Additional Lectures for Units 1, 2 and 3

CSAwesome

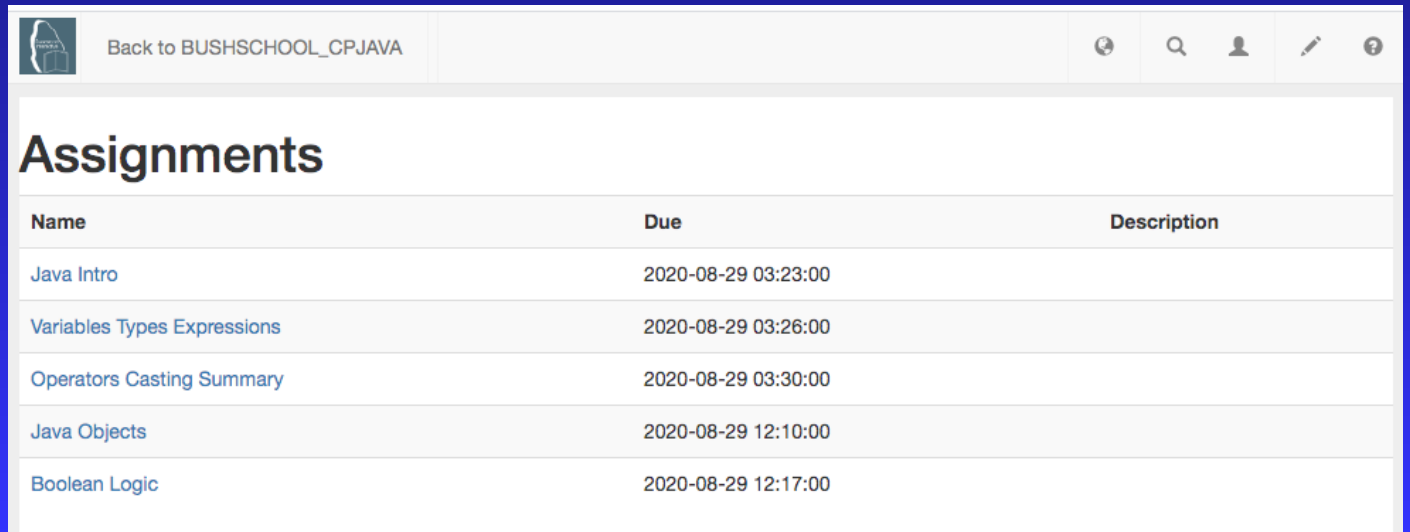
- Variables and Types
- Declarations and Initializations
- `%` (modulus) and integer division
- `if` statements, booleans
- Relational and Logical operators
- Random numbers
- + and `Strings`
- Methods (aka functions)

IMPORTANT

- Make sure you are signed in and the correct course is shown at the top left
- This ensures you get credit for your work!

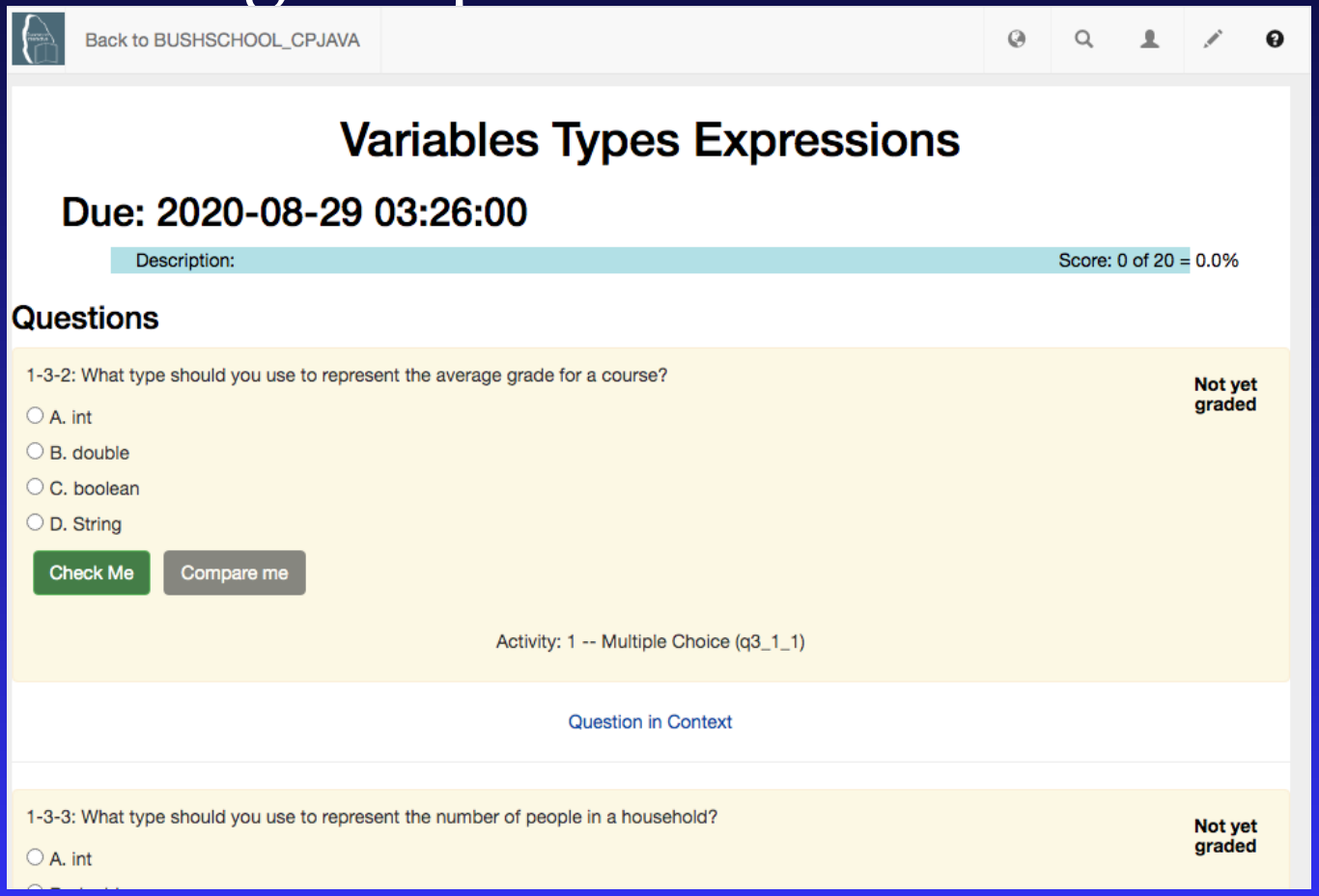
Assignments

- If you are logged in to the correct website, the link https://runestone.academy/runestone/assignments/chooseAssignment.html should bring up a list of your assignments



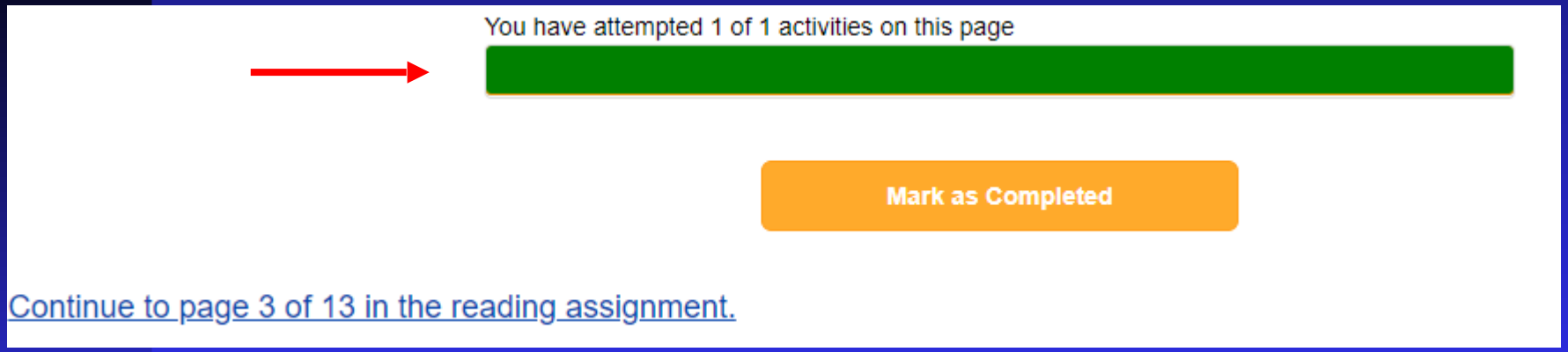
Assignments

- If you click the link, you should see only the assigned problems



Read the sections in order

- At the bottom of each page it will let you know if you have completed the required activities
- If you have, mark as completed and click the link to the next section



Unit1: Variables Types Expressions

Follow along in CSAwesome

- Kahoot in at the end of this section!

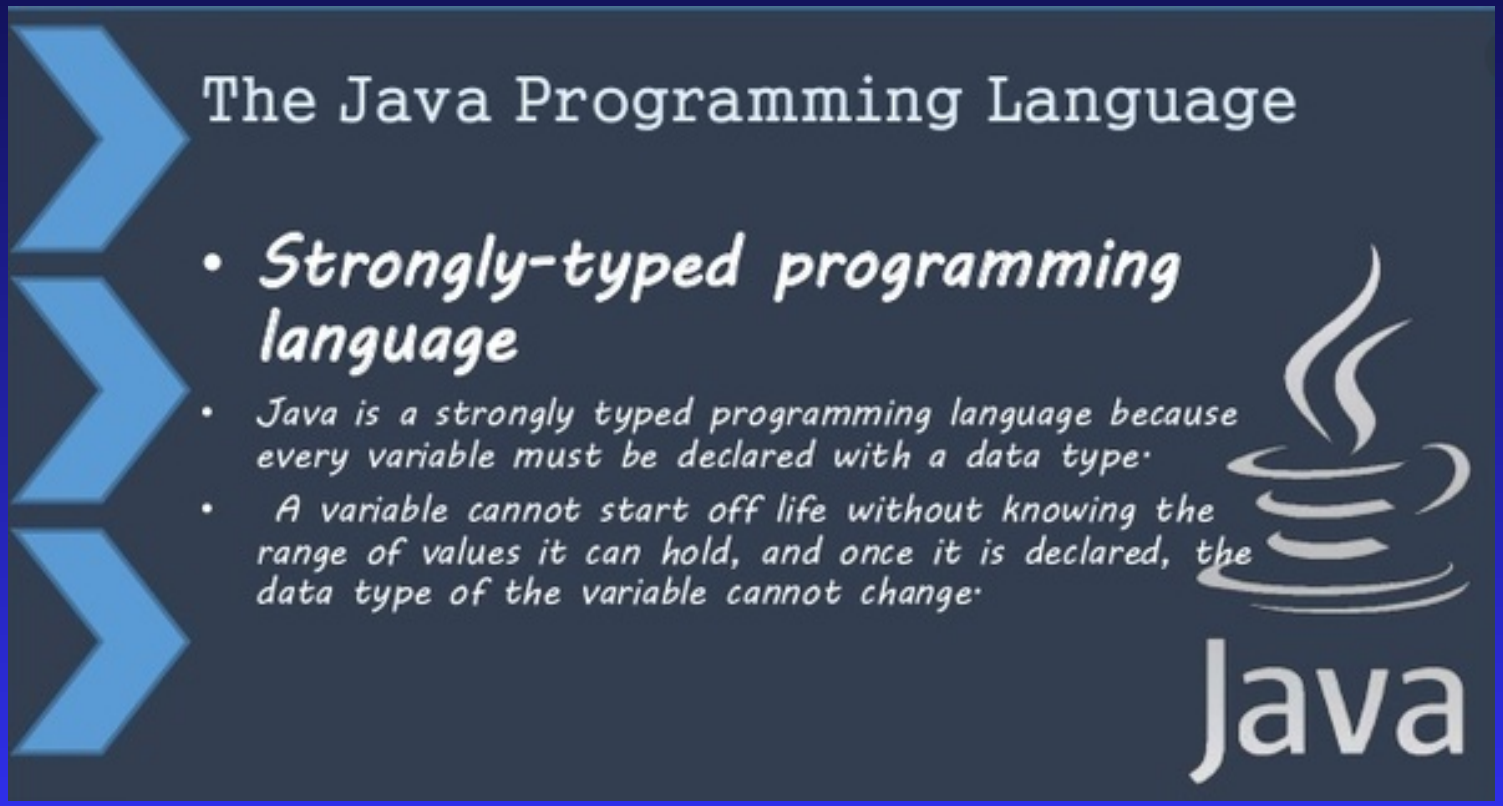
The parts of a basic program

- Java is picky!
- Capitalization matters
- Curly braces and semicolons will drive you crazy!

```
public class MyClass
{
    public static void main(String[] args)
    {
        System.out.println("Hi there!");
    }
}
```

Variables in Java have *types*

- This is different than JavaScript or Python



Variables: Parking space analogy

- Like a parking space, a variable can store a value (think ***vehicle***) until you need it later

Variables: Parking space analogy

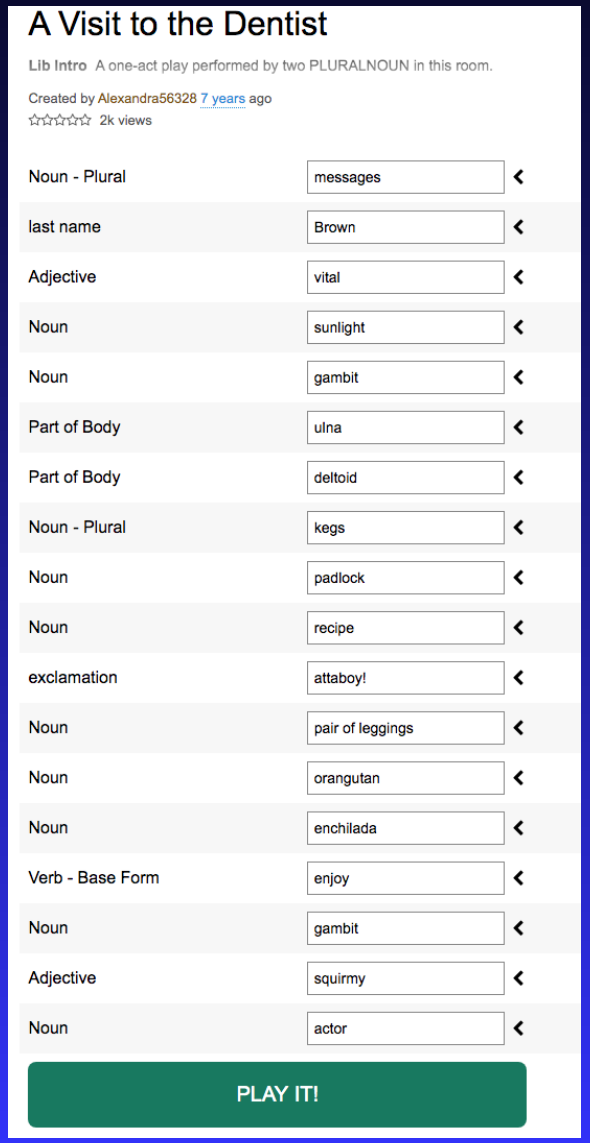
- Parking spaces are often labeled so you can find your car later when you need it
- Let's say that **`G210`** is the *name* of this parking space

Variables: Parking space analogy

- In addition to a *name*, parking spaces can have a *type* that sets size limits

Variables: MadLib analogy

- The blank in a MadLib is like a Java variable
- The blank is a space that can hold different words
- Each blank has a ***type:*** noun, verb, adjective or adverb
- Try it!



Madlib Hilarious Result!

A Visit to the Dentist

A one-act play performed by two messages in this room.

PATIENT : Thank you so very much for seeing me, Doctor Brown, on such vital notice.

DENTIST : What is your problem, young sunlight

PATIENT : I have a pain in my upper gambit, which is giving me a severe ulna ache.

DENTIST : Let me take a look. Open your deltoid wide. Good. Now I'm going to tap your kegs with my padlock.

PATIENT : Shouldn't you give me a an recipe killer

DENTIST : It's not necessary yet attaboy!! I think I see a an pair of leggings in your upper orangutan.

PATIENT : Are you going to pull my enchilada out

DENTIST : No. I'm going to enjoy your tooth and put in a temporary gambit.

PATIENT : When do I come back for the squirmy filling

DENTIST : A day after I cash your actor

Primitive data types

- In Java (unlike Python or JavaScript) variables have *types*
- Each type has size limits
- For this class, you need to know 5 basic (aka "primitive") types:
  - `int`
  - `float`
  - `double`
  - `boolean`
  - `char`

Primitive data types

- `int` holds a single integer value between -2,147,483,648 and 2,147,483,647
- `float` a decimal value with up to 7 digits
- `double` a decimal value with up to 15 digits
- A `boolean` can only hold values that evaluate to either `true` or `false`
- `char` holds a single letter, digit, space or punctuation mark and must be enclosed in *single quotes*, like this: `'G'`

A Java variable declaration

- The Java statement that creates a variable is called a ***declaration***
- Here's an example declaration

```
int num;
```

- The first word of the declaration is the type
- The second word is the name that the programmer choose
- The semicolon marks the end of a Java statement much like a period in English

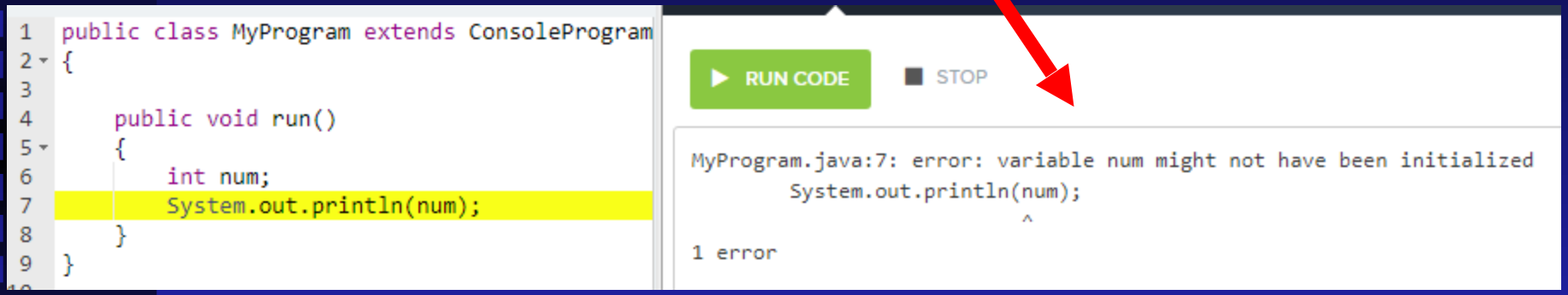
Variable names and keywords

- You can pretty much choose any name you want for a variable with a few limitations:
  - Variable names cannot
    - start with a number
    - contain a space
    - have a special meaning in Java
- Java has about 50 reserved words or ***keywords*** that you are not allowed to use as variable names such as **`public`**,

camelCase

- Because a Java variable name can't have spaces, a style called **camelCase** is usually used for variable names with more than one word
- **camelCase** capitalizes the first letter of each word except the first word
- Examples: **`firstName`**, **`numberOfDogs`**
- Java variable names are case-sensitive, so **`firstName`** is not the same as **`firstname`** or **`FirstName`**.

What's the error?



Variable initialization

- After a Java variable is declared, it needs to be *initialized*

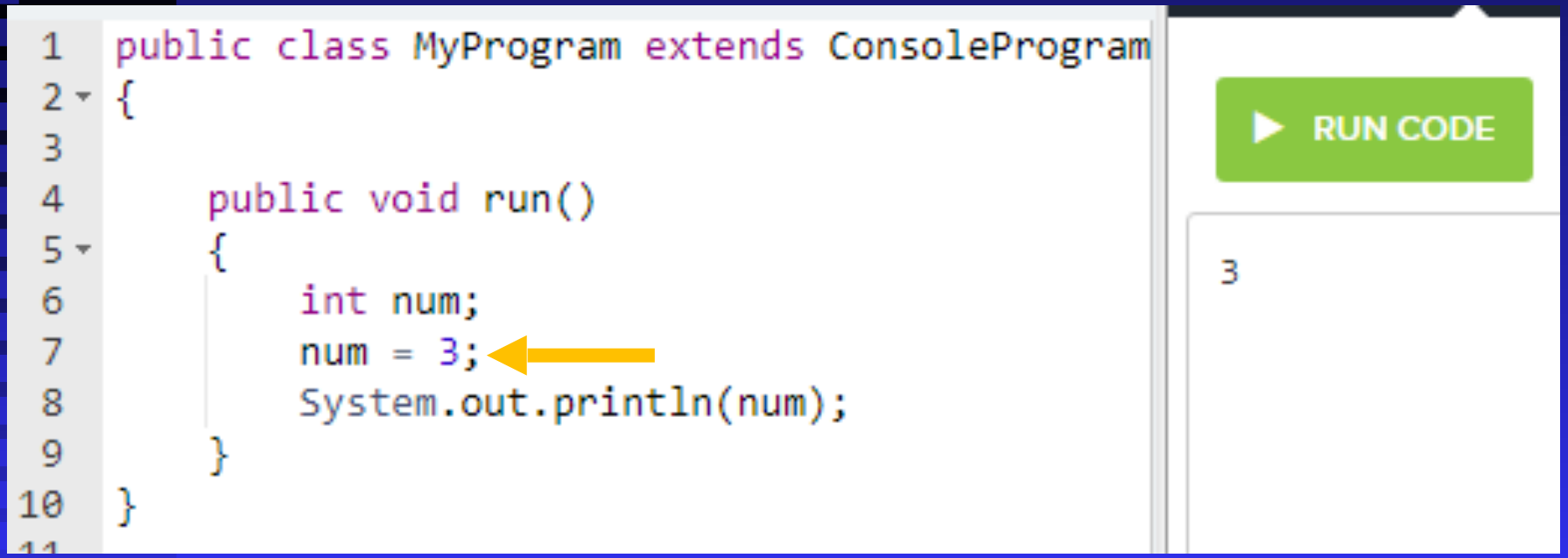
```
1  public class MyProgram extends ConsoleProgram
2  {
3
4      public void run()
5      {
6          int num;
7          System.out.println(num);
8      }
9  }
```

RUN CODE STOP

```
MyProgram.java:7: error: variable num might not have been initialized
        System.out.println(num);
                           ^
1 error
```

Variable initialization

- The English word "initial" means *first*
- We initialize a variable by assigning ("setting it equal to") its *first* value



Declare *and* initialize

- You can declare and initialize a variable with one line of code:

```
int num = 3;
```

- Just remember that the one line of code is doing two different steps: the ***declaration*** and the ***initialization***

`float` vs. `double`

- `float` is short for *floating point*, another name for *decimal point*
- `double` gets its name because of its size, it has about twice as many digits as a `float`

```
1  public class MyProgram extends ConsoleProgram
2  {
3
4      public void run()
5      {
6          double dNum = 5/9.0;
7          System.out.println(dNum);
8          System.out.println((float)dNum);
9      }
10 }
```

RUN CODE STOP

```
0.5555555555555556
0.5555556
```

What is the error?

```
1  public class Variables extends ConsoleProgram
2  {
3
4      public void run()
5      {
6          System.out.println(12345678901234567890);
7      }
8
9  }
```

Grader Results

CHECK CODE | EXPAND ALL | MINIMIZE ALL

Test	Pass	Message

Errors:

Variables.java: Line 6: integer number too large: 12345678901234567890

What is the error?

- We are trying to print an integer that is too large for Java's integer size

```
1  public class Variables extends ConsoleProgram
2  {
3
4      public void run()
5      {
6          System.out.println(12345678901234567890);
7      }
8
9  }
```

Grader Results

✔ CHECK CODE | EXPAND ALL | MINIMIZE ALL

Test	Pass	Message

Errors:

Variables.java: Line 6: integer number too large: 12345678901234567890

Assigning a value to a variable

- Changing the value of a variable is called an *assignment*
- We do this with the equals sign =
- In Java we call the (single) equals sign the *assignment operator*

```
1  public class MyProgram extends ConsoleProgram
2  {
3
4      public void run()
5      {
6          int num;
7          num = 3; //first assignment is called an "initialization"
8          System.out.println(num);
9          num = 5; //another assignment to change value to 5
10         System.out.println(num);
11     }
12 }
```

RUN CODE STOP

```
3
5
```

`String` variables

- A `String` variable can store text with any number of letters, digits, punctuation marks and spaces
- The beginning and end of the text is marked with double quotes (")

```
1  public class MyProgram extends ConsoleProgram
2  {
3
4      public void run()
5      {
6          String words = "I like Java";
7          System.out.println(words);
8
9      }
10 }
```

RUN CODE

```
I like Java
```

What is the error?

- The types **`String`** and **`int`** don't match

```
1 public class MyClass
2 {
3    public static void main(String[] args)
4    {
5        String s = 5;
6    }
7 }
```

Error

```
MyClass.java:5: error: incompatible types: int cannot be converted to String
       String s = 5;
                  ^
1 error
```

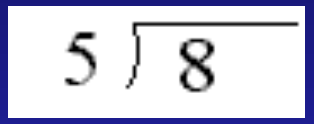
Arithmetic operators

`+ - * / %`

- Addition
- Subtraction
- Multiplication
- Division
- Modulus (also called *Mod* or *Modulo*) calculates the **remainder** of dividing two ***integers***
- ***It turns out that Modulus is extremely useful in loops and iterations!***

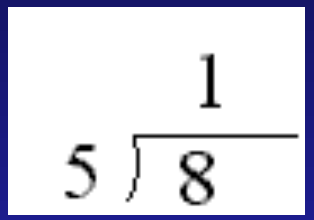
Modulus and integer division

Remember how you did math in grade school?



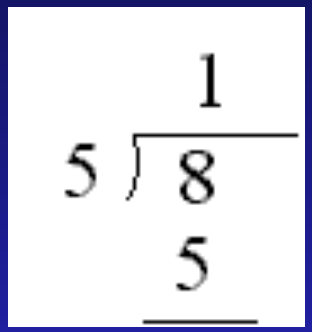
Modulus and integer division

Remember how you did math in grade school?



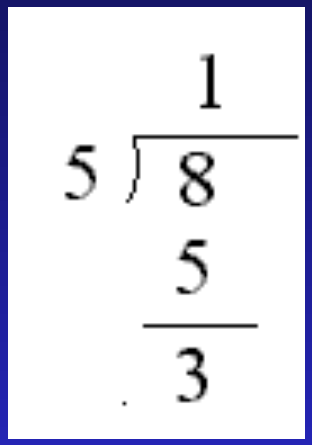
Modulus and integer division

Remember how you did math in grade school?



Modulus and integer division

Remember how you did math in grade school?

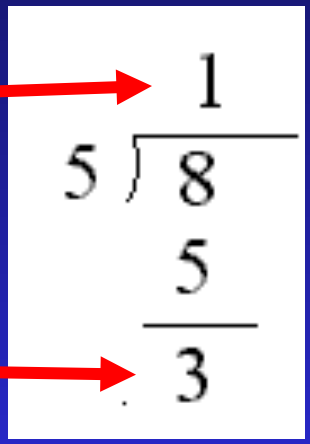


Modulus and integer division

The modulus operator gives the remainder of an integer division expression

8/5

8%5



integer vs. decimal division

- Java does division differently depending on the types of numbers
- When the types are mixed, the Java uses the larger type

Hello.java

```
1  public class Hello {
2
3      public static void main(String[] args) {
4          System.out.println(9/5);
5          System.out.println(9.0/5);
6          System.out.println(9/5.0);
7          System.out.println(9.0/5.0);
8      }
9  }
```

Power

```
1
1.8
1.8
1.8
```

Modulo Clock Widget

Evaluate these modulus expressions in Java. You may find the Modulo Clock Widget at www.yellkey.com/another helpful

1. 27 % 5
2. 9 % 4
3. 21 % 2
4. 5 % 8

Modulo Clock Widget

- `27 % 5`

Modulo Clock Widget

- 27 % 5

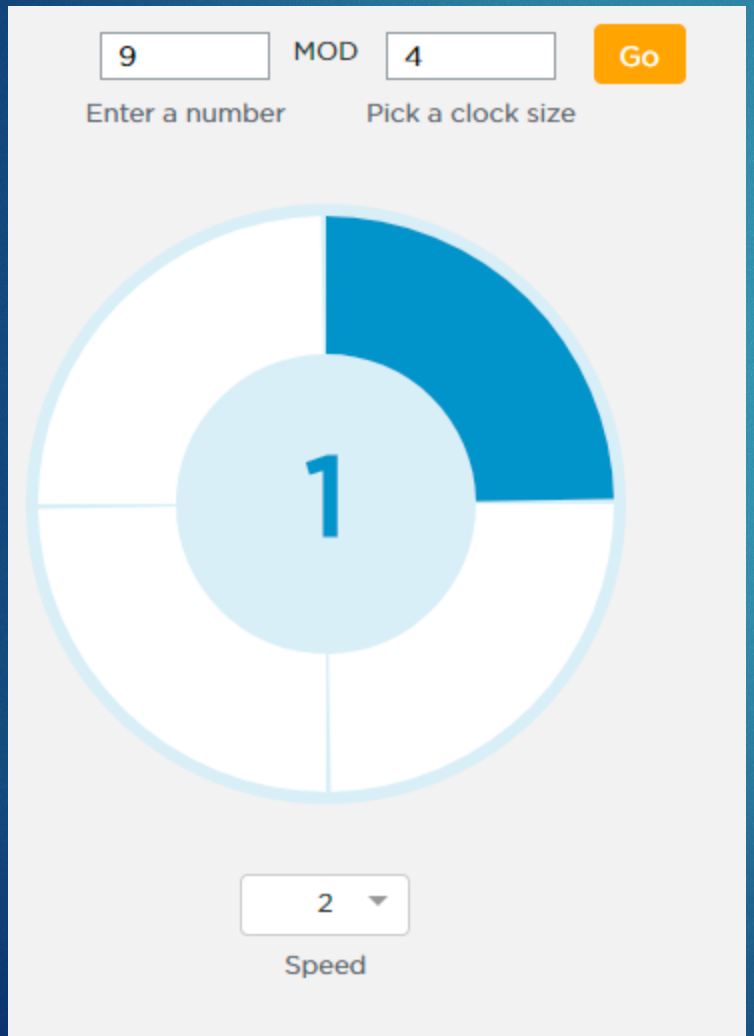
$$
\begin{array}{r}
5 \\
5\overline{)27} \\
25 \\
\hline
2
\end{array}
$$

Modulo Clock Widget

- `9 % 4`

Modulo Clock Widget

- 9 % 4



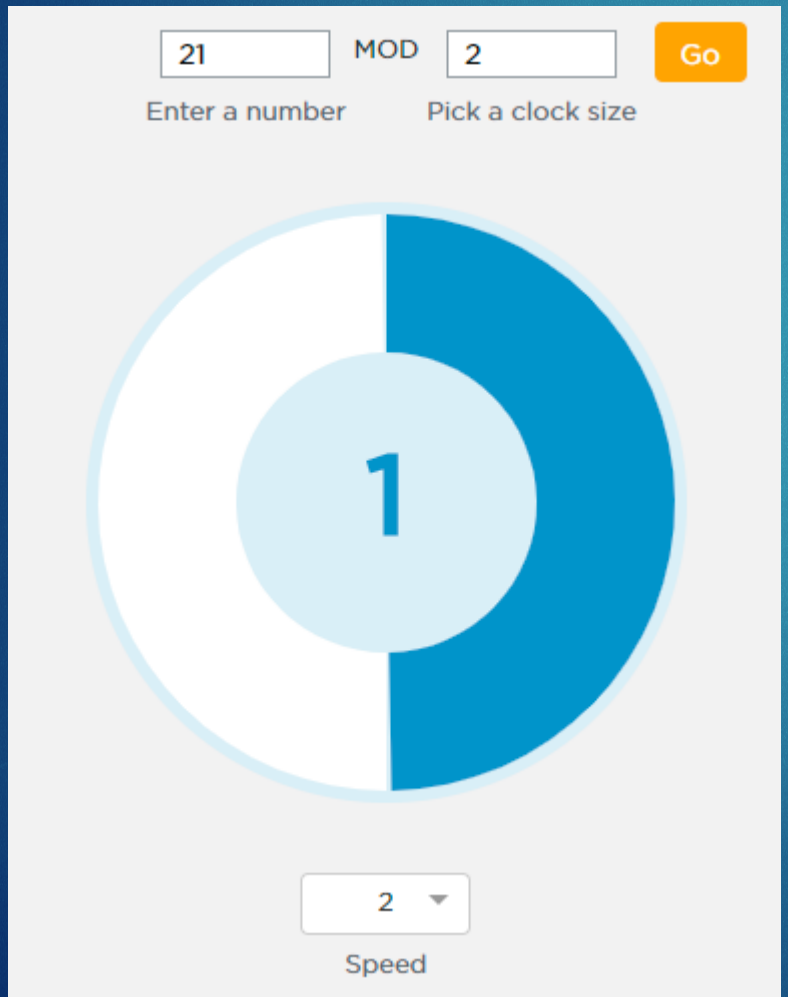
$$
\begin{array}{r}
2 \\
4\overline{)\,9} \\
\underline{8} \\
1
\end{array}
$$

Modulo Clock Widget

- `21 % 2`

Modulo Clock Widget

- `21 % 2`



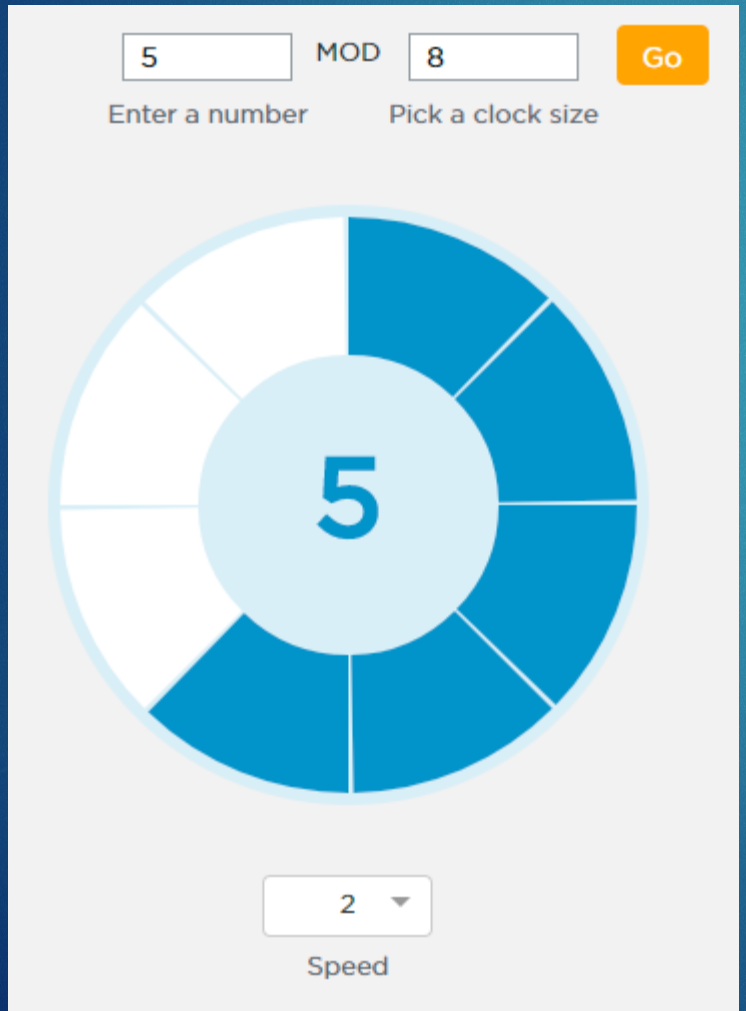
$$
\begin{array}{r}
10 \\
2\overline{)21} \\
\underline{20} \\
1
\end{array}
$$

Modulo Clock Widget

- `5 % 8`

Modulo Clock Widget

- 5 % 8



$$
\begin{array}{r}
0 \\
8\overline{)\,5} \\
0 \\
\hline
5
\end{array}
$$

Modulo Clock Widget

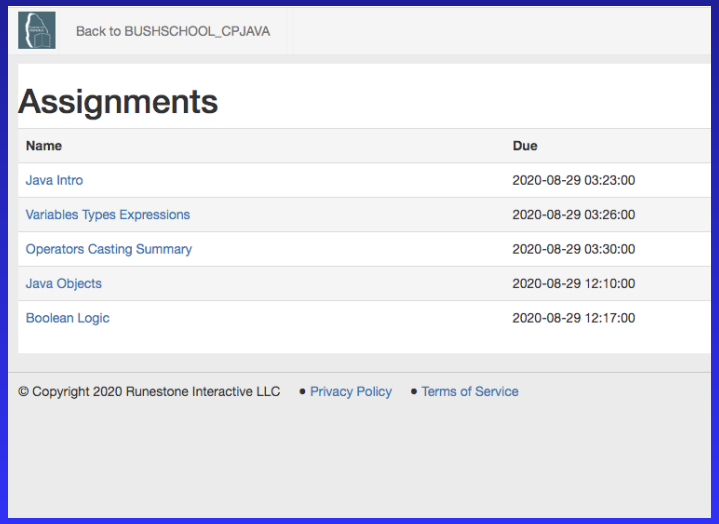
- Now see if you can answer these questions on your own *without using the widget*
- HINT: all of these are kind of easy if you understand how Modulo works - don't be fooled by the big number on the first one
  - 24 % 817234
  - 24 % 23
  - 15 % 4


Kahoot!
Integer Division in Java

Do CSAwesome Assignments

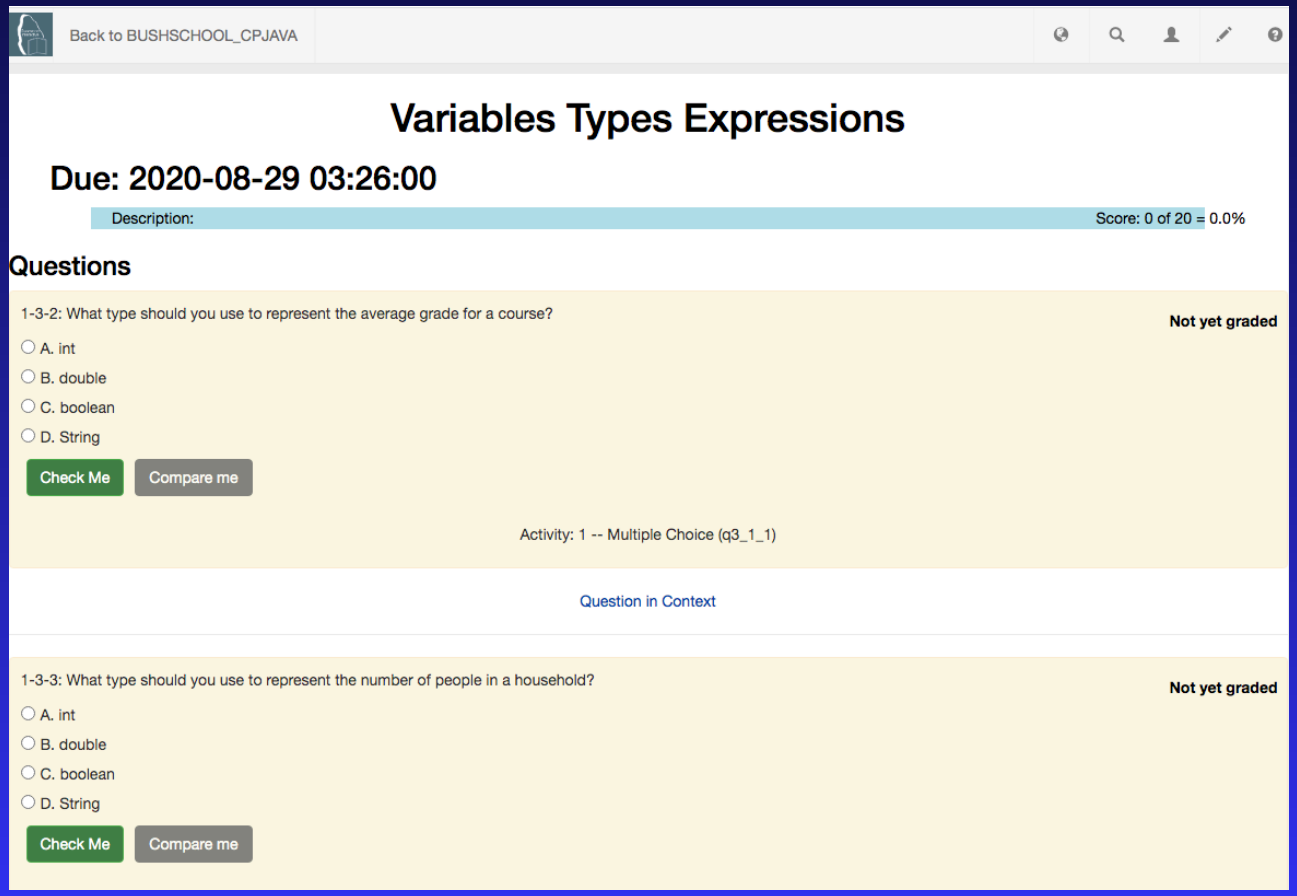
You only need to do the assigned problems

- If you are logged in to the correct website, the link https://runestone.academy/runestone/assignments/chooseAssignment.html should bring up a list of your specific assignments



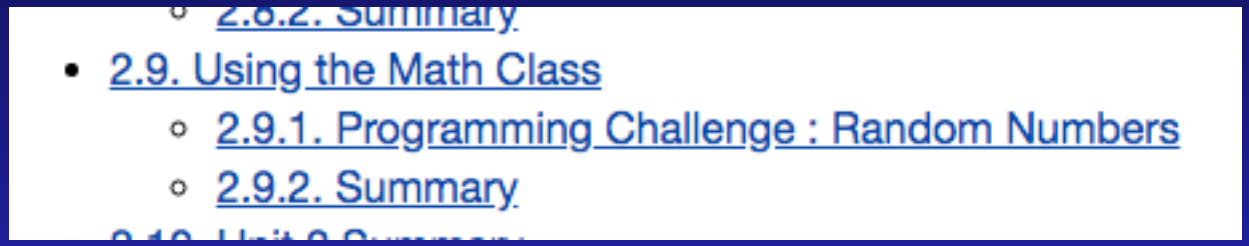
Assignments

- If you click a link, you should see only the assigned problems

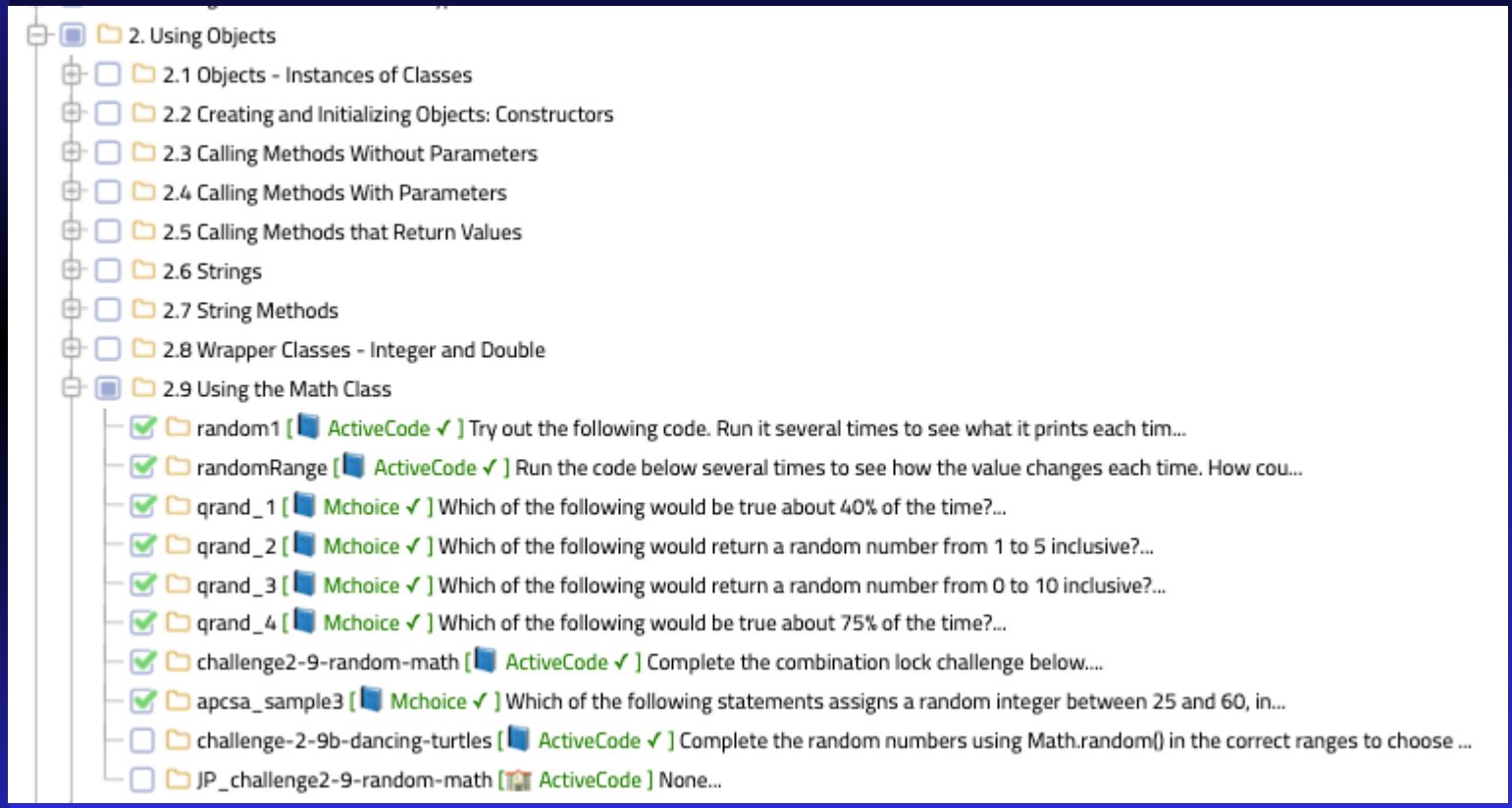


Unit2:Math.Random

Follow along in CSAwesome



Unit 2: Using Objects



`Math.random()`

- The AP exam uses **`Math.random()`** to generate random decimals
- **`Math.random()`** returns a **`double`** between 0 and 1 (including 0 but not 1)
- While there are other ways to make random numbers in Java, you will only be tested on **`Math.random()`**

Math.random()

- Notice that each call to Math.random() produces a different random decimal

```
1  public class MathChallenge
2  {
3     public static void main(String[] args)
4     {
5          System.out.println(Math.random());
6          System.out.println(Math.random());
7          System.out.println(Math.random());
8          System.out.println(Math.random());
9          System.out.println(Math.random());
10         System.out.println(Math.random());
11    }
12 }
13
```

```
0.06957121659941135
0.04700519712860429
0.5576605675173386
0.640005141147858
0.723164661431323
0.08066896566707715
```

Cast

- Java is *strongly typed*
- This means it's very strict about data types
- It won't let you do store a decimal in an `int` variable for example

```
int nNum = 5.6;
```

- To fix it you need a *cast*:

```
int nNum = (int)5.6;
```

- The cast tells Java just this once, *treat it as if it were an* `int`

`float` vs. `double`

- Java has two types of decimals: **`float`** and **`double`**
- **`double`** got its name because it can store roughly 2x as many digits as **`float`**
- For this class you will find that we need to know both

(float)Math.random()

```
1  public class MathChallenge
2  {
3     public static void main(String[] args)
4     {
5          System.out.println((float)Math.random());
6          System.out.println((float)Math.random());
7          System.out.println((float)Math.random());
8          System.out.println((float)Math.random());
9          System.out.println((float)Math.random());
10         System.out.println((float)Math.random());
11    }
12 }
13
```

```
0.64605314
0.20850246
0.14422274
0.92159355
0.21545891
0.47260916
```

(int)(Math.random()*2)

- A random **int** that is either 0 or 1

```
1  public class MathChallenge
2  {
3     public static void main(String[] args)
4     {
5         System.out.println((int)(Math.random()*2));
6         System.out.println((int)(Math.random()*2));
7         System.out.println((int)(Math.random()*2));
8         System.out.println((int)(Math.random()*2));
9         System.out.println((int)(Math.random()*2));
10        System.out.println((int)(Math.random()*2));
11    }
12 }
13
```

```
1
1
0
0
1
0
```

(int)(Math.random()*7)

- A random **int** in the range {0,1,2,3,4,5,6}

```
1  public class MathChallenge
2  {
3     public static void main(String[] args)
4     {
5          System.out.println((int)(Math.random()*7));
6          System.out.println((int)(Math.random()*7));
7          System.out.println((int)(Math.random()*7));
8          System.out.println((int)(Math.random()*7));
9          System.out.println((int)(Math.random()*7));
10         System.out.println((int)(Math.random()*7));
11    }
12 }
13
```

```
2
4
0
5
6
6
```

Watch out for this mistake!

- Why is the output always zero?

```
1  public class MathChallenge
2  {
3     public static void main(String[] args)
4     {
5          System.out.println((int)Math.random()*2);
6          System.out.println((int)Math.random()*2);
7          System.out.println((int)Math.random()*2);
8          System.out.println((int)Math.random()*2);
9          System.out.println((int)Math.random()*2);
10         System.out.println((int)Math.random()*2);
11    }
12 }
13
```

```
0
0
0
0
0
0
```

Watch out for this mistake

- `(int)Math.random()*2` always evaluates to zero because Java works left to right just like math
- The yellow cast is applied before the green multiplication

Watch out for this mistake

- `(int)Math.random()*2`
- Since `Math.random()` is cast as an `int` before being multiplied, the result is always zero
- The fix:
- `(int)(Math.random()*2)`

Problem: Generate a random integer between 0 and 101 (including 0 up to but not including 101)

- First we need a random decimal:

```
Math.random() * 101
```

- Gives a random decimal between 0.0 and 100.9999. . .

```
(int)(Math.random()*101)
```

- Gives a random integer between 0 and 100 inclusive

Some of the hardest `Math.random()` problems are when you need a random integer in a range that doesn't start with zero. For example: Create a random integer in the range {3,4,5,6,7,8,9,10}

Problem: Create a random integer in the range {3,4,5,6,7,8,9,10}

- First, we'll solve a simpler problem—subtract 3 so we start at zero

Problem: Create a random integer in the range {3,4,5,6,7,8,9,10}

- First, we'll solve a simpler problem—subtract 3 so we start at zero
- That, is, a random integer from *0 to 8 (including 0 but not 8)*
- `(int)(Math.random() * ??)`

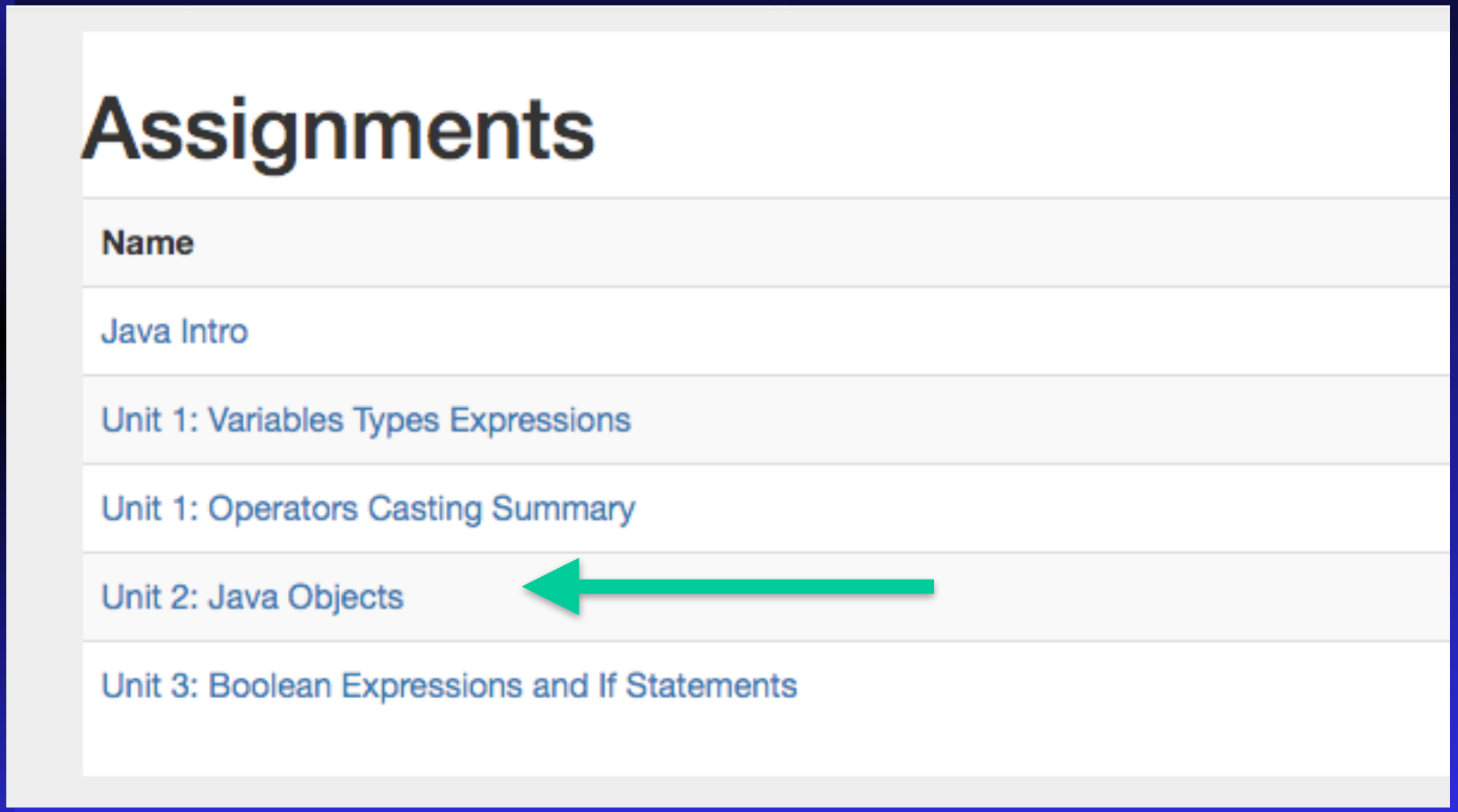
Problem: Create a random integer in the range {3,4,5,6,7,8,9,10}

- First, we'll solve a simpler problem—subtract 3 so we start at zero
- That, is, a random integer from *0 to 8 (including 0 but not 8)*
- `(int)(Math.random() * 8)`

Problem: Create a random integer in the range {3,4,5,6,7,8,9,10}

- Now, add 3 to shift the range back up
- `(int)(Math.random() * 8) + 3`

Do CSAwesome Assignment



Unit 3: booleans, logic and if statements

3. Boolean Expressions and If Statements

Question in context

- If you need more explanation click Question in context

Question in context

- That will take you to the part of the course where that topic is explained

3.1. Boolean Expressions

Boolean variables or expressions can only have true or false values.

3.1.1. Testing Equality (==)

Primitive values like ints and reference values like Strings can be compared using the operators == and != (not equal) which return boolean values.

Note

One = sign changes the value of a variable. Two == equal signs are used to test if a variable holds a certain value, without changing its value!

Watch the following video which shows what happens in memory as primitive types like int and reference types like Dog are compared with == in a physical model of Java memory.

CS Awesome: 3.1 Equal Equal (==) with ints and o...

Watch later Share

```
public static void main(String[] args)
{
    int x = 15;
    int y = 44;
    int z = 44;
    Dog fido = new Dog();
    Dog oreo = new Dog();
    Dog friend = oreo;
    System.out.println(x == y);
    System.out.println(y == z);
    System.out.println(fido == oreo);
    System.out.println(oreo == friend);
}
```

Activity: 1 -- Video: (bO9beJT0jwE)

Coding Exercise

What will the code below print out? Try to guess before you run it! Note that 1 equal sign (=) is used for assigning a value and 2 equal signs (==) for testing values.

Save & Run | Load History | Show CodeLens | Share Code

```
public class BoolTest1
{
  public static void main(String[] args)
  {
    int age = 15;
    int year = 14;
    // Will this print true or false?
    System.out.println( age == year );
    year = 15;
    // Will this print true or false?
    System.out.println( age == year );
    // Will this print true or false?
    System.out.println( age != year );
  }
}
```

Activity: 2 -- ActiveCode (bool1)

We can also use == or != to test if two reference values, like Turtle and String objects, refer to the same object. In the figure below, we are creating two separate Turtle objects called juan and mia. They do not refer to same object or turtle. Then, we create a reference variable called friend that is set to mia. The turtle mia will have two ways (**references** or **aliases**) to name her -- she's both mia and friend, and these variables refer to the same object (same Turtle) in memory. If two reference variables refer to the same object like the turtle on the right in the image below, the test with == will return true which you can see in the code below.

Figure 1: Turtle Reference Equality

`boolean` variables

- **`boolean`**s are Java's simplest and smallest primitive type
- **`boolean`**s can only store **`true`** or **`false`**, nothing else
- Here I'm getting an error because I'm trying to assign an integer to a **`boolean`** variable

```
1 public class MyClass
2 {
3    public static void main(String[] args)
4    {
5        boolean isBig = false;
6        boolean isSmal = true;
7        boolean howBig = 30;
8    }
9 }
```

Error

```
MyClass.java:7: error: incompatible types: int cannot be converted to boolean
       boolean howBig = 30;
                        ^
1 error
```

`boolean` vs. `String`

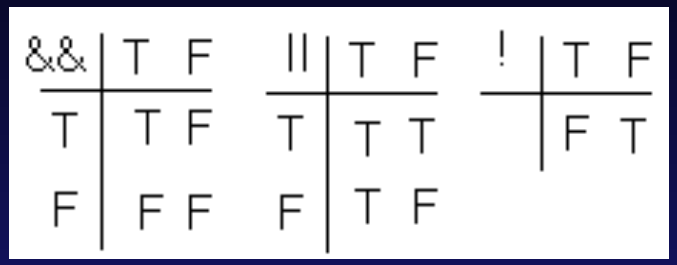
- **`true`** and **`false`** are **`boolean`**s
- **`"true"`** and **`"false"`** are **`String`**s
- Beginning Java programmers are often confused by the difference
- To Java, anything in double quotes is a **`String`** which, to Java, is completely different than a **`boolean`**

Logical operators

&& || !

- And
- Or
- Not
- Logical operators are sometimes also called *Boolean* operators
- Logical operators combine multiple booleans into expressions that evaluate to a single **`true`** or **`false`**

Truth Tables



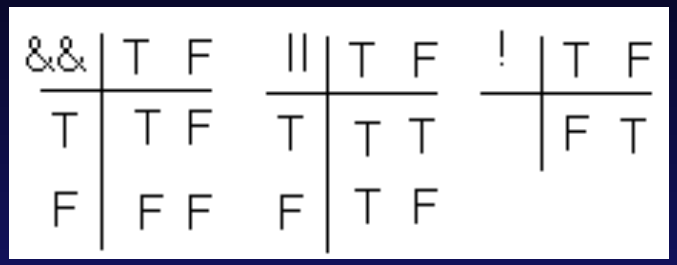
&&	T	F
T	T	F
F	F	F

||	T	F
T	T	T
F	T	F

!	T	F
	F	T

- Truth tables show the results combining **`boolean`**s with **`&& || `** or !
- Evaluate each expression:
  - **`false || true`**
  - **`true && false`**
  - **`!true`**

Truth Tables



&&	T	F
T	T	F
F	F	F

||	T	F
T	T	T
F	T	F

!	T	F
	F	T

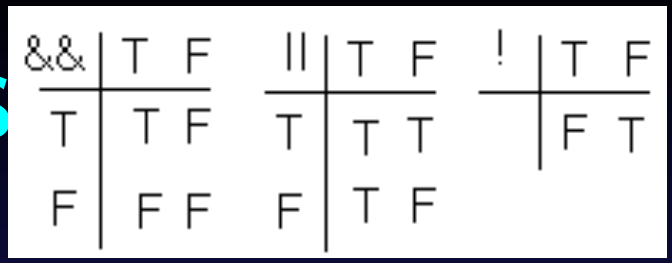
- Truth tables show the results combining **`boolean`**s with **`&& || `** or !
- Evaluate each expression:
  - **`false || true`** is **`true`**
  - **`true && false`** is **`false`**
  - **`!true`** is **`false`**

Relational (Comparison) Operators

> >= < <= != ==

- Is Greater Than
- Is Greater Than or Equal
- Is Less Than
- Is Less Than or Equal
- Is Not equal
- Is Equal to

Logical Operators



&&	T	F
T	T	F
F	F	F

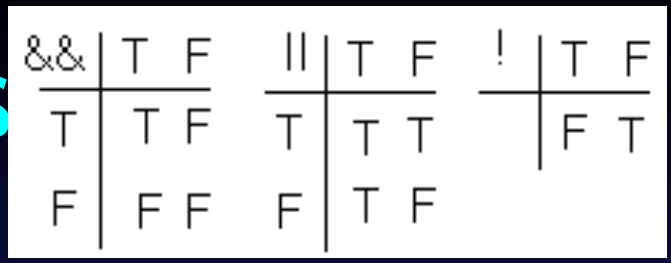
||	T	F
T	T	T
F	T	F

!	T	F
	F	T

- What would be printed?

```
int nCount = 150;
System.out.println((0<nCount)&&(nCount<=100));
```

Logical Operators



- What would be printed? **`false`**

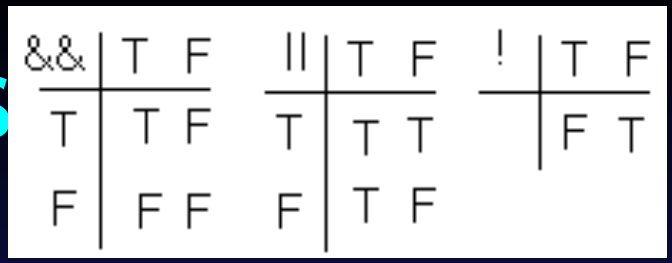
```
int nCount = 150;
System.out.println((0<nCount)&&(nCount<=100));
```

```
1  public class MyProgram extends ConsoleProgram
2  {
3      public void run()
4      {
5          int nCount = 150;
6          System.out.println((0<nCount)&&(nCount<=100));
7      }
8  }
```

RUN CODE

```
false
```

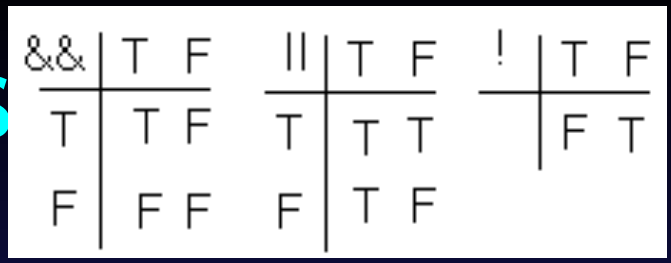
Logical Operators



- Now what would be printed?

```
int nCount = 50;
System.out.println((0<nCount)&&(nCount<=100));
```

Logical Operators



&&	T	F
T	T	F
F	F	F

||	T	F
T	T	T
F	T	F

!	T	F
	F	T

- What would be printed? **true**

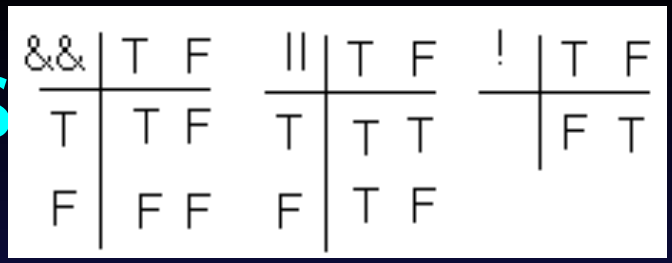
```
int nCount = 50;
System.out.println((0<nCount)&&(nCount<=100));
```

```
1  public class MyProgram extends ConsoleProgram
2  {
3      public void run()
4      {
5          int nCount = 50;
6          System.out.println((0<nCount)&&(nCount<=100));
7      }
8  }
```

RUN CODE

```
true
```

Logical Operators

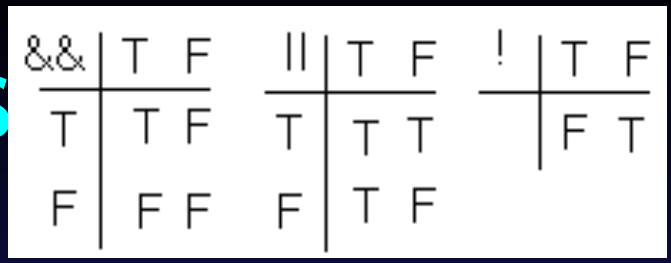


- Lets assign **nCount** 150 and replace the *and* with an *or*. Now what will be printed?

```
int nCount = 150;

System.out.println((0<nCount) || (nCount<=100));
```

Logical Operators



- Lets assign **nCount** 150 and replace the *and* with an *or*. Now what will be printed? **true**

```
int nCount = 150;
System.out.println((0<nCount)||(nCount<=100));
```

```
1  public class MyProgram extends ConsoleProgram
2  {
3      public void run()
4      {
5          int nCount = 150;
6          System.out.println((0<nCount)||(nCount<=100));
7      }
8  }
```

RUN CODE

```
true
```

! The *NOT* Operator

&&	T	F
T	T	F
F	F	F

||	T	F
T	T	T
F	T	F

!	T	F
	F	T

- In Java, ! is pronounced "not"
- It gives the opposite of whatever **boolean** comes after it

```
boolean isWhat = true;

System.out.println(!isWhat);
```

```
1  public class MyProgram extends ConsoleProgram
2  {
3      public void run()
4      {
5          boolean isWhat = true;
6          System.out.println(!isWhat);
7      }
8  }
9
```

RUN CODE

```
false
```

! The *NOT* Operator

&&	T	F
T	T	F
F	F	F

||	T	F
T	T	T
F	T	F

!	T	F
	F	T

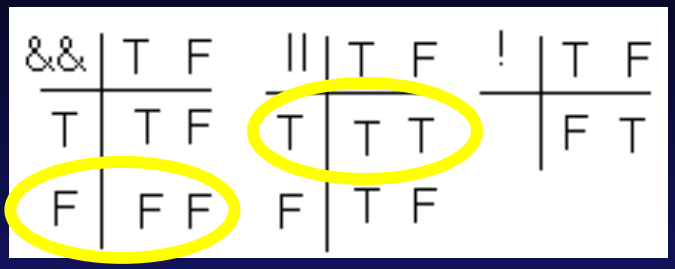
- ! is much more important than you would expect
- We'll frequently use ! to "flip" a boolean from **`true`** to **`false`** and back

```
1  public class MyProgram extends ConsoleProgram
2  {
3      public void run()
4      {
5          boolean isWhat = true;
6          isWhat = !isWhat;
7          System.out.println(isWhat);
8          isWhat = !isWhat;
9          System.out.println(isWhat);
10     }
11 }
12
```

RUN CODE

```
false
true
```

Short circuit evaluation



- **`true || anything`** is always **`true`**, so Java does not need to evaluate the expression ***`anything`***
- Likewise, **`false && anything`** is always false
- Ignoring the ***`anything`***, when possible, is called **short circuit evaluation**

Short circuit evaluation

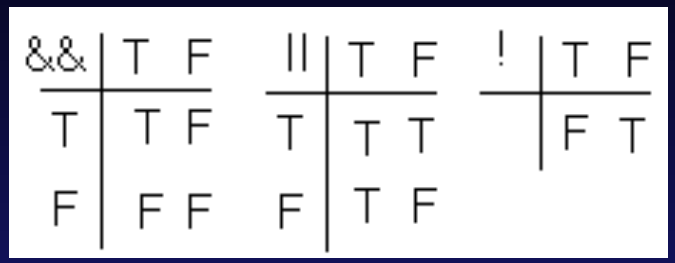
&&	T	F
T	T	F
F	F	F

||	T	F
T	T	T
F	T	F

!	T	F
	F	T

- What does
  `-3 < 0 || 567*2-17/3==1128`
  evaluate to?
- **`true`**
- Its not necessary to evaluate the second condition because the first condition is **`true`** and **`true || `*`anything`*** is always **`true`**

Short circuit evaluation



&&	T	F
T	T	F
F	F	F

||	T	F
T	T	T
F	T	F

!	T	F
	F	T

- What does
  `1==0 && 274/17 > 7846/461`
  evaluate to?
- **`false`**
- Its not necessary to evaluate the second condition because the first condition is **`false`** and **`false && anything`** is always **`false`**

Distributing the !

- The rules for distributing the `!` are known as De Morgan's laws

`!(A && B)` *is the same as* `!A || !B`

`!(A || B)` *is the same as* `!A && !B`

Distributing the !

- More examples of De Morgan's laws

```
!(x < 5 && y == 3)
```

is the same as

```
x >= 5 || y != 3
```

```
!(x >= 1 || y != 7)
```

is the same as

```
x < 1 && y == 7
```

How would Java evaluate these expressions?

&&	T	F
T	T	F
F	F	F

||	T	F
T	T	T
F	T	F

!	T	F
	F	T

```
char cGrade = 'B';
('A' <= cGrade) && (cGrade <= 'F')
```

How would Java evaluate these expressions?

&&	T	F
T	T	F
F	F	F

||	T	F
T	T	T
F	T	F

!	T	F
	F	T

```
char cGrade = 'B';
('A' <= cGrade) && (cGrade <= 'F')
//true
```

How would Java evaluate these expressions?

&&	T	F
T	T	F
F	F	F

||	T	F
T	T	T
F	T	F

!	T	F
	F	T

```
char cGrade = 'B';
('A' <= cGrade) && (cGrade <= 'F')
//true
int nAge = 16;
! (19 >= nAge && nAge >= 13)
```

How would Java evaluate these expressions?

&&	T	F
T	T	F
F	F	F

||	T	F
T	T	T
F	T	F

!	T	F
	F	T

```
char cGrade = 'B';
('A' <= cGrade) && (cGrade <= 'F')
//true
int nAge = 16;
! (19 >= nAge && nAge >= 13)
//false
```

How would Java evaluate these expressions?

&&	T	F
T	T	F
F	F	F

||	T	F
T	T	T
F	T	F

!	T	F
	F	T

```
char cGrade = 'B';
('A' <= cGrade) && (cGrade <= 'F')
//true
int nAge = 16;
! (19 >= nAge && nAge >= 13)
//false
int nAge = 16;
19 < nAge || nAge < 13
```

How would Java evaluate these expressions?

&&	T	F
T	T	F
F	F	F

||	T	F
T	T	T
F	T	F

!	T	F
	F	T

```
char cGrade = 'B';
('A' <= cGrade) && (cGrade <= 'F')
//true
int nAge = 16;
! (19 >= nAge && nAge >= 13)
//false
int nAge = 16;
19 < nAge || nAge < 13
//false
```

if statements

- If the condition in parentheses is **true**, the code in the curly braces (called a block) executes

```
if (x > 0) {
    System.out.println("x is positive");
}
```

if else statements

- If the condition in parentheses is **true**, the code in the first block executes otherwise the second block executes

```
if (x > 0) {
    System.out.println("x is positive");
}
else {
    System.out.println("x is NOT positive");
}
```

if else if statements

- The block under the first **true** condition runs, and then all other blocks are skipped
- If all conditions are **false**, the **else** block runs

```
if (temp > 80) {
    System.out.println("Hot");
} else if (temp > 65) {
    System.out.println("Warm");
} else if (temp > 50) {
    System.out.println("Cool");
} else {
    System.out.println("Cold");
}
```

if statements

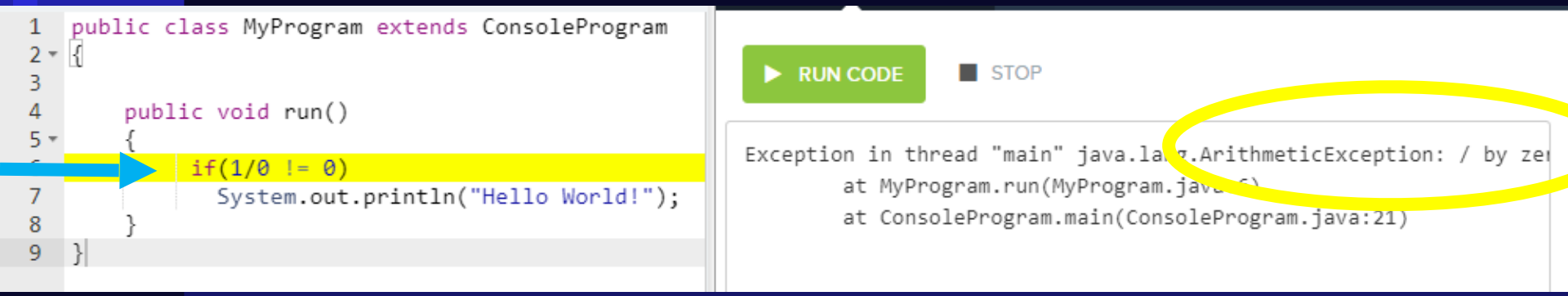
- Curly braces are optional if a block has only one statement

```
if (x > 0) {
    System.out.println("x is positive");
}
```

is the same as

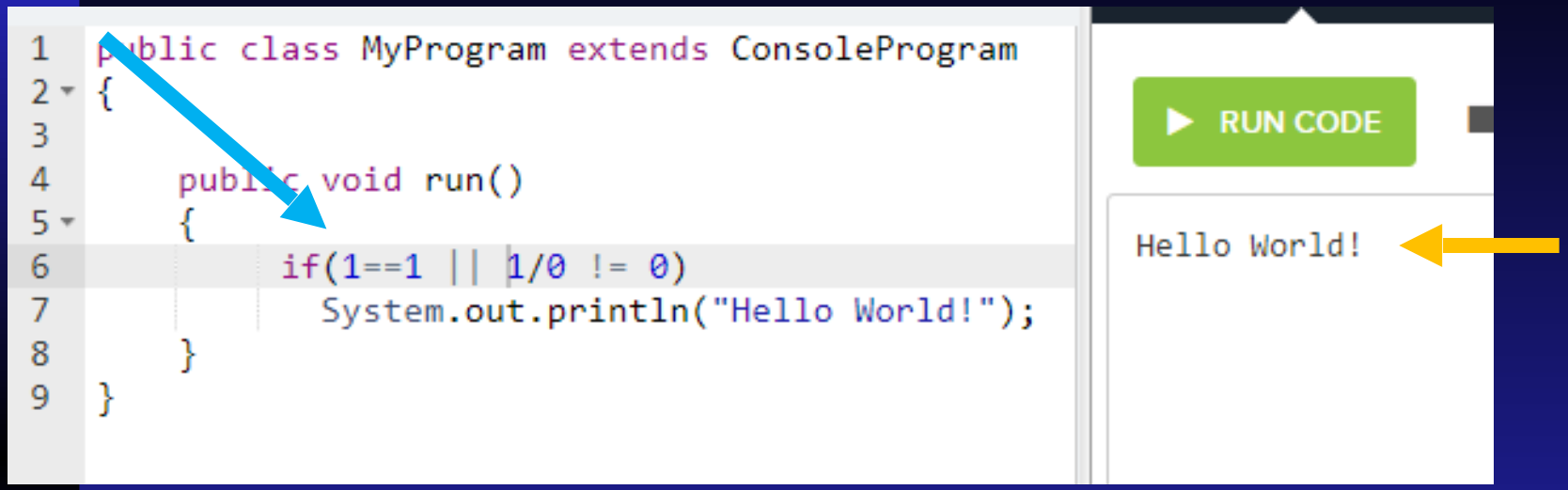
```
if (x > 0)
    System.out.println("x is positive");
```

More short circuit evaluation



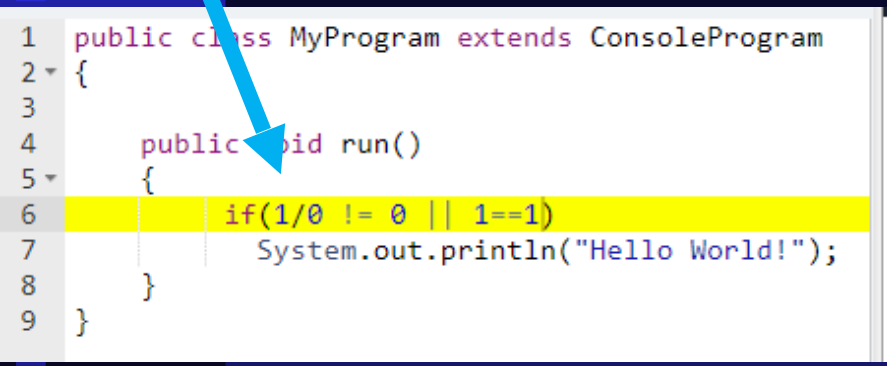
- Notice that `Hello World!` is not printed because of the divide by zero exception in the `if(1/0 != 0)`

More short circuit evaluation



- Now **`Hello World!`** ***is*** printed and there is no divide by zero exception
- Because **`1==1 || anything`** is **`true`**, Java does not execute the code that would cause an exception

Order matters!



- In this version, the code that causes the exception executes first so **`Hello World!`** is not printed

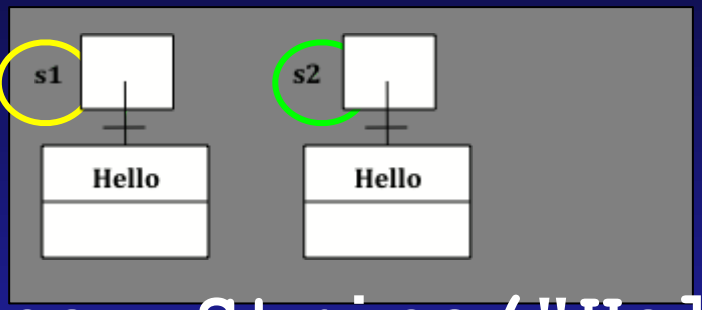
- Quizziz in 10 minutes!

`==` vs. `.equals()` for `Strings`

- `==` checks if the String variables refer to the same String
- `.equals` checks if two different Strings have the same characters

== vs. .equals() for Strings

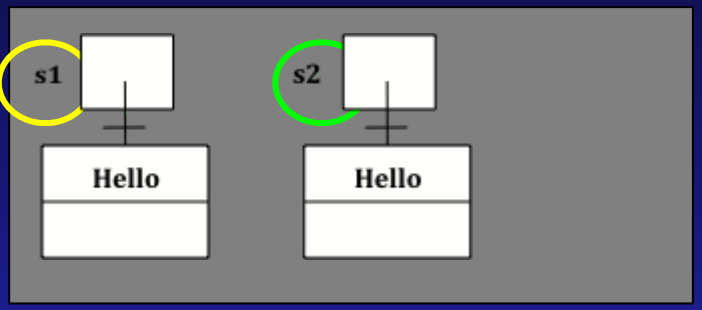
- **s1** and **s2** refer to two different **new Strings**




```
String s1 = new String("Hello");
String s2 = new String("Hello");
System.out.println(s1 == s2);
System.out.println(s1.equals(s2));
```

== vs. .equals() for Strings

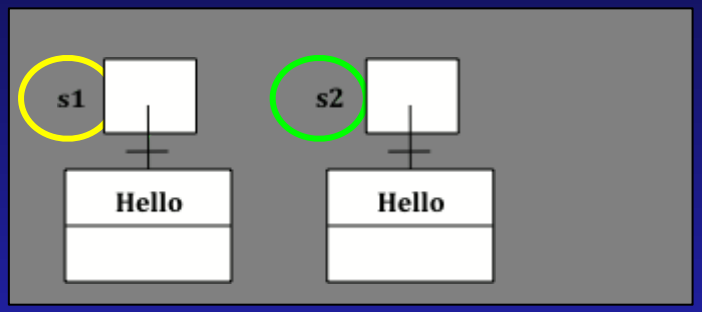
- **s1 == s2** is false because the two **String**s are different




```
String s1 = new String("Hello");
String s2 = new String("Hello");
System.out.println(s1 == s2); //false
System.out.println(s1.equals(s2)); //true
```

== vs. .equals() for Strings

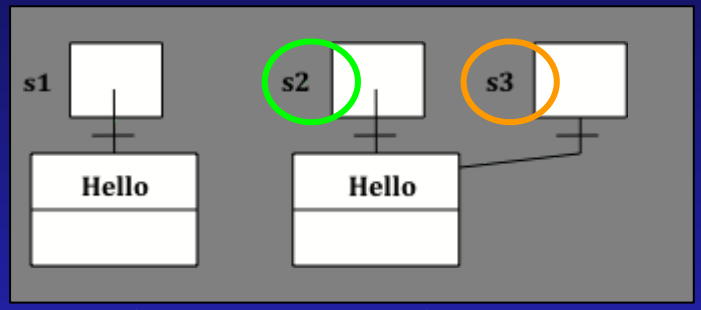
- **s1.equals(s2)** is **true** because the two different **String**s have the same characters




```
String s1 = new String("Hello");
String s2 = new String("Hello");
System.out.println(s1 == s2); //false
System.out.println(s1.equals(s2)); //true
```

== vs. .equals() for Strings

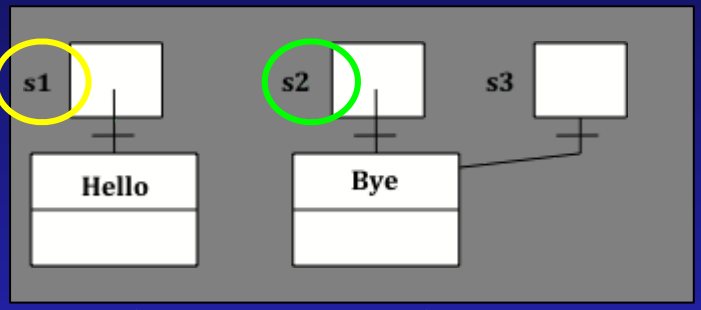
- **s2==s3** and **s2.equals(s3)** are both **true** because they are the same **String** with the same characters




```
String s2 = new String("Hello");
String s3 = s2;
System.out.println(s2 == s3); //true
System.out.println(s2.equals(s3)); //true
```

== vs. .equals() for Strings

- **s1==s2** and **s1.equals(s2)** are both **false** because they are different **String**s with different characters




```
String s1 = new String("Hello");
String s2 = new String("Bye");
System.out.println(s1 == s2); //false
System.out.println(s1.equals(s2)); //false
```

Null pointer error

- Uninitialized **Strings** are **null**
- You get a **null** pointer error when you try to use an uninitialized String

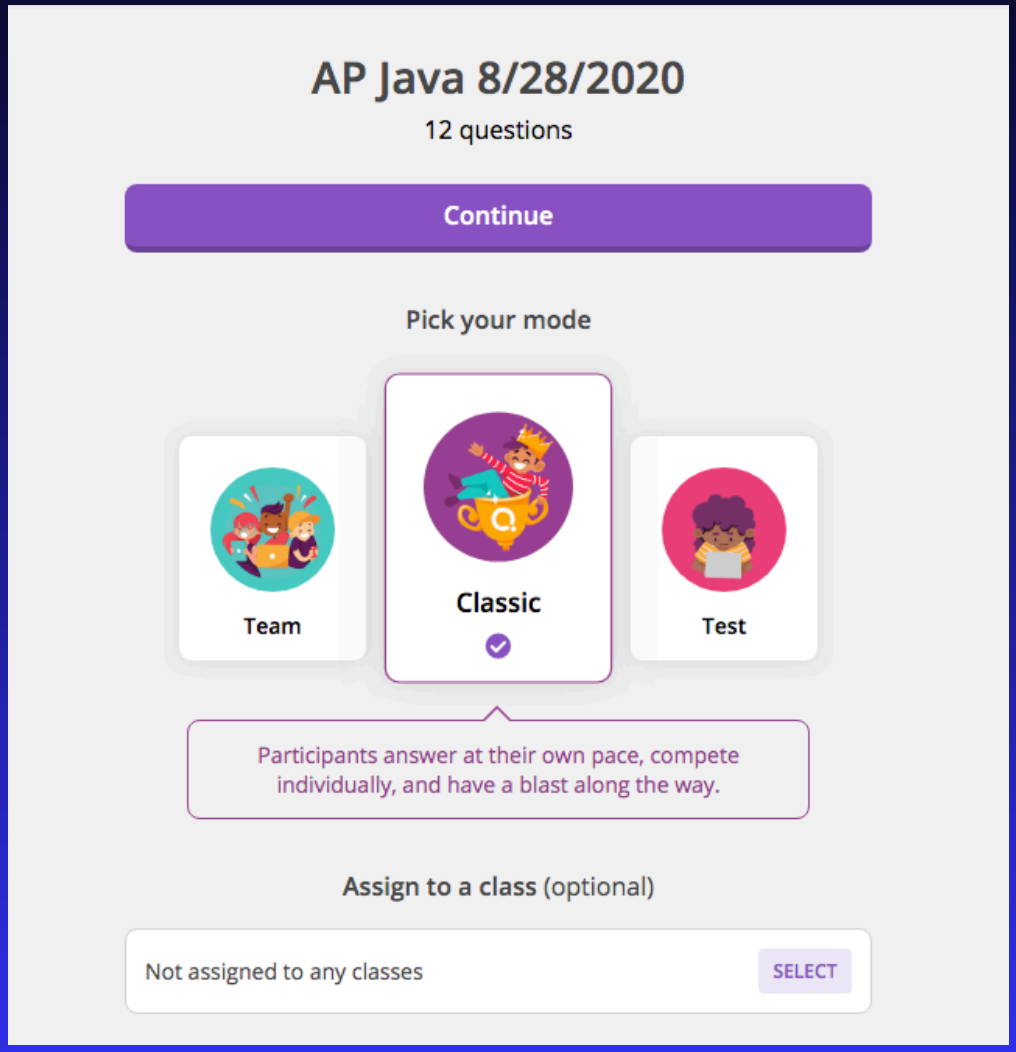
```
String s1; //unitialized
String s2 = new String("Hello");
System.out.println(s1); //null
System.out.println(s1.equals(s2));
                                  //error
```

Do CSAwesome Assignments

Confusing message on Activity 56

- At first I thought I got it wrong!
- If you get a check mark, it's correct

Quizziz


AP Java 8/28/2020
12 questions
Continue
Pick your mode
Team
Classic
Test
Participants answer at their own pace, compete individually, and have a blast along the way.
Assign to a class (optional)
Not assigned to any classes
SELECT