Agenda

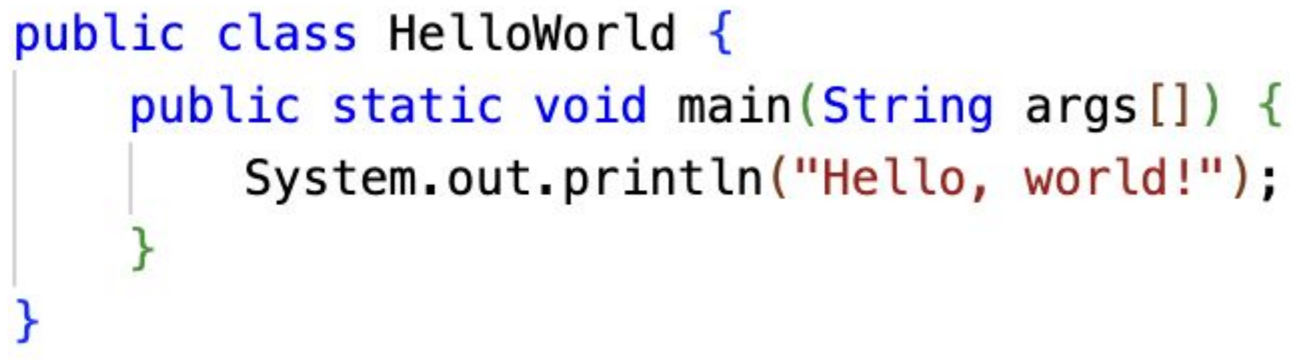
- **Dec 7 (Wednesday)**
  - Review Units 1, 2, and 3
  - Begin on File System Simulator
- **Dec 9 (Friday)**
  - Review Unit 4
  - File System Simulator Discussion / Demos
- **Dec 12 (Monday)**
  - Review Units 5 & 6
  - File System Simulator Discussion / Demos
- **Dec 14 (Wednesday)**
  - Final Exam

Final Review - Part 1
12/7/2022

Final Review - Part 1

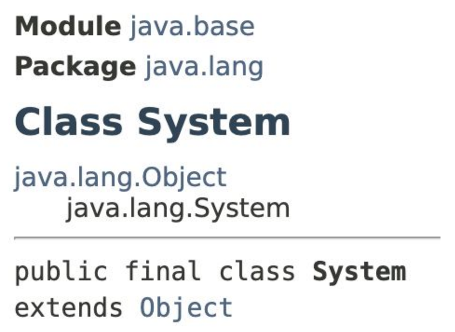
- Unit 1
- Unit 2
- Unit 3
- Begin on File System Simulator

1.1 Do we look at this through new eyes?



```java
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println("Hello, world!");
    }
}
```

```java
System.out.print("Hello, world!"); //prints on one continuous line

System.out.println("Hello, world!"); //adds a new line after printing
```

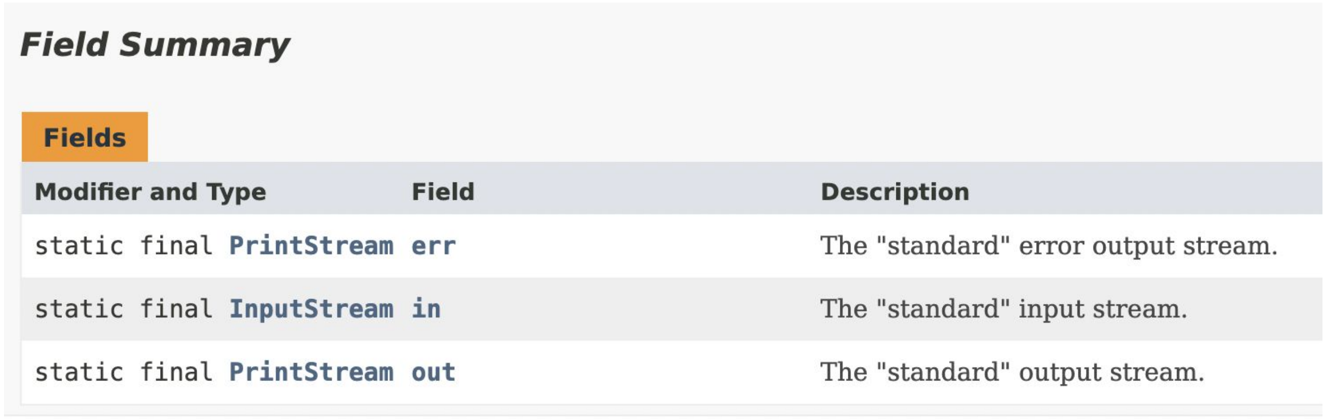


Module java.base
Package java.lang

Class System

java.lang.Object
    java.lang.System

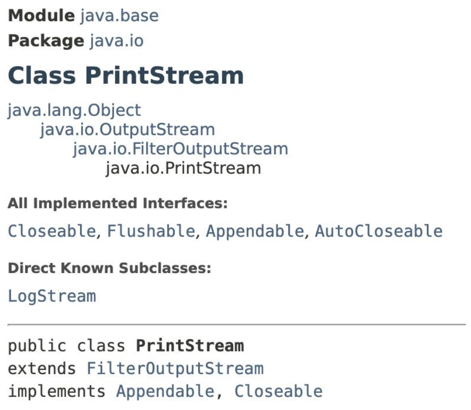
```
public final class System
extends Object
```



Field Summary

Fields

Modifier and Type	Field	Description
static final **PrintStream**	**err**	The "standard" error output stream.
static final **InputStream**	**in**	The "standard" input stream.
static final **PrintStream**	**out**	The "standard" output stream.



Module java.base
Package java.io

Class PrintStream

java.lang.Object
    java.io.OutputStream
        java.io.FilterOutputStream
            java.io.PrintStream

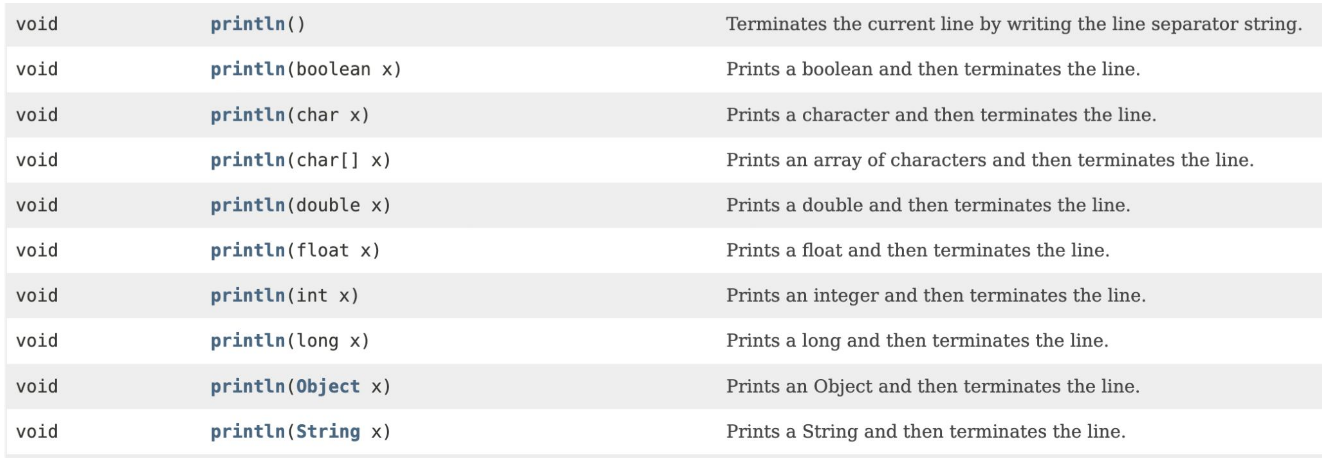
All Implemented Interfaces:

Closeable, Flushable, Appendable, AutoCloseable

Direct Known Subclasses:

LogStream

```
public class PrintStream
extends FilterOutputStream
implements Appendable, Closeable
```



void	**println**()	Terminates the current line by writing the line separator string.
void	**println**(boolean x)	Prints a boolean and then terminates the line.
void	**println**(char x)	Prints a character and then terminates the line.
void	**println**(char[] x)	Prints an array of characters and then terminates the line.
void	**println**(double x)	Prints a double and then terminates the line.
void	**println**(float x)	Prints a float and then terminates the line.
void	**println**(int x)	Prints an integer and then terminates the line.
void	**println**(long x)	Prints a long and then terminates the line.
void	**println**(**Object** x)	Prints an Object and then terminates the line.
void	**println**(**String** x)	Prints a String and then terminates the line.

1.3 Variables and Types

A variable is a named value located in computer memory. Its value may vary, because it can be assigned to repeatedly.

A variable is declared to be one of Java's data types.

The convention is variables in Java are named using `camelCase`.

	default value		
boolean stillPlaying;	false	***Variables may optionally be initialized:***	boolean stillPlaying = false;
char ch;	0		char ch = 'A';
double price;	0		double price = 3.95;
float ratio;	0		float ratio = 2.3f;
int i;	0		int i = 0;
long l;	0		long l = 120000000;
String s;	null		String s = "Hello";

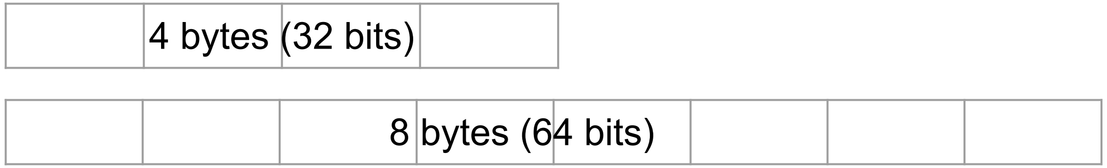
Java's primitive data types

`boolean`	true or false	1 bit	(in practice, often padded to 1 byte or more)
`byte`	integer in range -128..127	1 byte	(8 bits)
`char`	Unicode character	2 bytes (16 bits)	
`short`	integer in range -32768..32767	2 bytes (16 bits)	
`int`	integer in range -2,147,483,648 to 2,147,483,647	4 bytes (32 bits)	
`float`	single-precision floating point number, up to ~6-7 decimal digits	4 bytes (32 bits)	
`long`	integer -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	8 bytes (64 bits)	
`double`	double-precision floating point number, up to ~15 decimal digits	8 bytes (64 bits)	

`float` and `double` also have special values: `Infinity`, `-Infinity`, `NaN` (Not a Number)

Java object reference types

The size of an object reference is up to the Java implementation, but on today's computers, it's usually 4 or 8 bytes. It depends on the CPU, operating system, and JVM being used and its configuration.



Like everything on computers, it's a number in the end.

You never see the actual number though... Java hides it from you, as an abstraction.

You can think of computer memory as a giant array, and as an object reference as an index into that array. (It's a bit more complicated but that's the basic idea.) null is usually represented by 0.

Variable Names

Variable names must start with a letter. The rest of the name may include letters, digits, or _.

No spaces!

The convention is `camelCase` with the first letter in lowercase.

Variable names should be descriptive!

* Keywords are off limits (`for`, `if`, `class`, `double`, `static`, `int`, etc.)

Keyword `final`

The keyword `final` can be used in front of a variable declaration to make it a constant that cannot be changed. Constants are traditionally capitalized.

Syntax:

```
final type NAME = value;
```

EX:

```
final double PI = 3.14;
```

The String Concatenation Operator (+)

- Same symbol as addition
- Used to combine Strings and other data types into a result String

```
System.out.println("Hello, " + " friend!");
Will print out: Hello, friend!

System.out.println("Your score: " + 100);
Will print out: Your score: 100

// Variables can be included in the concatenation
int score = 500;
System.out.println("Your score: " + score);
Will print out: Your score: 500
```

1.4 Expressions

A combination of **constants**, **variables**, **operators** and **method calls** that evaluate to a **value.**

Examples:

- `11 * 3 + 5 - 8 / 4`
- `100`
- `x * 10`
- `radius * Math.sin(angle * Math.PI / 180.0)`

Assignment vs Equals

Assignment

```
x = 0
```

Equality

```
x == 0
```

Arithmetic Operators

Operator	**Meaning**	**Example**
+	addition	3 + x
-	subtraction	p - q
*	multiplication	6 * i
/	division	10 / 4
%	modulo (remainder)	11 % 3

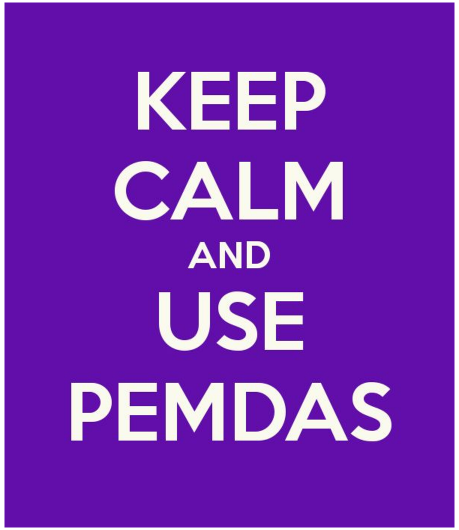
Order of Operations

Java evaluates expressions according to standard mathematical rules of precedence.

PEMDAS

- **P**arentheses
- **E**xponent (ignore this for now)
- **M**ultiply / **D**ivide / Modulus
- **A**ddition / **S**ubtraction

Note that Java evaluates expressions from left to right and from top to bottom.



Division

Dividing two `int` values results in a single `int`
(always truncated: rounded down)

```
    10 / 2 = 5         15 / 2 = 7         19 / 10 = 1
```

Dividing with a double results in a double

```
    15 / 2.0 = 7.5
```

Casting can be used to force double division:

```
int x = 9, y = 2;

double z = (double)x / y;
```

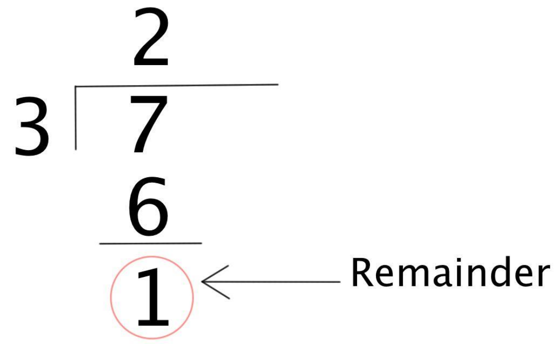
Modulo

The modulo operator (%) gives the remainder to an equivalent division problem. It's used most on ints, but also works on floats/doubles.

7 % 3 == 1

4 % 2 == 0

3 % 5 == 3



Check if a number is even or odd:
  x % 2 == 0 means x is even
  x % 2 == 1 means x is odd

Check if a number is a multiple of another:
  x % y == 0 means x is a multiple of y
  (divisible by y with a remainder of 0)

String Concatenation

Once Java starts working with Strings, it continues to look at thing as strings and interpret “+” operator as string concatenation operator.

Examples

- "Lucky number: " + 4 + 2   == "Lucky number: 42"

- 4 + 2 + " is lucky" == "6 is lucky"

1.5 Compound Assignment Operators

These shortcuts let you do **assignment** and a **math operation** in one step

Example Expression	**Shortcut form!**
`x = x + 3`	`x += 3`
`x = x - y`	`x -= y`
`x = x * 5.0`	`x *= 5.0`
`x = x / 2`	`x /= 2`
x = x % 3	`x %= 3`

Incrementing Compound Operators

Adding 1 and Subtracting 1 to a variable are so commonly used, that Java has special expressions for these operations.

Expression	**Shortcut form**	**Even Shorter Shortcut Form**
`x = x + 1`	`x += 1`	`x++`
`y = y - 1`	`y -= 1`	`y--`

*Note: You can also do ++x or --x. This would adjust the value of x **before** doing something with it. This is not on the AP exam but here's an example:*

```
int x = 6;
int y = ++x;
System.out.println(x) // Outputs 7
System.out.println(y) // Outputs 7
```

```
int x = 6;
int y = x++;
System.out.println(x) // Outputs 7
System.out.println(y) // Outputs 6
```

1.6 User Input

`java.util.Scanner` reads input from a stream. One stream you can read is `System.in`, which is usually connected to the user's keyboard!

```
import java.util.Scanner;
public class Demo {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.println("What is your name?");
    String name = scanner.nextLine();
    System.out.println("Well, hello, " + name + "!");
  }
}
```

If we want to do calculations, we need numeric input. To take numeric values as input from a `Scanner`, we can call different methods:

```
Scanner scan = new Scanner(System.in);
int x = scan.nextInt();
double y = scan.nextDouble();
scan.nextLine(); // Get past the newline
```

The final call to nextLine() is needed because scan.nextInt() and scan.nextDouble() read "tokens" from input but don't parse the newline.

Alternately, you could read strings in with scan.nextLine() and convert them to the desired types, using Integer.parseInt, Double.parseDouble, etc.

1.6 Promotion of values

Java will **promote** values from certain data types to certain other data types when used together.

```
int + double = double

int + String = String

double + String = String
```

1.6 Casting

```
(int) 4.29       ⇒  4
(double) 1       ⇒  1.0
```

Note: `String` *is not a primitive type and cannot be cast. Java will convert* `int` *and* `double` *to strings when concatenated with a string.*

```
      "" + 4.892     ⇒  "4.892"
```

2.1 Classes and Objects

CLASS	OBJECT (aka INSTANCE)
A class is a blueprint from which you can create the instance, i.e., objects.	An object is the instance of the class, which helps programmers to use variables and methods from inside the class.
Classes have logical existence.	Objects have a physical existence.
A class doesn't take any memory spaces when a programmer creates one. (The “idea” of a cat)	An object takes memory when a programmer creates one. (A real, live cat)
The class has to be declared only once. (i.e. “Cat”)	Objects can be declared several times depending on the requirement. (i.e. “Buttons”, “Mr. Bigglesworth”, “Garfield”)

Class

Object

```
String
```

```
String greeting = “Hello world!”;

String favoriteClass = “AP Computer Science”;

String bestTeacher = “Ms. Molina”;
```

Attributes (instance variable) and Behaviors (methods)

An **attribute** or **instance variable** is data the object knows about itself. For example a turtle object knows the direction it is facing or its color.

A **behavior** or **method** is something that an object can do. For example a turtle object can go forward 100 pixels.

2.2 Constructors

```
public class Dog {
    private String breed;
    private int age;
    private String color;

    public Dog() {
        breed = “pug”;
        age = 3;
        color = “brown”;
    }

    public Dog(String a, int b, String c) {
        breed = a;
        age = b;
        color = c;
    }
}
```

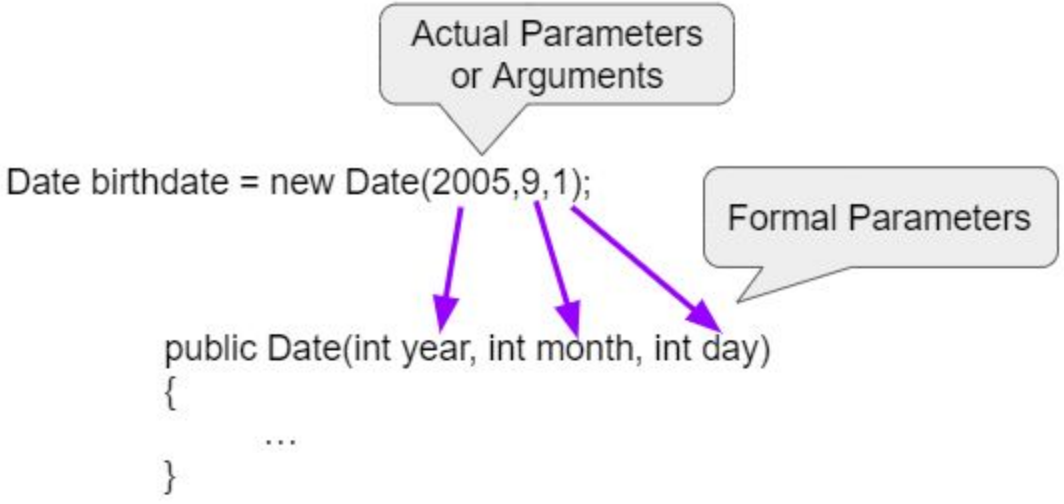
Default constructor

An overloaded constructor that takes parameters

```
World world1 = new World(); // creates a 640x480 world

World world2 = new World(300,400); // creates a 300x400 world


Turtle t1 = new Turtle(world1);

Turtle t2 = new Turtle(50, 100, world1);
```

Notice here that the order of parameters matters

Formal and Actual Parameters



```
Date birthdate = new Date(2005,9,1);

public Date(int year, int month, int day)
{
        ...
}
```

Date(2005,9,1) - This is **call by value** which means that copies of the actual parameter values are passed to the constructor. These values are used to initialize the object’s attributes.

2.3 Methods

A method is an **action** defined for a class that all instances of that class (objects) will support.

Methods can:

1. Provide access to an attribute of an instance
2. Update an attribute of an instance
3. Do something new and interesting with the information stored in an instance

Methods are called using the "." operator, which allows access to the public methods of a class.

Method declarations

Method declarations, such as `public void makeOlder(int years) { … }`

1. **Define whether the method is accessible to the outside world** (public / private)
    a. Public methods are available externally (e.g. `goodBoy.getAge()`) while private methods are not (calling `goodBoy.dogYears()` in `main` will cause an error)
2. **Determine what the method returns**
    a. Void methods return nothing
    b. String methods promise to return Strings, int methods to return ints
3. **Defines the variables (parameters) passed to the method**
    a. To be described in the next section
4. **Define the body of the method**
    a. The body is the statements of code that will execute when the method is called.

Abstraction – keeping things simple

One of the core concepts in computer science is **abstraction.** Abstraction means that you only need to understand how to interact with an object–you **don't need to understand how the code is actually implemented behinds the scenes.**

E.g. as a user, I should be indifferent between the following implementations:

Option 1

```
private int dogYears() {
    return 7*age;
}

public int getAge() {
    return dogYears();
}
```

Option 2

```
public int getAge() {
    return (age + age +
            age + age +
            age + age +
            age);
}
```

Option 3

```
public int getAge() {
    return 7*age;
}
```

The power of abstraction

Abstraction accomplishes two things:

1. It keeps things simple, minimizing what you need to know to write a program
2. It makes it possible for the class owner to change the technical implementation of the method without impacting its use
   a. e.g., option 3 may be faster for a computer to calculate than option 2… the programmer may want to switch their implementation from 2 to 3. Abstraction means that the user won’t notice a difference (besides faster code)

2.4 Method Parameters, Overloaded Methods

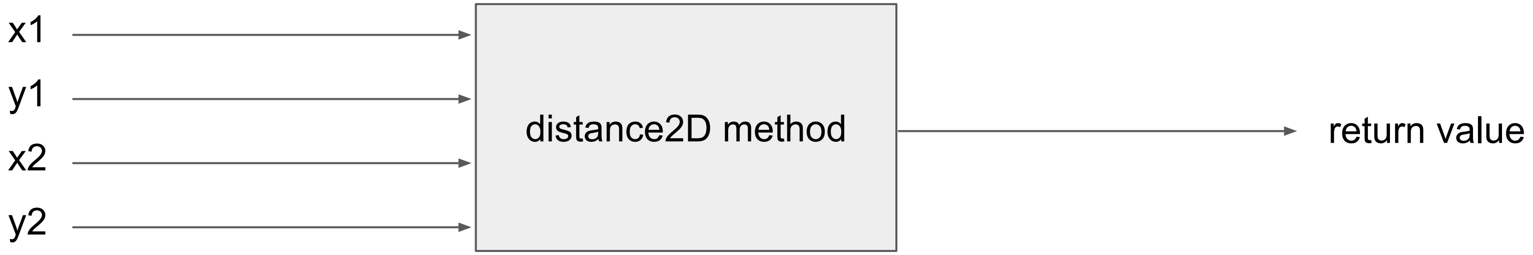
```java
// Person.java
public class Person {
  private String name;

  public Person(String name) {
    this.name = name;
  }

  // Greeting
  public void greet() {
    System.out.println(name + " says: Hello, world!");
  }

  // Greet a particular person
  public void greet(String otherName) {
    System.out.println(name + " says: Hello, " +
                       otherName + "!”);
  }
}
```

```java
// TestPerson.java

public class TestPerson {
  public static void main(String[] args) {
    Person amy = new Person("Amy");

    // Prints "Amy says: Hello, Ted!"
    amy.greet("Ted");

    // Prints "Amy says: Hello, Thursday!"
    amy.greet("Thursday");

    Person bob = new Person("Bob");
    // Prints "Bob says: Hello, Amy!"
    bob.greet("Amy");
  }
}
```

2.5 Return Values

Methods can take **inputs** (“arguments” or “parameters”), and they can also spit out a single **output** (“return value”)

```
public double distance2D(double x1, double y1, double x2, double y2)
```



Methods are like functions… what is the difference between a function and a method?

Return Value

Methods that don’t **return** anything have a **void** return type

```
public void printGreeting(String name) {
    System.out.println("Hello " + name + "!");
}
```

Return Value

- The **type** of the return value must match what is declared in the method declaration
- Right:
```
public int getNumberTimesThree(int value) {
    return 3 * value;
}
```
- Wrong (why?):
```
public int getNumberTimesThree(int value) {
    return 3.0 * value;
}
```
- Q: Java only lets you return one value from a method. How might you return multiple pieces of data at once?

Getter and Setter Methods

In Java, you'll commonly find that classes declare `getXYZ` and `setXYZ` methods for their properties (instance variables).

```
public class TurtleTestGetSet
{
  public static void main(String[] args)
  {
      World world = new World(300,300);
      Turtle yertle = new Turtle(world);
      System.out.println("Yertle's width is: " + yertle.getWidth()); // Yertle's width is: 15
(this is the default width)
      yertle.setWidth(200);
      yertle.setHeight(200);
      System.out.println("Yertle's width is: " + yertle.getWidth()); // Yertle's width is: 200
(this is the width after we've set it to 200 2 lines above
      yertle.turnRight();
      world.show(true);
  }
}
```

This is considered a **best practice**. Q: Why do you think that is?

toString Methods

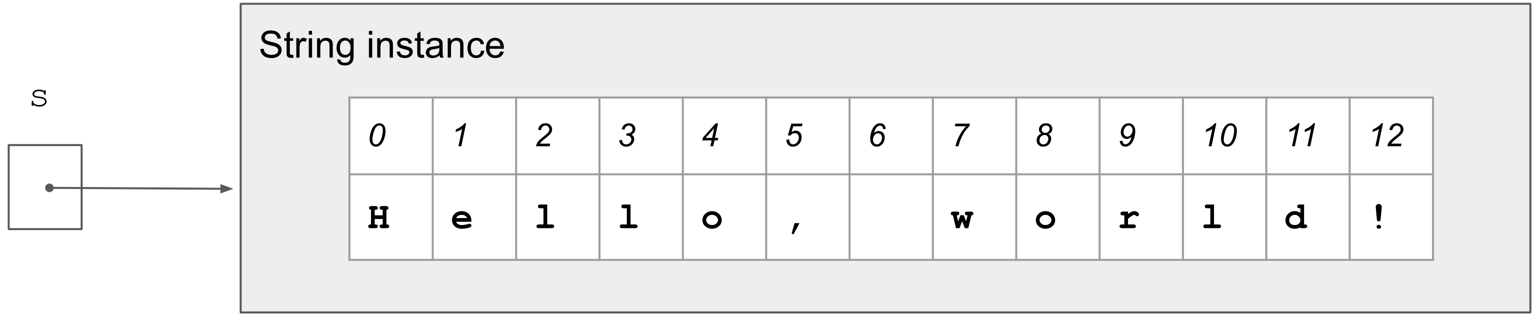
- In Java, all objects can be represented in String form by defining a **toString** method.
- This can be useful for programmers to get a visual or textual representation of an otherwise abstract object.
- How can we make the `toString` method to the right more descriptive?
- What gets printed if you don't define a `toString` method?

```java
class Student {
  private String name;
  private int age;

  Student(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public String toString(){
    return name;
  }
}

class HelloWorld {
    public static void main( String args[] ) {
      Student s = new Student("Jane",16);
      System.out.println(s.toString()); //”Jane”
      System.out.println(s); //Also ”Jane”
    }
}
```

2.6 Strings

Strings in Java are instances of the `java.lang.String` class that hold sequences of characters (`a`, `b`, `c`, `$`, etc.)

```
String s = “Hello, world!”;
```



Creating Strings

`String` is a class, so you can construct them with the `new` operator.

```
String s = new String(“Hello, world!”);
```

Strings can also be constructed using **string literals**:

```
String s = “Hello, world!”;
```

String comparison in Java

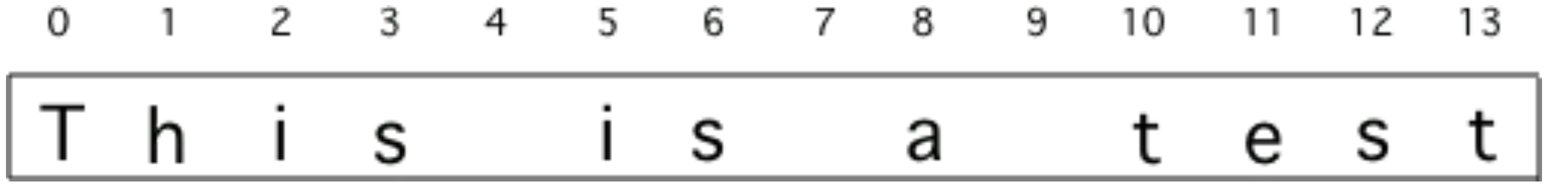
In many other languages, like JavaScript and Python, you can use the `==` operator to compare strings for equality.

In Java, the `==` operator compares object references, **not** what's in the referenced objects!

`s1.equals(s2)` is almost always what you want, not `s1 == s2`

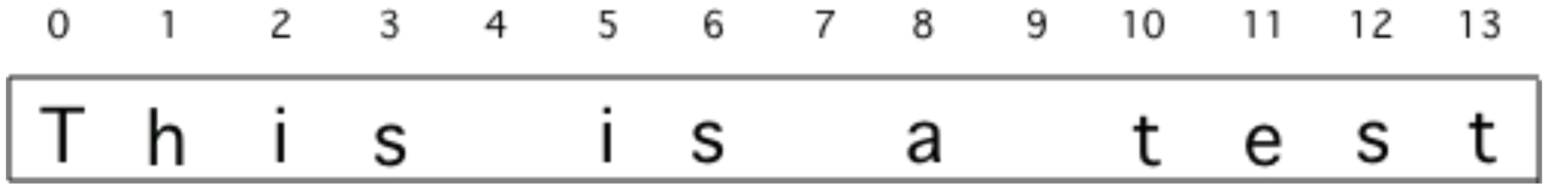
2.7 String Methods

- **`int length()`** method returns the number of characters in the string, including spaces and special characters like punctuation.
- **`String substring(int from, int to)`** method returns a new string with the characters in the current string starting with the character at the from index and ending at the character before the to index (if the `to` index is specified, and if not specified it will contain the rest of the string).
- **Remember: In Java, we always start counting from 0**



String Methods

- **`int indexOf(String str)`** method searches for the string str in the current string and returns the index of the beginning of str in the current string or -1 if it isn't found.
- **`int compareTo(String other)`** returns a negative value if the current string is less than the other string alphabetically, 0 if they have the same characters in the same order, and a positive value if the current string is greater than the other string alphabetically.
- **`boolean equals(String other)`** returns true when the characters in the current string are the same as the ones in the other string. This method is inherited from the Object class, but is **overridden** which means that the String class has its own version of that method.



Mutable vs Immutable

- **Mutable:** CAN CHANGE, **Immutable:** CANNOT CHANGE
- Strings are immutable. Any methods that seems to change a string actually just creates a **copy** of it, and returns the new version as its return value.

```
String str1 = "Hello!";
// Print str1 in lower case? Will str1 change?
str1.toLowerCase();
System.out.println("In lowercase: " + str1);
```

Why are strings immutable?

Immutability is a powerful concept in Computer Science. It can make it easier to reason about what a program does.

When you pass a `String` to a method, since `Strings` are immutable, you know that the method cannot change your `String` behind your back!

`s += ", world!";` is really the same as `s = s + ", world";`

(but is stylistically better)

Many modern programming languages have immutable strings, such as Python and JavaScript. Some chose to have mutable strings, like Ruby.

2.8 Wrapper classes

Wrapper classes are used to store primitive types inside of ordinary Java classes. `Integer` is a Java class, while `int` is not.

`Integer` has a single attribute storing the value of the `int` used to create the instance.

Constructors:

Integer i = new Integer(4);

Double d = new Double(2.718);

Getters:

int j = i.intValue();

double e = d.doubleValue();

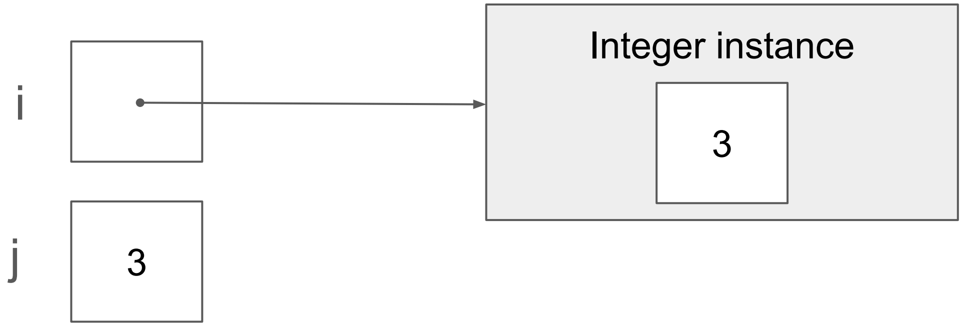
Wrapper classes

A variable whose type is a wrapper class like `Integer`, like any other class, is a reference to an object instance, not an object instance itself.

Using a wrapper class instead of a primitive type means more memory accesses, which is slower, and more memory used.

Wrapper object instances are also called "boxed primitives" – see the box?

```
Integer i = new Integer(4);

int j = 3;
```



Wrapper classes exist for every primitive type

Every primitive type in Java has a corresponding wrapper class.

`boolean`	`java.lang.Boolean`
`byte`	`java.lang.Byte`
`char`	`java.lang.Character`
`double`	`java.lang.Double`
`float`	`java.lang.Float`
`int`	`java.lang.Integer`
`long`	`java.lang.Long`
`short`	`java.lang.Short`

Autoboxing and unboxing

Originally, Java programmers had to explicitly convert between primitive types and the equivalent wrapper objects.

In Java 1.5 (released 9/30/2004), **autoboxing** and **unboxing** were added.

Autoboxing:

Integer i = 4; is the same as Integer i = new Integer(4);

Unboxing:

int j = i; is the same as int j = i.intValue();

Wrapper classes are immutable

The wrapper classes are immutable, like `String`. Once you create an instance of `Integer`, you can't change the `int` inside it.

The wrapper classes have getter methods like `intValue()`, `doubleValue()`, `booleanValue()`, `floatValue()`, but no setter methods.

```
Integer i = 3;
```

`i = 5;` ← This is really creating a whole new `Integer` object instance, and reassigning the value of `i`.

Why even use wrapper classes?

1) They contain useful methods and attributes:
    a) Integer.parseInt(string) converts a string representation of an integer into an int
    b) Integer.MIN_VALUE and Integer.MAX_VALUE store the largest and smallest possible 32-bit integers your computer can store. These lower and upper bounds on all computable integers are very useful in algorithm development
2) Storing primitive types within classes enables us to use Java language constructs that can only be applied to classes…
    a) More on this later with Arrays and Maps

2.9 The `Math` Class

This class implements standard mathematical functions and constants.

Math has only static methods and attributes. It cannot be instantiated with the new operator… it has no public constructor!

```
public class Main {
  public static void main(String args[]) {
    System.out.println(Math.PI);
    System.out.println(Math.sqrt(9));
  }
}
```

```
import static java.lang.Math.*;

public class Main {
  public static void main(String args[]) {
    System.out.println(PI);
    System.out.println(sqrt(9));
  }
}
```

Math methods

`static int abs(int x)`	Returns the absolute value of an `int` value
`static double abs(double x)`	Returns the absolute value of a `double` value
`static double sqrt(double x)`	Returns the square root of a double value.
`static double pow(double base, double exp)`	Returns $\texttt{base}^{\texttt{exp}}$.

Random number

```
static double random()
```

Returns a `double` value greater than or equal to `0.0` and less than `1.`

Example:

```
double randomValue = Math.random();

// Example output: 0.6573016382857277

// Generates a random int between 0 to 9
int random0To9 = (int) (Math.random() * 10);

// Generates a random int between 1 to 10
int random1To10 = (int) (Math.random() * 10) + 1;
```

3.1 Boolean Expressions

Relational Operators

x > y	true if x is greater than y, false otherwise
x < y	true if x is less than y, false otherwise
x >= y	true if x is greater than or equal to y, false otherwise
x <= y	true if x is less than or equal to y, false otherwise
x == y	true if x is equal to y, false otherwise
x != y	true if x is not equal to y, false otherwise

Tip: To remember >= and <=, think of it as the order in which you say it. Greater than (>) or equal to (=)

Relational Operators

x > y	true if x is greater than y, false otherwise
x < y	true if x is less than y, false otherwise
x >= y	true if x is greater than or equal to y, false otherwise
x <= y	true if x is less than or equal to y, false otherwise
x == y	true if x is equal to y, false otherwise
x != y	true if x is not equal to y, false otherwise

These don't work on objects, such as Strings! This is where you use the String.compareTo method.

3.2 if statement

```
if (boolean expression)
    then-statement
```

Example:

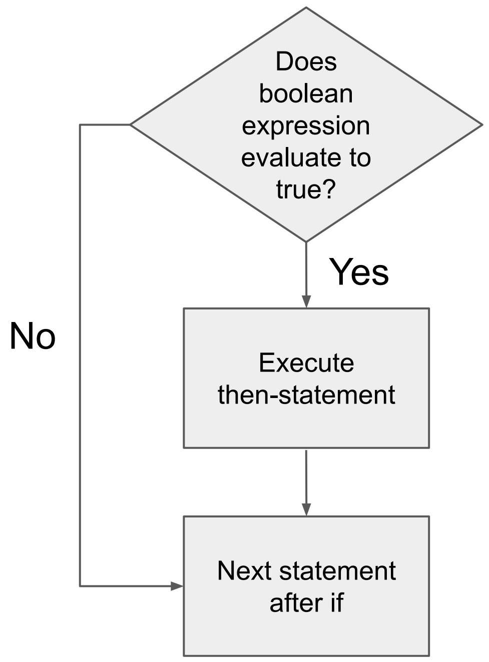
```
if (age >= 18) {
    System.out.println("You are eligible to vote!");
}
```

Legal, but not recommended:

```
if (age >= 18)
    System.out.println("You are eligible to vote!");
```

then-statement can be any statement, and a { block } is a statement.
It's recommended to always use blocks with if.

Flowchart of if



3.3 if-else statement

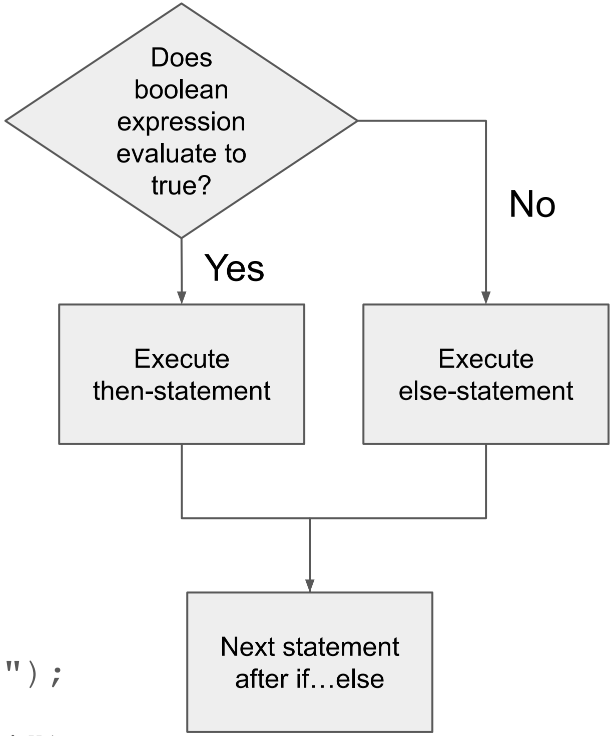
```
if (boolean expression)
    then-statement
else
    else-statement
```

The if statement has an optional else clause.

Example:

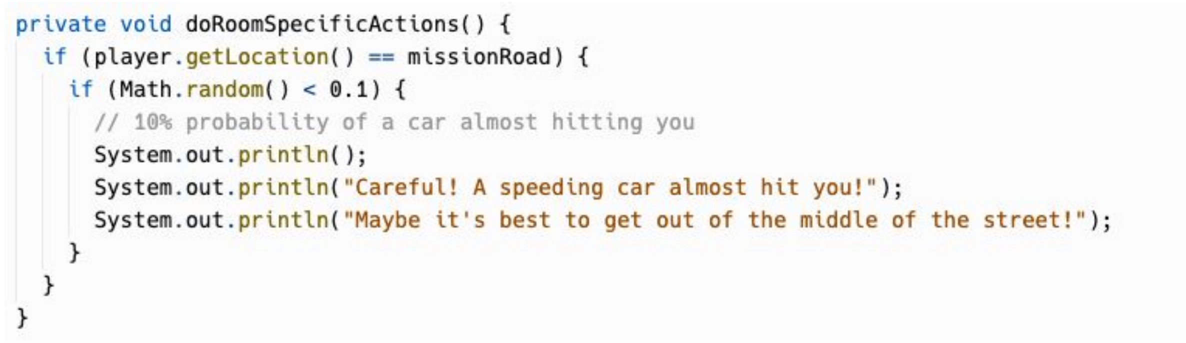
```
if (age >= 18) {
  System.out.println("You are eligible to vote!");
} else {
  System.out.println("You are too young to vote!");
}
```

Flowchart of if…else


Does boolean expression evaluate to true?
Yes
No
Execute then-statement
Execute else-statement
Next statement after if…else

Nested If-Statements

```
if (boolean expression) {
    if (boolean expression) {
        if (boolean expression) {
            <statement>;
            <statement>;
            …
        } else {
            <statement>;
            <statement>;
            …
        }
    }
} else {
    <statement>;
    <statement>;
    …
}
```



```java
private void doRoomSpecificActions() {
  if (player.getLocation() == missionRoad) {
    if (Math.random() < 0.1) {
      // 10% probability of a car almost hitting you
      System.out.println();
      System.out.println("Careful! A speeding car almost hit you!");
      System.out.println("Maybe it's best to get out of the middle of the street!");
    }
  }
}
```

Dangling Else

```
int x = 0;
if (x >= 0)
    if (x > 0) ←―――――――――――――――――――――――――――――― is paired with this if
        System.out.println("x is positive");
else ←――――――――――――――――――――――――――――――――――――――― this else
    System.out.println("x is negative");
```

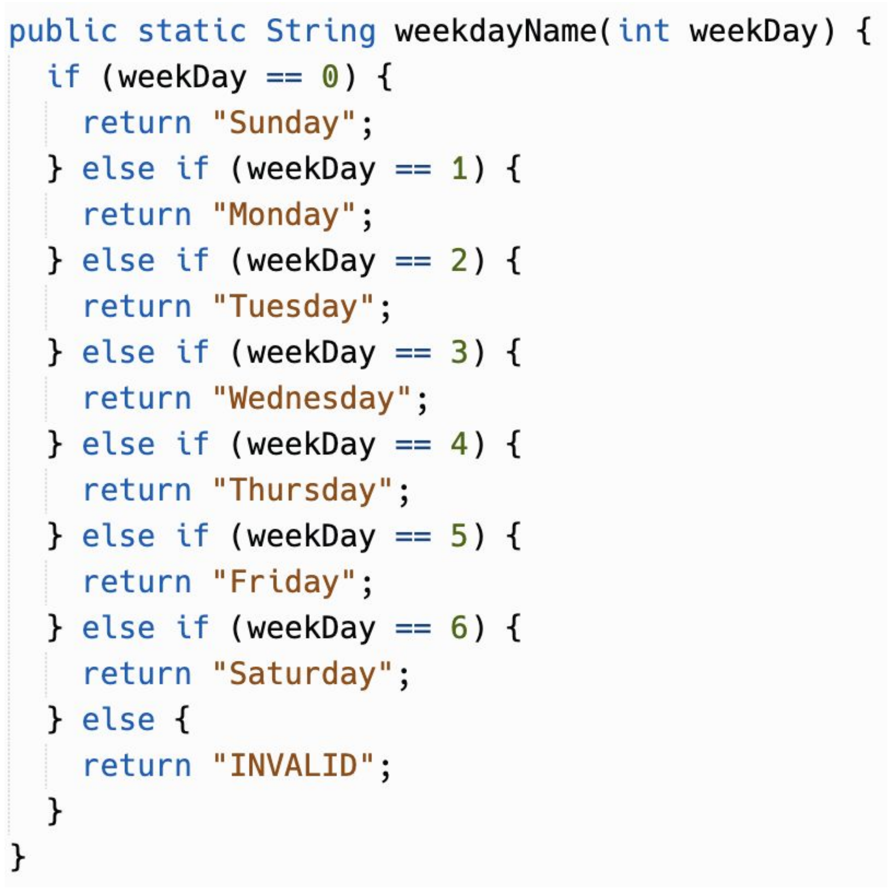
Prints "x is negative"!

The else clause will always be a part of the closest if statement if in the same block of code regardless of indentation…

Unless you use {}!

3.4 Multi-Selection: else-if

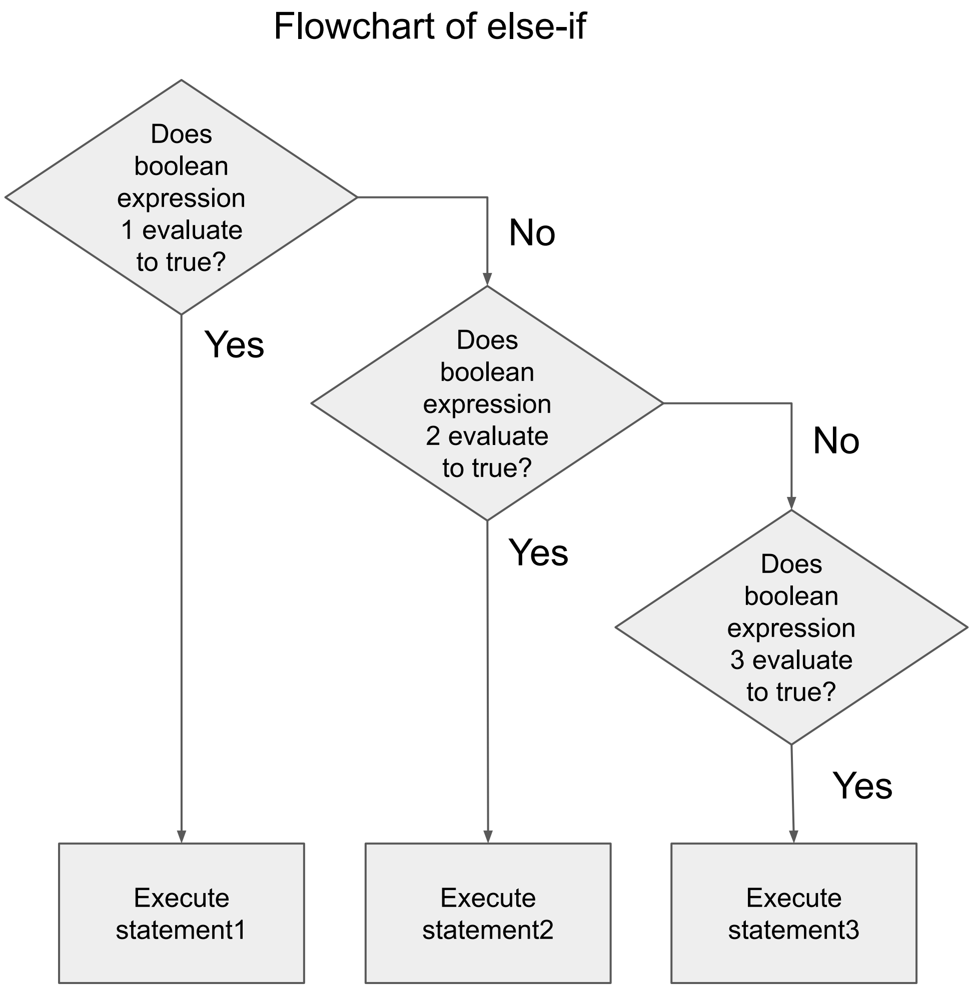
```
if (boolean expression 1)
     statement1
else if (boolean expression 2)
     statement2
else if (boolean expression 3)
     statement3
```



```java
public static String weekdayName(int weekDay) {
  if (weekDay == 0) {
    return "Sunday";
  } else if (weekDay == 1) {
    return "Monday";
  } else if (weekDay == 2) {
    return "Tuesday";
  } else if (weekDay == 3) {
    return "Wednesday";
  } else if (weekDay == 4) {
    return "Thursday";
  } else if (weekDay == 5) {
    return "Friday";
  } else if (weekDay == 6) {
    return "Saturday";
  } else {
    return "INVALID";
  }
}
```

else-if syntax

```
if (boolean expression 1)
    statement1
else if (boolean expression 2)
    statement2
else if (boolean expression 3)
    statement3
```



Flowchart of else-if
Does boolean expression 1 evaluate to true?
No
Yes
Does boolean expression 2 evaluate to true?
No
Yes
Does boolean expression 3 evaluate to true?
Yes
Execute statement1
Execute statement2
Execute statement3

3.5 Compound Boolean Expressions – Logical Operators

Logical And

p `&&` ***q***

Evaluates boolean expressions ***p*** and ***q***.

Evaluates to `true` if ***p*** and ***q*** are both `true`, `false` otherwise.

```
if (sunny && warm) {
     …
}
```

Logical Or

p `||` ***q***

Evaluates boolean expressions ***x*** and ***y***.

Evaluates to `true` if ***p*** or ***q*** are `true`, `false` otherwise.

```
if (christmas || halloween)
{
     …
}
```

Logical Not

`!`***p***

Evaluates boolean expression ***p***.

Evaluates to `true` if ***p*** is `false`.

Evaluates to `false` if ***p*** is `true`.

```
if (!day.equals("Sunday"))
{
     …
}
```

Why p and q? In logic textbooks, the "default" names for logical propositions are p and q.

Truth Table - &&

p	**q**	**p && q**
true	true	true
true	false	false
false	true	false
false	false	false

Truth Table - ||

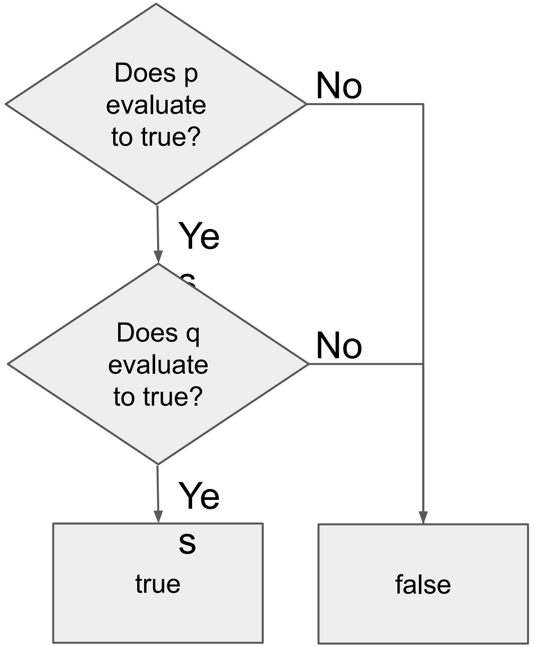
p	**q**	**p || q**
true	true	true
true	false	true
false	true	true
false	false	false

Truth Table - !

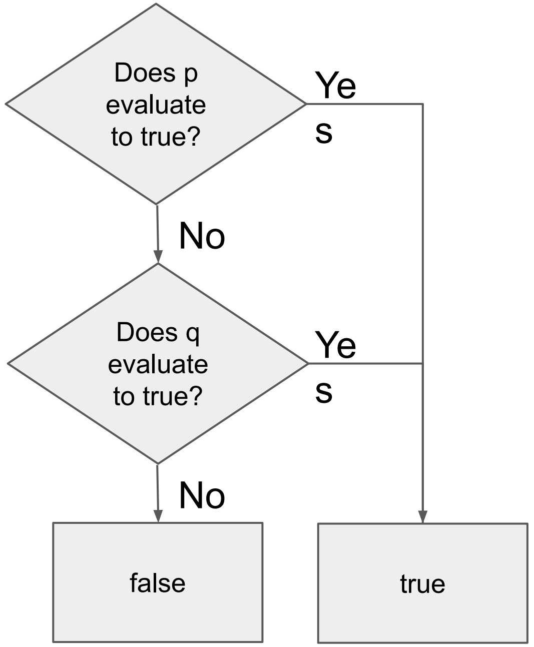
p	**!p**
true	false
false	true

Short-Circuit Evaluation

```
p && q
```



```
p || q
```



3.6 Equivalent Boolean Expressions

DeMorgan's Laws: These rules can simplify boolean expressions to make them easier to read or interpret.

- `!(a && b)` is equivalent to `!a || !b`
- `!(a || b)` is equivalent to `!a && !b`

De Morgan's Laws: Example

`YOU: I'm leaving for school!`

`MOM: Let's make sure you have everything.`

`MOM: Do you have both your phone and your lunch?`

`(is phone && lunch true?)`

`YOU: No. !(phone && lunch) is true`

`MOM: What are you missing, your phone? Your lunch? Both?`

`(!phone || !lunch) is true`

`!(phone && lunch) = !phone || !lunch`

phone	lunch	!(phone && lunch)	!phone || !lunch
F	F	!(F && F) → !F → **T**	!F || !F → T || T → **T**
F	T	!(F && T) → !F → **T**	!F || !T → T || F → **T**
T	F	!(T && F) → !F → **T**	!T || !F → F || T → **T**
T	T	!(T && T) → !T → **F**	!T || !T → F || F → **F**

De Morgan's Laws: Example

`!(a && b) = !a || !b`

You must do pull-ups AND run a mile to complete the fitness test.

```
    if (!(completedPullUps && completedOneMileRun)) {
        System.out.println(“You are not yet done with your
fitness test!");
    }
```

is the same as

```
    if (!completedPullUps || !completedOneMileRun) {
        System.out.println(“You are not yet done with your
fitness test!”);
    }
```

De Morgan's Laws: Truth Tables

`!(a && b) = !a || !b`

`!(a || b) = !a && !b`

`a`	`b`	`!(a && b)`	`!a || !b`
F	F	!(F && F) → !F → **T**	!F || !F → T || T → **T**
F	T	!(F && T) → !F → **T**	!F || !T → T || F → **T**
T	F	!(T && F) → !F → **T**	!T || !F → F || T → **T**
T	T	!(T && T) → !T → **F**	!T || !T → F || F → **F**

`a`	`b`	`!(a || b)`	`!a && !b`
F	F	!(F || F) → !F → **T**	!F && !F → T && T → **T**
F	T	!(F || T) → !T → **F**	!F && !T → T && F → **F**
T	F	!(T || F) → !T → **F**	!T && !F → F && T → **F**
T	T	!(T || T) → !T → **F**	!T && !T → F && F → **F**

Negated Relational Expressions

For negated relational expressions, **you can flip the operator and remove the !**

- !(c == d) is equivalent to (c != d)
- !(c != d) is equivalent to (c == d)
- !(c < d) is equivalent to (c >= d)
- !(c > d) is equivalent to (c <= d)
- !(c <= d) is equivalent to (c > d)
- !(c >= d) is equivalent to (c < d)

Example:
Simplify **!(x > 2 && y < 4)**

Apply !(a && b) == !a || !b
!(x > 2) || !(y < 4)

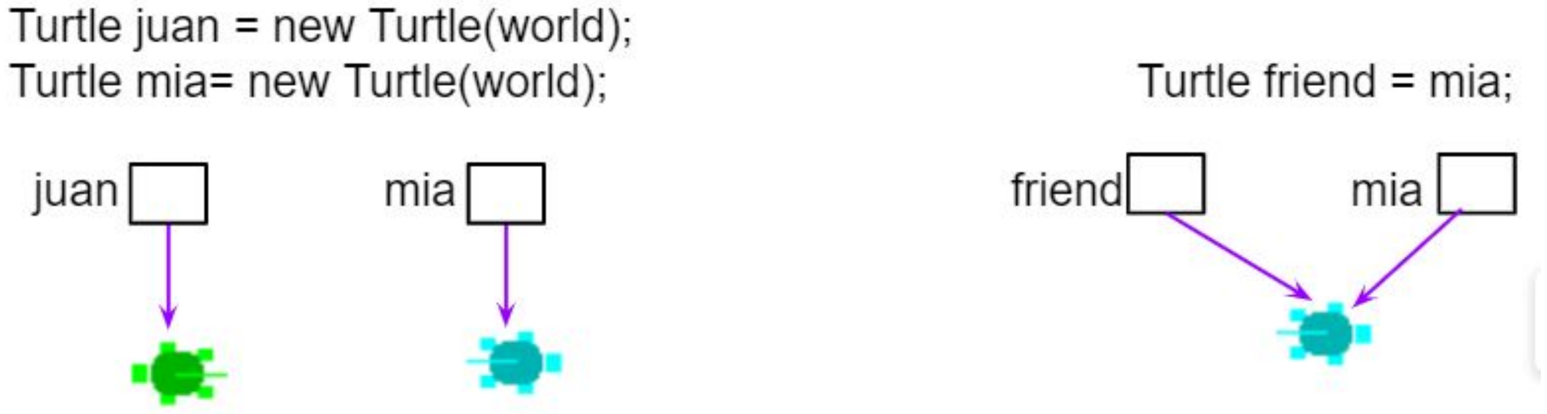
Apply !(c > d) == (c <= d)
(x <= 2) || !(y < 4)

Apply !(c < d) == (c >= d)
x <= 2 || y >= 4

3.7 Object Equality

For object types, == and != compare whether the two sides are references to the same object… not whether anything else about the objects are equal, such as the characters in two Strings.

For Strings, remember to use equals() and not the == or != operators.



```
Turtle juan = new Turtle(world);
Turtle mia= new Turtle(world);
```

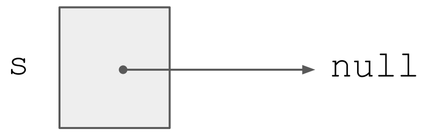
```
Turtle friend = mia;
```

null means nothing is pointed to

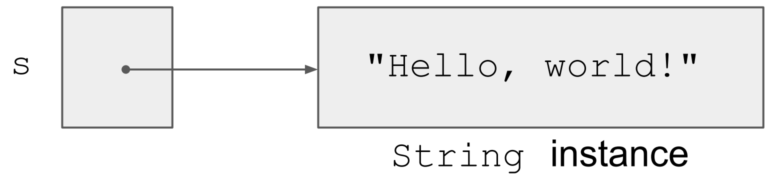
`String s;` is a reference variable of type `String`.

A `String` reference, like all object references, either points to a `String` object instance, or is `null`

```
String s;
```



`s = "Hello, world!";` ← `s` started out `null`, but now points to a legit string



NullPointerException

A variable `Dog dog;` **points to** an instance of `class Dog`.

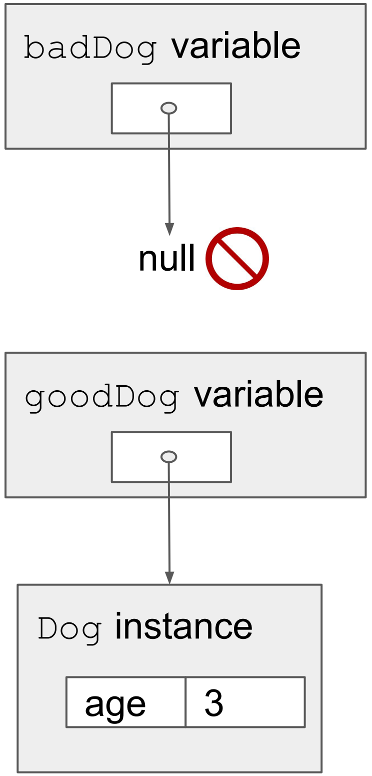
It starts out not pointing to any `Dog`, with the special value `null`.

You have to use `new Dog` to construct a Dog instance that the variable can point to.

If you don’t **initialize** a variable to point to a `Dog` instance, and you try to call a method, `NullPointerException` will be thrown.

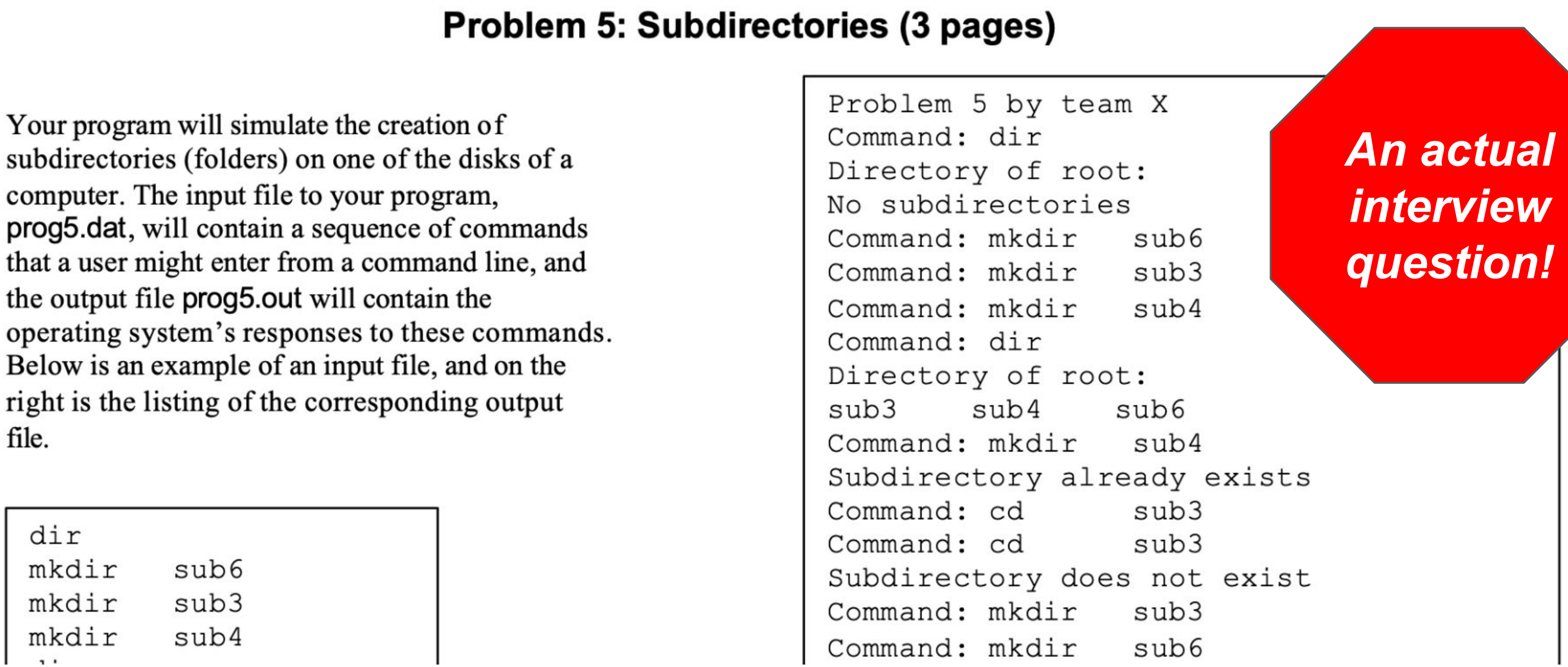
```
Dog badDog; // badDog == null
badDog.getAge(); // throws NullPointerException
```

```
Dog goodDog = new Dog(3); // goodDog points to instance
goodDog.getAge(); // no problem
```



Final Review Exercise
File System Simulator

Review Exercise: File System Simulator



Problem 5: Subdirectories (3 pages)

Your program will simulate the creation of subdirectories (folders) on one of the disks of a computer. The input file to your program, prog5.dat, will contain a sequence of commands that a user might enter from a command line, and the output file prog5.out will contain the operating system's responses to these commands. Below is an example of an input file, and on the right is the listing of the corresponding output file.

```
dir
mkdir    sub6
mkdir    sub3
mkdir    sub4
```

```
Problem 5 by team X
Command: dir
Directory of root:
No subdirectories
Command: mkdir    sub6
Command: mkdir    sub3
Command: mkdir    sub4
Command: dir
Directory of root:
sub3    sub4    sub6
Command: mkdir    sub4
Subdirectory already exists
Command: cd       sub3
Command: cd       sub3
Subdirectory does not exist
Command: mkdir    sub3
Command: mkdir    sub6
```

An actual interview question!

Review Exercise: File System Simulator

Directory	parentDirectory	directoryName	directories	files

File	parentDirectory	fileName	fileSize

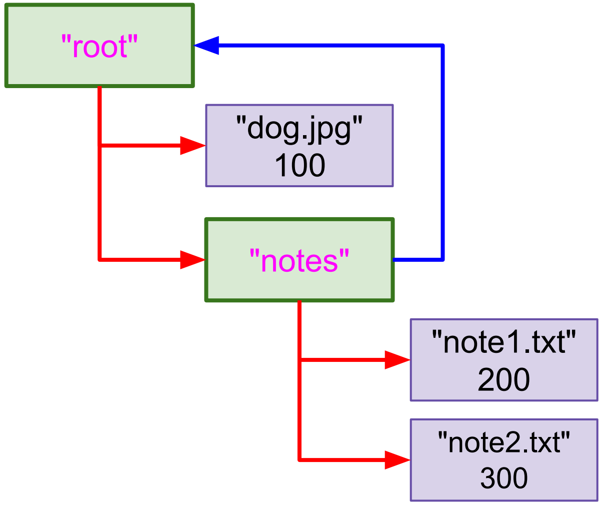
Review Exercise: File System Simulator

Directory	parentDirectory	directoryName	directories	files

File	parentDirectory	fileName	fileSize

Directories

- Have a name
- Have a list of children Files
- Have a list of children Directories
- Know their parent Directory



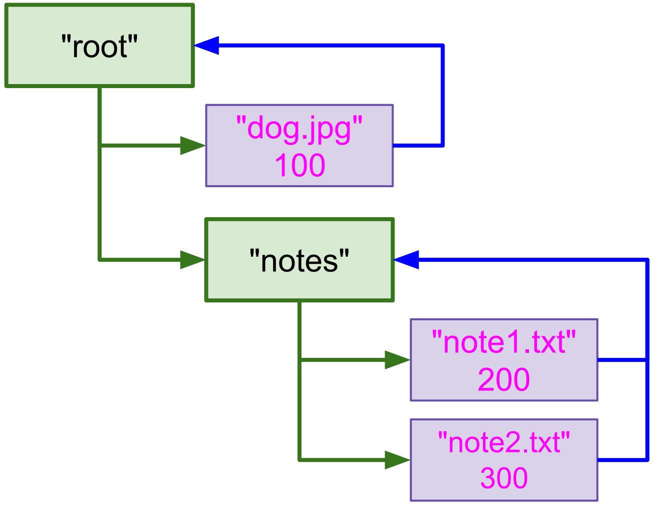
Review Exercise: File System Simulator

Directory	parentDirectory	directoryName	directories	files

File	parentDirectory	fileName	fileSize

Files

- Have a name
- Know their parent Directory



Review Exercise: File System Simulator

Command	Operation

`dir`	Print the contents of the current directory
	
	Example
	
	`Contents of /root/d2/`
	`<d4>`
	`<d5>`
	`t7.txt        800 bytes`
	`t8.txt        900 bytes`
	`t9.txt        1000 bytes`

`mkdir <directory name>`	Create a new directory with the name `<directory name>`; Fail if a directory named `<directory name>` already exists in the current directory

Review Exercise: File System Simulator

Command	Operation
`mkfile <file name> <file size>`	Create a new file with the name `<file name>` and size `<file size>`; Fail if a file named `<file name>` already exists in the current directory
`cd <directory name>`	Change the current directory to `<directory name>`; Fail if a directory named `<directory name>` does not exist in the current directory
`up`	Change the current directory to the parent of the current directory; Fail if the current directory is already the topmost directory
`verify`	Run tests
`quit`	Quit

Review Exercise: File System Simulator

Replit: **File System Simulation**

What other operations can you add?

Final Review - Part 2
12/9/2022

Final Review - Part 2

- Unit 4
- File System Simulator Discussion / Demos

4.1

The while loop

Iteration

Iteration, in the context of computer programming, is a process wherein a set of instructions are repeated a specified number of times or until a condition is met.

Each time the set of instructions is executed is called an **iteration**.

Another term for iteration is **loop**… the program "loops back" to an earlier step and repeats.

while syntax

`while` (*boolean expression*)

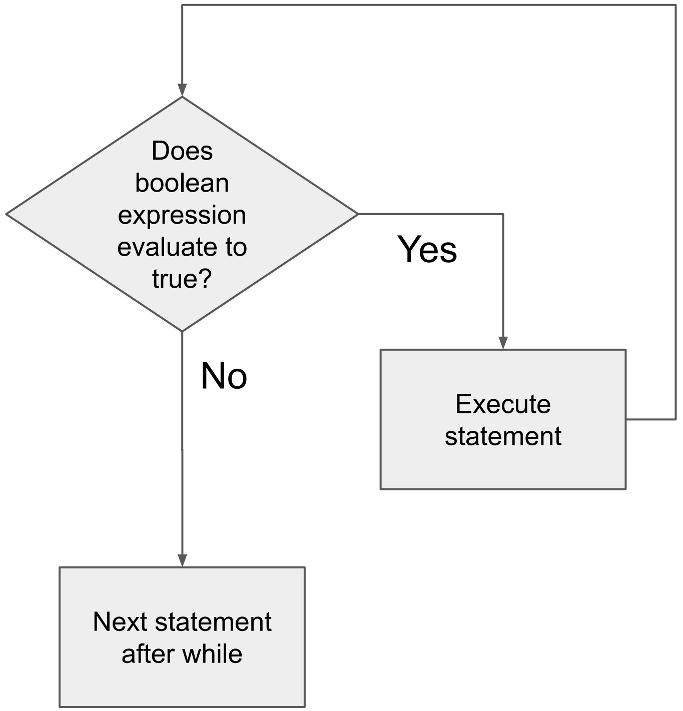
statement

Example:

```
int i = 1;
while (i <= 100) {
     System.out.println(i);
     i++;
}
```

It's like an if statement that keeps repeating itself.
Like if, we recommend curly braces always.
(But Java does not require it.)

Flowchart of while


Does boolean expression evaluate to true?
Yes
No
Execute statement
Next statement after while

++ and -- operators

```
int i = 1;
while (i <= 100) {
     System.out.println(i);
     i++;
}
```

```
int i = 1;
while (i <= 100) {
     System.out.println(i++);
}
```

```
void printManyTimes(String s, int count) {
  while (count > 0) {
    System.out.println(s);
    count--;
  }
}
```

```
void printManyTimes(String s, int count) {
  while (--count >= 0) {
     System.out.println(text);
  }
}
```

`i++` means post-increment, so it evaluates to the current value of `i`, then increments it. The increment/decrement is a **side effect**.
Opinions differ on style here.

= (assignment) can be used in the condition

Your loop condition may depend on some code that repeats every iteration:

```
String command = getNextCommand();
while (command != null) {
    executeCommand(command);
    command = getNextCommand();
  }
}
```

The assignment operator can be used in the condition to avoid the repetition:

```
String command;
while ((command = getNextCommand()) != null) {
    executeCommand(command);
  }
}
```

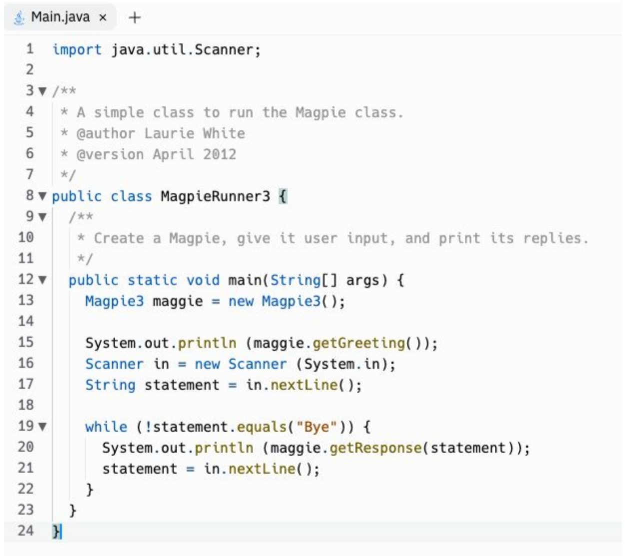
Programmers may differ on the style here. (The first style was used in Magpie.)

Sentinel values

A sentinel value is a special value that tells the loop to terminate.
The code from the previous slide is an example of this:
getNextCommand returns null when there are no more commands to execute.

```
String command;
while ((command = getNextCommand()) != null) {
    executeCommand(command);
  }
}
```

Input-controlled loops

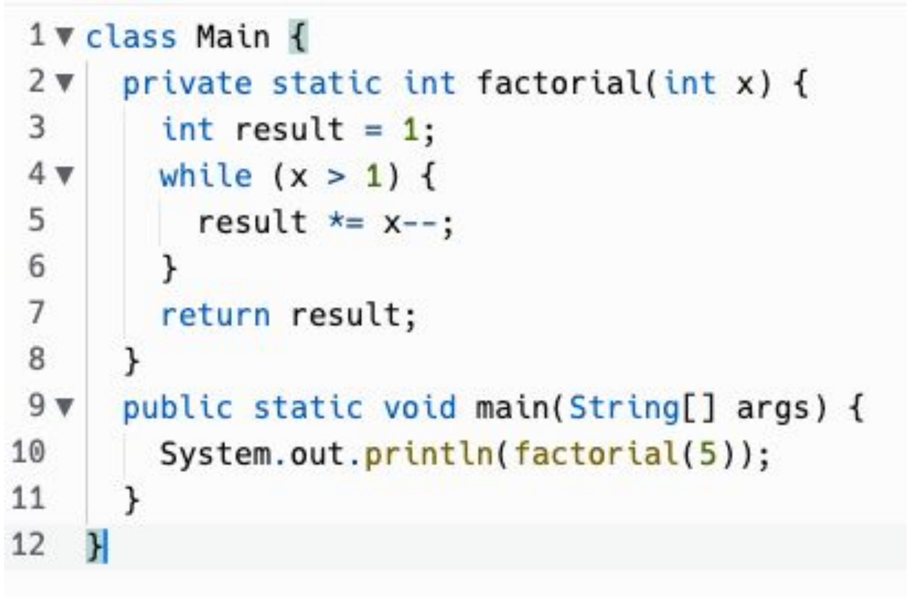

Main.java

```java
import java.util.Scanner;

/**
 * A simple class to run the Magpie class.
 * @author Laurie White
 * @version April 2012
 */
public class MagpieRunner3 {
  /**
   * Create a Magpie, give it user input, and print its replies.
   */
  public static void main(String[] args) {
    Magpie3 maggie = new Magpie3();

    System.out.println (maggie.getGreeting());
    Scanner in = new Scanner (System.in);
    String statement = in.nextLine();

    while (!statement.equals("Bye")) {
      System.out.println (maggie.getResponse(statement));
      statement = in.nextLine();
    }
  }
}
```

The while loop is often used for **input-controlled loops**, where **input** is being taken from the user, or from a file on disk, or the network.

An example is the MagpieRunner, where the while loop continues until the user enters "Bye", the sentinel value which tells Magpie that no more input is coming and terminates the loop.

Tracing loops



```java
class Main {
  private static int factorial(int x) {
    int result = 1;
    while (x > 1) {
      result *= x--;
    }
    return result;
  }
  public static void main(String[] args) {
    System.out.println(factorial(5));
  }
}
```

x	**result**
5	1
4	5
3	20
2	60
1	120

do…while syntax

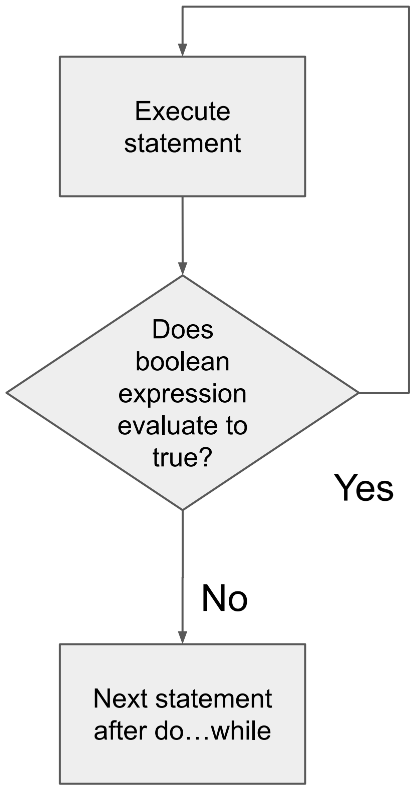
```
do
      statement
while  (boolean expression)
```

Sometimes, you want to check some condition AFTER the body of the loop has run, not before.

Example:

```
String name;
do {
  System.out.println("Enter your name.");
  name = scanner.nextLine();
} while (name.length() == 0);
```

Flowchart of while


Execute statement
Does boolean expression evaluate to true?
Yes
No
Next statement after do…while

Infinite loops

```
void serveRequestsForever() {
  while (true) {
    handleNextRequest();
  }
}
```

This may seem strange, but it has its place.
Sometimes the loop isn't really infinite, but the termination condition of the loop is complicated.
There are ways to break out of a loop, even an infinite one (break, return).

There may also be **unintentional** infinite loops in your code that you need to fix!

4.2

The for loop

Counter-controlled loops

We looked at input controlled loops, which are often done using while.

For statements are often used to do **counter-controlled loops**, where the loop is repeated a specific number of times, and a numeric counter is used to track which iteration the loop is on.

However, really, any of the loop statements in Java can be used to write any possible program. Which loop to use is a matter of what you think best expresses the intent of the program.

for syntax

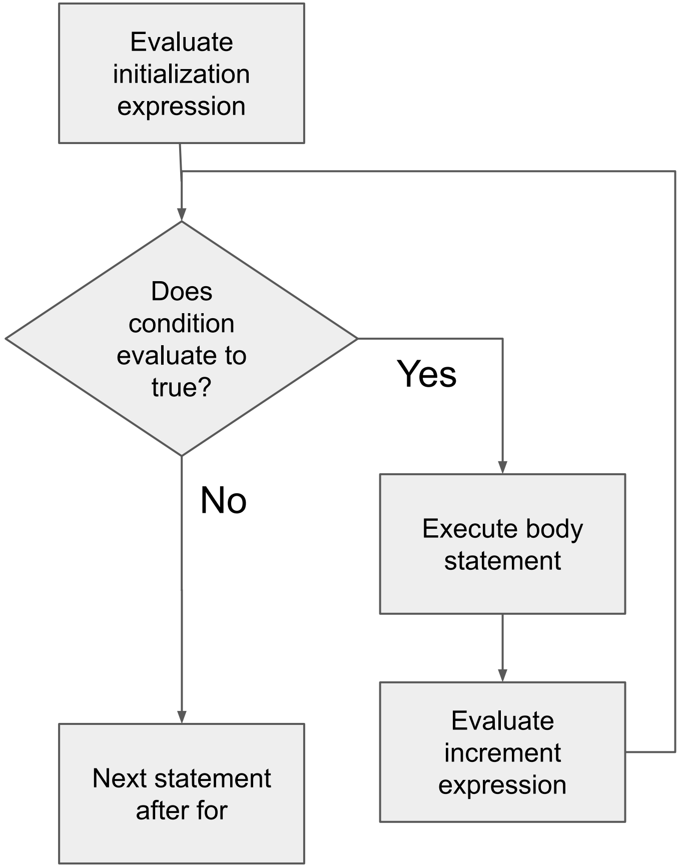
`for` (*initialization; condition; increment*)

statement

Example:

```
int i;
for (i = 1; i <= 100; i++) {
    System.out.println(i);
}
```

Flowchart of for

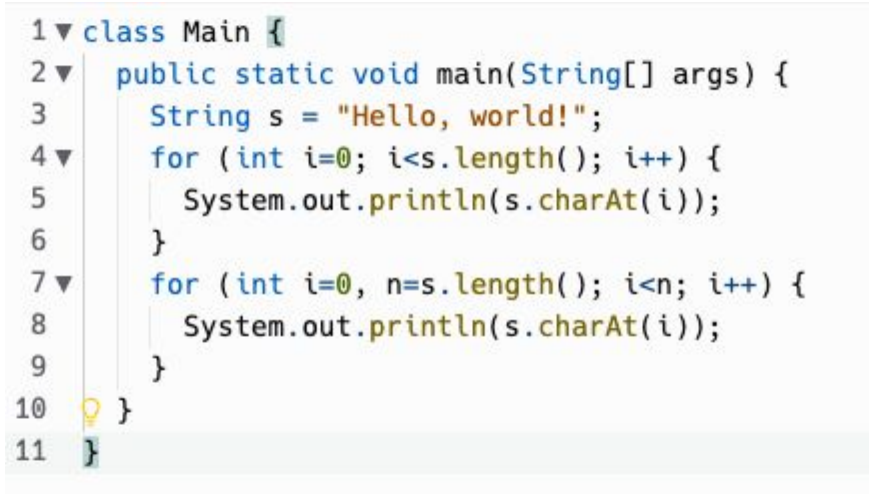

Evaluate initialization expression
Does condition evaluate to true?
Yes
No
Execute body statement
Evaluate increment expression
Next statement after for

for syntax

`for` (*initialization; condition; increment*)

statement

initialization may also declare variables, even multiple variables (but only of the same type)
The scope of any variable declarations is purely the for loop itself.

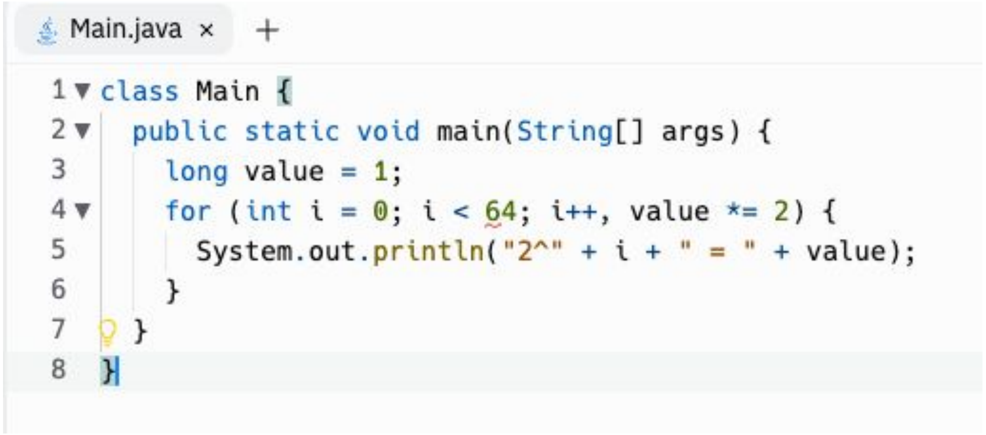


```
class Main {
  public static void main(String[] args) {
    String s = "Hello, world!";
    for (int i=0; i<s.length(); i++) {
      System.out.println(s.charAt(i));
    }
    for (int i=0, n=s.length(); i<n; i++) {
      System.out.println(s.charAt(i));
    }
  }
}
```

for syntax

The *increment* expression can use "," to update multiple variables.

And you're not limited to ++, --, although they are the most common things to see there.



Main.java

```
class Main {
  public static void main(String[] args) {
    long value = 1;
    for (int i = 0; i < 64; i++, value *= 2) {
      System.out.println("2^" + i + " = " + value);
    }
  }
}
```

for syntax

```
for (initialization; condition; increment)
    statement
```

initialization, condition, and ***increment*** are all optional

```
for (;;) { … }
```

is an infinite loop, the same as while (true) { … }

Why would you omit initialization?
Sometimes the initialization needed takes multiple statements and can't be easily expressed in a single expression, so you do it before the for loop and omit the initialization.

Why would you omit increment?
Increment logic may similarly get complicated and be better expressed within the body of the loop.

for syntax

`for` (*initialization; condition; increment*)

    statement

for statements are very frequently used with numeric counters, usually integers.
But they don't have to be… the expressions can be most anything.

```
// Cast spell to magically look east as far as possible.
for (Room room = player.getLocation();
     room != null;
     room = room.getEast()) {
  printRoomContents(room);
}
```

`for` is very flexible in this way.
Some languages like BASIC have a FOR statement that only can initialize and increment a numeric counter.

Getting out of a loop, or the current loop iteration

return

You can exit a while or for loop by returning out of the enclosing method.

One student wrote this for the TruthGame pickNext method:

```
public int pickNext() {
  for (int i=1; i<=3; i++) {
    if (isTruth(i)) {
      return i;
    }
  }
  return -1;
}
```

break

break exits a while or for loop and just moves on to the next statement after the loop.

```
while (true) {
  String command = scanner.nextLine();
  if (command.equals("quit")) {
    break;
  }
  …
}
System.out.println("Well, goodbye, then!");
```

continue

continue jumps back to the top of the loop and re-evaluates the condition. It's useful for skipping the body of the loop, like to ignore blank lines.

```
while (scanner.hasNextLine()) {
  String line = scanner.nextLine();
  if (line.equals("")) {
    continue;
  }
  processNonEmptyLine(line);
}
```

4.3

Looping over Strings

10/14/2022

Refresher: String Methods!

`int length()`	Returns the number of characters in a String object.	`str.length()`

`int indexOf(String str)`
 	
`int indexOf(String str, int fromIndex)`
	Returns the index of the first occurrence of `str` [starting at `fromIndex`, if provided].
	
	Returns -1 if not found.
		`str.indexOf("ing")`
		`str.indexOf("ch", 9)`

`String substring(int from, int to)`
 	
`String substring(int from)`
	Returns substring beginning at index `from` and ending at (`to - 1`) [or `length()-1`, if `to` isn't provided].	`str.substring(7, 10)`
		`str.substring(3)`
		`str.substring(i, i+1)`

`char charAt(int index)`[*]	Returns the character in the string at `index`.	`str.charAt(2)`

* *Note that charAt is not part of AP Computer Science A Java Subset*

String Transformations Using Loops

Many loops over strings are to **transform** a string into another string, e.g., remove spaces, reverse it. This can be approached in multiple ways. Here are two approaches:

Approach #1: Transform the same String variable repeatedly until you achieve the desired result.

```
s = "Let us remove all spaces"
s ← "Letus remove all spaces"
s ← "Letusremove all spaces"
s ← "Letusremoveall spaces"
s ← "Letusremoveallspaces"
```

(Remember, Java Strings are immutable, meaning they cannot be modified. When `s` changes, you aren't modifying the same String instance… you are repeatedly changing what the String reference variable `s` points to.)

Approach #2: Loop over the source String, leaving it unchanged, to build up a new result String.

```
s = "Let us remove all spaces"
result ← "Let"
result ← "Letus"
result ← "Letusremove"
result ← "Letusremoveall"
result ← "Letusremoveallspaces"
```

Neither approach is always better. It depends on the problem you're solving. You may need to **benchmark** the code both ways to find what works better.

String loops with while

Example: Removes spaces from a String

```
public static String removeSpaces(String s) {
  int i = s.indexOf(" ");

  // while there is a " " in the string
  while (i >= 0) {
    // Remove the " " at index by concatenating
    // substring up to index and then rest of the string.
    s = s.substring(0, i) + s.substring(i+1);
    i = s.indexOf(" ");
  }

  return s;
}
```

When to use `while` vs `for` with strings?

Looking for a certain character or substring?

Don’t know how many times the loop needs to run?

↓

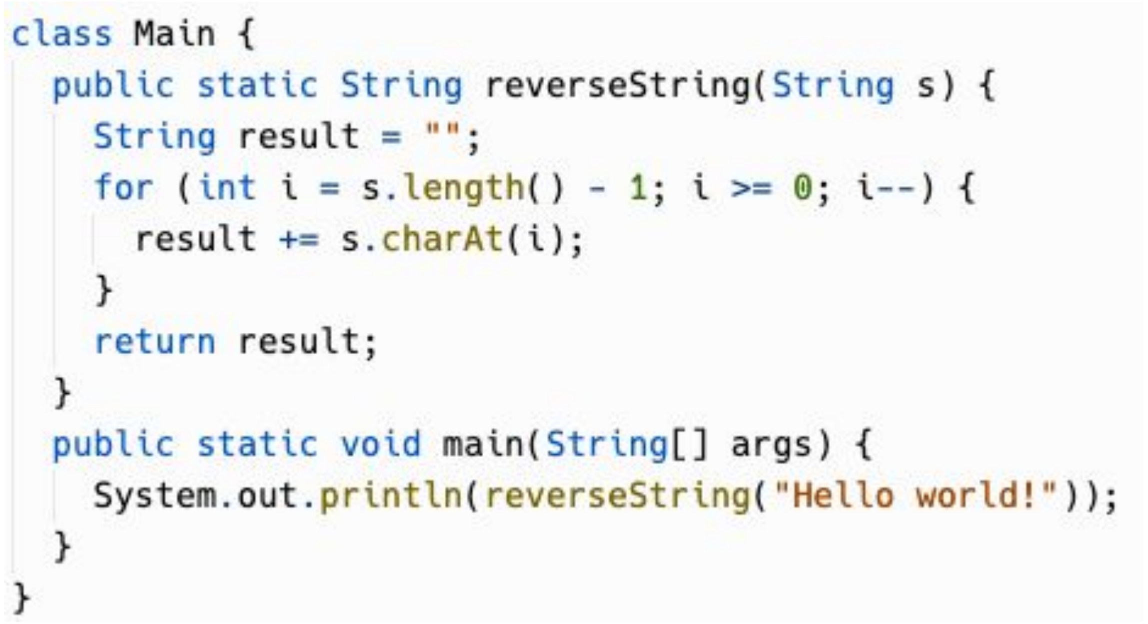
`while`

Want to visit every character (e.g. reversing a string, checking if palindrome)?

↓

`for`

Reverse String Using `for` Loops

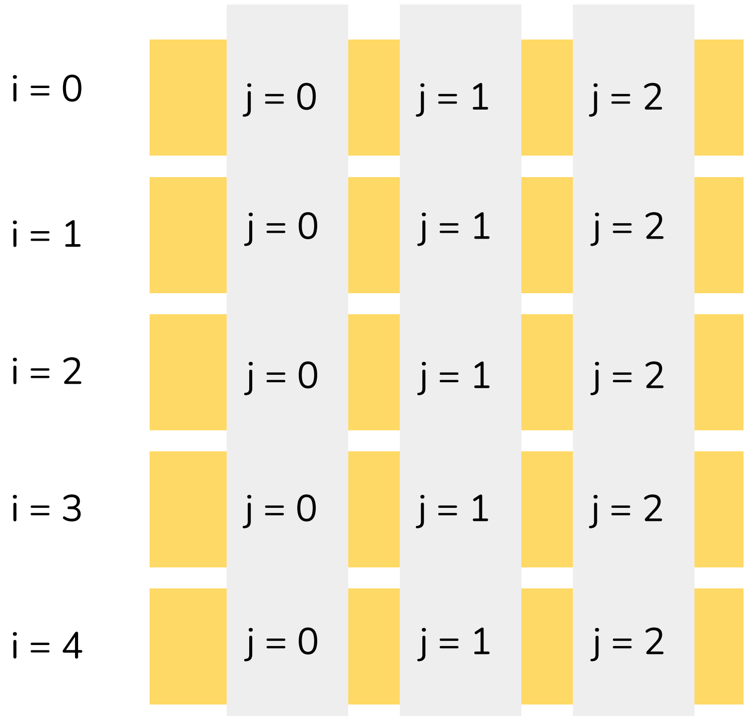


```
class Main {
  public static String reverseString(String s) {
    String result = "";
    for (int i = s.length() - 1; i >= 0; i--) {
      result += s.charAt(i);
    }
    return result;
  }
  public static void main(String[] args) {
    System.out.println(reverseString("Hello world!"));
  }
}
```

Nested Loops

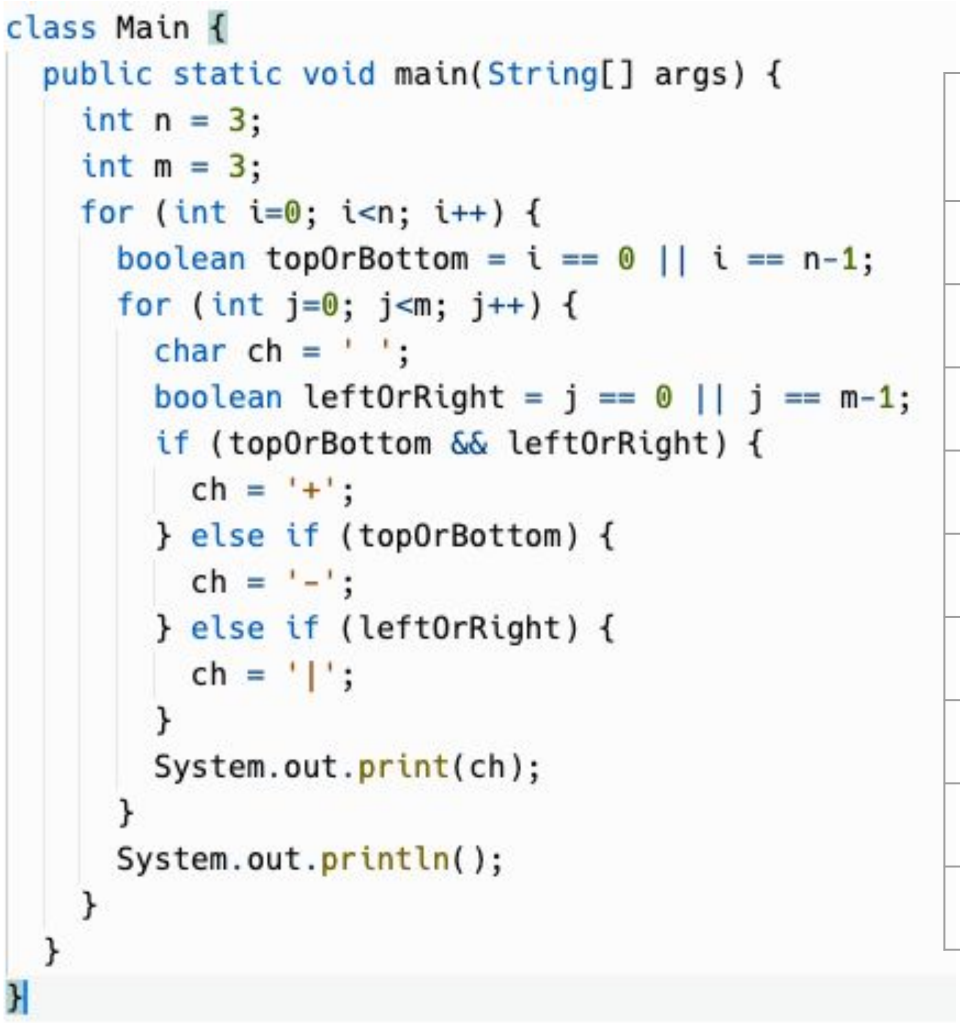
Chapter 4.4

Rows and Columns



```
for (int i = 0; i < 5; i++) {
  for(int j = 0; j < 3; j++) {
    System.out.print(j);
  }
  System.out.println();
}
```

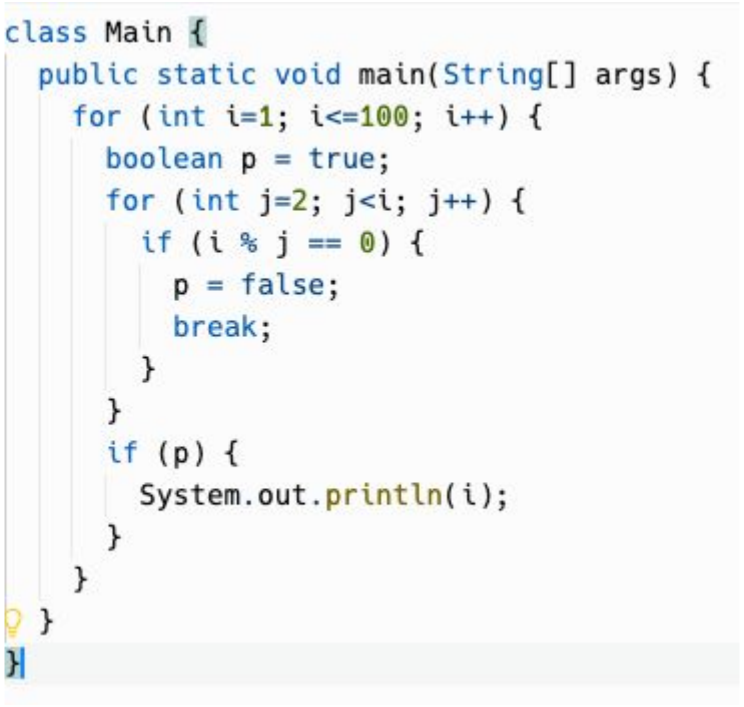
- The **outer loop** iterates through the <u>rows</u>
- The **inner loop** iterates through the <u>columns</u>
- The inner loop runs in its entirety on each iteration of the outer loop



```java
class Main {
  public static void main(String[] args) {
    int n = 3;
    int m = 3;
    for (int i=0; i<n; i++) {
      boolean topOrBottom = i == 0 || i == n-1;
      for (int j=0; j<m; j++) {
        char ch = ' ';
        boolean leftOrRight = j == 0 || j == m-1;
        if (topOrBottom && leftOrRight) {
          ch = '+';
        } else if (topOrBottom) {
          ch = '-';
        } else if (leftOrRight) {
          ch = '|';
        }
        System.out.print(ch);
      }
      System.out.println();
    }
  }
}
```

i	j	topOrBottom	leftOrRight	output
0	0	true	true	+
0	1	true	false	-
0	2	true	true	+
1	0	false	true	|
1	1	false	false	(space)
1	2	false	true	|
2	0	true	true	+
2	1	true	false	-
2	2	true	true	+

Tracing through primes



```
class Main {
  public static void main(String[] args) {
    for (int i=1; i<=100; i++) {
      boolean p = true;
      for (int j=2; j<i; j++) {
        if (i % j == 0) {
          p = false;
          break;
        }
      }
      if (p) {
        System.out.println(i);
      }
    }
  }
}
```

i	j	p (prime)	output
1	2	true	1
2	2	true	2
3	2	true	
3	3	true	3
4	2	false	
5	2	true	
5	3	true	
5	4	true	
5	5	true	5

Runtime Analysis

Chapter 4.5

What makes a good algorithm?

From CodeHS:

What makes a good algorithm?

- Correctness
- Easy to understand by someone else
- Efficiency (run time and memory requirements)

Correctness refers to whether the algorithm solves the given problem.

Easy to understand can be a function of the complexity of the algorithm design, as well as how the code is laid out, use of appropriate variable names, and the use of comments.

Efficiency can be looked at in several ways, which we will explore in this section.

Runtime analysis

How long an algorithm takes to execute (run) is called its **running time** or **runtime**.

Runtime analysis is the process of understanding how an algorithm or complete program will perform when it is run.

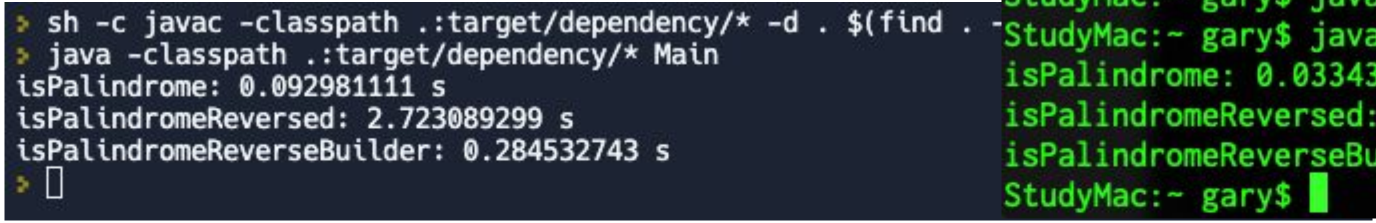
It includes how long the program takes to run, as well as other factors like how much memory is consumed.

Timing Execution

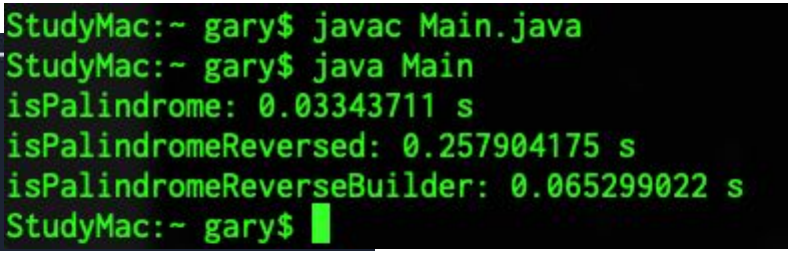
`System.currentTimeMillis()` – milliseconds since "Unix Epoch" (January 1, 1970 00:00:00 UTC time zone). This is **wall clock time**… if you adjust the date/time on your computer, the value will change.

`System.nanoTime()` – nanoseconds since an arbitrary point in time (maybe since CPU booted). Independent of "wall clock" … it will continue increasing even if you futz with computer's date and time.

- Different computers will give you different results, so timing benchmarks are only valid on the same hardware.
- A lot of other stuff can be happening in other processes on a modern computer.
- Different programming languages have different performance characteristics.
- Even the same programming language may behave differently in different environments.



```
> sh -c javac -classpath .:target/dependency/* -d . $(find . -
> java -classpath .:target/dependency/* Main
isPalindrome: 0.092981111 s
isPalindromeReversed: 2.723089299 s
isPalindromeReverseBuilder: 0.284532743 s
> 
```

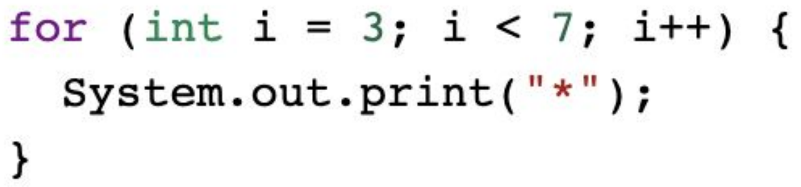


```
StudyMac:~ gary$ javac Main.java
StudyMac:~ gary$ java Main
isPalindrome: 0.03343711 s
isPalindromeReversed: 0.257904175 s
isPalindromeReverseBuilder: 0.065299022 s
StudyMac:~ gary$ 
```

Statement Execution Count

Getting an absolute number of statements executed is tricky, and some of the things we may want to count are actually just expressions, not full statements.

Here, one way we might do it is to "point" it as follows:



```
for (int i = 3; i < 7; i++) {
  System.out.print("*");
}
```

int i = 3; counts as 1 point

This part repeats for i=3, 4, 5, 6:
i < 7 counts as 1 point
System.out.print("*") counts as 1 point
i++ counts as 1 point

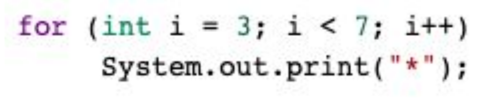
1 + 3 * 4 = 13 "statements" total

There could be different counting methods here. On the AP exam, you'll probably be asked something more like: How many times is System.out.print called in this loop? That is, count how many times a **specific** statement is executed.

Loop Execution Count

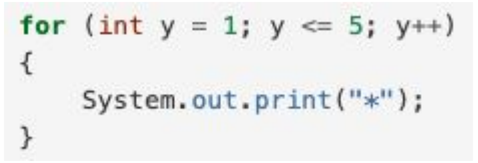
You can use a trace table to figure out how many times a loop executes. But, you may be able to shortcut that just looking at it.

With a < condition, the number of iterations is (ending value - starting value) = 7 - 3 = 4 iterations.



```
for (int i = 3; i < 7; i++)
    System.out.print("*");
```

When a <= is involved, the number of iterations is (ending value - starting value + 1) = 5 - 1 +1 = 5 iterations.

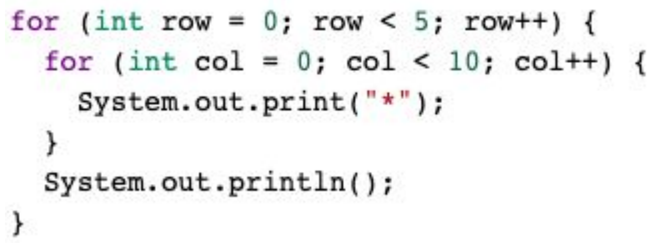


```
for (int y = 1; y <= 5; y++)
{
    System.out.print("*");
}
```

Both of these shortcuts assume the increment expression is just adding 1 to the counter.

Loop Execution Count

For nested loops: The number of times a nested for loop body is executed is the number of times the outer loop runs multiplied by the number of times the inner loop runs (outer loop runs * inner loop runs).



```java
for (int row = 0; row < 5; row++) {
  for (int col = 0; col < 10; col++) {
    System.out.print("*");
  }
  System.out.println();
}
```

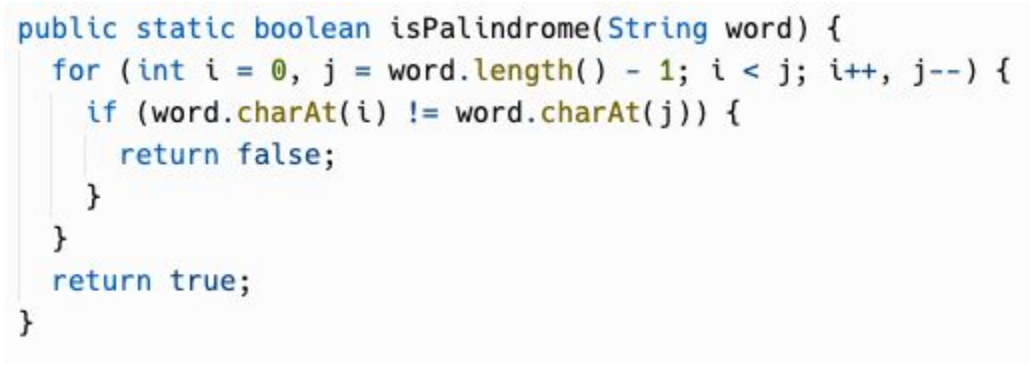
Best and worst case

Let's count the loop executions in this implementation of isPalindrome.

The loop execution count depends on the input.

The **best case** is a zero-length string. Next, a one-letter string. Next, the first and last letters do not match, which would be a loop execution count of 0 (but a statement execution count of 3, for the init expression, condition check, and return statement.)

The **worst case** is… if the string is really a palindrome, and we have to check every pair of characters! That would be word.length() / 2 loop executions.



```java
public static boolean isPalindrome(String word) {
  for (int i = 0, j = word.length() - 1; i < j; i++, j--) {
    if (word.charAt(i) != word.charAt(j)) {
      return false;
    }
  }
  return true;
}
```

Final Review Exercise
File System Simulator

File System Simulator

- Some Thoughts So Far
  - `ArrayList` versus `Array` to manage child lists of `Files` and `Directories`
    - Don't assume a limit on the number of `Files` or `Directories`!
    - The children are of type `File` and `Directory` (not `String`)
  - **`Directory.directoryExists ==`**
    ```
        (Directory.getDirectory(directoryName) != null);
    ```
  - **`Directory.fileExists ==`**
    ```
        (Directory.getFile(fileName) != null);
    ```
  - The `dir` command should list child `Files` and child `Directories`
- Are there any questions?
- Does anyone have any new functionality that they added that they would like to show?

Final Review - Part 3
12/12/2022

Final Review - Part 3

- Unit 5
- Unit 6
- File System Simulator Discussion / Demos

5.1
Anatomy of a Java Class

Definitions

Class - Blueprint for an object; instructions for how construct an object. **There can be ONLY ONE of these -> "Dog"**

Object - A particular instance of a class; use the new operator to create an object instance from a Class. **There can be MANY of these -> "A 3 year-old German Shepherd named Roscoe", "A 1 year-old Golden Retriever named Lucy"**

Properties and Methods - **Properties** are attributes (name, age, breed) and **Methods** are operations (play, eat, sleep); and each can be either `public` (available outside the Object) or `private` (available from only the inside of an Object)

Instance Variables

- Also known as attributes, properties, or fields
- Holds the data of an object
- **Every Object instance has their own values for these properties**

```
 Dog scout = new Dog("Scout", 10);
Dog bailey = new Dog("Bailey", 5);


  scout.name.equals("Scout") == true
 bailey.name.equals("Bailey") == true

scout.name.equals(bailey.name) == false
```

Instance Methods

- Define the behavior generically in the Class
- **So that it can be used by every Object instance**

```
public void feedDog() {
    System.out.println("Gave " + name + " a bowl of food.");
}
```

```
Dog scout = new Dog("Scout", 10);
Dog bailey = new Dog("Bailey", 5);

scout.feedDog();
"Gave Scout a bowl of food."

bailey.feedDog();
"Gave Bailey a bowl of food."
```

Private vs Public

Private

An instance variable or method that can only be accessed within the class

- On the AP Exam all instance variables should be private
- Some methods can be private if they are only used internally

Public

An instance variable or method that can be accessed outside of a class like in the main method

- Most methods are public

Object-Oriented Design

A design philosophy used by programmers when developing larger programs

1. Decide what classes you'll need to solve a problem
2. Define the data (instance variables) and functionality (methods) for the classes
3. Utilize classes and objects to solve your problem

Data Encapsulation

1. Data (instance variables) and the code acting on it (methods) are wrapped together in a single implementation and the details are hidden.
2. Data is safe from harm by keeping it private

5.2 & 5.3
Constructors, Comments, and Conditions

The Anatomy of a Constructor

Things to keep in mind:

- The Constructor name (in red) must **always** match the name of the class (in blue)
- Always prepend the **public** keyword before your constructor name
- Constructors have **no return type**. **Not even void!**
- The constructor definition is often included before other method definitions (but is not required)
- Instance variables are often assigned values within a Constructor; But they can also be intialized in place (i.e. `private boolean isAlive = true;`)

```
public class ClassName
{
   // Instance Variable Declarations
   private String name;
   private int age;
   private boolean isAlive = true;

   // Constructor - same name as Class, no return type
   public ClassName(String initName, int initAge)
   {
      name = initName;
      age = initAge;
   }

   // Other methods ...
}
```

Types of Constructors

- **Default Constructor**
- No-Argument Constructor
- Parameterized Constructor
- Overloaded Constructors

```
public class Person
{
  // No constructor defined
  // Java creates Default Constructor
}
```

- Automatically generated by the Java compiler when you do not supply a Constructor
- Does not do any kind of specialized initialization of the Instance beyond whatever inplace variable initialization that exists
- Instance variables without inplace initialization will default to something "reasonable"
  - `int -> 0`
  - `boolean -> false`
  - `String -> null`
  - `Object -> null`

Types of Constructors

- Default Constructor
- **No-Argument Constructor**
- Parameterized Constructor
- Overloaded Constructors

```
public class Person
{
  private String name;
  private int age;
  public Person() {
    name = "Billy";
    age = 25;
  }
}
```

- A Constructor you define that takes no arguments
- Can perform any kind of initialization that the the Instance requires
- Every Instance is initialized exactly the same
- No values can be passed in during **`new`** to customize the Instance

When you define a Constructor (any kind)
Java will NOT create a Default Constructor!

Types of Constructors

- Default Constructor
- No-Argument Constructor
- **Parameterized Constructor**
- Overloaded Constructors

```
public class Person
{
  private String name;
  public Person(String initName) {
    name = initName;
  }
}
```

- A Constructor you define that takes arguments
- Can perform any kind of initialization that the the Instance requires
- Each Instance is initialized individually based on the values passed into **`new`**

Types of Constructors

- Default Constructor
- No-Argument Constructor
- Parameterized Constructor
- **Overloaded Constructors**

```
public class Person
{
  private String name;
  private int age;
  public Person() {
    name = "Name unknown";
  }
  public Person(String initName) {
    name = initName;
    age = 30;
  }
  public Person(String initName, int initAge) {
    name = initName;
    age = initAge;
  }
}
```

- A Class may have multiple Constructors
- Each Constructor must be named the same as the Class; have no return type; and have a distinct set of parameters (types)
- These are useful when you want to provide default values for some Instance variables - While allowing other Instance variables to be set via `new`

Comments

- Comments are a way for you to annotate your code
- This is text in your program that is never run by Java and is added for the benefit of the person reading the code
- You can also "comment out" a block of code during development to assist in the development or debugging process
- There are 3 ways to write comments in Java

Types of Comments

- **Single-Line Comment**
- Multi-Line Comment
- Documentation Comment

- A single-line comment starts with a double forward-slash (//)
- Can start anywhere - i.e. does not need to be in column 0
- All characters following the double forward-slash are ignored until newline or end of file

```
thisCodeWillRun();
// thisCodeWillNotRun();
thisCodeWillRun();

thisCodeWillRun(); // woo-hoo!
```

Types of Comments

- Single-Line Comment
- **Multi-Line Comment**
- Documentation Comment

- A multi-line comment starts with a forward-slash asterisk (`/*`)
- Can begin anywhere - i.e. does not need to be in column 0
- All characters - including newlines - are considered part of the comment until a asterisk forward-slash (`*/`) is encountered
- Your editor may have a key command that automatically converts a block of code into a multi-line comment

```
/*
thisCodeWillNotRun();
thisCodeWillNotRun();
*/


/* thisCodeWillNotRun(); */


thisCodeWillRun(); /* woo-hoo! */
```

Types of Comments

- Single-Line Comment
- Multi-Line Comment
- **Documentation Comment**

- A variant of the multi-line comment syntax - Documentation Comments start with a forward-slash asterisk asterisk (`/**`)
- Typically found just prior to the definition of a function or method
- All characters - including newlines - are considered part of the Documentation Comment until a asterisk forward-slash (`*/`) is encountered
- Within a Documentation Comment - other standard components may be supported

```
/**
 * Documentation comment
 *
 */
myMethod()
```

Preconditions and Postconditions

DEFINITELY on AP exam

Preconditions and Postconditions

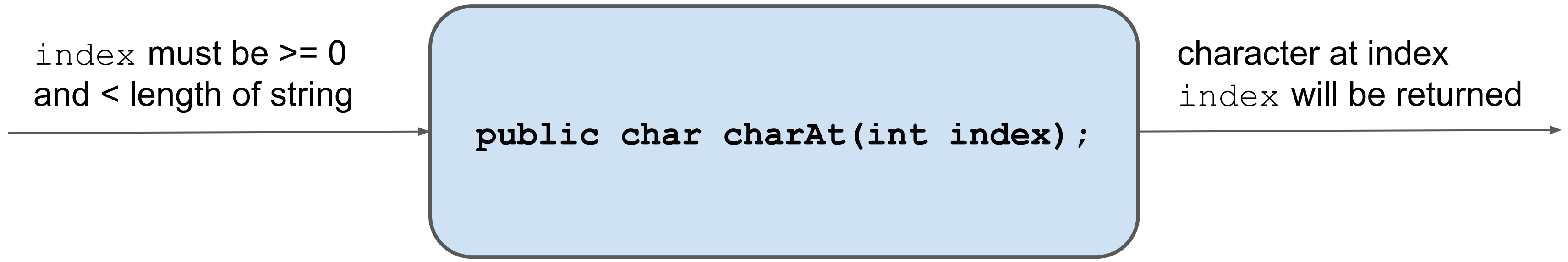
Preconditions and **postconditions** are a "contract" that describes what a method requires about its inputs, and what it promises as output.

Precondition

`index` must be >= 0
and < length of string

Postcondition

character at index
`index` will be returned



```
public char charAt(int index);
```

The caller must satisfy this requirement when calling `charAt`.

In return, the desired character will be returned.

Preconditions

Preconditions are part of the method's documentation, and may exist only as comments.

There is no expectation that the method will check to ensure preconditions are satisfied.

They may or may not be enforced by the method's code – the programmer using the method should read the documentation and understand the "contract" the method offers.

```
/**
 * Precondition: num2 is not zero.
 * Postcondition: Returns the quotient of num1 and num2.
 */
public double divide(double num1, double num2)
{
   return num1 / num2;
}
```

Who enforces preconditions?

Sometimes preconditions ARE enforced by the method's code.

An actual implementation of Java's `String.charAt`:

```
public char charAt(int index) {
      if ((index <  0) || (index >= value.length)) {
            throw new StringIndexOutOfBoundsException(index);
      }
      return value[index];
}
```

Here, an exception is thrown if the precondition is not satisfied.

Throwing an exception in Java is a common way to handle failed preconditions.

Postconditions

A **postcondition** is a condition that is true after running the method. It is what the method promises to do.

Postconditions describe the outcome of running the method, for example what is being returned or the changes to the instance variables.

Examples:

- **`String.compareTo()`** The method returns 0 if the string is equal to the other string. A value less than 0 is returned if the string is less than the other string (less characters) and a value greater than 0 if the string is greater than the other string (more characters).
- **`Math.random`** Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.

5.4 & 5.5
Accessor Methods and Mutator Methods

Creating Classes

- Variables & Methods
- Public vs Private
- Constructors
- **Accessor Methods**
- Mutator Methods

```
public class Person
{
  private String name;
  public Person(String initName) {
    name = initName;
  }
  public String getName() {
    return name;
  }
}
```

- `public` methods used to provide read-only access to `private` instance variables within the Object
- Sometimes these are called "get methods" or "getters"
- These `public` methods have a return type that matches the type of the `private` variable being returned
- Accessor methods are commonly named `get+VariableName` and do not have parameters

Creating Classes

- Variables & Methods
- Public vs Private
- Constructors
- Accessor Methods
- **Mutator Methods**

```
public class Person
{
  private String name;
  public Person(String initName) {
    name = initName;
  }
  public void setName(String newName) {
    name = newName;
  }
}
```

- `public` methods used to modify internal `private` instance variables
- Sometimes these are called "set methods" or "setters"
- These `public` methods typically have a `void` return type and a parameter that matches the type of the `private` instance variable being modified
- Mutator methods are commonly named `set+VariableName` and have a single parameter

5.6: Writing Methods

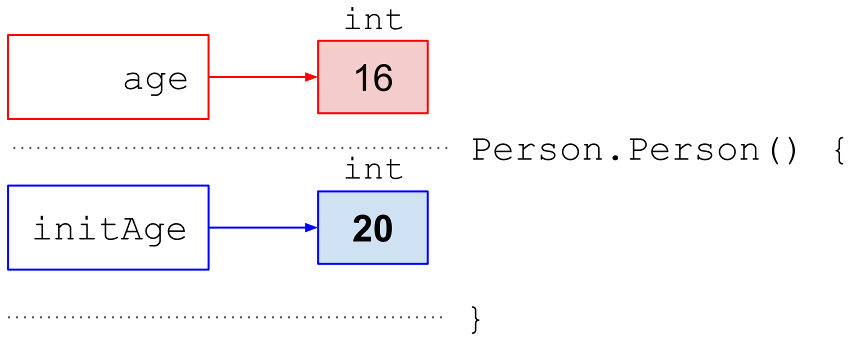
Methods

- We have already covered about HOW to create Methods - but we have not spent much time talking about WHEN you should consider moving code into a Method
- Some of the WHENs
    - You have the same (or very nearly the same) block of code written in multiple places
    - You want to reduce complexity (improve development velocity / reduce code brittleness)
    - You want to write tests for a block of code
    - You have Methods that are excessively long (more than a single page)

Method Parameters: **Pass by Value** / Pass by Reference

- Parameters passed into Methods behave differently if they are primitive types or Object references
- Primitive Types (byte, short, int, long, float, double, boolean, char)
  - A copy is made when the Method is invoked for use during the life of the Method
  - The Method cannot alter the value of the variable (passed as a parameter) that exists in the caller of the Method

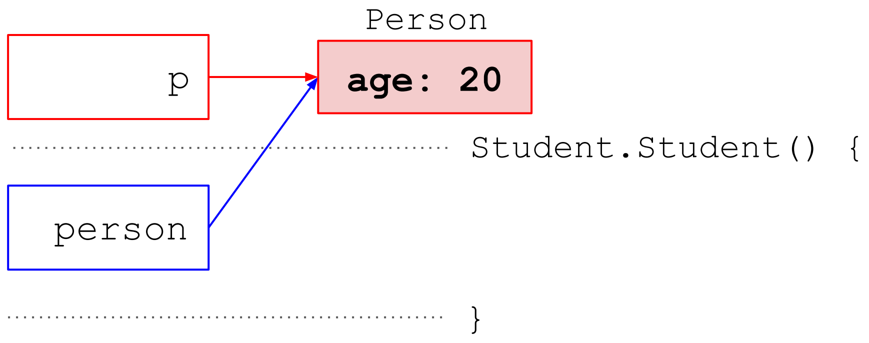
```
int age = 16;
Person p = new Person(age);
// age is guaranteed to still be 16

public class Person {
  public Person(int initAge) {
    // initAge is a COPY of age
    initAge = 20; // does not alter the value of age in the caller
  }
}
```



Method Parameters: Pass by Value / **Pass by Reference**

- Object Types / Classes
  - A reference to the Object is passed into the Method
  - The Method can alter the internal value of the Object passed as a parameter; And the caller can see these changes when it access the Object

```
int age = 16;
Person p = new Person(age);
// p.age == 16
Student s = new Student(p);
// p.age == 20

public class Student {
  public Student(Person person) {
    // person is a REFERENCE to the same object in the caller
    person.age = 20; // alters the Person passed in from the the caller
  }
```



Method Parameters: Pass by Value / **Pass by Reference**

- Object Types / Classes
  - A reference to the Object is passed into the Method
  - The Method can alter the value of the variable (passed as a parameter) that exists in the caller of the Method - **RARELY USED / NOT A BEST PRACTICE**

```
Person p = new Person();
p.age = 16;
Student s = new Student(p);
// p.age now equals 20 - LIKELY UNEXPECTED BEHAVIOR

public class Student {
  public Student(Person person) {
    // person is a REFERENCE to the same object in the caller
    person.age = 20; // alters the Person passed in from the the caller
  }
}
```

Sections 5.7, 5.8, 5.9

Statics
Scope and Access
`this`

Statics

- Static variables & methods belong to Classes - not Instances of a Class
  - There is only one copy of a `static` variable & method
  - They can be `public` or `private`
- Static variables & methods are accessed using the name of the class to which they belong and a dot (`.`) With a couple of exceptions: Statics accessing statics; And static imports
  - `Math.PI`
  - `Math.random()`
  - `Math.sqrt()`
- The `main()` method is a static method - It is only ever run one time for a program - And the JVM needs to run it without creating an Instance of a Class

Statics

- **Statics can directly access other Statics**
- Statics cannot directly access non-Statics
- Non-Statics can directly access Statics

```
class Person {
  private static int numPeople = 0;
  private String name;
  public Person(String initName) {
    numPeople++;
    name = initName;
  }
  public static int getNumPeople() {
    // System.out.println(name);
    return numPeople;
  }
  public void report() {
    System.out.println(name + " is one of " + numPeople + " people");
  }
}
```

```
Person.getNumPeople();

A> 0
```

Statics

- Statics can directly access other Statics
- **Statics cannot directly access non-Statics**
- Non-Statics can directly access Statics

```
public static int getNumPeople() {
  // System.out.println(name);
  return numPeople;
}
```

```
class Person {
  private static int numPeople = 0;
  private String name;
  public Person(String initName) {
    numPeople++;
    name = initName;
  }
  public static int getNumPeople() {
    // System.out.println(name);
    return numPeople;
  }
  public void report() {
    System.out.println(name + " is one of " + numPeople + " people");
  }
}
```

Statics

- Statics can directly access other Statics
- Statics cannot directly access non-Statics
- **Non-Statics can directly access Statics**

```
class Person {
  private static int numPeople = 0;
  private String name;
  public Person(String initName) {
    numPeople++;
    name = initName;
  }
  public static int getNumPeople() {
    // System.out.println(name);
    return numPeople;
  }
  public void report() {
    System.out.println(name + " is one of " + numPeople + " people");
  }
}
```

```
Person p1 = new Person("Julie");
Person p2 = new Person("Bobby");

p1.report()

B> Julie is one of 2 people

p1.getNumPeople();

C> 2
```

Scope and Access Control

- **Class Level Scope** Instance and static variables inside a Class.
- **Method Level Scope** Local variables (including parameter variables) inside a method.
- **Block Level Scope** Loop variables and other local variables defined inside of blocks of code with { }

```
public class Person {
  private String name;
  private String email;

  public void print(int length) {
    for (int i = 0; i < length; i++) {
      System.out.println(name.charAt(i));
    }
  }
}
```

Scope and Access Control

- **Class Level Scope** Instance and static variables inside a Class.
- **Method Level Scope** Local variables (including parameter variables) inside a method.
- **Block Level Scope** Loop variables and other local variables defined inside of blocks of code with { }

```
public class Person {
  private String name;
  private String email;

  public void print(int length) {
    for (int i = 0; i < length; i++) {
      System.out.println(name.charAt(i));
    }
  }
}
```

Scope and Access Control

- **Class Level Scope** Instance and static variables inside a Class.
- **Method Level Scope** Local variables (including parameter variables) inside a method.
- **Block Level Scope** Loop variables and other local variables defined inside of blocks of code with { }

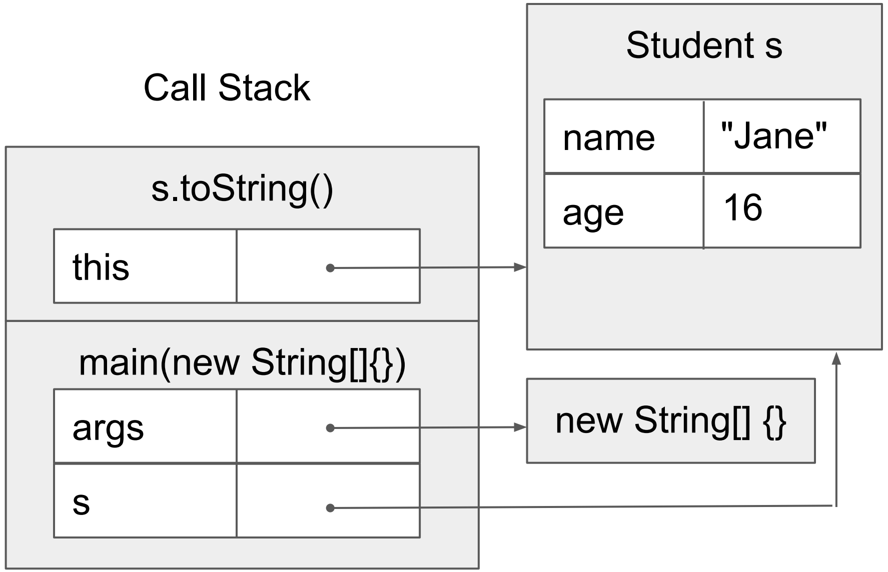
```
public class Person {
  private String name;
  private String email;

  public void print(int length) {
    for (int i = 0; i < length; i++) {
      System.out.println(name.charAt(i));
    }
  }
}
```

`this` (keyword)

- Within an Instance method of a Class - `this` refers to the current Instance (and refers to the Instance being created in a Constructor)
- Static methods cannot refer to `this` (since there is no Instance when using static methods)

```
class Person {
  private static int numPeople = 0;
  private String name;
  public Person(String name) {
    numPeople++;
    this.name = name;
  }
  public static int getNumPeople() {
    return numPeople;
  }
  public void report() {
    System.out.println(name + " is one of " + numPeople + " people");
  }
}
```

`this` is a reference to the current object instance



```
class Student {
  private String name;
  private int age;

  Student(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public String toString(){
    return name;
  }
}

class HelloWorld {
    public static void main( String args[] ) {
      Student s = new Student("Jane",16);
      System.out.println(s.toString()); //"Jane"
      System.out.println(s); //Also "Jane"
    }
}
```

VERY common pattern.

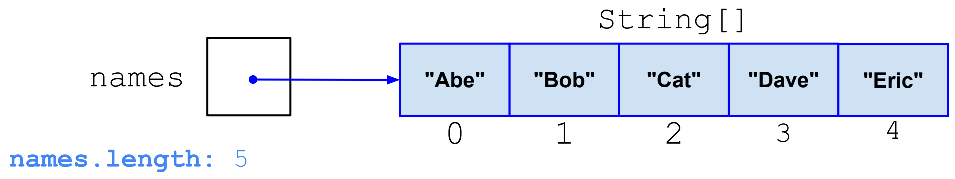
Q: What's the difference here between name and this.name?

Unit 6: Arrays

6.1 Arrays

Arrays are a "table" of values of the same type. Each value can be get/set using its 0-based index.

```
String[] names = {"Abe", "Bob", "Cat", "Dave", "Eric"};
```



For any Java type, you can use *type*`[]` for an Array of that type.

The length of an Array is decided when it is created, and cannot change.

Arrays - Creation

Arrays are created with an **initializer list** or the **`new`** operator:

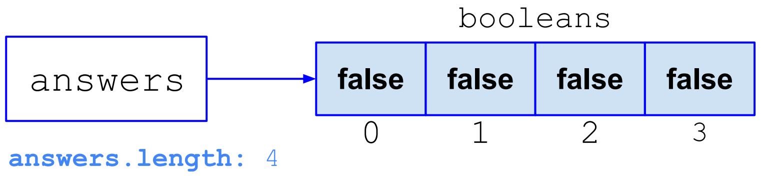
```
boolean[] answers = {true, false, false, true};
int[] scores = {100, 84, 95, 78};

double[] prices = new double[20];
String[] questions = new String[5];

int numStudents = 10;
Student[] students = new Student[numStudents];
```

If you declare an Array variable and do not initialize it, it will default to **null**, like other object types.

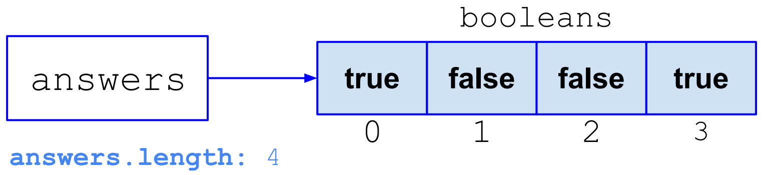
```
Student[] students; // Contains null
```

Arrays: initialized with default values, unless specified

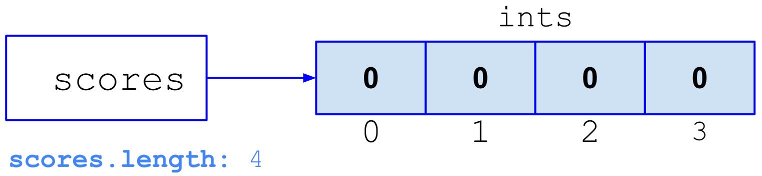
```
boolean[] answers = new boolean[4];
```



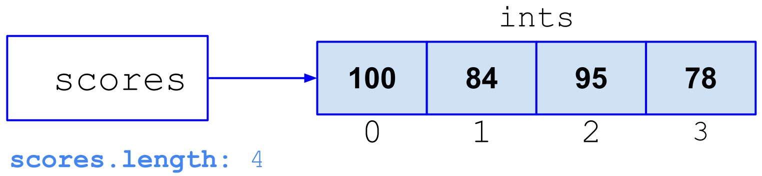
```
boolean[] answers = {true, false, false, true};
```



```
int[] scores = new int[4];
```

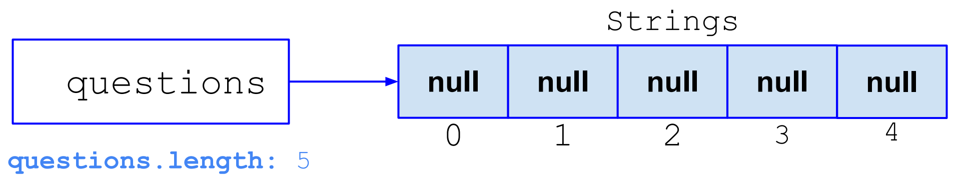


```
int[] scores = {100, 84, 95, 78};
```

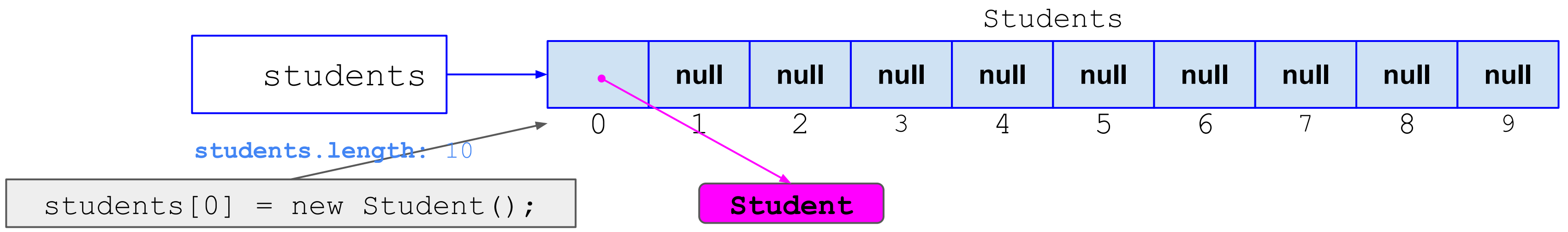


Arrays of object type are initialized to **null**

```
String[] questions = new String[5];
```



```
Student[] students = new Student[10];
```



Re-assigning Array variables

Array variables are just like any other object typed variable:

```
boolean[] answers = {true, false, false, true};
```

That is, you can later point them to a different object (array):

```
answers = new boolean[4];
```

The `new` operator also can take an initializer list.

```
answers = new boolean[] { true, false, false, true };
```

Array length and the [] operator

`*array*.length`	`*array*[index]`	`*array*[index] = *value*`
A public final instance variable which returns the length of the array. Note that it's a variable (`array.length`), not a method like `String.length()`	Evaluates to the *index*'th element of array. Indices are 0-based. If *index* is out of bounds, `ArrayIndexOutOfBoundsException` is thrown.	Sets the *index*'th element of the array to value. Indices are 0-based. If *index* is out of bounds, `ArrayIndexOutOfBoundsException`is thrown. Remember, the length of an array cannot change.

Example:
`boolean[] answers = {true, false,`
`false, true};`
`System.out.println(answers.length);`
`A> 4`
	Example:
	`int[] scores = {32, 5, 87, 44};`
	`System.out.println(scores[2]);`
	`B> 87`
		Example:
		`int[] scores = {100, 84, 95, 78};`
		`scores[1] = 48;`
		`System.out.print(scores[1]);`
		`C> 48`

6.2 Traversing Arrays with `for` loops

for loops are frequently used to traverse (visit every element) of an array.

The most common pattern is:

```
int[] scores = {95, 100, 91, 85 };
for (int i = 0; i < scores.length; i++) {
    System.out.println(scores[i]);
}
```

The range of valid Array indexes (for non-empty Arrays) is `0` to `Array.length - 1`, so starting at `i=0` and looping on `i<`*`array`*`.length` will loop over every element.

Traversing arrays all different ways

Reverse order

```
int[] scores = {95, 100, 91, 85 };
for (int i=scores.length-1;
     i>=0;
     i--) {
  System.out.println(scores[i]);
}
```

Subarray

```
int[] scores = {95, 100, 91, 85 };
for (int i=startIndex;
     i<endIndex;
     i++) {
  System.out.println(scores[i]);
}
```

Note: In this example, `endIndex` is exclusive, like `String.substring`. The interval `[startIndex, endIndex)` will be returned.

Skipping elements

```
int[] scores = {95, 100, 91, 85 };

// Print every other element
for (int i=0, n=scores.length;
     i<n;
     i += 2) {
  System.out.println(scores[i]);
}
```

Traversing Arrays with `for` loops

You can use a `for` loop to traverse an Array from back to front!

```
int[] scores = {95, 100, 91, 85 };
for (int i = scores.length - 1; i >= 0; i--) {
    System.out.println(scores[ i]);
}
```

...or to traverse any arbitrary range of elements

```
int[] scores = {95, 100, 91, 85 };
for (int i = 1; i <= 2; i++) {
    System.out.println(scores[ i]);
}
```

6.3 Traversing Arrays with `for-each` loops

- An alternate way to loop through Objects that support the Iterable interface

```
for (type arrayItemVariable : arrayVariable) {
     arrayItemVariable is a copy of arrayVariable[0]
     arrayItemVariable is a copy of arrayVariable[1]
     arrayItemVariable is a copy of arrayVariable[...]
     arrayItemVariable is a copy of arrayVariable[arrayVariable.length-1]
     then the loop terminates
}
```

Traversing Arrays with `for-each` loops

```
for (type arrayItemVariable : arrayVariable) {
    arrayItemVariable resolves to arrayVariable[...]
}
```

```
String[] colors = {"red", "orange", "purple"};
```

```
System.out.println("begin");
for(String color: colors){
  System.out.println(" " + color);
}
System.out.println("end");
```

Output:

```
begin
  red
  orange
  purple
end
```

Traversing Arrays with `for-each` loops

The type of the `for-each` variable MUST match the type of the values stored in the Array

```
String[] colors = {"red", "orange", "purple"};

for(int color: colors){
  System.out.println(" " + color);
}
```

✗

***Note:** color must be of type String since colors is an Array that contains Strings*

Comparing `for` and `for-each` loops

`for`

- Direct access to any element in the Array, in any order, using index and `[]`
- You always know the index – so using parallel Arrays is easy!
- Code has more "boilerplate" and may require more variable declarations
- Can change the value of an Array element during the loop
- Many iteration possibilities: Reverse order, skip elements, visit only a sub-array, etc.

`for-each`

- Sequential access to the elements in the Array - **must always go from first to last**
- You do not know the index – so using Parallel Arrays requires tracking it yourself
- Code is more concise and may require less variable declarations
- Cannot change the value of an Array element during the loop
- More limited in functionality, but the code is "short and sweet."
- Might be slower since it's Iterable/Iterator under the hood ... benchmark if unsure

6.4 Common Array Algorithms

Here are some common algorithms that you should be familiar with for the AP CS A exam:

- Determine the minimum or maximum value in an array
- Compute a sum, average, or mode of array elements
- Search for a particular element in the array
- Determine if at least one element has a particular property
- Determine if all elements have a particular property
- Access all consecutive pairs of elements
- Determine the presence or absence of duplicate elements
- Determine the number of elements meeting specific criteria
- Shift or rotate elements left or right
- Reverse the order of the elements

Minimum and Maximum Value

These require a "tracking value" for the smallest or largest value found so far.

	25	70	9	3	15	16	19
minValue =	25	25	9	3	3	3	$\boxed{3}$

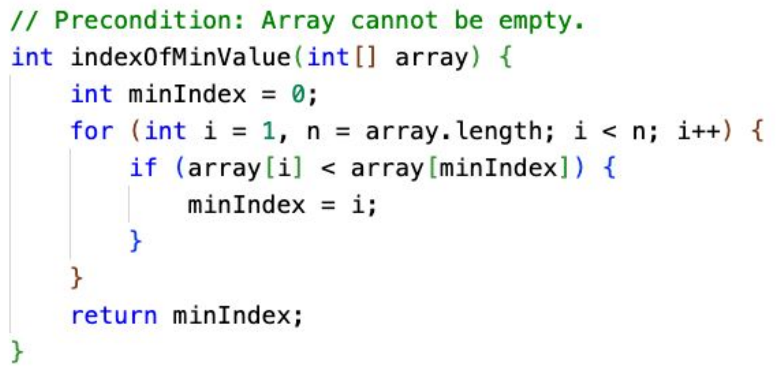
One trick is to "seed" the tracking value with the first element, and skip it in the loop.

```
// Precondition: Array cannot be empty.
int findMinValue(int[] array) {
    int minValue = array[0];
    for (int i = 1, n = array.length; i < n; i++)
        if (array[i] < minValue) {
            minValue = array[i];
        }
    }
    return minValue;
}
```

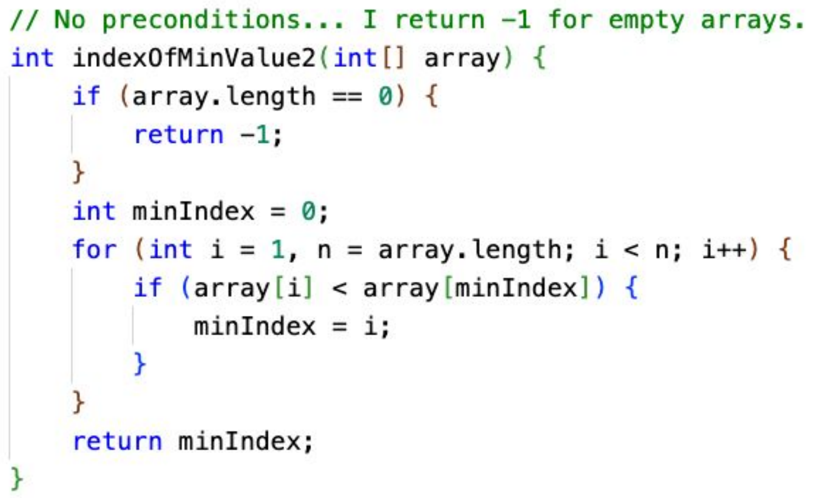
```
// Precondition: Array cannot be empty.
int findMaxValue(int[] array) {
    int maxValue = array[0];
    for (int i = 1, n = array.length; i < n; i++) {
        if (array[i] > maxValue) {
            maxValue = array[i];
        }
    }
    return maxValue;
}
```

Minimum and Maximum Value

With a little modification, you could return the array index instead of the value.



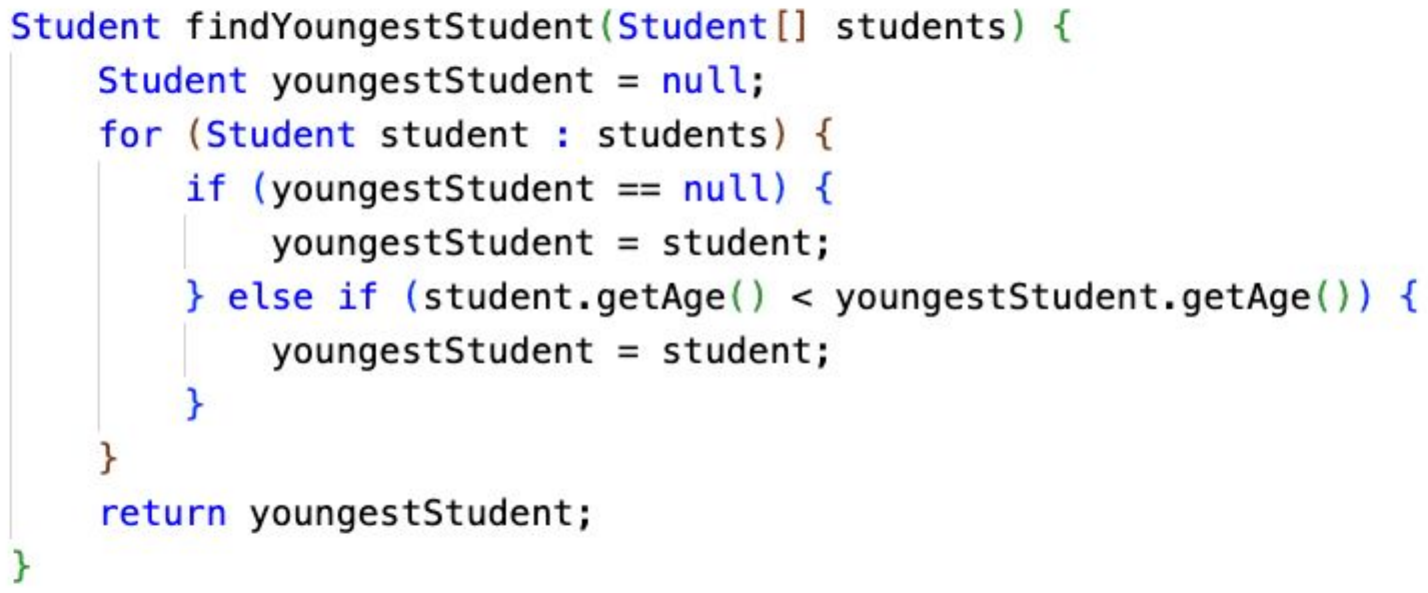
```java
// Precondition: Array cannot be empty.
int indexOfMinValue(int[] array) {
    int minIndex = 0;
    for (int i = 1, n = array.length; i < n; i++) {
        if (array[i] < array[minIndex]) {
            minIndex = i;
        }
    }
    return minIndex;
}
```



```java
// No preconditions... I return -1 for empty arrays.
int indexOfMinValue2(int[] array) {
    if (array.length == 0) {
        return -1;
    }
    int minIndex = 0;
    for (int i = 1, n = array.length; i < n; i++) {
        if (array[i] < array[minIndex]) {
            minIndex = i;
        }
    }
    return minIndex;
}
```

Minimum and Maximum Value of Objects

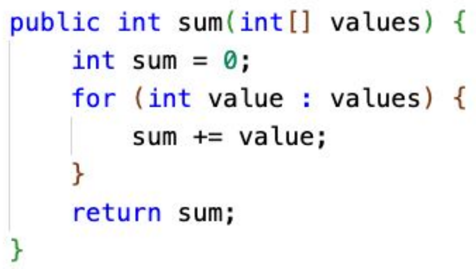
You might be dealing with an array of something other than numbers.



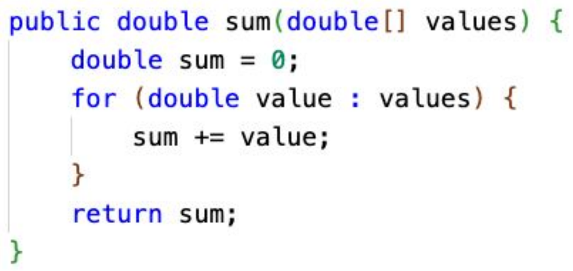
```java
Student findYoungestStudent(Student[] students) {
    Student youngestStudent = null;
    for (Student student : students) {
        if (youngestStudent == null) {
            youngestStudent = student;
        } else if (student.getAge() < youngestStudent.getAge()) {
            youngestStudent = student;
        }
    }
    return youngestStudent;
}
```

Sum and Average

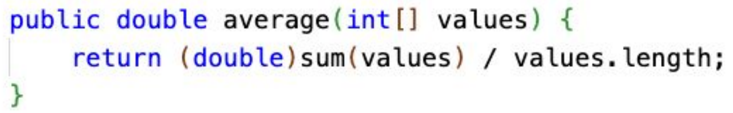
If dealing with ints, remember to cast to double when calculating average.
(Also known as the arithmetic mean.)



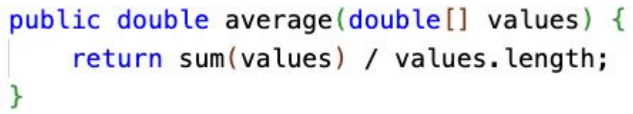
```java
public int sum(int[] values) {
    int sum = 0;
    for (int value : values) {
        sum += value;
    }
    return sum;
}
```



```java
public double sum(double[] values) {
    double sum = 0;
    for (double value : values) {
        sum += value;
    }
    return sum;
}
```



```java
public double average(int[] values) {
    return (double)sum(values) / values.length;
}
```



```java
public double average(double[] values) {
    return sum(values) / values.length;
}
```

Calculations aren't always over int[] or double[]...

What if you are calculating the average age of a class of Students?

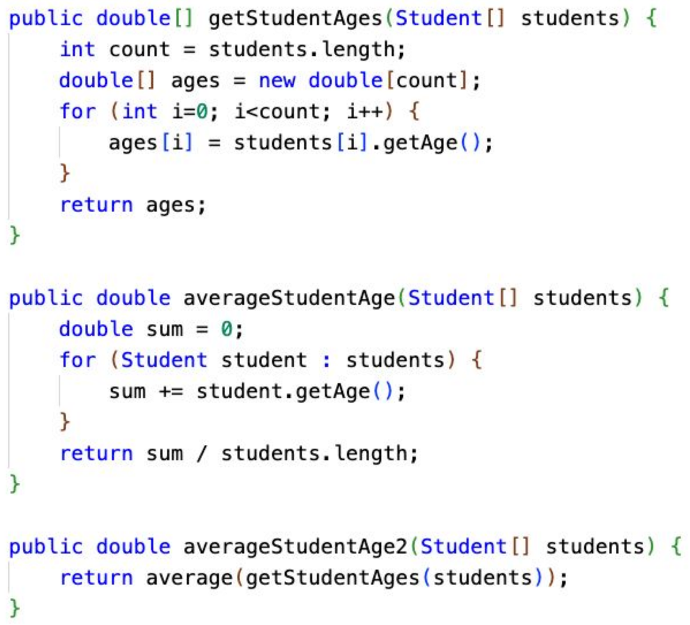
```java
class Student {
    private String name;
    private int age;
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
    public int getAge() { return age; }
}
```

```java
private Student[] students = {
    new Student("Alice", 16),
    new Student("Bob", 15),
    new Student("Carleton", 17),
    new Student("David", 17)
};
```

```java
public double averageStudentAge(Student[] students) {
    double sum = 0;
    for (Student student : students) {
        sum += student.getAge();
    }
    return sum / students.length;
}
```

Transforming arrays to do calculations

It can make sense to transform an Array into another Array, and then do a calculation.

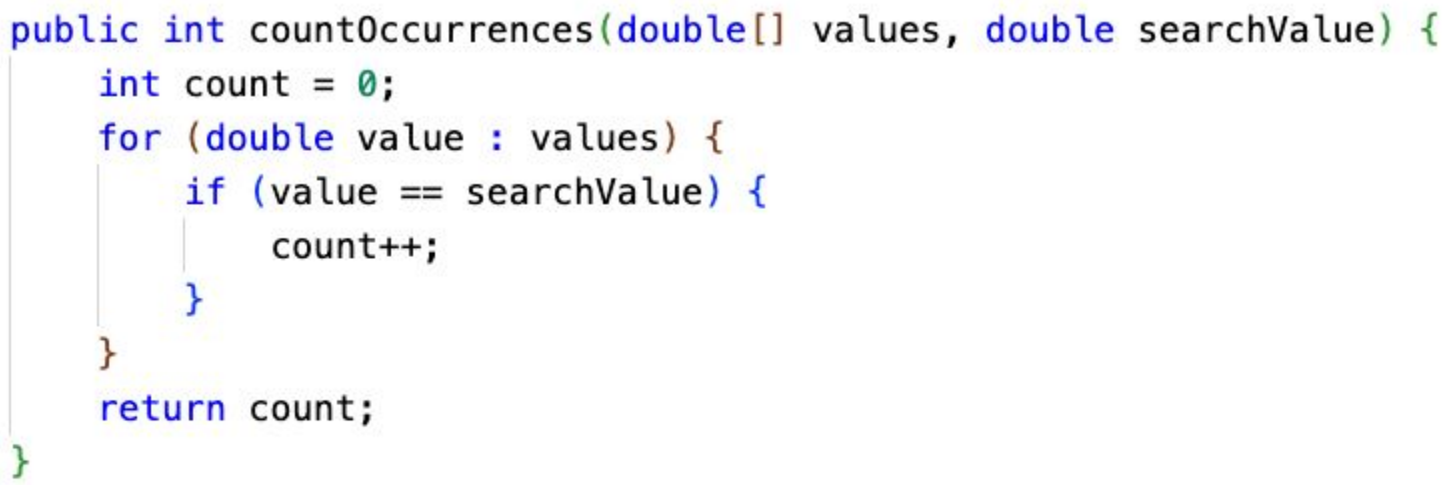


```java
public double[] getStudentAges(Student[] students) {
    int count = students.length;
    double[] ages = new double[count];
    for (int i=0; i<count; i++) {
        ages[i] = students[i].getAge();
    }
    return ages;
}

public double averageStudentAge(Student[] students) {
    double sum = 0;
    for (Student student : students) {
        sum += student.getAge();
    }
    return sum / students.length;
}

public double averageStudentAge2(Student[] students) {
    return average(getStudentAges(students));
}
```

Creating the ages Array takes time and memory... but it can make sense, depending on the situation.

- The age data might be used more than once
- You might be interfacing with code, such as a third-party library, that doesn't know about Students
- Your math-heavy code stays in its "domain" ... it only needs to know about math, not Students, and can be reused for things other than Students.
- Performance and memory usage may not be critical

Determine number of elements meeting specific criteria



```java
public int countOccurrences(double[] values, double searchValue) {
    int count = 0;
    for (double value : values) {
        if (value == searchValue) {
            count++;
        }
    }
    return count;
}
```

Mode: Most frequently occurring value

We can calculate the mode using a nested loop. For each element, count the number of occurrences of that element, and track which element has the maximum number of occurrences.

17	17	9	9	9	9	1	17	19	3	3	5

frequency (of value at current index)

3	skip	4	skip	skip	skip	1	3	1	2	skip	1

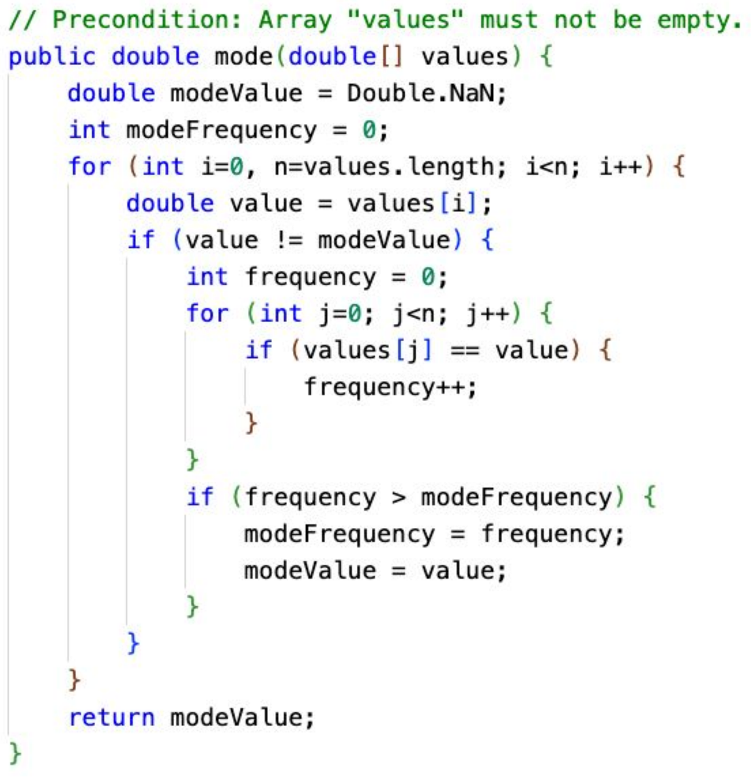
modeValue

17	17	9	9	9	9	9	9	9	9	9	**9**

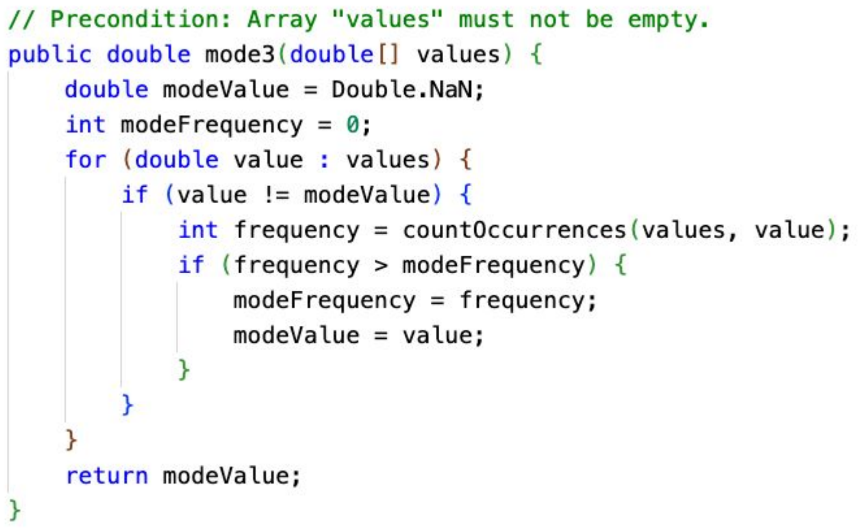
modeFrequency

3	3	4	4	4	4	4	4	4	4	4	**4**

Mode of unsorted array



```java
// Precondition: Array "values" must not be empty.
public double mode(double[] values) {
    double modeValue = Double.NaN;
    int modeFrequency = 0;
    for (int i=0, n=values.length; i<n; i++) {
        double value = values[i];
        if (value != modeValue) {
            int frequency = 0;
            for (int j=0; j<n; j++) {
                if (values[j] == value) {
                    frequency++;
                }
            }
            if (frequency > modeFrequency) {
                modeFrequency = frequency;
                modeValue = value;
            }
        }
    }
    return modeValue;
}
```



```java
// Precondition: Array "values" must not be empty.
public double mode3(double[] values) {
    double modeValue = Double.NaN;
    int modeFrequency = 0;
    for (double value : values) {
        if (value != modeValue) {
            int frequency = countOccurrences(values, value);
            if (frequency > modeFrequency) {
                modeFrequency = frequency;
                modeValue = value;
            }
        }
    }
    return modeValue;
}
```

Mode of sorted array, with helper method

```java
public int lengthOfRun(double[] values, int index) {
    double value = values[index];
    int length = 0;
    while (index < values.length && values[index] == value) {
        index++;
        length++;
    }
    return length;
}
```

```java
// Precondition: Array "values" must be sorted.
public double modeOfSortedArray2(double[] values) {
    double modeValue = Double.NaN;
    int modeFrequency = 0;
    int i = 0;
    while (i < values.length) {
        int frequency = lengthOfRun(values, i);
        if (frequency > modeFrequency) {
            modeFrequency = frequency;
            modeValue = values[i];
        }
        i += frequency;
    }
    return modeValue;
}
```

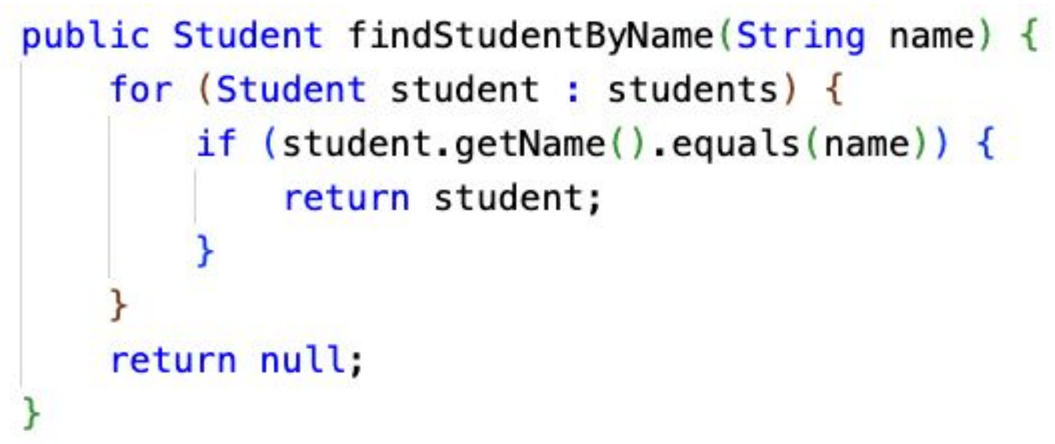
1	1	2	2	2	3	3	3	3	4	4	5

lengthOfRun(values, index)

2		3			4				2		1

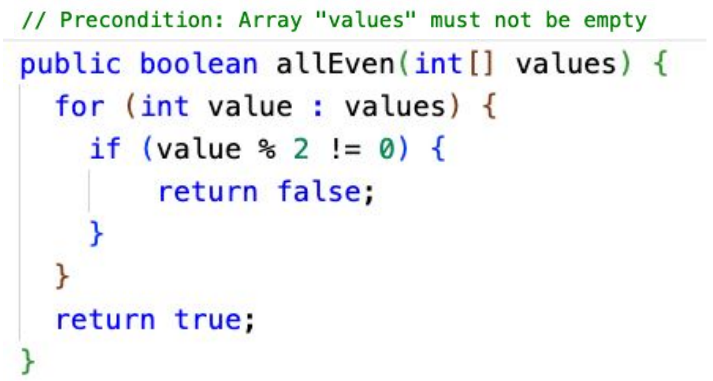
Search for a particular element in the array

This is known as **linear search**, the simplest (and least efficient) of search algorithms. We'll learn others later.



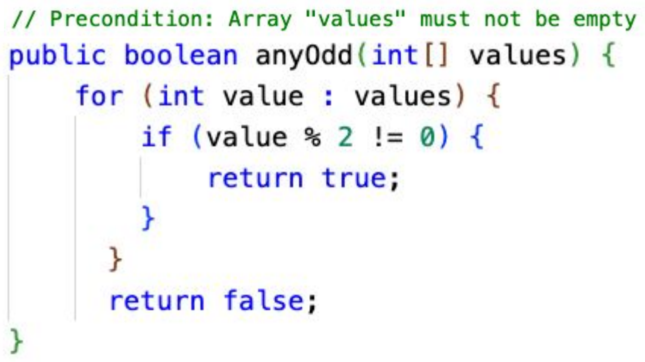
```java
public Student findStudentByName(String name) {
    for (Student student : students) {
        if (student.getName().equals(name)) {
            return student;
        }
    }
    return null;
}
```

Determine if all elements have a particular property

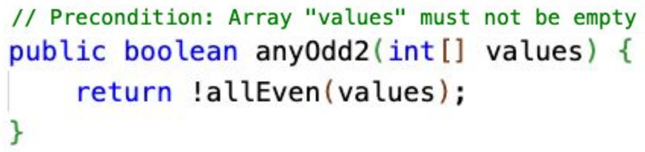


```java
// Precondition: Array "values" must not be empty
public boolean allEven(int[] values) {
  for (int value : values) {
    if (value % 2 != 0) {
      return false;
    }
  }
  return true;
}
```

Determine if at least one element has a particular property



```java
// Precondition: Array "values" must not be empty
public boolean anyOdd(int[] values) {
    for (int value : values) {
        if (value % 2 != 0) {
            return true;
        }
    }
    return false;
}
```



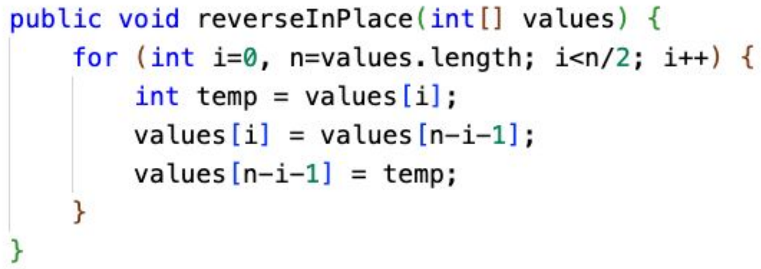
```java
// Precondition: Array "values" must not be empty
public boolean anyOdd2(int[] values) {
    return !allEven(values);
}
```

These algorithms are the inverse of each other:

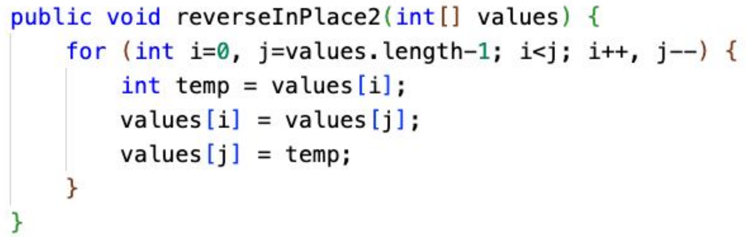
If **all** of the numbers all even, then there are **not any** odd numbers.

If there are **any** odd numbers, then **not all** of the numbers are even.

Reverse an array (in place)



```
public void reverseInPlace(int[] values) {
    for (int i=0, n=values.length; i<n/2; i++) {
        int temp = values[i];
        values[i] = values[n-i-1];
        values[n-i-1] = temp;
    }
}
```



```
public void reverseInPlace2(int[] values) {
    for (int i=0, j=values.length-1; i<j; i++, j--) {
        int temp = values[i];
        values[i] = values[j];
        values[j] = temp;
    }
}
```

The top implementation is fine, but the bottom one uses i and j instead of just i.

I find it easier to reason about what this algorithm is doing by having two index counters, "racing" toward each other from each end of the array.

Computers have many **registers** for storage of frequently used variables, so there is really no additional cost to having two variables instead of one. It can be even faster, since less arithmetic is performed.

Check for presence of duplicate elements

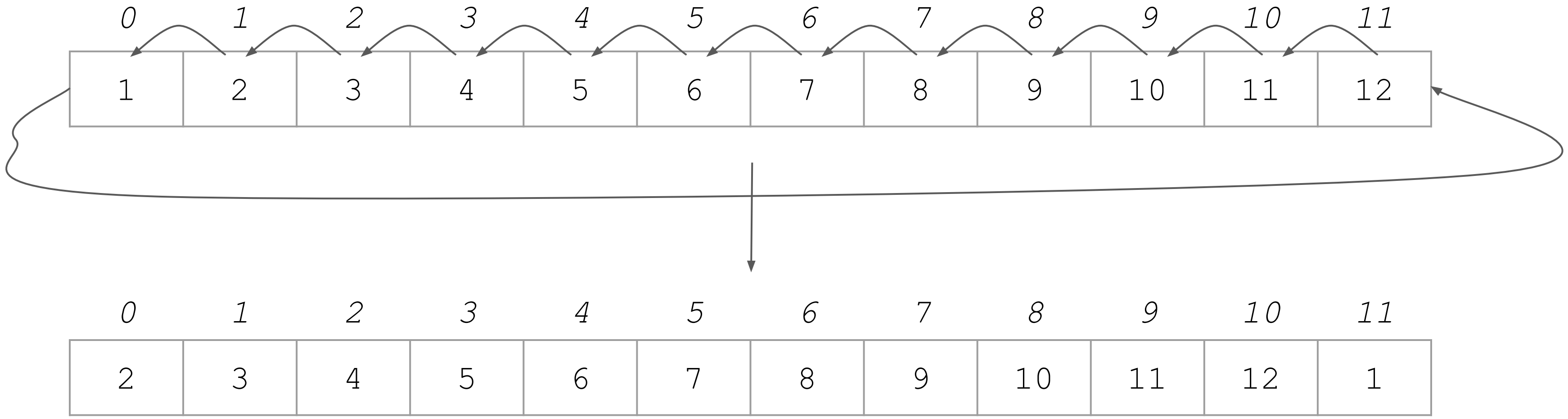
```java
public boolean hasDuplicates(double[] values) {
    for (int i=0, n=values.length; i<n; i++) {
        for (int j=i+1; j<n; j++) {
            if (values[i] == values[j]) {
                return true;
            }
        }
    }
    return false;
}
```

```java
// Precondition: "values" must be sorted
public boolean sortedArrayHasDuplicates(double[] values) {
    for (int i=1, n=values.length; i<n; i++) {
        if (values[i-1] == values[i]) {
            return true;
        }
    }
    return false;
}
```

Shift or rotate an array

Here, we rotate an array of numbers to the left by 1 position.

a[i] = a[i+1] for all i. The first element gets moved to the last position.



Rotate array to the left, in place

```java
// Precondition: Array "values" must not be empty.
public void rotateLeft(double[] values) {
    double firstValue = values[0];
    for (int i=0, n=values.length; i<n-1; i++) {
        values[i] = values[i+1];
    }
    values[values.length-1] = firstValue;
}
```

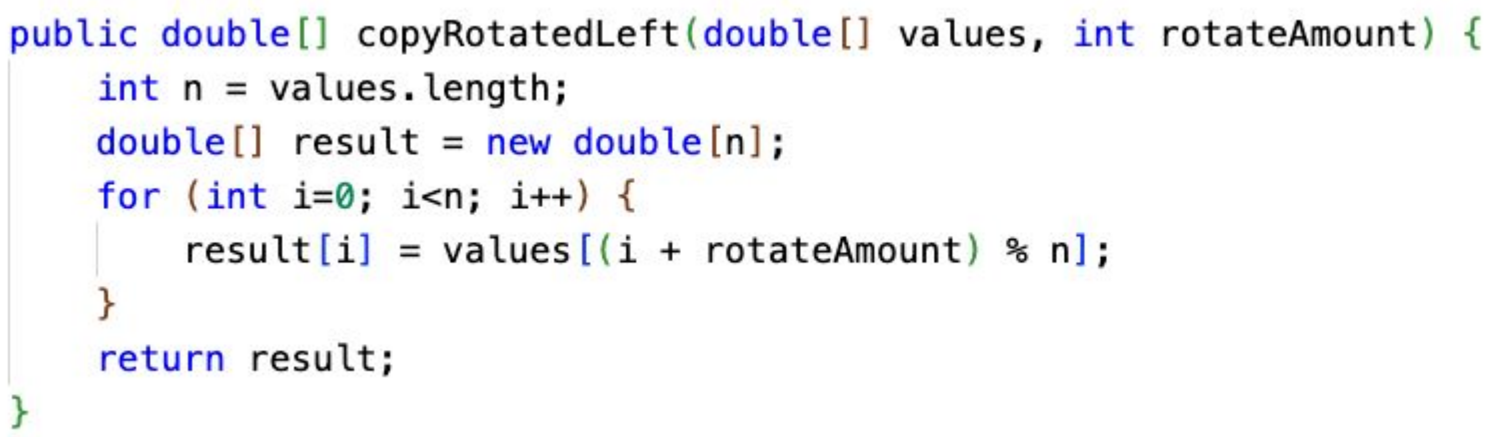
0	1	2	3	4	5	6	7	8	9	10	11
1	2	3	4	5	6	7	8	9	10	11	12

↓

0	1	2	3	4	5	6	7	8	9	10	11
2	3	4	5	6	7	8	9	10	11	12	1

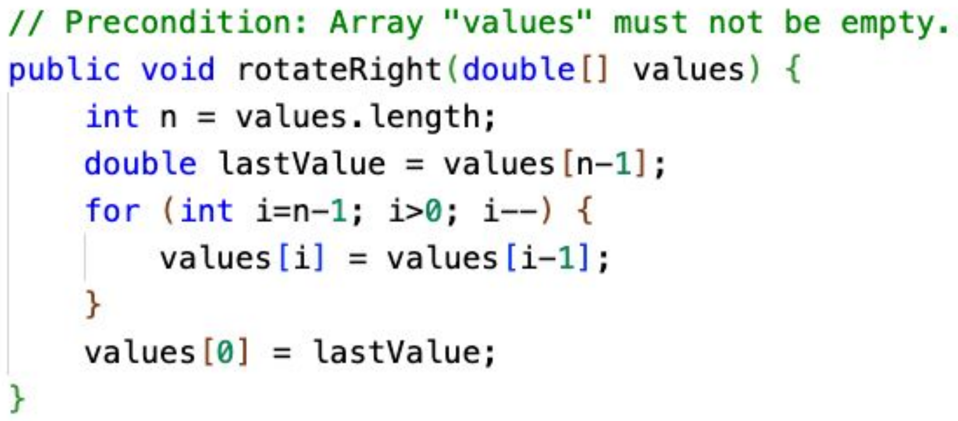
Rotate array to the left multiple positions, returning copy

Here's another instance where the % operator is very useful.



```java
public double[] copyRotatedLeft(double[] values, int rotateAmount) {
    int n = values.length;
    double[] result = new double[n];
    for (int i=0; i<n; i++) {
        result[i] = values[(i + rotateAmount) % n];
    }
    return result;
}
```

Rotate array to the right, in place



```java
// Precondition: Array "values" must not be empty.
public void rotateRight(double[] values) {
    int n = values.length;
    double lastValue = values[n-1];
    for (int i=n-1; i>0; i--) {
        values[i] = values[i-1];
    }
    values[0] = lastValue;
}
```

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
1	2	3	4	5	6	7	8	9	10	11	12

$\downarrow$

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
12	1	2	3	4	5	6	7	8	9	10	11

Final Review Exercise
File System Simulator

File System Simulator

- Some Thoughts So Far
  - Seeing additional `System.out` statements letting the user know what is going on 👍
  - Including a new `help` command
- Are there any questions?
- Does anyone have any new functionality that they added that they would like to show?
  - `find(fileName)` - Ryan