Variables and Expressions in Java

One of the most powerful features of a programming language is the ability manipulate a computer's memory. A variable is a named item that allows programmers to stores values in RAM. Values may be numbers, text, images, sounds, and other types of data. In strictly-typed language like Java you have to include the type with a variable definition. Once the variable has been declared a specific type, that variable can only hold values of that type. Any attempts to store a value of a different type will result in compile errors (There are a few exceptions to this)

```
String message;
int x;
double cost;
```

These statements are declarations, because they define the variable names and the associated types. Each variable has a type that determines what kind of values it can store. For example, the int type can store integers, and the double type can store fractions. Some types begin with a capital letter and some with lowercase. We will learn the significance of this distinction later, but for now you should take care to get it right. There is no such type as Int or string.

```
String firstName;
String lastName;
int hour, minute;
```

This example declares two variables with type String and two with type int. When a variable name contains more than one word, like firstName, it is conventional to capitalize the first letter of each word except the first. Variable names are case-sensitive, so firstName is not the same as firstname or FirstName.

This example also demonstrates the syntax for declaring multiple variables with the same type on one line: hour and minute are both integers. Note that each declaration statement ends with a semicolon. You can use any name you want for a variable. But there are about 50 reserved words, called keywords, that you are not allowed to use as variable names. These words include public, class, static, void, and int, which are used by the compiler to analyze the structure of the program.

You can find the complete list of keywords at **http://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html,** but you don't have to memorize them. Most programming editors provide "syntax highlighting", which makes different parts of the program appear in different colors.

Now that we have declared variables, we want to use them to store values. We do that with an assignment statement.

```
firstName = "Tony";
lastName = "Iommi";
hour = 3;
minute = 30;
```

Variable Initialization

We don't have to break variable creation into two steps like the examples shown above. We can declare and assign values on the same line. This is called initialization.

```
String firstName = "Tony;
String lastName = "Iommi";
int hour = 3, minute = 30;
```

Number Systems

The decimal number system is the one that we are most familiar with, we use it every day. Decimal is what we refer to as a ***positional number system***. That is, the position of the digits gives meaning to the value they represent. The other number systems (binary, hexadecimal and octal) are also positional, so once you understand the underlying theory of decimal we can easily apply that to understand the other systems.

Let's look at an example:

If I have the number 31415, what this actually represents is:

30,000 + 1,000 + 400 + 10 + 5

$3 * 10^4$	30,000
$1 * 10^3$	1,000
$4 * 10^2$	400
$1 * 10^1$	10
$5 * 10^0$	5

Decimal is ***base 10***. This means we have 10 symbols for representing values (0 - 9). As we move through each position we multiply that number by 10 to the power of that position (starting at 0 at the far right side).

Note: Anything to the power of 0 is 1

Decimal is convenient as a number system as every time we increase the power all we need to do is add another 0. For each digit in the number, add the number of 0's required by the position and you've got it's positional value. Then each digit naturally lines up in the overall number.

Binary: Binary follows the same pattern as Decimal except that instead of being base 10, it is instead ***base 2***. Instead of having 10 symbols to represent values we have two (0 and 1). Using the positional theory, we multiply each digit by 2 to the power of its position.

Let's look at an example:

If I have the binary number 101010, this translates into decimal as:

32 + 0 + 8 + 0 + 2 + 0 = 42

$1 * 2^5$	32
$0 * 2^4$	0
$1 * 2^3$	8
$0 * 2^2$	0
$1 * 2^1$	2
$0 * 2^0$	0

The following table shows the first 16 values of the base 10 and base 2 positional number systems.

decimal (base 10)	binary (base 2)	expansion
0	0	0 ones
1	1	1 one
2	10	1 two and zero ones
3	11	1 two and 1 one
4	100	1 four, 0 twos, and 0 ones
5	101	1 four, 0 twos, and 1 one
6	110	1 four, 1 two, and 0 ones
7	111	1 four, 1 two, and 1 one
8	1000	1 eight, 0 fours, 0 twos, and 0 ones
9	1001	1 eight, 0 fours, 0 twos, and 1 ones
10	1010	1 eight, 0 fours, 1 two, and 0 ones
11	1011	1 eight, 0 fours, 1 two, and 1 one
12	1100	1 eight, 1 four, 0 twos, and 0 ones
13	1101	1 eight, 1 four, 0 twos, and 1 one
14	1110	1 eight, 1 four, 1 two, and 0 ones
15	1111	1 eight, 1 four, 1 two, and 1 one
16	10000	1 sixteen, 0 eights, 0 fours, 0 twos, and 0 ones

Converting between binary and decimal numbers is fairly simple, as long as you remember that each digit in the binary number represents a power of two.

Convert 1011001012 to the corresponding base-ten number.[

Count these digits off from the RIGHT, starting with zero:

Digits:	1	0	1	1	0	0	1	0	1
Position:	8	7	6	5	4	3	2	1	0
Weight:	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Value:	$1x2^8$	$0x2^7$	$1x2^6$	$1x2^5$	$0x2^4$	$0x2^3$	$1x2^2$	$1x2^1$	$1x2^0$

$1\times2^8 + 0\times2^7 + 1\times2^6 + 1\times2^5 + 0\times2^4 + 0\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$

$= 1\times256 + 0\times128 + 1\times64 + 1\times32 + 0\times16 + 0\times8 + 1\times4 + 0\times2 + 1\times1$

$= 256 + 64 + 32 + 4 + 1$

$= 357$

Then 101100101 converts to 357_{10}

Repeated Division-by-2 Method

We have seen above how to convert binary to decimal numbers, but how do we convert a decimal number into a binary number. An easy method of converting decimal to binary number equivalents is to write down the decimal number and to continually divide-by-2 (two) to give a result and a remainder of either a “1” or a “0” until the final result equals zero.

So for example. Convert the decimal number 357 into its binary number equivalent.

LSB = Least Significant Bit (position zero)

MSB = Most Significant Bit (position (length + 1))

Division	Quotient	Remainder	Position
357 // 2	178	**1**	**0**
178 // 2	89	**0**	**1**
89 // 2	44	**1**	**2**
44 // 2	22	**0**	**3**
22 // 2	11	**0**	**4**
11 // 2	5	**1**	**5**
5 // 2	2	**1**	**6**
2 // 2	1	**0**	**7**
1 // 2	0	**1**	**8**

Java Primitive Data Types

The following list describes the basic data types supported by Java. Each of these numeric types allocate a specific number of bits. The quantity of bits determines the range of values that can be stored in the type. Notice that the String type is not in this list. That is because the type String is an object represented by a class. These **primitive** types are not objects, they are simply typed memory allocations with no associated methods. The different numeric types allow programmers control over the amount of memory allocated for each variable. For example . . . if you know that a range of integers will never surpass 100, then you can save memory space by allocating a **byte** instead of an **int.** Be smart about your data type choices.

These types will either be **signed or unsigned.**

Signed: Allows negative numbers. An important detail is that the MSB in a signed type is not used in the calculation of the value. It is used to denote **the sign bit**. If the bit is on the number is negative, if the bit is off, the number is positive. There is much more to this, but there is no need to get into here. If you are interested in this neat little tangent, research **2's complementation.**

Unsigned: Supports positive numbers only **and uses all bits** to represent the value.

byte: The byte data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). An unsigned 8 bit type would give the range of 0 – 255. With a signed type we still get these 256 unique values, they are just divided into positive and negative ranges with 0 in the middle. The byte data type can be useful for saving memory in large arrays, where the memory savings actually matters. They can also be used in place of int where their limits help to clarify your code; the fact that a variable's range is limited can serve as a form of documentation.

short: The short data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). As with byte, the same guidelines apply: you can use a short to save memory in large arrays, in situations where the memory savings actually matters.

int: By default, the int data type is a 32-bit signed two's complement integer, which has a minimum value of -2^{31} and a maximum value of $2^{31}-1$. In Java SE 8 and later, you can use the int data type to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of $2^{32}-1$. Use the Integer class to use int data type as an unsigned integer. See the section The Number Classes for more information. Static methods like compareUnsigned, divideUnsigned etc have been added to the Integer class to support the arithmetic operations for unsigned integers.

long: The long data type is a 64-bit two's complement integer. The signed long has a minimum value of -2^{63} and a maximum value of $2^{63}-1$. In Java SE 8 and later, you can use the long data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of $2^{64}-1$. Use this data type when you need a range of values wider than those provided by int. The Long class also contains methods like compareUnsigned, divideUnsigned etc to support arithmetic operations for unsigned long.

float: The float data type is a single-precision 32-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the Floating-Point Types, Formats, and Values section of the Java Language Specification. As with the recommendations for byte and short, use a float (instead of double) if you need to save memory in large arrays of floating point numbers. This data type should never be used for precise values, such as currency. For that, you will need to use the

java.math.BigDecimal class instead. Numbers and Strings covers BigDecimal and other useful classes provided by the Java platform.

double: The double data type is a double-precision 64-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the Floating-Point Types, Formats, and Values section of the Java Language Specification. For decimal values, this data type is generally the default choice. As mentioned above, this data type should never be used for precise values, such as currency.

boolean: The boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.

char: The char data type is an unsigned 16 bit allocation; a single 16-bit Unicode character. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive). This gives us exponentially more values than a standard ASCII character which is 8 bits. The **char type** in C/C++ for instance allocates 8 bits of storage and thus can only store 256 unique characters. By choosing to adhere to the Unicode standard, Java can store character sets from many different languages. If you are interested in more details on this, check out https://home.unicode.org/

Default Values

It's not always necessary to assign a value when a field is declared (but it is a good idea). Fields that are declared but not initialized will be set to a reasonable default by the compiler. Generally speaking, this default will be zero or null, depending on the data type. Relying on such default values, however, is generally considered bad programming style.

The following chart summarizes the default values for the above data types.

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

Constants and Literals

A value that appears in a program, like 2.54 (or " in ="), is called a literal. In general, there’s nothing wrong with literals. But when numbers like 2.54 appear in an expression with no explanation, they make code hard to read. And if the same value appears many times, and might have to change in the

future, it makes code hard to maintain. Values like that are sometimes called **magic numbers** (with the implication that being “magic” is not a good thing). A good practice is to assign magic numbers to variables with meaningful names. If we wanted to store the conversion factor for **centimeters per inch** it’s helpful to provide a name for this value.

double cmPerInch = 2.54;

This version is easier to read and less error-prone, but it still has a problem. Variables can vary, but the number of centimeters in an inch does not. Once we assign a value to cmPerInch, it should never change. Java provides a language feature that enforces that rule, the keyword final.

final double CM_PER_INCH = 2.54;

Declaring that a variable is final means that it cannot be reassigned once it has been initialized. If you try, the compiler reports an error. Variables declared as final are called constants. By convention, names for constants are all uppercase, with the underscore character (_) between words.

Printing Variables

The following examples and many more examples to come utilize the **JShell** tool. Read about it here: https://docs.oracle.com/javase/9/jshell/introduction-jshell.htm#JSHEL-GUID-630F27C8-1195-4989-9F6B-2C51D46F52C8

I recommend following along with these examples to get the hang of JShell. Use this tool whenever you are experimenting with a language feature and you do not want to create, save, compile and execute source code. This tool is similar to Python’s IDLE and command line interpreter.

You can display the value of a variable using print or println. The following statements declare a variable named firstLine, assign it the value "Hello, again!", and display that value.

```
jshell> String firstLine = "Hello, again!";
firstLine ==> "Hello, again!"

jshell> System.out.println(firstLine);
Hello, again!
```

When we talk about displaying a variable, we generally mean the value of the variable. To display the name of a variable, you have to put it in quotes.

```
jshell> System.out.print("The value of firstLine is ");
   ...> System.out.println(firstLine);
The value of firstLine is Hello, again!
```

Conveniently, the syntax for displaying a variable is the same regardless of its type. For example:

```
jshell>     int hour = 11;
   ...>     int minute = 59;
   ...>     System.out.print("The current time is ");
   ...>     System.out.print(hour);
   ...>     System.out.print(":");
   ...>     System.out.print(minute);
   ...>     System.out.println(".");
hour ==> 11
minute ==> 59
The current time is 11:59.
```

Notice the difference between print and println: Print does not cause a line break, println does cause a line break. That is print “el” n . . . not print “one” n. **println => “print line”** (include a line break in the display).

Arithmetic operators

Operators are symbols that represent simple computations. Java supports the following arithmetic operators

Operator	Result
+	Addition (also unary plus)
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
–=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
––	Decrement

The following JShell snippet converts a time of day to minutes:

```
jshell>     int hour = 11;
   ...>     int minute = 59;
   ...>     System.out.print("Number of minutes since midnight: ");
   ...>     System.out.println(hour * 60 + minute);
hour ==> 11
minute ==> 59
Number of minutes since midnight: 719
```

In this program, **hour * 60 + minute** is an expression, which represents a single value to be computed. When the program runs, each variable is replaced by its current value, and then the operators are applied. The values operators work with are called **operands**.

Expressions are generally a combination of numbers, variables, and operators. When complied and executed, they become a single value.

For example:

- the expression 1 + 1 has the value 2.
- In the expression hour - 1, Java replaces the variable with its value, yielding 11 - 1, which has the value 10.
- In the expression hour * 60 + minute, both variables get replaced, yielding 11 * 60 + 59. The multiplication happens first, yielding 660 + 59. Then the addition yields 719.

Addition, subtraction, and multiplication all do what you expect, but you might be surprised by division. For example, the following fragment tries to compute the fraction of an hour that has elapsed:

```
jshell>     System.out.print("Fraction of the hour that has passed: ");
   ...>     System.out.println(minute / 60);
Fraction of the hour that has passed: 0
```

The output is:

Fraction of the hour that has passed: 0

This result often confuses people. The value of minute is 59, and 59 divided by 60 should be 0.98333, not 0. The problem is that Java performs **integer division when the operands are integers**. By design, integer division always rounds toward zero, even in cases like this one where the next integer is close.

As an alternative, we can calculate a percentage rather than a fraction:

```
jshell>
   ...>     System.out.print("Percent of the hour that has passed: ");
   ...>     System.out.println(minute * 100 / 60);
Percent of the hour that has passed: 98
```

The new output is:

Percent of the hour that has passed: 98

Again the result is rounded down, but at least now it's approximately correct.

Remember: The data type of the operands determines the type of the expression.

Remember: The data type of the variable that stores a result does not determine the type of the expression

Floating-point numbers
A more general solution is to use floating-point numbers, which can represent fractions as well as integers. In Java, the default floating-point type is called **double**, which is short for **double-precision**. You can create double variables and assign values to them using the same syntax we used for the other types:

```
jshell> double pi = 3.14159;
pi ==> 3.14159
```

Java performs "floating-point division" when one or more operands are double values. So we can solve the problem we saw in the previous section:

```
jshell> double minute = 59.0;
   ...> System.out.print("Fraction of the hour that has passed: ");
   ...> System.out.println(minute / 60.0);
minute ==> 59.0
Fraction of the hour that has passed: 0.9833333333333333
```

Although floating-point numbers are useful, they can be a source of confusion. For example, Java distinguishes the integer value 1 from the floating-point value 1.0, even though they seem to be the same number. They belong to different data types, and strictly speaking, you are not allowed to make assignments between types. The following is illegal because the variable on the left is an int and the value on the right is a double:

```
jshell> int x = 1.1;
|  Error:
|  incompatible types: possible lossy conversion from double to int
|  int x = 1.1;
|          ^-^
```

It is easy to forget this rule because in many cases Java automatically converts from one type to another. The following is legal and is considered poor style, but notice that Java automatically converted the integer 1 to double 1.0.

```
jshell> double y = 1;
y ==> 1.0
```

The preceding example should be illegal, but Java allows it by converting the **int value 1** to the **double value 1.0** automatically. This leniency is convenient, but it often causes problems for beginners. Let's revisit the integer division issue

```
jshell> double y = 1 / 3;
y ==> 0.0
```

In this scenario the integers 1 and 3 are divided using **integer division** which results in a truncated value of 0. The fractional portion of the result is lost. The assignment of the result to the double variable y **triggers the auto-conversion to a double**, but this happens **after** the division takes place so all precision is lost. Here is the correct way to achieve the fractional results.

```
jshell> double y = 1.0 / 3.0;
y ==> 0.3333333333333333
```

If you have integer variables that you wish to perform double division on you can employ type casting. Type casting is the process of converting values of one type to values of another type.

```
jshell> int a = 1;
   ...> int b = 2;
   ...> double c = (double)a / b;
a ==> 1
b ==> 2
c ==> 0.5
```

The type cast occurs on this token **(double)a** and is telling the compiler to treat the integer variable **a** as a double. Notice how variable **c** contains the correct result. What about variable **b**? This expression is called a **mixed expression** and Java will automatically perform a **widening type cast** to the smaller type. Computers cannot process expressions with mixed types, so it is common for this automatic type cast to normalize the expression to a single type. Once this conversion is performed the expression can be evaluated.

Type casting is when you assign a value of one data type to another type.

In Java, there are two types of casting:

Widening Cast: (automatic) - converting a smaller type to a larger type can happen automagically as we have just seen. Smaller to large means going from a smaller number of bits to a larger number of bits. A widening cast can happen in any of the following situations. (follow the arrows)

`byte` → `short` → `char` → `int` → `long` → `float` → `double`

Narrowing Cast: (manual) - converting a larger type to a smaller type has the opportunity to lose precision, as you are converting from a larger size of allocated bits to a smaller size of allocated bits. When you remove bits from a value you can have potential undesired behavior. Due to this potential loss in precision, a narrowing cast must be performed manually.

double → float → long → int → char → short → byte

Here's an example of a narrowing cast that discards bits and results in a loss of data.

```
jshell> int a = 12345678;
   ...> byte b = (byte)a;
a ==> 12345678
b ==> 78
```

What's going on here? A Java integer is 32 bits of storage. A Java byte is 8 bits of storage. That type cast essentially truncated 24 bits off of the variable **a**. Use this wisely!

Let's take a close look:

An int is 32 bits. 12345678 in binary is

00000000-00000000-010111100-01100001-01001110

The dashes separate 8 bit chunks called **bytes** and are inserted here just to increase legibility.

When we perform a narrowing type cast from an **int** to a **byte,** Java takes the low order 8 bits and discards the rest

Discarded

00000000-00000000-010111100-01100001-01001110

What's left is the bit string 01001110 which is 78_{10}

```
jshell> int a = 12345678;
   ...> byte b = a;
a ==> 12345678
|  Error:
|  incompatible types: possible lossy conversion from int to byte
|  byte b = a;
|           ^
```

If you try this assignment without an explicit type cast the compiler will complain. Notice the error message **Possible lossy conversion from int to byte**

Moral of the story: Widening casts happen automatically. Narrowing casts you must perform manually and **have a very good reason for doing so!** We will encounter said very good reason when we venture into class inheritance. Stay tuned.

Application Programming Interface (API)

Let's take a moment to examine an incredibly important concept . . . the API. An application programming interface . . . or API . . . is a communication protocol that allows programmer's to interact with various software components. In our case we are interacting with the expansive Java library. In building applications, an API simplifies programming by **abstracting** he underlying implementation and only exposing objects or actions the developer needs. While a graphical interface for an email client might provide a user with a button that performs all the steps for fetching and highlighting new emails, an API for file input/output might give the developer a function that copies a file from one location to another without requiring that the developer understand the file system operations occurring behind the scenes. Popular software applications like Facebook, Instagram and Google Maps provide APIs for programmers to link their functionality into customized programs.

For example: Facebook's API "Graph" provides simplistic ways for programmers to retrieve all posts marked as public. This can happen outside of Facebook.

For Example: How many times have you seen a Google Map embedded into some software application . . . like a map to your favorite pizza place?
These applications provide a way for us to interface with their underlying code base without us needing to know anything about the underlying implementation. This is the genius of abstraction.

An API is usually related to a software library. The API describes and prescribes the "expected behavior" (a specification) while the library is an "actual implementation" of this set of rules.
A single API can have multiple implementations (or none, being abstract) in the form of different libraries that share the same programming interface.

The separation of the API from its implementation can allow programs written in one language to use a library written in another. For example, because Scala and Java compile to compatible bytecode, Scala developers can take advantage of any Java API.

Ok, what does that mean for us?

This means that you need to begin getting used to researching the Java API specification. This is a fancy way of saying **Java documentation.** The APIs can be found here:

https://www.oracle.com/technetwork/java/api-141528.html

Choose the API to match the version of Java that you are working with. Remember, this document is the **specification.** You need to download the Java SE library to actually write code. You use the specification to figure out the expected behaviors of the language. I selected the API spec for Java SE 9 which contains the following specifications.

Java® Platform, Standard Edition & Java Development Kit Version 9 API Specification

This document is divided into three sections:

Java SE

The Java Platform, Standard Edition (Java SE) APIs define the core Java platform for general-purpose computing. These APIs are in modules whose names start with `java`.

JDK

The Java Development Kit (JDK) APIs are specific to the JDK and will not necessarily be available in all implementations of the Java SE Platform. These APIs are in modules whose names start with `jdk`.

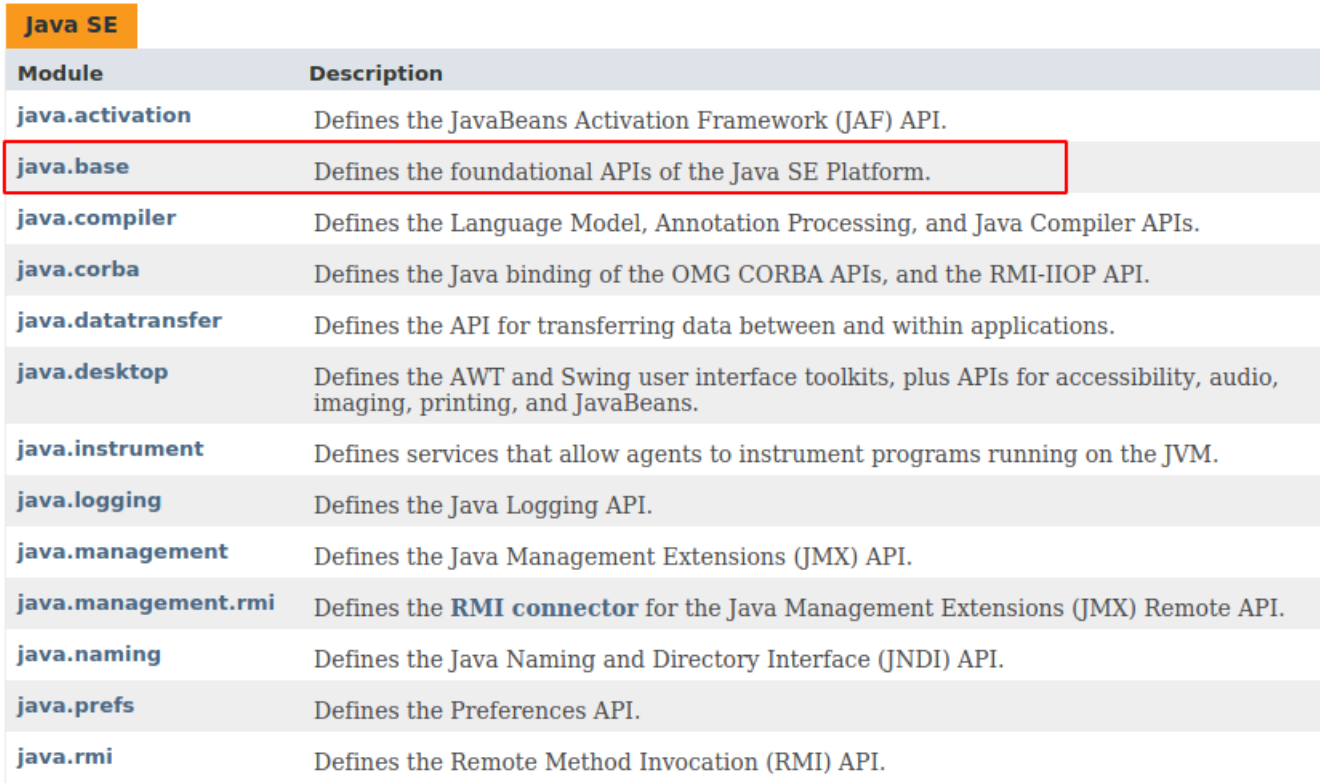
JavaFX

The JavaFX APIs define a set of user-interface controls, graphics, media, and web packages for developing rich client applications. These APIs are in modules whose names start with `javafx`.

This API specification is organized around the idea of **modules** and **packages.** The package is a fundamental unit of organization for the Java language. A package is simply a directory of related classes. As we have seen above with **java.io** package, we would expect classes dealing with input and output to be located here. The **java.util** package contains utility classes like the Scanner and the Arrays class. These classes must be imported before they can be used. The **java.lang** package contains the String class and other commonly used classes. These classes are so common that the **java.lang** package does not need to be imported. It is automatically provided by the Java compiler.

Strings in Java and the String API

Let’s take an opportunity to delve into the Java API by analyzing the String API. Follow the link above and select the API for your version of Java. You can find the String API in the **java.base** module



Java SE

Module	Description
java.activation	Defines the JavaBeans Activation Framework (JAF) API.
java.base	Defines the foundational APIs of the Java SE Platform.
java.compiler	Defines the Language Model, Annotation Processing, and Java Compiler APIs.
java.corba	Defines the Java binding of the OMG CORBA APIs, and the RMI-IIOP API.
java.datatransfer	Defines the API for transferring data between and within applications.
java.desktop	Defines the AWT and Swing user interface toolkits, plus APIs for accessibility, audio, imaging, printing, and JavaBeans.
java.instrument	Defines services that allow agents to instrument programs running on the JVM.
java.logging	Defines the Java Logging API.
java.management	Defines the Java Management Extensions (JMX) API.
java.management.rmi	Defines the **RMI connector** for the Java Management Extensions (JMX) Remote API.
java.naming	Defines the Java Naming and Directory Interface (JNDI) API.
java.prefs	Defines the Preferences API.
java.rmi	Defines the Remote Method Invocation (RMI) API.

Clicking on this link exposes **exported packages** of the foundational Java APIs. An exported package is one that programmers can take advantage of for our projects. Take a minute to familiarize yourself as you will be coming back often throughout the semester and maybe more so if you get a job programming in Java.

Warning: This is a serious rabbit hole!! Try not to be overwhelmed. Treat this as an exciting learning opportunity. There are all types of tutorials and demonstrations contained in this documentation. These APIs should be the first place you look when trying to accomplish a task. It is possible that there is already an optimized solution. There will also be times in this class when I forbid the usage of the API because I am trying to focus you on a specific concept that I want you to be able to implement yourself.

The String API spec is contained in the **java.lang** package. This package contains commonly used objects and does not need to be imported. Click on this package

Scroll down until you see the **Class Summary** section. This is an alphabetical listing of all the classes in the **java.lang** package. Find the String class

Packages

Exports

Package	Description
java.io	Provides for system input and output through data streams, serialization and the file system.
java.lang	Provides classes that are fundamental to the design of the Java programming language.
java.lang.annotation	Provides library support for the Java programming language annotation facility.
java.lang.invoke	The `java.lang.invoke` package contains dynamic language support provided directly by the Java core class libraries and virtual machine.
java.lang.module	Classes to support module descriptors and creating configurations of modules by means of resolution and service binding.
java.lang.ref	Provides reference-object classes, which support a limited degree of interaction with the garbage collector.
java.lang.reflect	Provides classes and interfaces for obtaining reflective information about classes and objects.
java.math	Provides classes for performing arbitrary-precision integer arithmetic (`BigInteger`) and arbitrary-precision decimal arithmetic (`BigDecimal`).
java.net	Provides the classes for implementing networking applications.

The String API spec contains all the information you need to know about working with String objects.

Much of this information may be over your head at this point but by the end of the semester you should be able to understand **all of it!** The important information for us now are the methods that the String class offers us. Scroll down to the area marked **Method Summary.** These are all the methods that you can invoke upon a valid String object.

This documentation gives us the following crucial information on how to interact with a Java class.

1. The method name
2. The argument list
3. The return type

Note: For you Python folks, **Java, as a strictly typed language** must declare return types for methods.

Module java.base
Package java.lang

Class String

java.lang.Object
    java.lang.String

All Implemented Interfaces:
`Serializable`, `CharSequence`, `Comparable<String>`

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

This documentation tells you the type of variable that you will need to store the results of a method call.

Method Summary

All Methods	Static Methods	Instance Methods	Concrete Methods	Deprecated Methods

Modifier and Type	Method	Description
char	charAt(int index)	Returns the char value at the specified index.
IntStream	chars()	Returns a stream of int zero-extending the char values from this sequence.
int	codePointAt(int index)	Returns the character (Unicode code point) at the specified index.
int	codePointBefore(int index)	Returns the character (Unicode code point) before the specified index.
int	codePointCount(int beginIndex, int endIndex)	Returns the number of Unicode code points in the specified text range of this String.
IntStream	codePoints()	Returns a stream of code point values from this sequence.
int	compareTo(String anotherString)	Compares two strings lexicographically.
int	compareToIgnoreCase(String str)	Compares two strings lexicographically, ignoring case differences.
String	concat(String str)	Concatenates the specified string to the end of this string.

Things to know about Java Strings:

1. A String is a sequence of individual characters of primitive type **char**
2. Strings are immutable and cannot be modified in any way. The methods that operate on Strings will return **new String objects**, and will not modify the existing String object.
3. Java has a very interesting approach to handling allocation of String memory. We will cover this in more detail later.
4. The class String includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase.
5. The Java language provides special support for the string concatenation operator (+), and for conversion of other objects to strings.

Demonstration of the most common String operations

1. Find the length of a string. Use the **length()** method

```
jshell> String s1 = "Shamash";
   ...> s1.length();
s1 ==> "Shamash"
$6 ==> 7
```

2. Extract a single character. Use the **charAt()** method. Notice from the API spec that this returns a **char** and should be treated as a primitive character type.

```
jshell> String s1 = "Shamash";
   ...> char letter = s1.charAt(1);
s1 ==> "Shamash"
letter ==> 'h'
```

3. Find the position of a character in a String. Use the **indexOf()** method.

```
jshell> String s1 = "Shamash";
   ...> int index = s1.indexOf('h');
s1 ==> "Shamash"
index ==> 1
```

4. Find the last position of a character in a String. Use **lastIndexOf()**

```
jshell> String s1 = "Shamash";
   ...> int index = s1.indexOf('h');
s1 ==> "Shamash"
index ==> 1

jshell> int last_index = s1.lastIndexOf('h');
last_index ==> 6
```

5. Extract a sub-string. Use the **substring()** method

```
jshell> String s1 = "Shamash";
   ...> String sub = s1.substring(1, 4);
s1 ==> "Shamash"
sub ==> "ham"
```

Research the API for all of the details. You will hear me say this hundreds of times this semester. Expect it, it is required.

Java Arrays

An array is a sequence of values; the values in the array are called elements. For folks with only Python experience, an array in Java is analogous to a List in Python. **There are some very important differences though**. Pay close attention to this section.

You can make an array of ints, doubles, or any other type, **but all the values in an array must have the same type**. To create an array, you have to declare a variable with an array type and then create the array itself. Array types look like other Java types, except they are followed by square brackets ([]). For example, the following lines declare that counts is an “integer array” and values is a “double array”:

```
jshell> int[] counts;
counts ==> null

jshell> double[] values;
values ==> null
```

Notice above that the variables count and values have been created with an array type but are initially **null**. The keyword **null** is important in object oriented programming and represents the absence of an initialized object. You will be seeing this concept in many more situations, especially when we get into classes and objects. Arrays are **objects** not **primitives,** so array variables can be **null**, meaning they do not actually point to a valid sequence yet. To create the array you have to use the keyword **new**. This keyword will actually allocate the memory and associate the allocated memory with the variable. At this point the variable will not be null.

```
jshell> counts = new int[10];
counts ==> int[10] { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }

jshell> values = new double[10];
values ==> double[10] { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 }
```

These can be combined into a single line.

```
jshell> int[] counts = new int[10];
counts ==> int[10] { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }

jshell> double[] values = new double[10];
values ==> double[10] { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 }
```

Notice the array creation syntax. The integer value in the second set of [ ] represents the **length** of the array, or the number of elements the array will hold. This brings up another important aspect of Java arrays: **they are statically sized**. Once an array has been created, the size **cannot be changed.** The arrays above have been initialized with 10 default values of the specified type. These default values are supplied because the array is of a primitive type: **double and int**. If we were to create an array of objects there would be different results.

Check out this array of type String. Remember, **strings are objects**. Because this array is of an object type, the initial values are **null.** This is a different approach from a language like C++. The C++ compiler will actually call the **default constructor** on each of the objects in an array. More on default constructors later.

```
jshell> String[] names = new String[10];
names ==> String[10] { null, null, null, null, null, null, null, null, null, null }
```

Once an array has been created it can be indexed just like Python lists and C++ arrays.

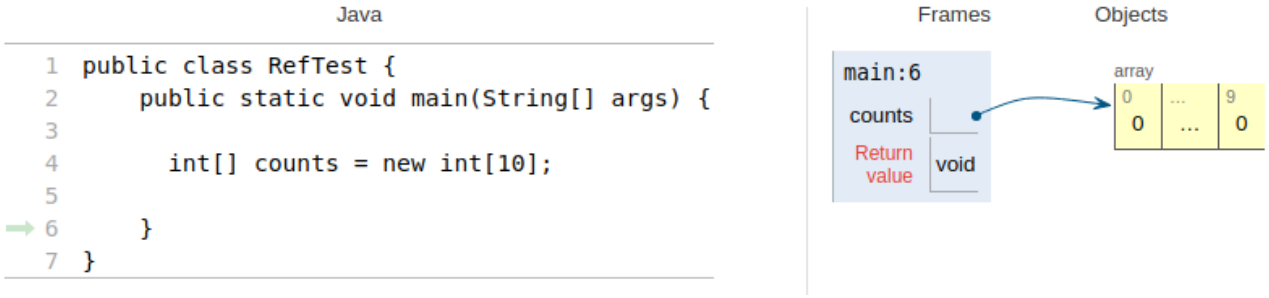
An important concept at this point is to understand the difference between an object and a reference. Examine this example where I attempt to print the contents of the **names** array.

```
jshell> System.out.println(names);
[Ljava.lang.String;@69663380
```

These results are unexpected. No errors occurred but the elements were not printed. JShell is nice enough to give the base type of the array and then some strange number. Without getting too much in the weeds with the details, this number is a value that maps the variable to the object itself. This value is called a **hash code** and it is not a direct physical RAM address but an internal JVM address that the run-time environment uses to provide object access through a variable. We will call these object variables: **references.**

Rule: Object variables refer to the object, they are not the object themselves.

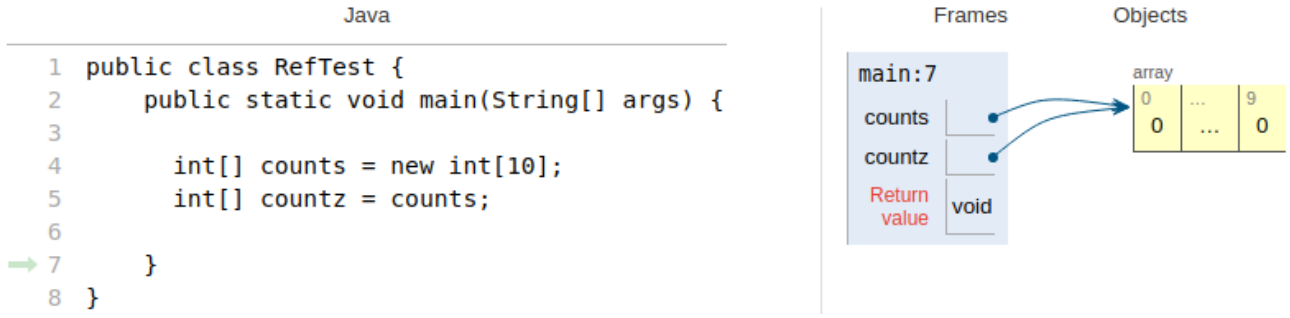
Moral of the Story: Object variables do not actually **store** the values they **"contain".** They simply refer to them., or point to them in memory. A reference variable in Java holds the address of the object. The PythonTutor variant for Java does a good job of graphically illustrating the relationship between object variables and the object itself. The arrow is showing you that the variable "**counts" points to the array object.** Again, we will use the term **reference.** The variable refers to the object.



Notice what happens when we assign one object variable to another object variable. If you look at the second print statement you will see that the variable **namez** prints the same hash code as **names.** That is because **they both refer to the same object.** This is a bit of an over simplification but it is illustrative of the point.

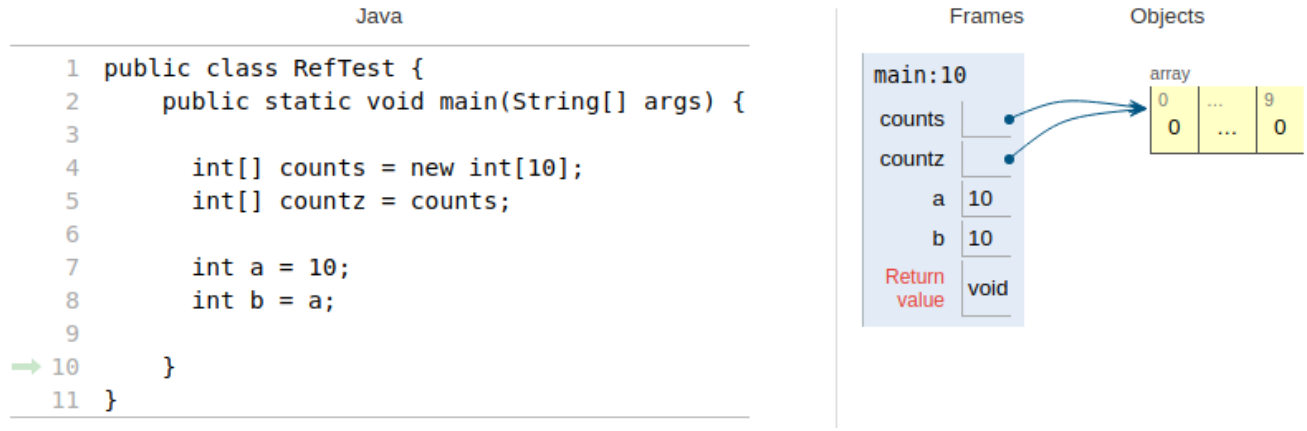
```
jshell> String[] names = new String[10];
names ==> String[10] { null, null, null, null, null, null, null, null, null, null }

jshell> System.out.println(names);
[Ljava.lang.String;@69663380

jshell> String[] namez = names;
namez ==> String[10] { null, null, null, null, null, null, null, null, null, null }

jshell> System.out.println(namez);
[Ljava.lang.String;@69663380
```

JavaTutor makes this clear . . . there is only a single object, but there are two references to this object.



One array, two pointers to the array. Make sure you wrap your head around this concept. It is quite important.

Notice that this does not occur when dealing with primitives



In the example above, line 8 generates a copy of the value in variable a. This is due to these variables being of **primitive type** not **object type.**

Two Dimensional Arrays (Arrays of Arrays)

A two dimensional array, is an array where each element is also an array. It is helpful to view this structure as two dimensional table that consists of an intersection of rows and columns.

This code creates the following table:

int[ ][ ] table = new int[3][4];

	Column 1	Column 2	Column 3	Column 4
Row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 3	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Initializing an 2D Array

int[ ][ ] a = { {1, 2, 3}, {4, 5, 6, 9}, {7} };

The statement above will create the following situation

	Column 1	Column 2	Column 3	Column 4
Row 1	1 a[0][0]	2 a[0][1]	3 a[0][2]	
Row 2	4 a[1][0]	5 a[1][1]	6 a[1][2]	9 a[1][3]
Row 3	7 a[2][0]			

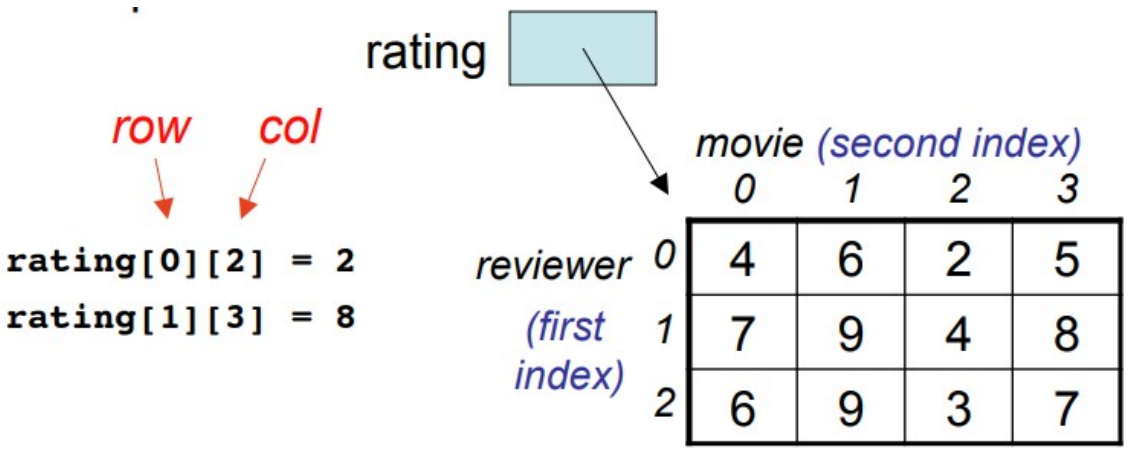
Often data come naturally in the form of a table, e.g., spreadsheet, which need a two-dimensional array.

Examples:

- Lab book of multiple readings over several days
- Periodic table
- Movie ratings by multiple reviewers
  - Each row is a different reviewer
  - Each column is a different movie

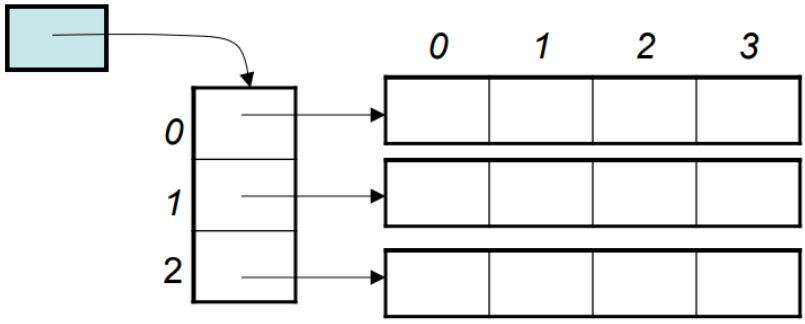
Two-dimensional (2D) arrays are indexed by two subscripts, one for the row and one for the column. •

Example:



Notes:

- Each element in the 2D array must by the same type, either a primitive type or object type.
- Subscripted variables can be use just like a variable: **rating[0][3] = 10;**
- Array indices must be of type int and can be a literal, variable, or expression. r**ating[3][j] = j;**
- If an array element does not exists, and you attempt to index it, the JREwill give you an ArrayIndexOutOfBoundsException
- A 2D array is a 1D array of (references to) 1D arrays. **int[][] rating = new int[3][4];**



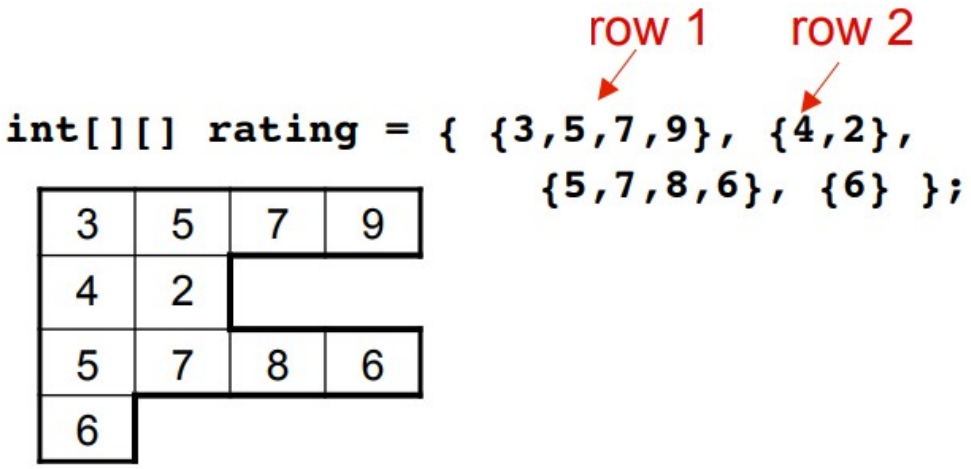
Size of 2D Arrays

Given **int[][] rating = new int[3][4];** What is the value of rating.length? Answer: 3, the number of rows (first dimension)

What is the value of rating[0].length? Answer: 4, the number of columns (second dimension)

Ragged Arrays

Since a 2D array is a 1D array of references to 1D arrays, each of these latter 1D arrays (rows) can have a different length.



We will be revisiting arrays and 2 dimensional arrays in much more detail in the future. Be sure you understand this concept. Experiment with arrays.