simple ones for now)

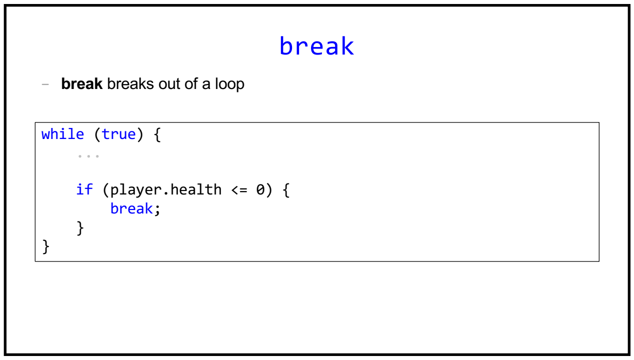
```
for (double a = 10.0; (a > 1.01); a = Math.sqrt(a)) {
    ...
}
```

```
for (int j = n - 1, i = 0; (i < n); j = i++) {
    ...
}
```

```
for (;;) {
    ...
}
```



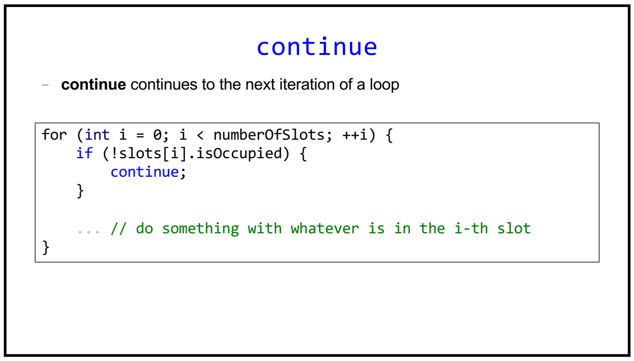
break and continue



break

- **break** breaks out of a loop

```
while (true) {
    ...
    if (player.health <= 0) {
        break;
    }
}
```

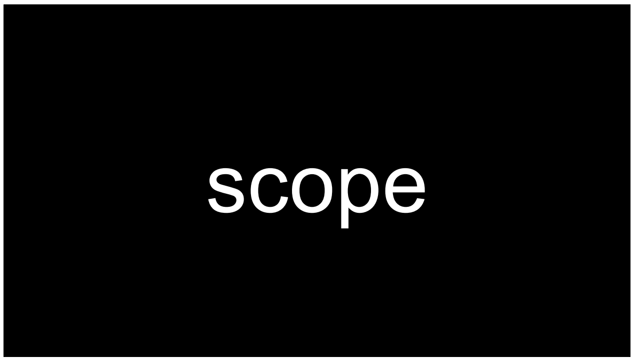


continue

- **continue** continues to the next iteration of a loop

```
for (int i = 0; i < numberOfSlots; ++i) {
    if (!slots[i].isOccupied) {
        continue;
    }

    ... // do something with whatever is in the i-th slot
}
```



scope

{ ... }

scope

- a **scope** is a region of code in which variables live
  - in Java, a scope is (usually) defined by a pair of curly braces
    - OUTER_SCOPE { INNER_SCOPE } OUTER_SCOPE

```
{
    int i;
    {
        int j;
        // you can find i here
        // you can find j here
    }
    // you can find i here
    // you can NOT find j here
}
```

common scope-related errors

cannot find symbol (1/2)

```
Error: cannot find symbol
  symbol:   variable foo
  location: class Main

class Main {
    public static void main(String[] arguments) {
        if (...) {
            int foo = 0;
        } else {
            int foo = 1;
        }
        System.out.println(foo);
    }
}
```

cannot find symbol (2/2)

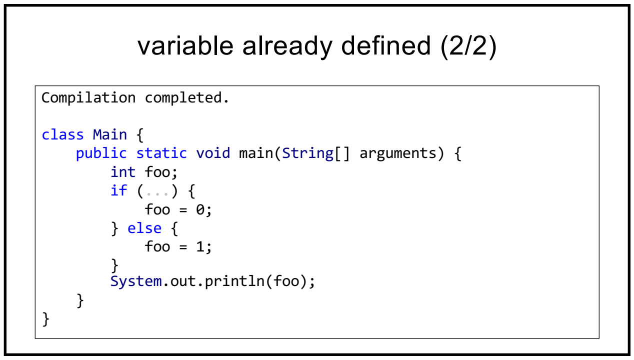
```
Compilation completed.

class Main {
    public static void main(String[] arguments) {
        int foo;
        if (...) {
            foo = 0;
        } else {
            foo = 1;
        }
        System.out.println(foo);
    }
}
```

variable already defined (1/2)

```
Error: variable foo is already defined in method
main(java.lang.String[])

class Main {
    public static void main(String[] arguments) {
        int foo;
        if (...) {
            int foo = 0;
        } else {
            foo = 1;
        }
        System.out.println(foo);
    }
}
```

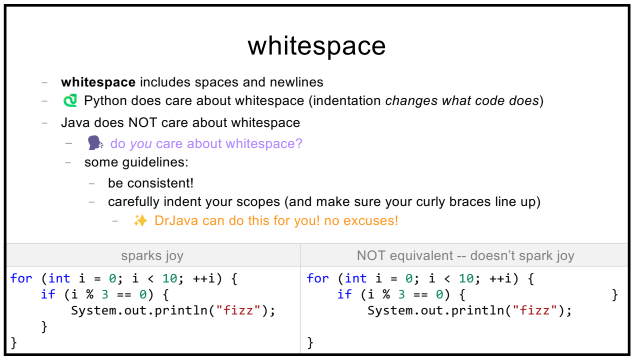


variable already defined (2/2)

```
Compilation completed.

class Main {
    public static void main(String[] arguments) {
        int foo;
        if (...) {
            foo = 0;
        } else {
            foo = 1;
        }
        System.out.println(foo);
    }
}
```

whitespace



whitespace

- **whitespace** includes spaces and newlines
- 🐍 Python does care about whitespace (indentation *changes what code does*)
- Java does NOT care about whitespace
  - 🗣 do *you* care about whitespace?
  - some guidelines:
    - be consistent!
    - carefully indent your scopes (and make sure your curly braces line up)
      - ✨ DrJava can do this for you! no excuses!

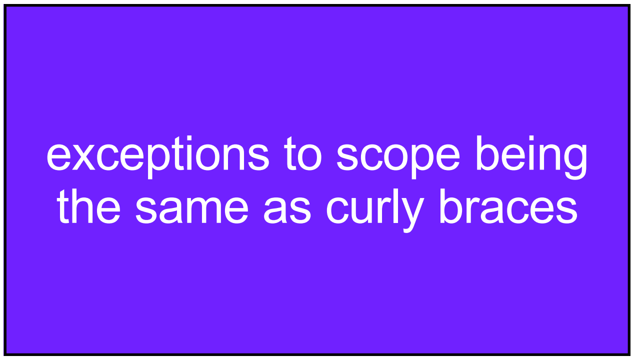
sparks joy

```
for (int i = 0; i < 10; ++i) {
    if (i % 3 == 0) {
        System.out.println("fizz");
    }
}
```

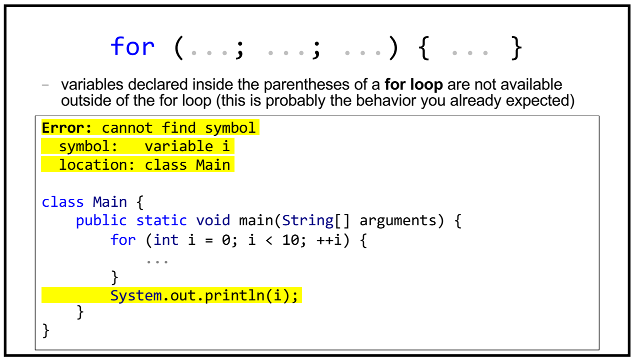
NOT equivalent -- doesn't spark joy

```
for (int i = 0; i < 10; ++i) {
    if (i % 3 == 0) {                          }
        System.out.println("fizz");

}
```



exceptions to scope being the same as curly braces

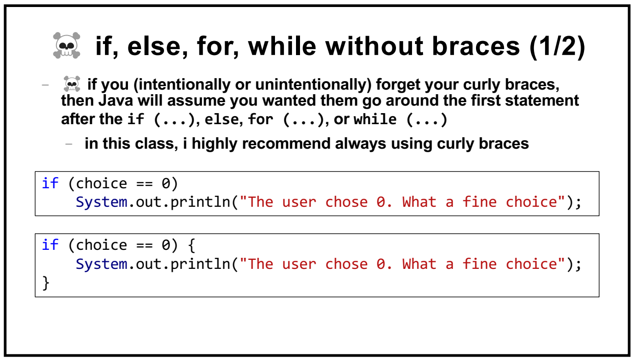


`for (...; ...; ...) { ... }`

- variables declared inside the parentheses of a **for loop** are not available outside of the for loop (this is probably the behavior you already expected)

```
Error: cannot find symbol
  symbol:   variable i
  location: class Main

class Main {
    public static void main(String[] arguments) {
        for (int i = 0; i < 10; ++i) {
            ...
        }
        System.out.println(i);
    }
}
```



## ☠️ if, else, for, while without braces (1/2)

- ☠️ **if you (intentionally or unintentionally) forget your curly braces, then Java will assume you wanted them go around the first statement after the `if (...)`, `else`, `for (...)`, or `while (...)`**
  - **in this class, i highly recommend always using curly braces**

```
if (choice == 0)
    System.out.println("The user chose 0. What a fine choice");
```

```
if (choice == 0) {
    System.out.println("The user chose 0. What a fine choice");
}
```

## ☠️ if, for, while without braces (2/2)

```
if (choice == 0)
    System.out.println("The user chose 0. What a fine choice");
else
    System.out.println("The user did not choose 0.");
    System.out.println("How avant-garde!");
```

```
if (choice == 0) {
    System.out.println("The user chose 0. What a fine choice");
} else {
    System.out.println("The user did not choose 0.");
} // whoops!
    System.out.println("How avant-garde!");
```

ANNOUNCEMENTS

`hexDigitSum` CORRECTION -- don't print error message and return; instead, just crash using an **`assert false;`** 😳 👍

today is **Fun Friday!** 🎉 🎉 🎉 🔥 🚨 🚨

WARMUP

compress this code!

```
boolean isEven;
if (i % 2 == 0) {
    isEven = true;
} else if (i % 2 != 0) {
    isEven = false;
}

if (isEven) {
    ...
}
```

🙆 TODAY

goto and big O notation

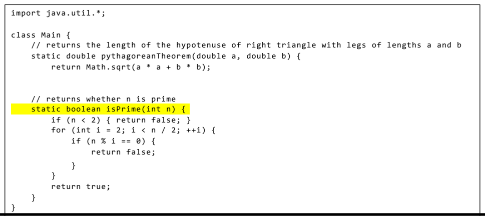
homework hints

homework hints (1/2)

- `char` is an integer type!
  - `// char c`
  - ~~`48 <= c`~~
  - `'0' <= c`
- we have ✨Documentation✨ linked on the website! (it's really good!)
  - ~~`for (char c : string.toCharArray()) {`~~
  - `for (int i = 0; i < string.length(); ++i) {`
    - `char c = string.charAt(i);`
  - `// i know it's more typing but you'll thank me (much) later`
- the starter code has ✨Examples✨ (they're relevant!)

homework hints (2/2)

- if you get a weird **`"illegal start of expression"`** compiler error...
  - highlight your code and press Tab!
    - you're probably missing a curly brace somewhere above the error



```
import java.util.*;

class Main {
    // returns the length of the hypotenuse of right triangle with legs of lengths a and b
    static double pythagoreanTheorem(double a, double b) {
        return Math.sqrt(a * a + b * b);


    // returns whether n is prime
    static boolean isPrime(int n) {
        if (n < 2) { return false; }
        for (int i = 2; i < n / 2; ++i) {
            if (n % i == 0) {
                return false;
            }
        }
        return true;
    }
}
```

Java doesn't have a

goto

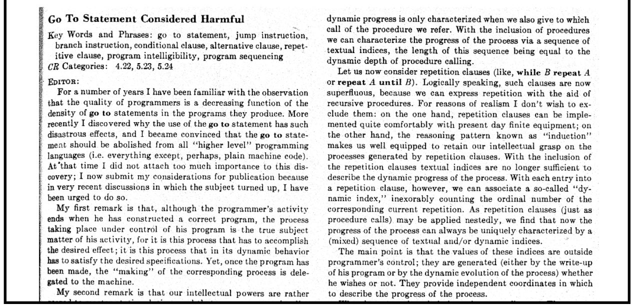
```
goto label;
```

- a **goto** "goes to" some **label**
  - it turns out loops are just convenient alternatives to goto's
  - goto's are considered *spooky* 👻

```
while (condition) {
    ...
}
```

```
label BEGINNING_OF_WHILE:
if (!condition) {
    goto END_OF_WHILE;
}
...
goto BEGINNING_OF_WHILE;
label END_OF_WHILE:
```

Edgar Dijkstra: Go To Statement Considered Harmful



Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

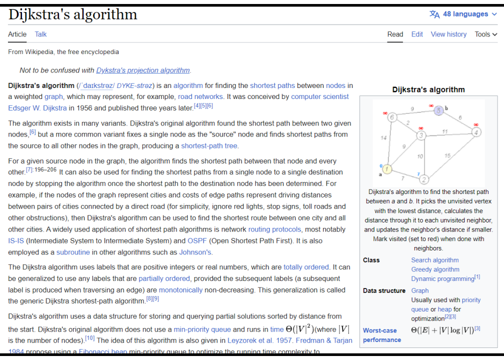
My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather

dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.



Dijkstra's algorithm

From Wikipedia, the free encyclopedia

Not to be confused with Dykstra's projection algorithm.

Dijkstra's algorithm (/ˈdaɪkstrəz/ *DYKE-strəz*) is an algorithm for finding the shortest paths between nodes in a weighted graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.[4][5][6]

The algorithm exists in many variants. Dijkstra's original algorithm found the shortest path between two given nodes,[6] but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.

For a given source node in the graph, the algorithm finds the shortest path between that node and every other.[7]: 196–206 It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined. For example, if the nodes of the graph represent cities and costs of edge paths represent driving distances between pairs of cities connected by a direct road (for simplicity, ignore red lights, stop signs, toll roads and other obstructions), then Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. A widely used application of shortest path algorithms is network routing protocols, most notably IS-IS (Intermediate System to Intermediate System) and OSPF (Open Shortest Path First). It is also employed as a subroutine in other algorithms such as Johnson's.

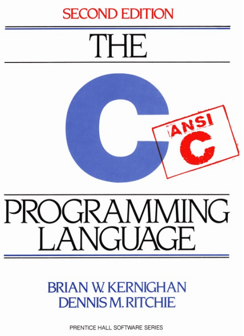
The Dijkstra algorithm uses labels that are positive integers or real numbers, which are totally ordered. It can be generalized to use any labels that are partially ordered, provided the subsequent labels (a subsequent label is produced when traversing an edge) are monotonically non-decreasing. This generalization is called the generic Dijkstra shortest-path algorithm.[8][9]

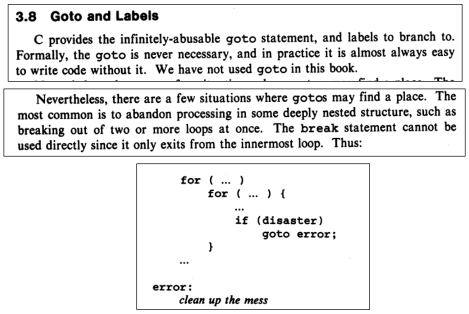
Dijkstra's algorithm uses a data structure for storing and querying partial solutions sorted by distance from the start. Dijkstra's original algorithm does not use a min-priority queue and runs in time $\Theta(|V|^2)$(where $|V|$ is the number of nodes).[10] The idea of this algorithm is also given in Leyzorek et al. 1957. Fredman & Tarjan 1984 propose using a Fibonacci heap min-priority queue to optimize the running time complexity to

Dijkstra's algorithm

Dijkstra's algorithm to find the shortest path between *a* and *b*. It picks the unvisited vertex with the lowest distance, calculates the distance through it to each unvisited neighbor, and updates the neighbor's distance if smaller. Mark visited (set to red) when done with neighbors.

Class	Search algorithm Greedy algorithm Dynamic programming[1]
Data structure	Graph Usually used with priority queue or heap for optimization[2][3]
Worst-case performance	$\Theta(\|E\| + \|V\| \log \|V\|)$[3]





3.8 Goto and Labels

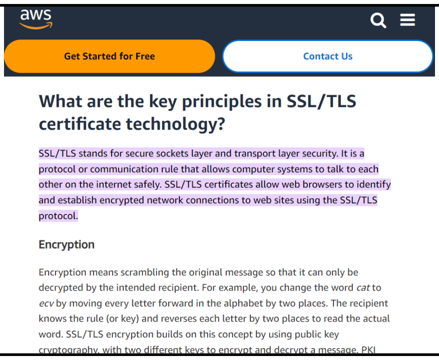
C provides the infinitely-abusable goto statement, and labels to branch to. Formally, the goto is never necessary, and in practice it is almost always easy to write code without it. We have not used goto in this book.

Nevertheless, there are a few situations where gotos may find a place. The most common is to abandon processing in some deeply nested structure, such as breaking out of two or more loops at once. The break statement cannot be used directly since it only exits from the innermost loop. Thus:

```
    for ( ... )
        for ( ... ) {
            ...
            if (disaster)
                goto error;
        }
    ...

error:
    clean up the mess
```

story time ☕

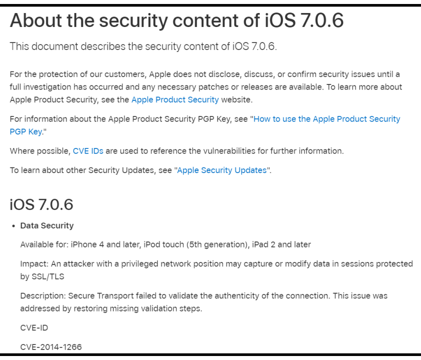


What are the key principles in SSL/TLS certificate technology?

SSL/TLS stands for secure sockets layer and transport layer security. It is a protocol or communication rule that allows computer systems to talk to each other on the internet safely. SSL/TLS certificates allow web browsers to identify and establish encrypted network connections to web sites using the SSL/TLS protocol.

Encryption

Encryption means scrambling the original message so that it can only be decrypted by the intended recipient. For example, you change the word *cat* to *ecv* by moving every letter forward in the alphabet by two places. The recipient knows the rule (or key) and reverses each letter by two places to read the actual word. SSL/TLS encryption builds on this concept by using public key cryptography, with two different keys to encrypt and decrypt a message. PKI



About the security content of iOS 7.0.6

This document describes the security content of iOS 7.0.6.

For the protection of our customers, Apple does not disclose, discuss, or confirm security issues until a full investigation has occurred and any necessary patches or releases are available. To learn more about Apple Product Security, see the Apple Product Security website.

For information about the Apple Product Security PGP Key, see "How to use the Apple Product Security PGP Key."

Where possible, CVE IDs are used to reference the vulnerabilities for further information.

To learn about other Security Updates, see "Apple Security Updates".

iOS 7.0.6

- **Data Security**

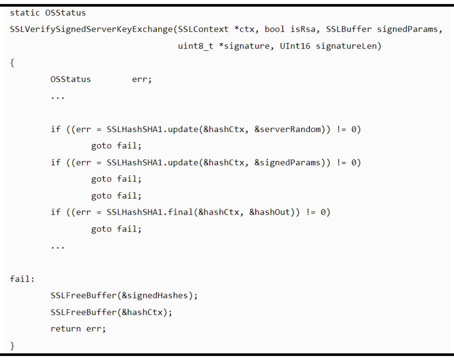
  Available for: iPhone 4 and later, iPod touch (5th generation), iPad 2 and later

  Impact: An attacker with a privileged network position may capture or modify data in sessions protected by SSL/TLS

  Description: Secure Transport failed to validate the authenticity of the connection. This issue was addressed by restoring missing validation steps.

  CVE-ID

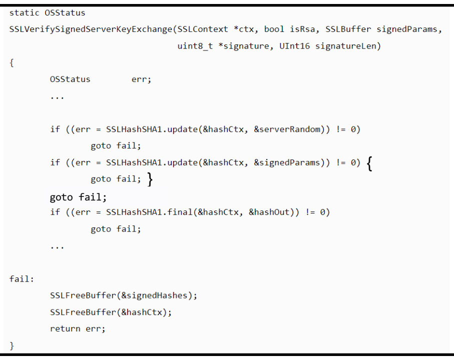
  CVE-2014-1266



```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
        OSStatus        err;
        ...

        if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
                goto fail;
        if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
                goto fail;
                goto fail;
        if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
                goto fail;
        ...

fail:
        SSLFreeBuffer(&signedHashes);
        SSLFreeBuffer(&hashCtx);
        return err;
}
```

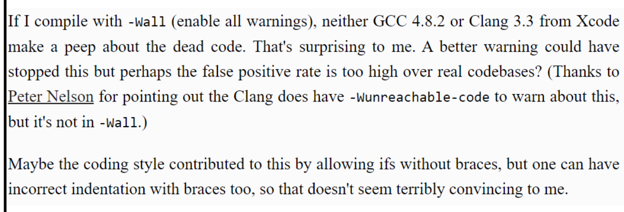


```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
        OSStatus        err;
        ...

        if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
                goto fail;
        if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0) {
                goto fail; }
        goto fail;
        if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
                goto fail;
        ...

fail:
        SSLFreeBuffer(&signedHashes);
        SSLFreeBuffer(&hashCtx);
        return err;
}
```

did the bug *really* happen because they left out the curly braces?



If I compile with `-Wall` (enable all warnings), neither GCC 4.8.2 or Clang 3.3 from Xcode make a peep about the dead code. That's surprising to me. A better warning could have stopped this but perhaps the false positive rate is too high over real codebases? (Thanks to Peter Nelson for pointing out the Clang does have `-Wunreachable-code` to warn about this, but it's not in `-Wall`.)

Maybe the coding style contributed to this by allowing ifs without braces, but one can have incorrect indentation with braces too, so that doesn't seem terribly convincing to me.

but one can have
incorrect indentation with braces too

did the bug *really* happen because they left out the curly braces?

no, the bug happened because they didn't code in DrJava!

highlight your code and press Tab!

big O

big O (1/2)

- **big O** describes a function's "limiting behavior"
  - to find a mathematical function's big O notation...
    - 1. throw away the coefficients
    - 2. find the fastest growing term
    - 3. the function is O(FASTEST_GROWING_TERM)

  - **e.g.,** $f(n) = 7n^2 + 100n + 4732$
    - 1. throw away coefficients to get $n^2 + n + 1$
    - 2. fastest growing term is n^2
    - 3. $f(n)$ is $O(n^2)$

big O (2/2)

- what is $f(n) = 77n^7 + 2^n$ in big O notation?
  - $n^7 + 2^n$
  - 2^n 🧪 is this true?
  - $\mathcal{O}(2^n)$
- what is $f(n) = 100$ in big O notation?
  - 1
  - 1 🧠 what does this *mean*?
  - $\mathcal{O}(1)$
- what is $f(n) = n + \log(n)$ in big O notation?
  - $n + \log(n)$
  - n 🧪 is this true?
  - $\mathcal{O}(n)$

1. throw away the coefficients
2. find the fastest growing term
3. the function is $\mathcal{O}$(FASTEST_GROWING_TERM)

🗣 what is the big O of your **`isPrime(int n)`**?

can you do better?

Caaaaaaaaaaaarl

- **e.g.,** Imagine a classroom with n students. I want to figure out if any students are named Carl.
  - I need an ✨Algorithm✨ -- `boolean isAnyoneNamedCarl(Student[] students);`
  - What is the big O of the following algorithms?
    - **Algorithm 1:** Ask each student, one at a time, “Are you named Carl?”
    - **Algorithm 2:** Pass a paper around the room, and have each student write their name on it. Then take the paper, and read through it.
    - **Algorithm 3:** The students draw straws. The student who draws the short straw must leave. On their way out of the room, ask them whether their name is Carl. Repeat this procedure until the room is empty.
    - **Algorithm 4:** Play Kahoot. The winner legally changes their name to Carl.