

✨ discuss with 3+ people ✨

$+$, $-$, $/$, and $*$ are called "operators";
why?

what does $2 + 2$ like... $*$ mean $*$?

what is $7 / 2$?

who has finished
Tutorial-00?

thank you to those who emailed DrJava questions!

note: my email responses tend to be really short;
it actually honestly doesn't mean i'm annoyed;
i just have to write emails really fast because there
are ~64 students in this class 😊👍

ps screenshots are really helpful if you email me!

is there anyone
here who is not yet
enrolled in CS 136?

questions
before we
begin?

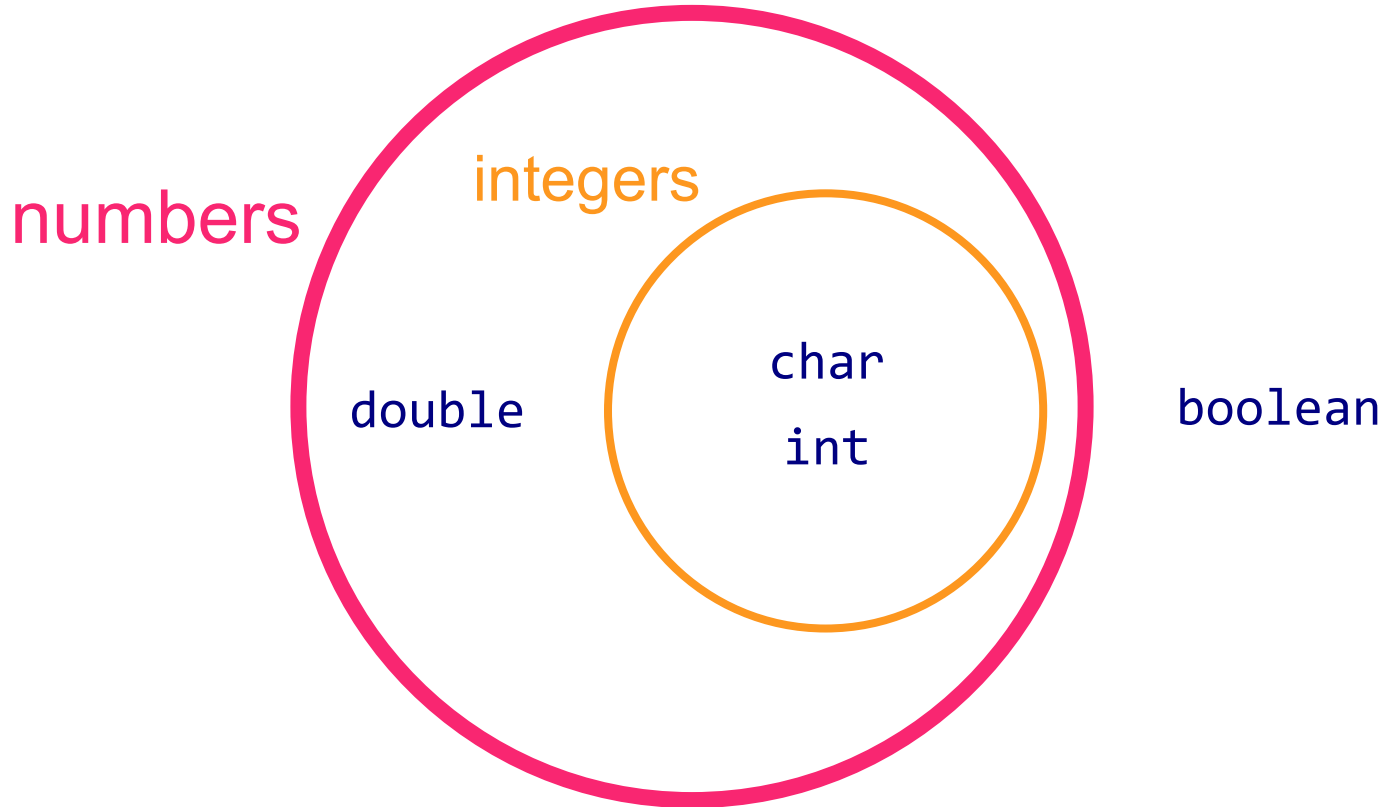
primitives

some primitive data types

boolean, char, double, int

- a `boolean` stores a truth value
 - `true`, `false`
- a `char` stores a character
 - `'\0'`, `'a'`, `'Z'`, `'!'`
- 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"

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**
 - `int i = 4;`
 - `int j = (i *= 42);`

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

- $x \% N$ returns the **remainder** of x / N
and is read "**x modulo N**"
 - `int foo = 17 % 5; // 2 ("17 divided by 5 is 3 remainder 2")`
-  **% probably doesn't do what you expect for negative numbers**
 - **instead, use** `((x % N + N) % N)`
 - 

logical operators

logical operators

- `||` 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

- `// 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 a 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 {  
    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 {  
    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 (. . .) { . . . }

- a **while loop** repeats a block of code
 - `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 once, and then maybe also a bunch more times

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

for

for (. . . ; . . . ; . . .) { . . . } (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 (. . . ; . . . ; . . .) { . . . } (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
 - OUTSIDE_SCOPE { INSIDE_SCOPE } OUTSIDE_SCOPE

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	NOT equivalent -- doesn't spark joy
<pre>for (int i = 0; i < 10; ++i) { if (i % 3 == 0) { System.out.println("fizz"); } }</pre>	<pre>for (int i = 0; i < 10; ++i) { if (i % 3 == 0) { System.out.println("fizz"); } }</pre>

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 assumed)

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, for, while without braces (1/2)

-  if you (intentionally or unintentionally) forget to type the curly braces, Java assumes you wanted them go around the first statement after the condition
 - in this class, i highly recommend always using the 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!");
```

let's do the
homework

(including infinite loop example)

big O

how to read/write big O notation

- **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 $\mathcal{O}(\text{FASTEST_GROWING_TERM})$
- **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 $\mathcal{O}(n^2)$

how to read/write big O notation

- **e.g.**, what is $f(n) = 77n^7 + 2^n$ in big O notation?

- $n^7 + 2^n$

- 2^n  is this true?

- $\mathcal{O}(2^n)$

- **e.g.**, what is $f(n) = 100$ in big O notation?

- 1

- 1  what does this mean?

- $\mathcal{O}(1)$

- **e.g.**, what is $f(n) = n + \log(n)$ in big O notation?

- $n + \log(n)$

- n  is this true?

- $\mathcal{O}(n)$

how to read/write big O notation

- **e.g.**, Imagine a classroom with n students. I want to figure out if any students are named Carl.
 - I need an ✨ Algorithm ✨, **e.g.**, `boolean isAnyoneNamedCarl(Student[] students);`
 - In big O, what is longest amount of time each of these algorithms could take to run?
 - **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.



examples

// return whether n is prime

```
boolean isPrime(int n) {  
    for (int i = 0; i < n; ++i) {  
        if (n % i == 0) {  
            return false;  
        }  
    }  
    return true;  
}
```

e.g., when called on $\{ 0.0, 4.2, -10.0, 99.0 \}$, returns 3

when called on {}, returns -1

```
int findIndexOfMaxEle
    int result = -1;
    double maxElement
    for (int i = 0; i
        if (array[i]
            result =
                maxElemen
        }
    }
    return result;
}
```

returns the number of digits an integer has (in base-10)

e.g., when called on 427, returns 3

```
int getNumberOfDigits(int n) {
    int result = 0;
    while (n != 0) {
        ++result; // result = result + 1;
        n /= 10;  // n = n / 10;
    }
    return result;
}
```

print 😊 😄 ❤️ ♦️ ♣️

! " # \$ % & ' () * + , -

./0123456789:;<=>?@ABCDEFGHIJK

NOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz

nopqrstuvwxyz{|}~△

```
for (char c = 0; c < 128; ++c)
    System.out.print(c);
}
```

array

- an **array** is

```
int[] foo;
```

```
double[] baz;
```

```
String[] bar;
```

accessing an array

- 🕒 accessing an array is $\mathcal{O}(1)$
 - **i.e.**, "accessing an array takes a constant number of CPU cycles"

getting the value of array's i-th element

```
int currentValue = array[i];
```

setting the value of array's i-th element

```
array[i] = newValue;
```

creating an array

- 🕒 creating an array is $\mathcal{O}(n)$

creating a new integer array with 7 elements

```
int[] array = new int[7];
```

creating a new String array with n elements

```
String[] array = new String[n];
```



TODO list

- the **list** (*aka* sequence) abstract data type is
 - an array list's *length* is the number of elements stored inside it
 - an array list stores its elements inside of an array
 - i call this array the array list's “internal array”
 - an array list's **capacity** is the length of this internal array

array list

- the **array list** (*aka* dynamic array, stretchy buffer, **vector**) data structure implements the list abstract data type using an array
 - an array list's *length* is the number of elements stored inside it
 - an array list stores its elements inside of an array
 - i call this array the array list's “internal array”
 - an array list's **capacity** is the length of this internal array

array list

-  an array list's length is NOT the same thing as its capacity
 - e.g., imagine an array list that is currently storing 3 `String`s in an internal `String[]` of length 8. This array list has length 3 and capacity 8. Its internal array might be ["Blango", "Sprout", "Sparket", null, null, null, null, null,], where the null “elements” are empty slots in the internal array. There are $8 - 3 = 5$ empty slots
-  even though another name for an array list is a "vector," the array list is NOT related to the vector from math and physics

ArrayList() { ... }

- a new array list should have...
 - length equal to 0
 - internalArray equal to a new array with length equal to some starting capacity, e.g., 4

```
void add(ElementType element) { ... }
```

- to **add** (*aka append, push back*) an element to an array list...
 - if the internal array is full...
 - create a new array two times the length of the current internal array
 - copy the elements of the current internal array over into the new array
 - update the array list's internal array reference to refer to the new array
 - write the element to the first available empty slot in the internal array
 - increment the array list's length
- 🕒 adding an element to an array list has worst-case runtime of $\mathcal{O}(n)$ and amortized worst-case runtime of $\mathcal{O}(1)$
 - if we're out of space, it takes $\mathcal{O}(n)$ time to copy the elements over, however this happens only every $\mathcal{O}(n)$ adds

about the lectures













how to read side by side code

- i often show equivalent code side by side in order to...
 - relate new concepts to old concepts
 - compare and contrast different design decisions, *e.g.*,

a piece of code	equivalent code	equivalent code	equivalent code
<pre>boolean isEven; if (i % 2 == 0) { isEven = true; } else if (i % 2 != 0) { isEven = false; } if (isEven) { ... }</pre>	<pre>boolean isEven; if (i % 2 == 0) { isEven = true; } else { isEven = false; } if (isEven) { ... }</pre>	<pre>boolean isEven = (i % 2 == 0); if (isEven) { ... }</pre>	<pre>if (i % 2 == 0) { ... }</pre>

how to read emojis

- i use emojis to help you read and study
 -  info only relevant inside the world of CS136
 -  fun Java fact! (*i.e.*, NOT relevant to C/C++; please forget after CS136)
 -  comparison to Python
 -  **common misconception or potential source of bugs**
 -  spoilers/hints
 -  big O runtime
 -  optional (NOT on exams) but sparks joy
 -  question for you to think about
 -  question for your to talk about
 -  question for you to experiment with
 - 