Naming things

- in programming, we use variables as a named place to store information
- in programming, we try to make variables have *reasonable* and *readable* names---rarely will we use 1-letter variables in real code
- in Java, tradition says that multiword variables be done in camelCase, e.g., `numChairs` would be a good variable name for the number of chairs

Naming things

Declaring

- in Java, before using a variable we must ***declare*** it by stating the type of the variable
- Declaring looks like this: `int num = 3;` or like this: `int max;`

Naming things

Assignment statements

- in Java, we put information in a variable by using an ***assignment statement*** like `num = 4;` or `max = 9001;`
- Remember that `=` should be pronounced as "gets" or "stores" i.e., `num = 4` says "num *gets* 4" to remind us that the variable is *getting* or *storing* the number 4 (for now)

Naming things

Understanding assignment statements

- Remember that a line like `num = num + 1` is perfectly reasonable in programming
    - first we figure out the right-hand side
    - assuming num is currently storing 4, the right-hand side becomes `4+1` which **evaluates** to `5`
    - the line has essentially become `num = 5`
    - `5` is stored in `num` (replacing/overwriting the `4` that was *previously* stored there)

Naming things

Understanding assignment statements

- Consider the following code segment:

```
int x = 2;
int y = x;
x = 5;
```

Naming things

Understanding assignment statements

- Consider the following code segment:

  ```
  int x = 2;
  int y = x;
  x = 5;
  ```

  - note that after running, `y` still stores `2`

Naming things

types

- recall that programmers must care about the ***type*** of each value, that is the "sort of thing that it is"
- the **type** of a value determines how you can use it and how it interacts with other values
- big types to know for now: (*CAPITALIZATION MATTERS!*)
  - `int` : for integers
  - `double` : for "decimal-y" numbers
  - `boolean` : for `true` / `false` things
  - `String` : for strings aka pieces of text (could include letters, spaces, digits, punctuation, etc.)

Only do things sometimes aka conditionals

if example

- Use if when we only want something to happen sometimes

```java
if (num == 5) {
    System.out.println("The number is 5.");
}
System.out.println("Always happens");
```

Only do things sometimes aka conditionals

if details

```
if (cond) {
  body0;
  body1;
}
after0;
```

- for now, all of the "punctuation" is *necessary*
- `body0` and `body1` only run when `cond` is true
- `after0` runs no matter what
- `cond` should *evaluate* to `true` / `false`

Only do things sometimes aka conditionals

if else example

- use if...else when we want one of two things to happen depending on some condition

```java
if (num == 5) {
    System.out.println("The number is 5.");
} else {
    System.out.println("The number is not 5.");
}
System.out.println("Always happens");
```

Only do things sometimes aka conditionals

if...else details

```
if (cond) {
  b0;
  b1;
} else {
  b2;
}
a0;
```

- for now, all of the "punctuation" is *necessary*
- `b0` and `b1` only run when `cond` is `true`
- `b2` only runs when `cond` is `false`
- `a0` runs no matter what
- `cond` should *evaluate* to `true` / `false`
- `else` *needs* an `if` to "attach" to

Only do things sometimes aka conditionals

Comparison

- Remember that we use `=` for **assignment statements**!
- If we want to ask whether two things are equal, we use `==`
- We can also use `<`, `>`, `<=`, `>=`, `!=` (not equals)
- The result of these comparisons is a `boolean` value

Repeating things aka loops

while loop example

```
int num = 0;
while (num < 3) {
  System.out.println(num);
  num = num + 1;
}
```

Repeating things aka loops

while loop example

```
int num = 0;
while (num < 3) {
  System.out.println(num);
  num = num + 1;
}
```

- the above code prints 0, 1, 2 (on separate lines)

Repeating things aka loops

while loop explained

```
while (cond) {
  b0;
  b1;
}
a0;
```

- for now, all of the "punctuation" is *necessary*
- first `cond` is evaluated if it's true, we run the body
- `b0` and `b1` are the body, so they're run
- when we reach the end of the body, we go back up and evaluate `cond`; if it's true we run the body...
- eventually once `cond` evaluates to `false` we skip over the body and go right to `a0`

Using magic spells aka calling a static method

Java caveat

- In Java, there aren't really such a thing as functions
- For now, the closest thing we have are static methods
- For now, static methods will seem a lot like functions, but eventually the difference will be clear
- static is a terrible name (solid contender for worst name in CS) and has essentially nothing to do with staying still/unchanging

Using magic spells aka calling a static method

lil' bit of vocab

- the "inputs" to a static method are called ***arguments***
- the "result"/output-to-the-program of a static method is called the ***return value***

Using magic spells aka calling a static method

example

```java
double squareArea = 16.0;
double squareSide = Math.sqrt(squareArea);
```

- the static method is _____
- the argument to the static method is _____
- the return value of the static method is _____

Using magic spells aka calling a static method

example

```java
double squareArea = 16.0;
double squareSide = Math.sqrt(squareArea);
```

- the static method is `Math.sqrt()`
- the argument to the static method is `squareArea`
- the return value of the static method is `4.0`

Using magic spells aka calling a static method

return value/output IMPORTANT NOTE

- **NOTE:** a **return value** is output for a different part of the program to use; it is NOT output to the user

Using magic spells aka calling a static method

Dot notation

- when using a static method from a different file/library, we put the file name before a dot and then the name of the method
- when using a static method in the same file it's defined, we are allowed to leave off the thing before the dot and the dot