2023-04-14

AP CS A Exam

- Date of Exam
  - **Wednesday, May 3, 2023 at 12 PM**
- Section 1: Multiple Choice
  - 40 Questions
  - 90 Minutes
  - 50% of Exam Score
- Section 2: Free-Response
  - 4 Questions
  - 90 Minutes
  - 50% of Exam Score
- Additional Information and Past Questions
  - College Board: AP Computer Science A Exam

Upcoming Schedule

Monday	Wednesday	Friday
		04/14/2023 (90) • Review: Units 1-4 **• AP CS Question 1: Methods and Control Structures**
04/17/2023 (90) • Review: Unit 5, Unit 9 **• AP CS Question 2: Classes**	**04/19/2023 (90)** • Review: Units 6-7 **• AP CS Question 3: Array/ArrayList**	**04/21/2023 (45)** • Review: Unit 10 • More recursion exercises like we did on Apr-7
04/24/2023 (90) • Review: Unit 8 **• AP CS Question 4: 2D Array**	**04/26/2023 (90)** **• AP CS Multiple Choice Game**	**04/28/2023 (45)** • Review: Unit 7, Unit 10 • Algorithms: Iterative/recursive binary search, selection sort, insertion sort, merge sort
05/01/2023 **• FINAL**	**05/03/2023** **• AP EXAM**	

Units 1-4
AP CS FRQ 1
(Methods and Control Structures)

Print Statements and Comments

- Print something, but do not terminate with a newline (`'\n'`)
  - `System.out.print("Hello, World!");`
- Print something, terminated by a newline
  - `System.out.println("Hello, World!");`
- Print just a newline
  - `System.out.println();`
- Single Line Comments
  - `// comments here`
- Block Comments:
  -
```
/* comment  starts
   continues
   comment ends */
```

Primitive Data Types

Primitive	Wrapper	Size	Description
`byte`	`Byte`	1 byte	Store integers from -128 (-2^7) to 127 (2^7-1)
`short`	`Short`	2 bytes	Store integers from -32,768 (-2^{15}) to 32,767 (2^{15}-1)
`int`	`Integer`	4 bytes	Store integers from -2,147,483,648 (-2^{31}) to 2,147,483,647 (2^{31}-1)
`float`	`Float`	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits.
`double`	`Double`	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits.
`long`	`Long`	8 bytes	Store integers -9,223,372,036,854,775,808 (-2^{63}) to 9,223,372,036,854,775,807 (2^{63}-1)
`boolean`	`Boolean`	1 bit	Stores `true` or `false`
`char`	`Character`	2 bytes	Stores a single 16-bit Unicode character: `'M', '3', '!', 'é'`

Variables in Java

There are two types of variables in Java:

- **Primitive variables**
  - These hold primitive data types like `ints`, `doubles`, and `booleans`.
  - Autoboxing and Auto-unboxing can automatically convert a primitive to its corresponding Wrapper object and back
- **Object or Reference variables**
  - These hold a reference to an Object. For example, `Strings`. A reference is a way to find the object (like the tracking number on a package).

To create a variable you need its data type and name - This is called declaring a variable

```
type variableName;
```

Most Java coding-standards prefer **camelCase** when naming variables.

Variables in Java

The assignment operator in Java is a single equals sign

```
a = b; // assign b to a
```

You can combine variable declaration and assignment

```
type variableName = value;
```

You can use the final keyword with this form to create a **constant** that cannot be changed

```
final type VARIABLE_NAME = value;
```

Most Java coding standards prefer `ALL_CAPS_WITH_SNAKE_CASE` when naming **constant** variables.

Relational Operators

Operator	Meaning	Example
`==`	Equal To	`a == b`
`!=`	Not Equal To	`age != 21`
`>`	Greater Than	`average > 30`
`<`	Less Than	`grade < 60`
`>=`	Greater Than or Equal	`age >= 18`
`<=`	Less Than or Equal	`height <= 6`

Arithmetic Operators

Operator	Meaning	Example
`+`	Addition	`3 + x`
`-`	Subtraction	`p - q`
`*`	Multiplication	`6 * i`
`/`	Division	`10 / 4`
`%`	Mod (remainder)	`11 % 3`

Compound Assignment Operators

Operator	Example	Compound Assignment Operator	Example
`+`	`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`

Increment / Decrement Operators

Operator	Equivalent 1	Equivalent 2	Example
`post-increment`	`x = x + 1`	`x += 1`	`x++`
`post-decrement`	`y = y - 1`	`y -= 1`	`y--`

Note: *pre-increment and pre-decrement operators are also available (not on the AP exam)*

```
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
```

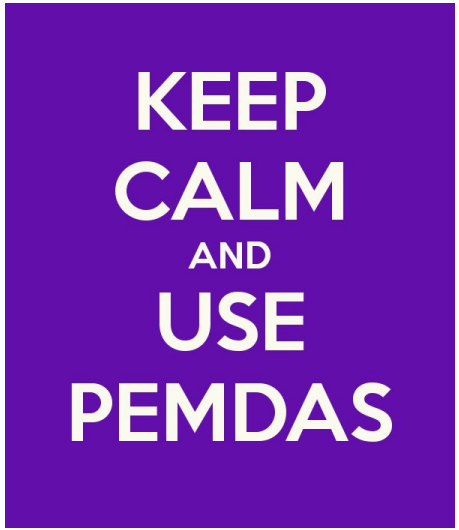
Order of Operations

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

Java evaluates expressions from left to right and from top to bottom



Operator Precedence

Java has an order in which it evaluates operators, just like PEMDAS tells you to multiply/divide before adding/subtracting when doing math.

So `x+2 < y+3` and `(x+2) < (y+3)` are equivalent because + has higher precedence than <.

When in doubt, use parentheses, and sometimes it's best to add parentheses to make code more readable.

Operator Precedence

Operators	Precedence
postfix	*expr*++ *expr*--
unary	++*expr* --*expr* +*expr* -*expr* ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Division

- Integer Division
  - Always results in an integer (no decimal; rounded down)
    - `10 / 2 = 5`
    - `15 / 2 = 7`
    - `19 / 10 = 1`
    - `3 / 10 = 0`
- Double Division
  - Any double in the expression will cause the result to be a double
    - `10.0 / 2 = 5.0`
    - `15.0 / 2 = 7.5`
    - `19.0 / 10 = 1.9`
    - `3.0 / 10 = 0.3`
- Average of Integers
  - `(3 + 3 + 11) / 3 = 5` **(eh...)**
  - `(3 + 3 + 11) / 3.0 = 5.667` **(...probably the value you want)**

String Concatenation

- Same Symbol as Addition (+)
  - Any string in the expression will cause the result to be a string
    - `"hello " + "friend" == "hello friend"`
    - `"your score: " + 100 == "your score: 100"`
    - `(78.6 + 10) + " degrees" == "88.6 degrees"`
    - `"" + true == "true"`
- Comparing Strings
  - `String.compareTo()` or `String.equals()` should be used when comparing String objects when you are interested in comparing the **contents of the String** (usually the case)
    - `bobStr.compareTo("bob")`
    - `catStr.equals(dogStr)`
  - Using the equals operator `(==)` to compare two String objects will determine if the Object **references are the same** (the variables refer to the **same instance** of the String Object)

Type Promotion and Type Casting

- We have already seen that if an expression combines
  - An `int` and a `double` - the result will be a `double`
  - A `String` and a number - the result will be a `String`
  - This is called **Type Promotion** - and is done automatically by Java to ensure (as much as possible) that you do not accidentally lose precision with your variables
- However you can override this in Java by explicitly defining the types you want to use (even if it means losing precision)
  - This is called **Type Casting** and the syntax is `(type) variableName;`
  - `(int) 4.29 // equals 4`
  - `(double) 1 // equals 1.0`
  - We will see more of this operator later when we talk about **OOP** and **Inheritance**

Classes and Objects (the basics)

- A **Class** in Java is a blueprint for an Object.
  - It describes the data it will contain - `static` and **instance variables** - a.k.a **attributes** or **properties**) and the operations (**methods**) it supports
  - There is only one **Class** for a specific type and they do not make use of any program resources
- You create an **instance** of a **Class** directly via the **`new`** keyword (or indirectly via the automatic construction that happen for **Strings** or for **primitive** types to their **Wrappers**).
  - There can be many instances of a **Class** and each instance consumes program resources
- Java provides **thousands** of **Classes** as part of its standard library
- Classes are equivalent to **types** in Java - so a variable that has an Object type can refer to an instance of that Object - uninitialized Object type variables are initialized to **`null`** - Which will throw **`NullPointerException`** if you try and use it!

`type variableName = value;`	`Planet mars = new Planet("mars");` `Planet jupiter; // null`

Primitive Wrappers

Primitive	Wrapper
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`float`	`Float`
`double`	`Double`
`long`	`Long`
`boolean`	`Boolean`
`char`	`Character`

- Java provides Wrappers for all the primitive types to better facilitate passing information around the Java library (which heavily relies on things being Object types)
- The Primitive Wrapper classes are special in that they support **auto-boxing,** automatic construction of the objects (i.e., you don't need to call the **constructor**).

Equivalent Examples:

`Integer i = 5;`	`Integer i = new Integer(5);`
`Boolean b = true;`	`Boolean b = new Boolean(true);`
`Character c = 'A';`	`Character c = new Character('A');`
`Double d = 1.256;`	`Double d = new Double(1.256);`

Classes and Objects (the basics)

- Properties
  - Can be **`public`**, **`private`**, or **`protected`**; And **`static`**, or **`non-static`**
  - **`Static`** properties can be accessed like this
    - `ClassName.propertyName`
  - **`Non-Static`** properties (a.k.a **instance properties**) are accessed like this
    - `classInstance.propertyName`
- Methods
  - Can be **`public`**, **`private`**, or **`protected`**; And **`static`**, or **`non-static`**
  - Require a **`return type (`**can be **`void`** if nothing is returned)n
  - The same method name can occur more than once; But each version of the method must have a unique set of arguments. This is called **Method Overloading**.
  - **`Static`** methods can be invoked like this
    - `ClassName.staticMethod()`
  - **`Non-Static`** methods (a.k.a **instance methods**) are invoked like this
    - `classInstance.method()`

Classes and Objects (the basics)

```
class Planet {
  private String name;

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

  public getName() {
    return name;
  }

  public static int getNumPlanets() {
    return 8; // sorry Pluto
  }
}
```

```
Planet.getNumPlanets(); // 8

Planet p = new Planet("Mars");

p.getName(); // "Mars"
p.getNumPlanets(); // 8
```

Classes and Objects (the basics)

- **Method Parameters**
  - Methods can have zero or more parameters
  - We call them **Formal Parameters**
    - When referring to the method parameters in the class definition
      - `public void greet(String name)`
  - We call them **Actual Parameters**
    - When referring to the method parameters passed at runtime
      - `amy.greet("Ted")`
- **Method Overloading**
  - The same method name can occur more than once; But each version of the method must have a unique set of argument types (argument names and return types are ignored).
  - This is called Method Overloading.
    - `public void greet(){}`
    - `public void greet(String otherName){}`
    - `public void greet(int aNumber){}`

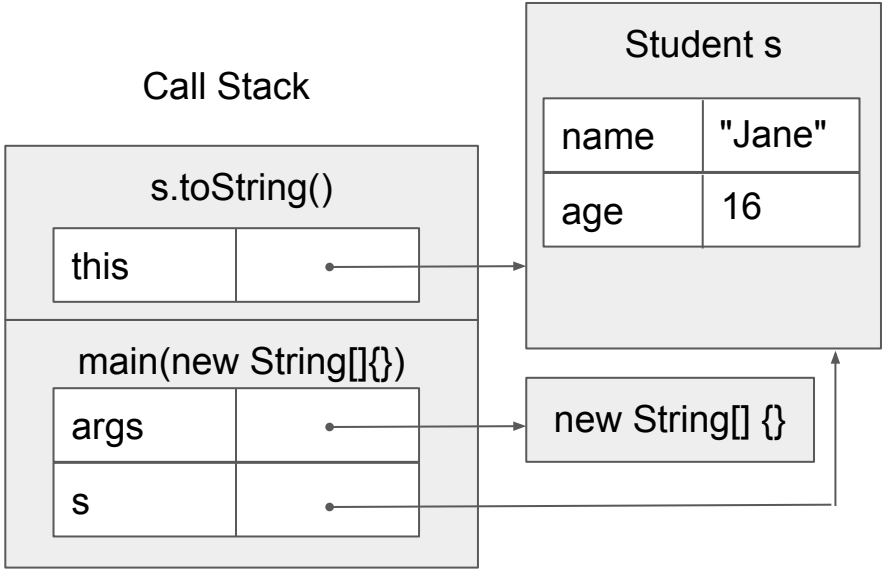
Classes and Objects (the basics)

- **Methods vs Functions**
  - A method represents an action supported by some class of object in Object-Oriented Programming (OOP). Java is an OOP language. Some programming languages have functions. Some have methods. Some have functions and methods!
  - **In Java, there are only methods; Every method belongs to a class.**
- **Returning From A Method**
  - Use the **`return`** keyword to return a value from a Method; Java only allows a single value to be returned from a Method (although as we will see later you could return an Array or ArrayList that itself is a single value - but contains multiple values)
  - Every Method that has a **`non-void`** return type must return a value; And that value must have the type specified as the return type in the Method signature; You can call **`return`** with no argument if the Method return type is **`void`** and it is useful for the flow of your program

VALID	INVALID (WHY?)

`public int getNumberTimesThree(int value) {`
`  return 3.0 * value;`
`}`
	`public int getNumberTimesThree(int value) {`
	`  return 3.0 * value;`
	`}`

`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?

The String Class

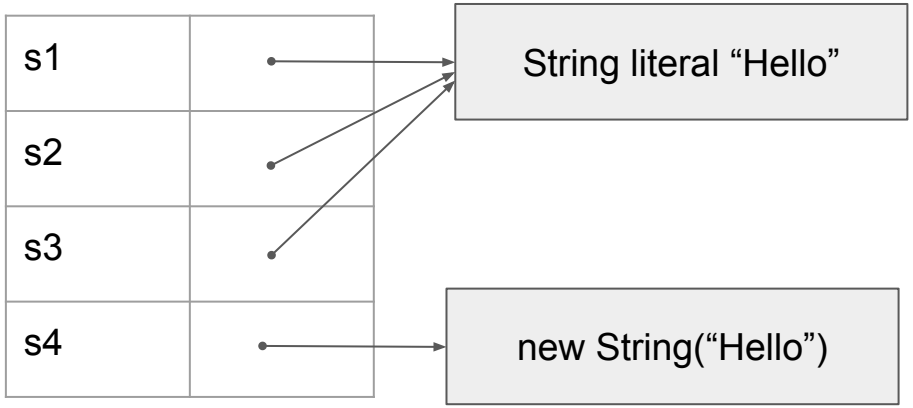
- Strings in Java are instances of the `java.lang.String` class that hold sequences of characters (a, b, c, $, etc.). The parent class of `java.lang.String` is `java.lang.Object`, which is the top-most parent class of all Java classes.
- Strings in Java are **IMMUTABLE** - they cannot change. If you want to make use of a **MUTABLE** string then you can use `java.lang.`**`StringBuilder`**
- Strings can be appended to each other to create a new string using the + or += operator. This is also called concatenation.
- 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`**

Main.java

```java
public class Main {
  public static void main(String args[]) {
    String s1 = "Hello";
    String s2 = "Hello";
    String s3 = s2;
    String s4 = new String("Hello");
    System.out.println("s1 == s2: " + (s1 == s2));
    System.out.println("s2 == s3: " + (s2 == s3));
    System.out.println("s1 == s4: " + (s1 == s4));
    System.out.println("s1.equals(s4): " + s1.equals(s4));
  }
}
```

Shell × Console ×

```
> sh -c javac -classpath .:target/dependency/* -d . $(find . -type f -name '*.java')
> java -classpath .:target/dependency/* Main
s1 == s2: true
s2 == s3: true
s1 == s4: false
s1.equals(s4): true
>
```



Java Language Standard 15.29:

Constant expressions of type String are always "interned" so as to share unique instances, using the method `String.intern`.

String Class Methods (some of them)

Return Type	Method Name	Formal Arguments	Example
`int`	`length`	`N/A`	`s.length()`
`String`	`substring`	`int from, int to`	`s2 = s1.substring(1,3)`
`int`	`indexOf`	`String str`	`idx = s.indexOf("cat")`
`int`	`compareTo`	`String str`	`iRes = s.compareTo("Bob")`
`boolean`	`equals`	`String other`	`b = s.equals("dog")`
`boolean`	`isEmpty`	`N/A`	`b = s.isEmpty()`

Remember: Like most "indexed" structures in Java
String indices start at zero and the last element index will be length-1

The Math Class

- Provides 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!

Math Class Methods (some of them)

Return Type	Method Name	Formal Arguments	Example
int	abs	int x	Math.abs(-123)
double	abs	double x	Math.abs(-123.25)
double	pow	double base, double exp	Math.pow(2,10)
int	sqrt	int x	Math.sqrt(16)
double	random	N/A	d = Math.random()

Write code that generates a random int between 0 to 9

```
int random = (int) (Math.random() * 10);
```

Write code that generates a random int between 1 and 10

```
int random = (int) (Math.random() * 10) + 1;
```

Control Flow: `if` statement

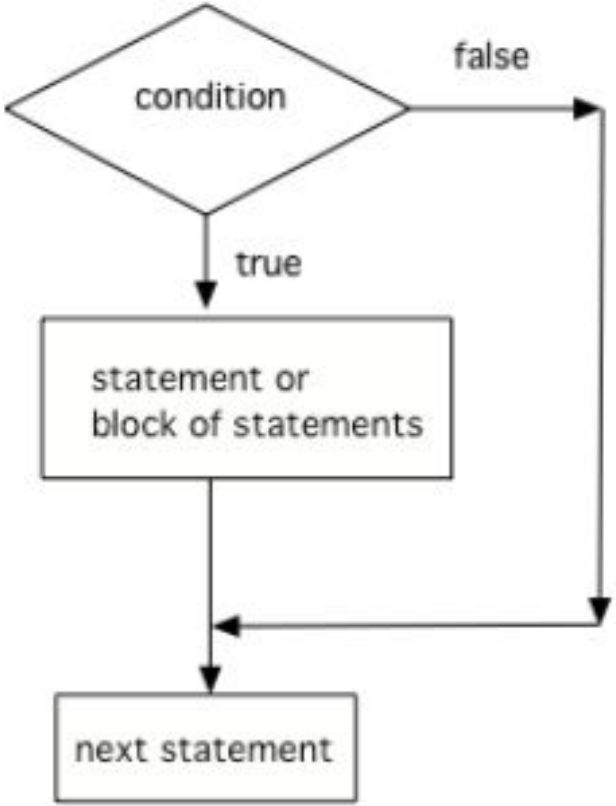
```
if (boolean expression) {
    statements
}

next statement
```

Curly braces are not required for a single statement.

(Curly braces around one statement is really a one-statement block.)

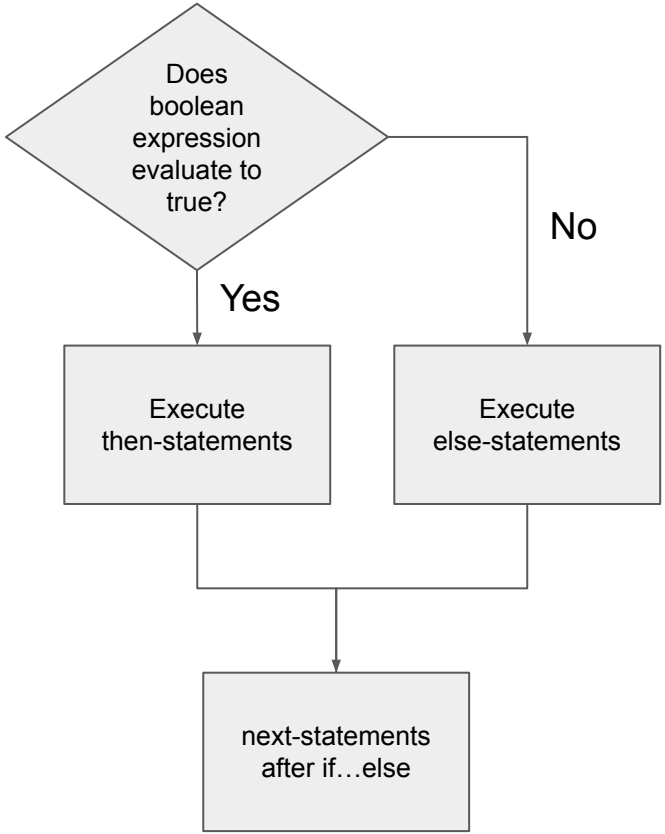
But we recommend it, and it's common to many Java coding standards.



Control Flow: `if-else` statement

```
// Condition
if (boolean expression) {
  then-statements
} else {
  else-statements
}

next-statements
```

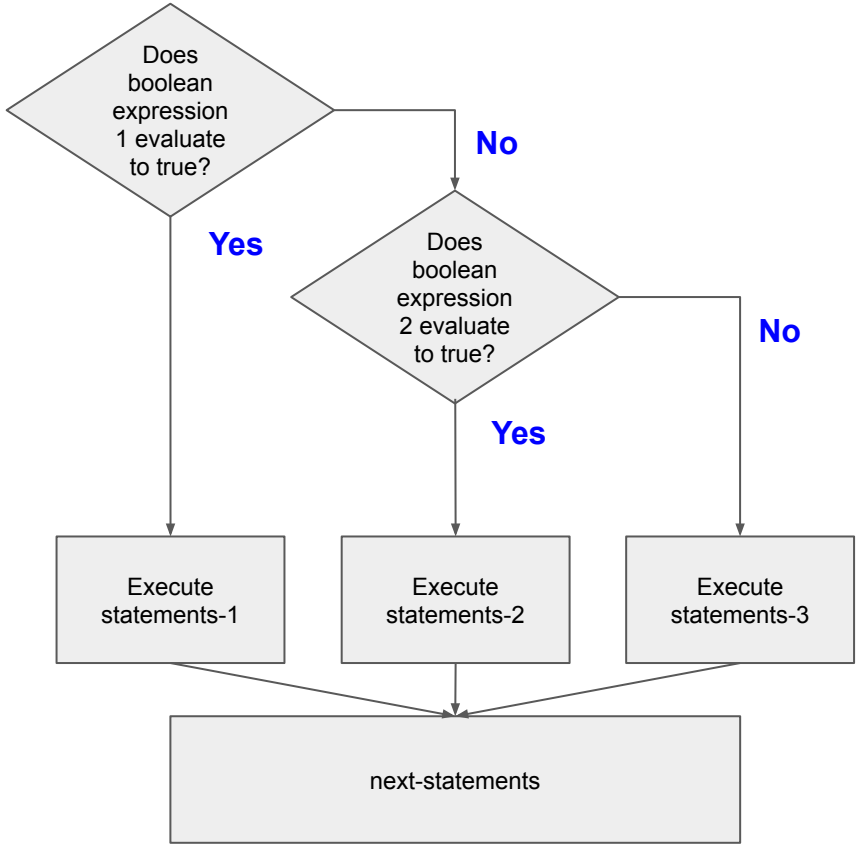


Control Flow: `else-if`

`if-else` statements can be chained together to handle many alternative cases.

```
if (boolean expression 1) {
  statements-1
} else if (boolean expression 2) {
  statements-2
} else {
  statements-3
}

next-statements
```



Control Flow: Nesting

Reminder: `if` statements and `if-else` statements are **statements**; so, they can become the statements inside other `if` or `if-else` **statements**! **Always double-check your curly-braces because spacing doesn't mean anything to the Java compiler!**

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

Dangling Else

```
int x = 0;
if (x >= 0)
    if (x > 0)
        System.out.println("x is positive");
    else
        System.out.println("x is negative");
```

is paired with this **if**

this **else**

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 `{ }`!

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 - ||

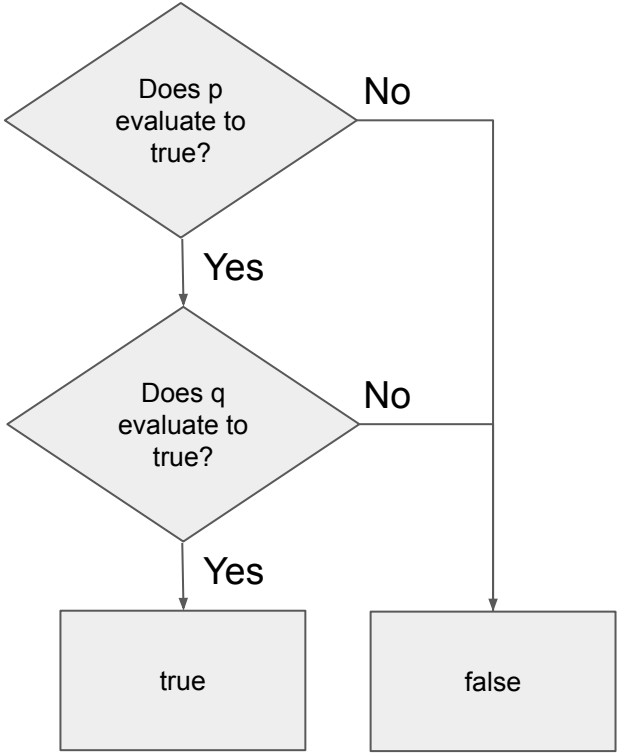
p	q	p || q
true	true	true
true	false	true
false	true	true
false	false	false

Truth Table - !

p	!p
true	false
false	true

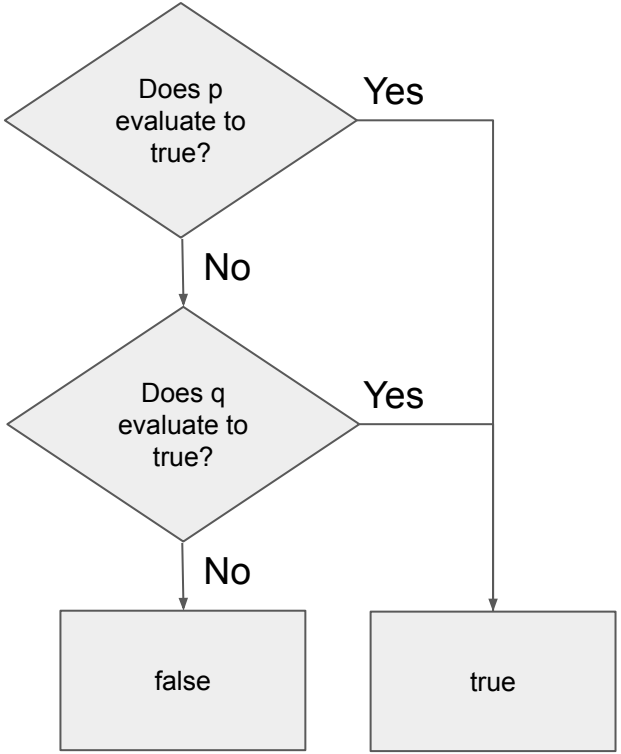
Short-Circuit Evaluation with &&

```
p && q
```



Short-Circuit Evaluation with ||

```
p || q
```



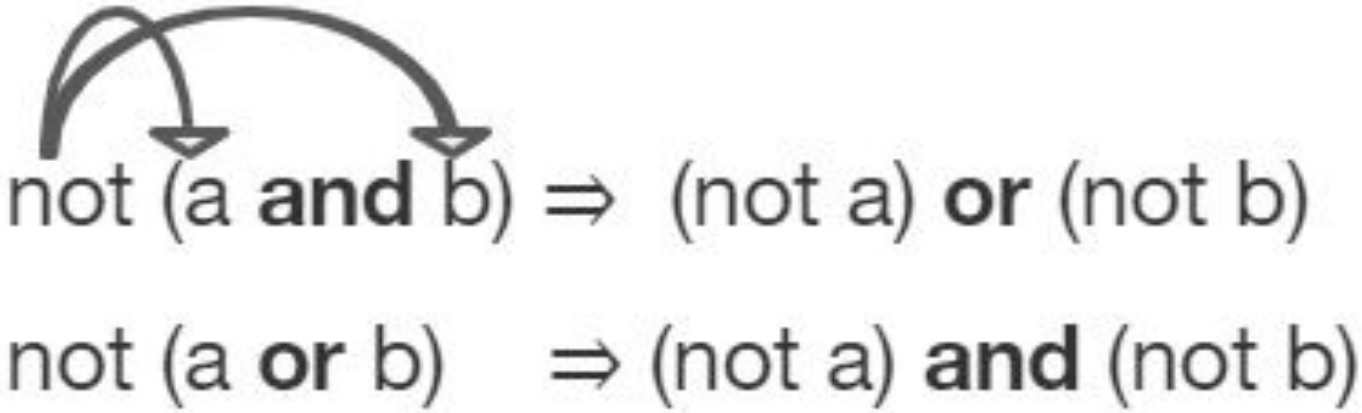
De Morgan's Laws

Augustus De Morgan (27 June 1806 – 18 March 1871) was a British mathematician and logician. He formulated De Morgan's Laws.

$$(P \wedge Q) \iff \neg(\neg P \vee \neg Q),$$
$$(P \vee Q) \iff \neg(\neg P \wedge \neg Q).$$

DeMorgan's Laws

Rules by which we can simplify Booleans to make them easier to read or interpret



DeMorgan's Laws

In Java:

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

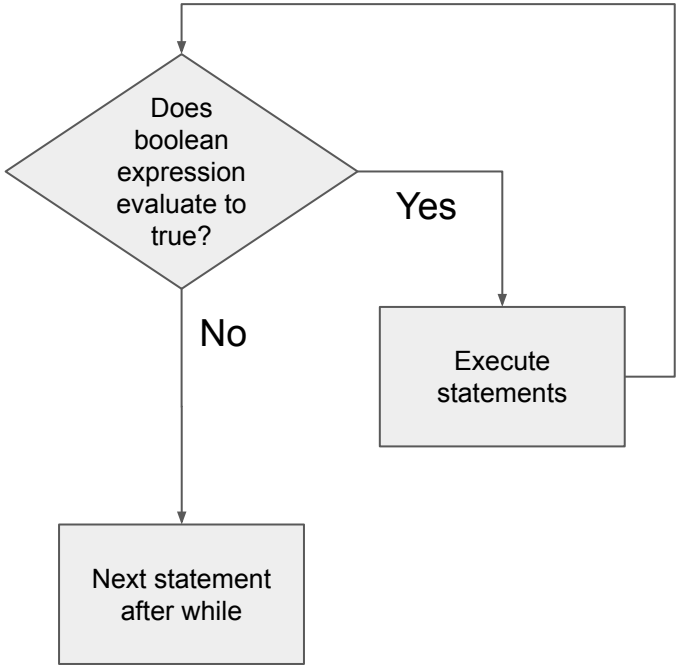
a	b	!(a && b)	!a || !b
T	T	F	F
T	F	T	T
F	T	T	T
F	F	T	T

Iteration and Looping: `while` statement

```
while (boolean expression) {
   statements
}
```

Example:

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

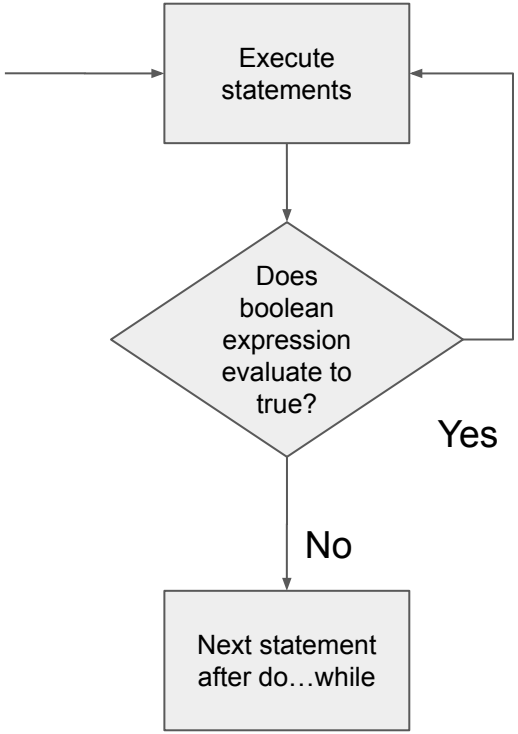


Iteration and Looping: `do-while` statement

```
do {
   statements
} while (boolean expression)
```

Example:

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

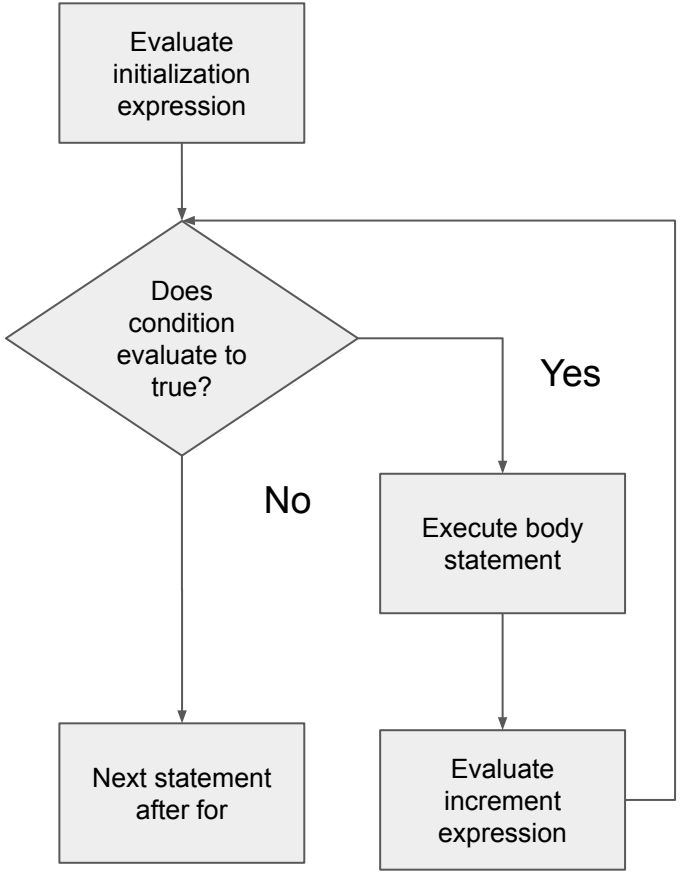


Iteration and Looping: `for` statement

```
for (initialization; condition; increment) {
   statements
}
```

Example:

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



Iteration and Looping

Infinite Loop	Occasionally Useful; But beware of the **accidental** Infinite Loop
	
	`while (true) {`
	`  handleNextRequest();`
	`}`

`break`
 	
`while`, `do-while`, `for`, `for-each`
	Exit the current loop
	
	`for (int i = 1; i <= 100; i++) {`
	`  if (55 == i) {`
	`  break;`
	`  }`
	`  System.out.println(i);`
	`}`

Iteration and Looping

`continue`
 	
`while`, `do-while`, `for`, `for-each`
	Jump to the beginning (or the end for `do-while`) and evaluate the condition (or the next item for `for-each`
	
	`for (int i = 1; i <= 100; i++) {`
	`  if (0 == i % 2) {`
	`  continue;`
	`  }`
	`  System.out.println(i);`
	`}`

`return`	Return immediately from the current method

AP CS FRQ 1

(25 minutes)

2022 AP Computer Science A - Free-Response Questions

Complete (1.A) and (1.B)

AP CS FRQ 1 - Review

(15 minutes)

Sample Responses and Scoring Commentary - FRQ-1