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 store 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 data 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, and we will cover these as we progress)

```
// illustrate some variable declarations
 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; // can declare two variables on a single line
```

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***. Choose a style and stick with it.

The accepted Java style is called camel case: **https://en.wikipedia.org/wiki/Camel_case**

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 and organize 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 like VS Code and Eclipse 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;
```

Java Primitive Data Types

The following list describes the basic data types supported by Java. 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.

The information describing these data types includes their ***bit size.*** If you are unfamiliar with the binary numbering system, now would be a good time to familiarize yourself.

https://www.computerhope.com/jargon/b/binary.htm

The number of bits assigned to a data type will dictate the allowable range of values for variables of this type. For instance, an integer in Java is a 32-bit value. That means that for each ***int*** declared in a program, 32 bits will be set aside in RAM for the values. Values that are larger or smaller than 32 bits allow will not be able to be stored in these variables. This is an important concept and quite different from the way Python handles data. When the documentation says that a typed is ***signed*** that simply means that it will support both positive and negative values. If a type is described as ***unsigned*** that means that it will only allow positive values. The number theory for these concepts is a bit beyond the scope of this course and will be covered in subsequent CS courses.

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). 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 signed 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 signed 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 signed 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 a single unsigned 16-bit Unicode character. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive).

Default Values

It's not always necessary to assign a value when a field is declared. 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 and you should get into the habit of initializing your variables.

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. Python does not support this concept.

Printing Variables

The following examples and many more examples to come utilize the **JShell** tool. This tool is similar to Python's IDLE and the command line Python interpreter and allows you to experiment with language features ***without*** having to create, save, compile and execute a Java program.

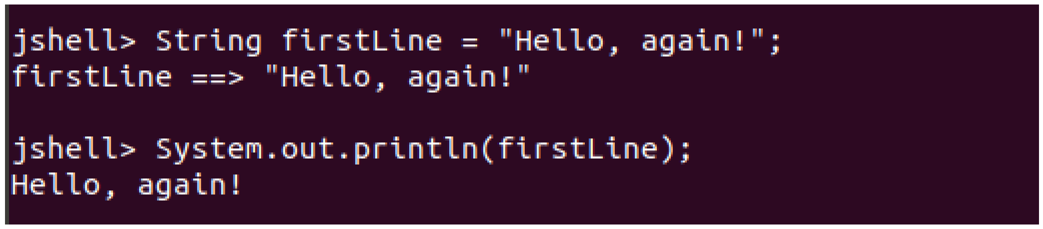
It's is just for experimentation . . . it is not a development tool.

Read about it here:

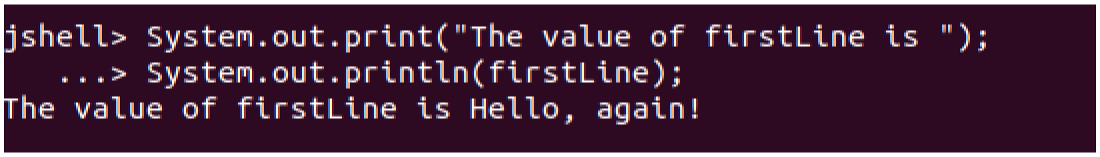
https://docs.oracle.com/javase/9/jshell/introduction-jshell.htm#JSHEL-GUID-630F27C8-1195-4989-9F6B-2C51D46F52C8

I recommend following along with these examples to get the hang of JShell.

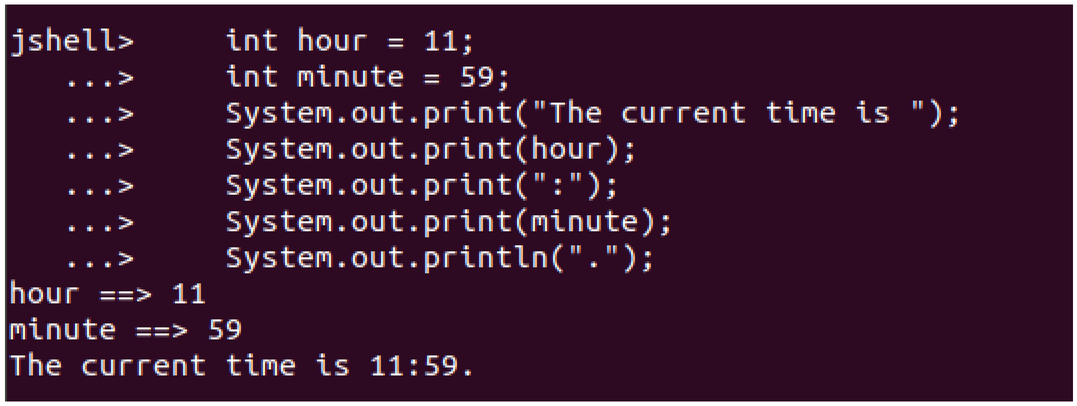
You can display the value of a variable using the methods ***print*** or ***println***. The following statements declare a variable named firstLine, assign it the value "Hello, again!", and display that value.


```
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!
```

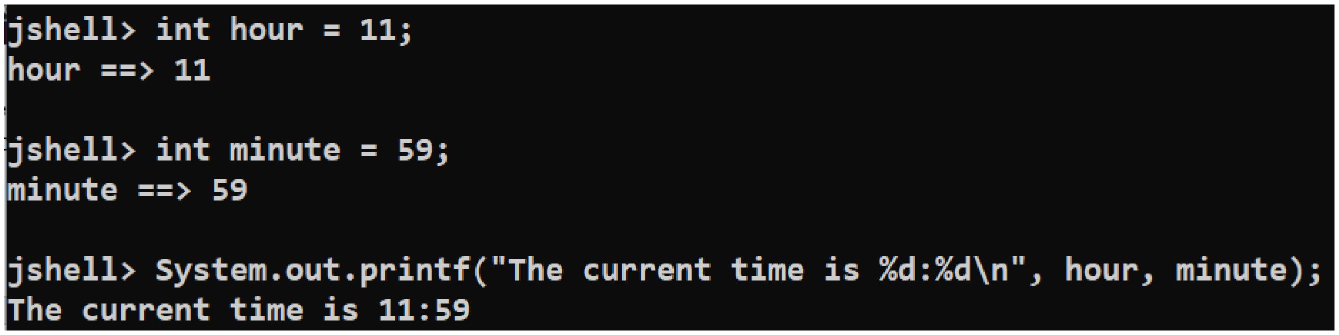
Conveniently, the syntax for displaying a variable is the same regardless of its type. For example:


```
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.
```

println: Prints the values and forces a line break
print: Prints the values and does not force a line break

printf: Allows printing with a specified format

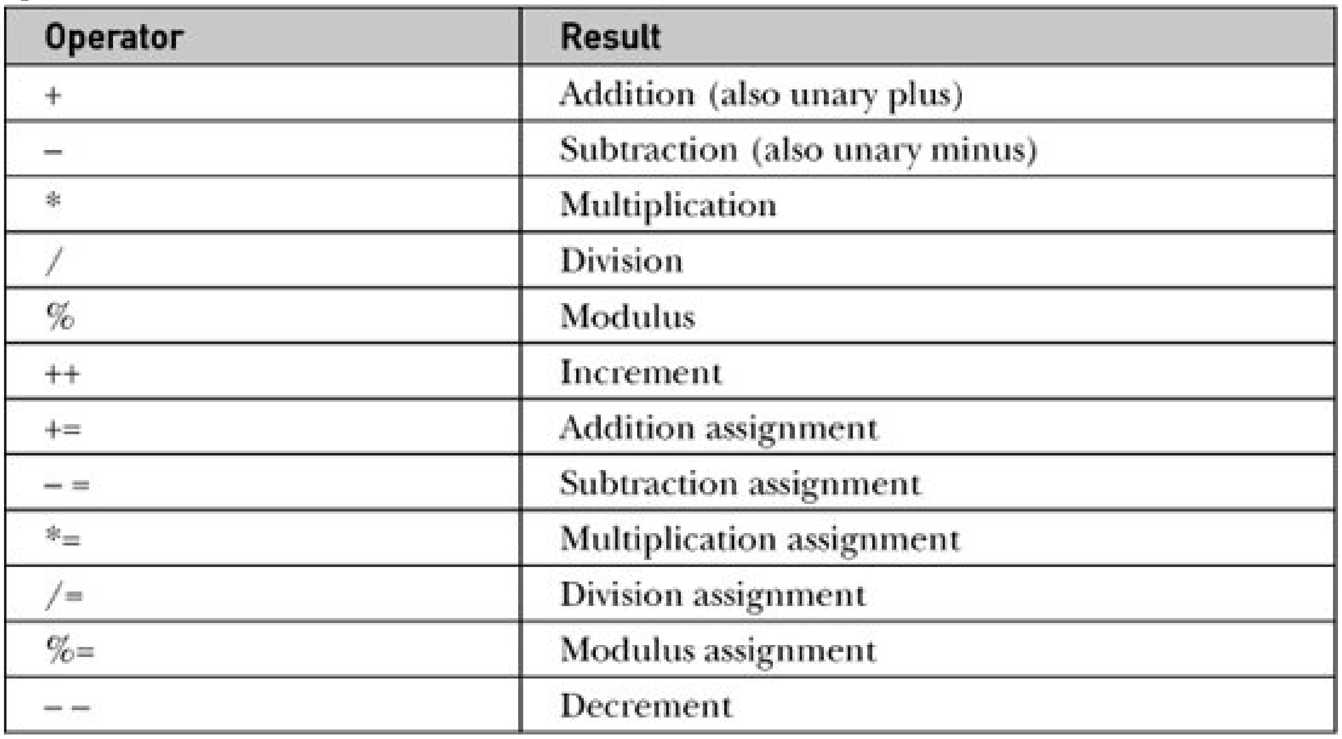
https://www.journaldev.com/28692/java-printf-method



```
jshell> int hour = 11;
hour ==> 11

jshell> int minute = 59;
minute ==> 59

jshell> System.out.printf("The current time is %d:%d\n", hour, minute);
The current time is 11:59
```

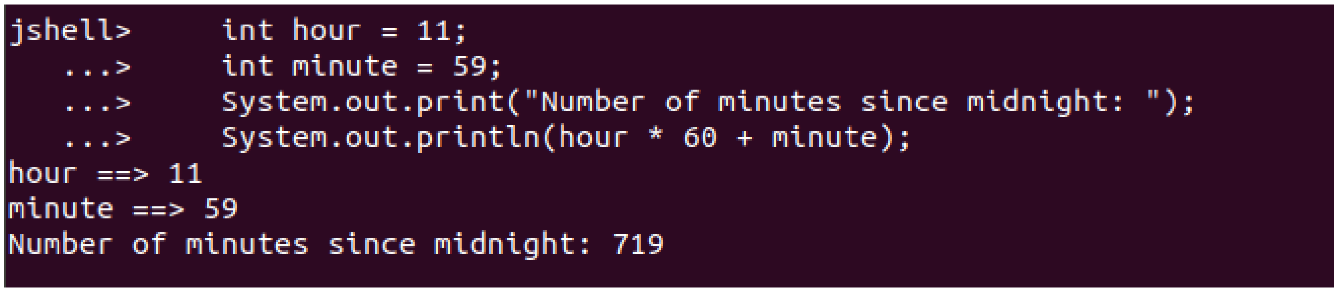
Arithmetic operators

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 are resolved to a single value.

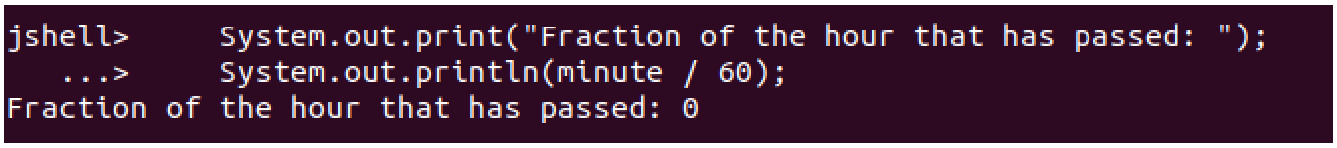
For example:

- 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

(maybe not after CS1, but you will find the rules for division are quite different from Python. This is due to the strict data-typing of Java).

Continuing with the example from above, the following fragment tries to compute the fraction of an hour that has elapsed:



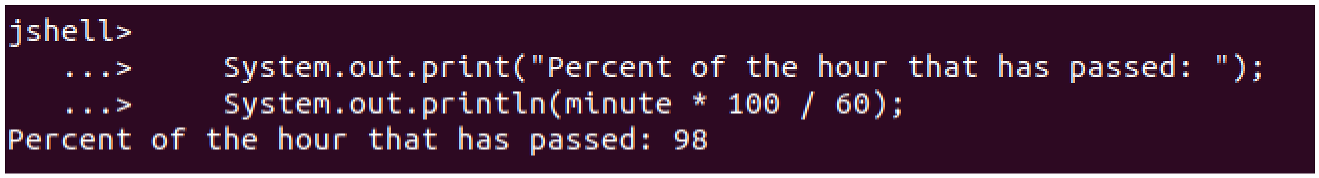
```
jshell>     System.out.print("Fraction of the hour that has passed: ");
   ...>     System.out.println(minute / 60);
Fraction of the hour that has passed: 0
```

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
```

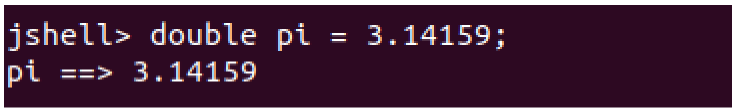
The new output is: **Percent of the hour that has passed: 98**

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.

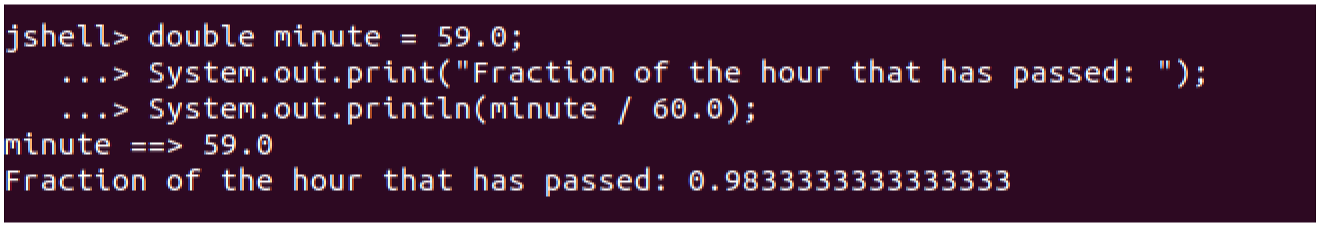
Remember: The data type of the variable that stores a result ***does not determine*** the type of the expression

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:



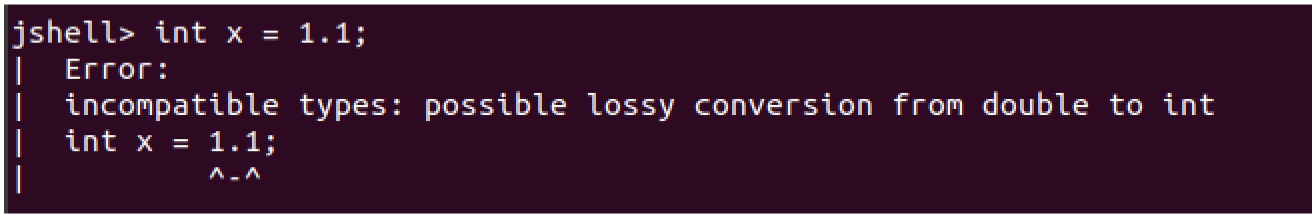
```
jshell> double pi = 3.14159;
pi ==> 3.14159
```

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:



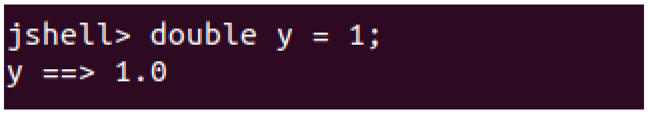
```
jshell> double minute = 59.0;
   ...> System.out.print("Fraction of the hour that has passed: ");
   ...> System.out.println(minute / 60.0);
minute ==> 59.0
Fraction of the hour that has passed: 0.9833333333333333
```

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 (another major difference from Python). The following is illegal because the variable on the left is an int and the value on the right is a double:



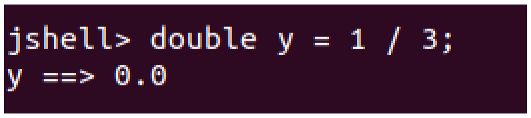
```
jshell> int x = 1.1;
|  Error:
|  incompatible types: possible lossy conversion from double to int
|  int x = 1.1;
|          ^-^
```

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.



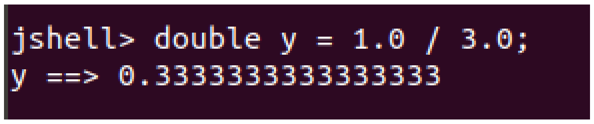
```
jshell> double y = 1;
y ==> 1.0
```

The preceding example should be illegal, but Java allows it by converting the i**nt 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



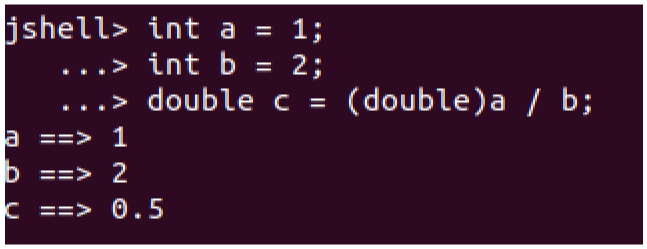
```
jshell> double y = 1 / 3;
y ==> 0.0
```

In this scenario the integers 1 and 3 are divided using **integer division** which results in a truncated value of 0. The 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.



```
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 directly 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 convert a value of one data type to another type.

In Java, there are two types of casting:

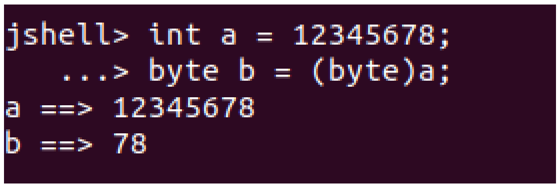
Widening Casting: (automatic) - converting a smaller type (bit size) to a larger type (bit size) can happen automagically as we have just seen. A widening cast can happen in any of the following situations. Notice how the bit sizes increase from left to right . . . you are ***widening*** the bit size.

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

Narrowing Casting: (manual) - converting a larger type to a smaller size type has the opportunity to lose precision as you are converting from a larger size of allocated memory to a smaller size of allocated memory. 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.

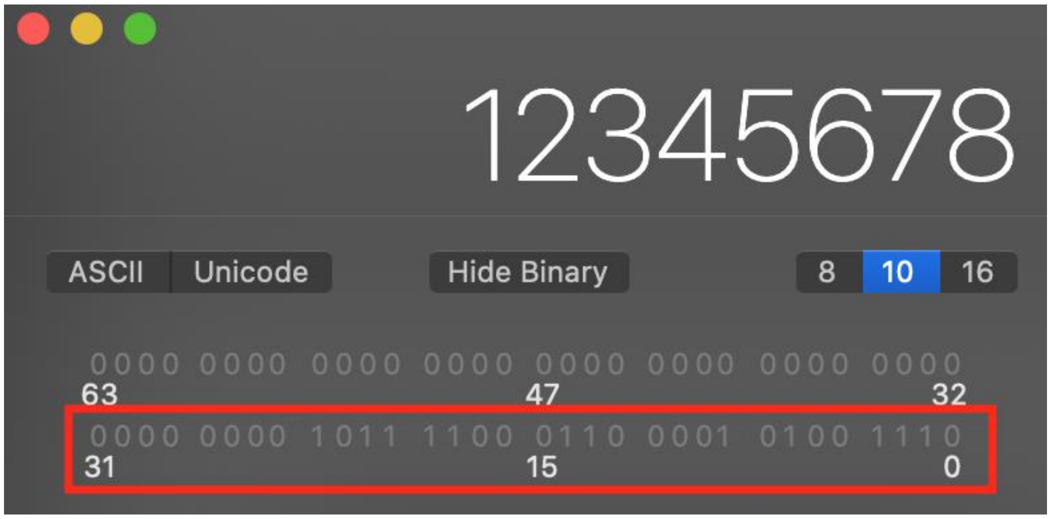
`double` → `float` → `long` → `int` → `char` → `short` → `byte`

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

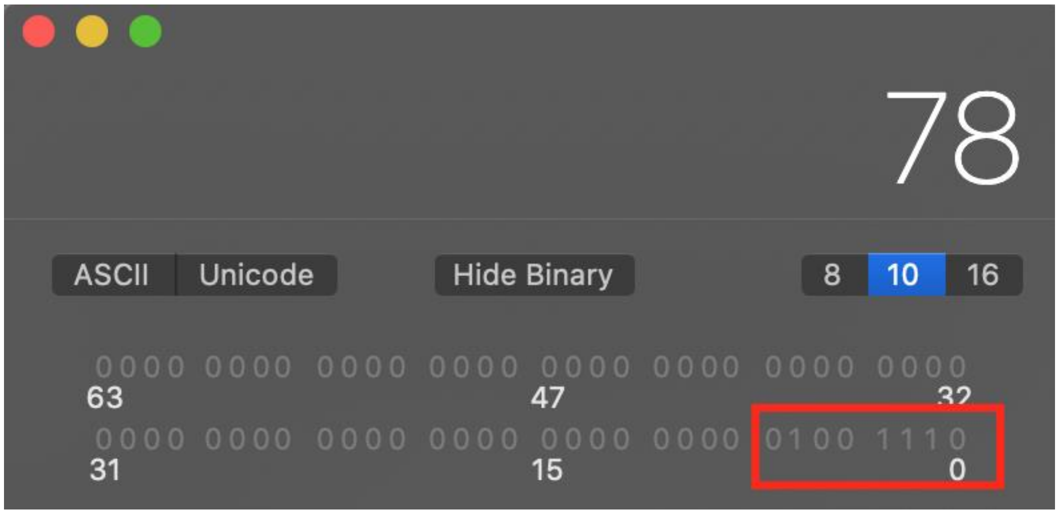


```
jshell> int a = 12345678;
   ...> byte b = (byte)a;
a ==> 12345678
b ==> 78
```

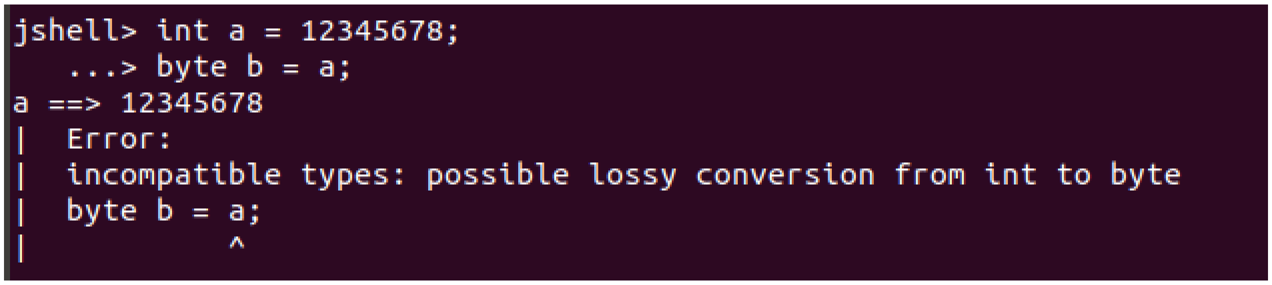
What's going on here? A Java integer is 32 bits of storage. A Java byte is 8 bits of storage. That type cast essentially truncated 24 bits off of the variable **a**. Use this wisely!



```
12345678

0000 0000 0000 0000 0000 0000 0000 0000
63                  47                32
0000 0000 1011 1100 0110 0001 0100 1110
31                  15                 0
```



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**



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

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 programmers 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** the 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.

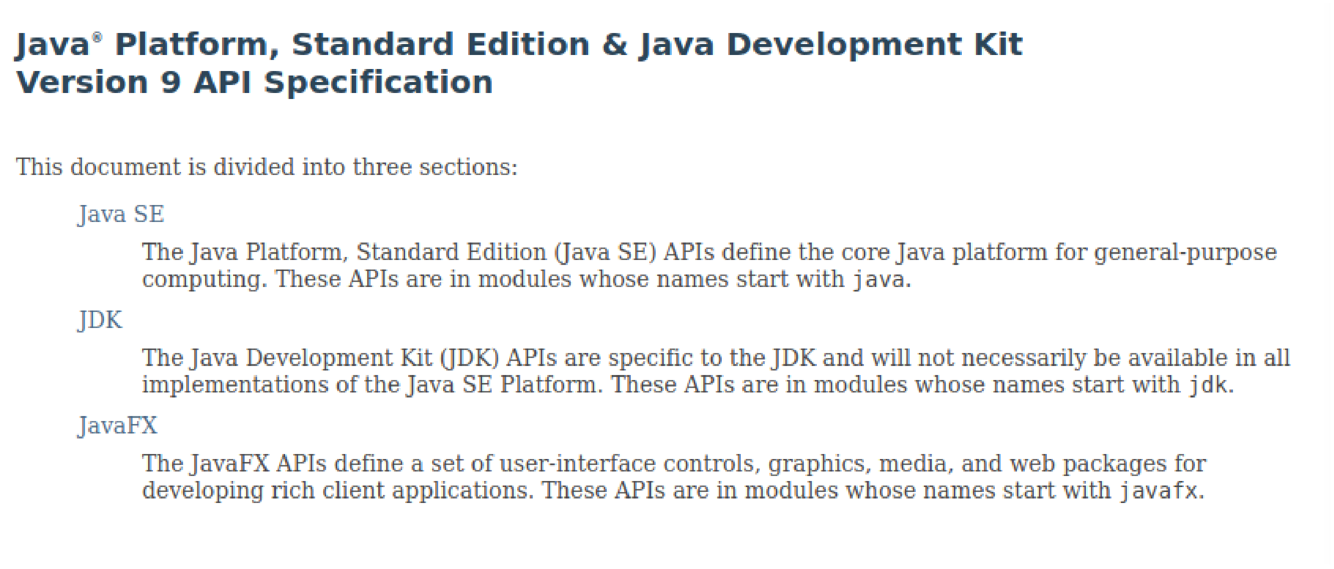
The separation of the API from its implementation can allow programs written in one language to use a library written in another. For example, because Scala and Java compile to compatible bytecode, Scala developers can take advantage of any Java API.

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`.

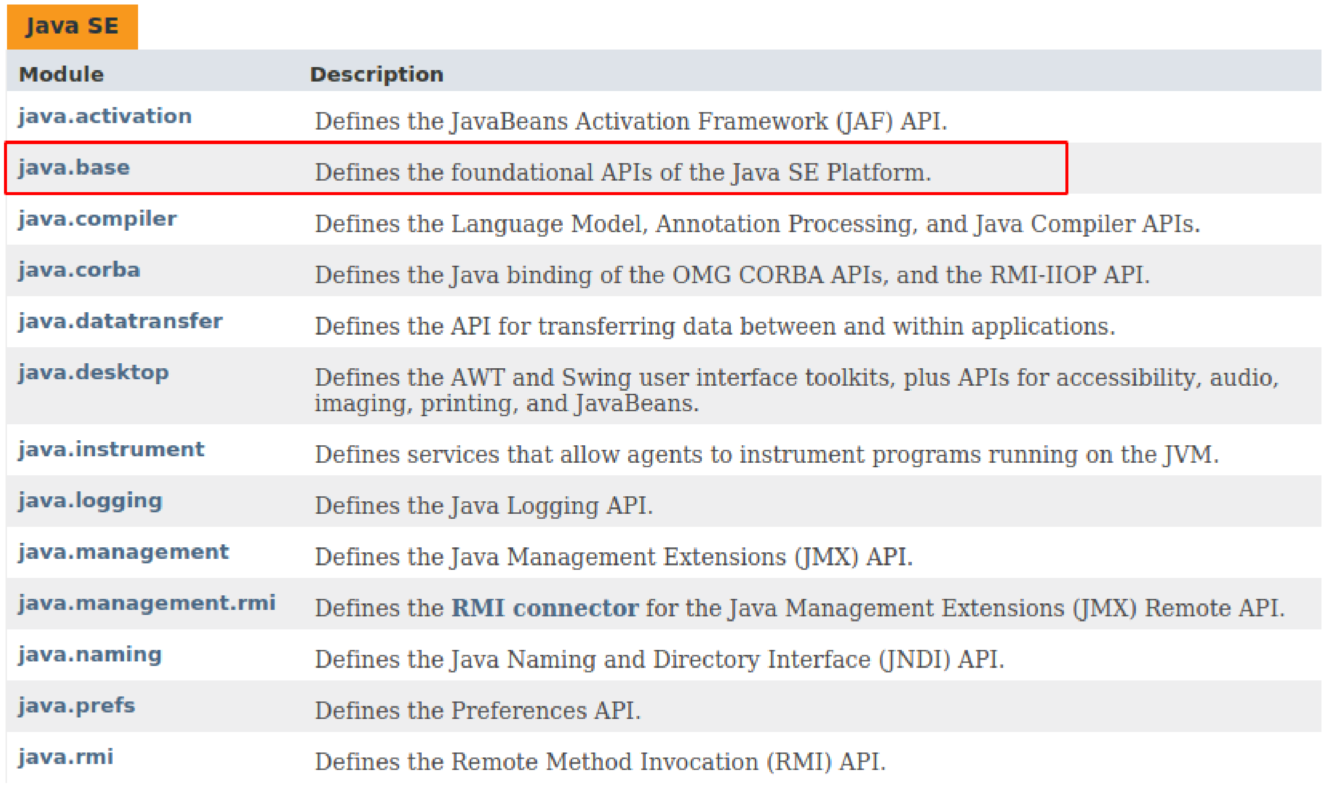
JavaFX

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 selecting the API for your version of Java. You can find it in the **java.base** module


Java SE

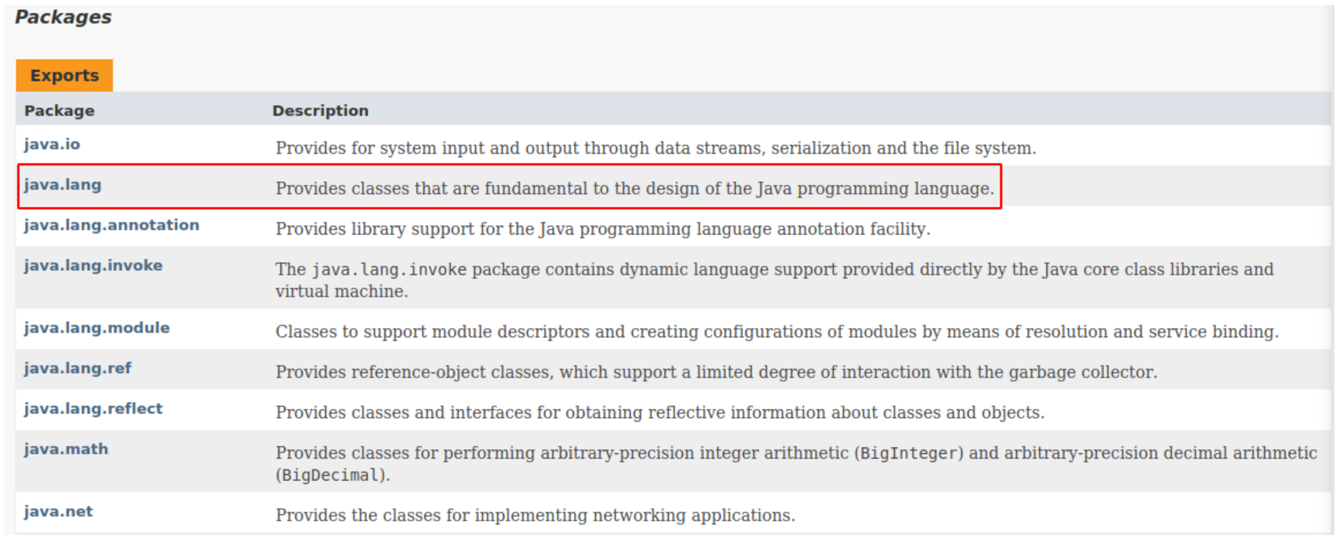
Module	Description
java.activation	Defines the JavaBeans Activation Framework (JAF) API.
java.base	Defines the foundational APIs of the Java SE Platform.
java.compiler	Defines the Language Model, Annotation Processing, and Java Compiler APIs.
java.corba	Defines the Java binding of the OMG CORBA APIs, 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.

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

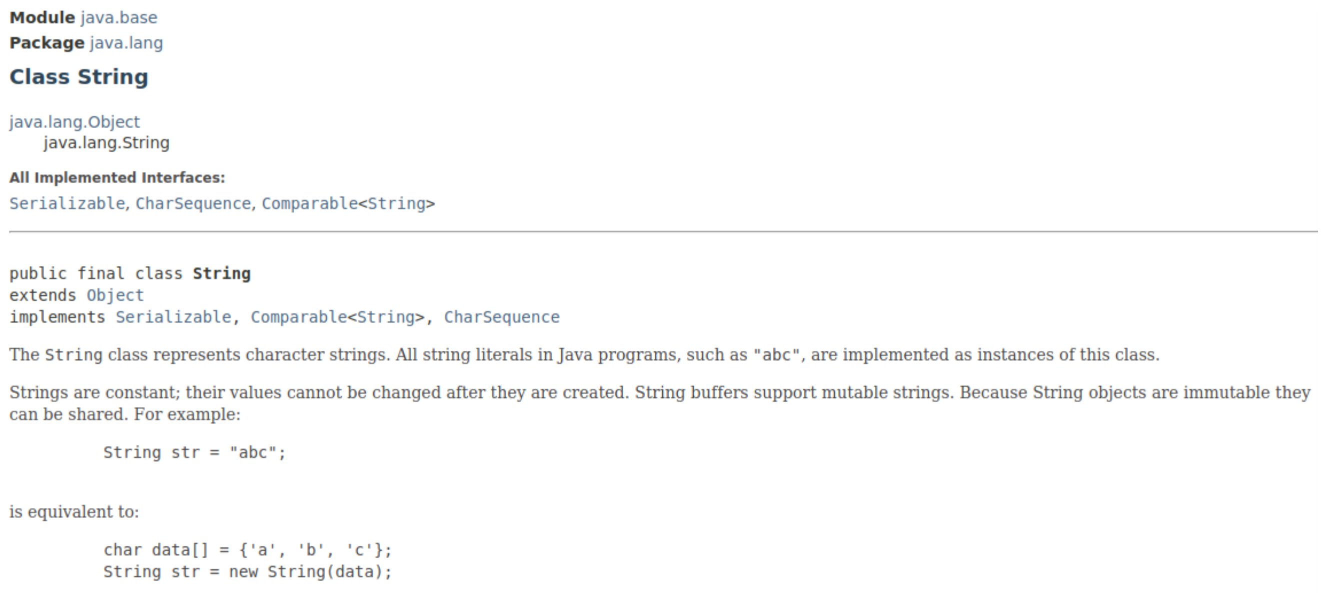


Packages

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.



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:

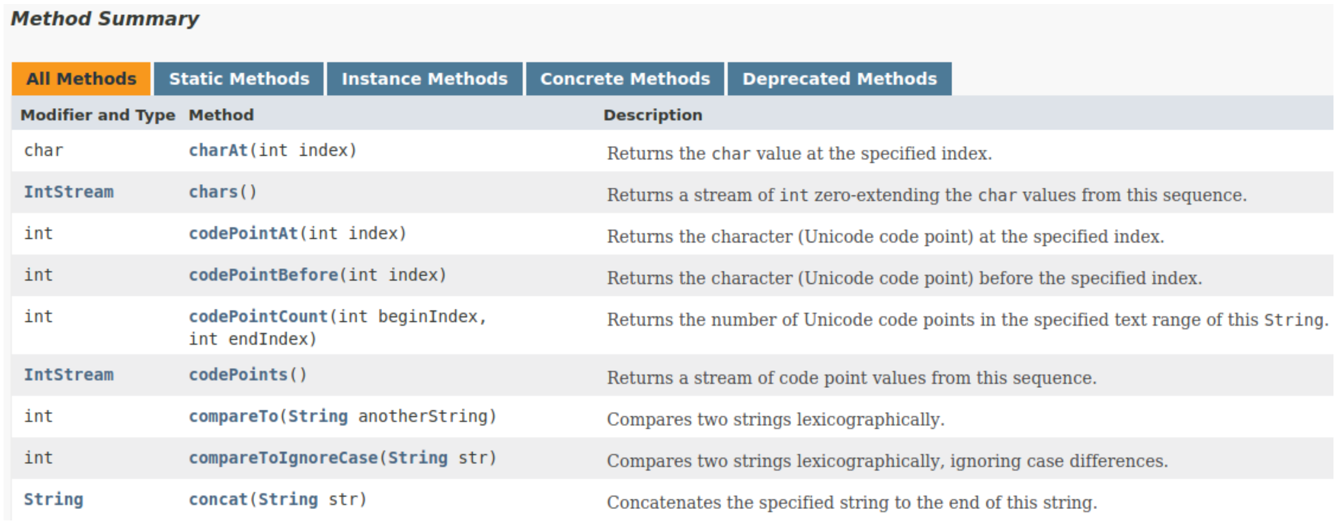
```
     char data[] = {'a', 'b', 'c'};
     String str = new String(data);
```

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 Scanner 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 names
2. The argument lists
3. The return types

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

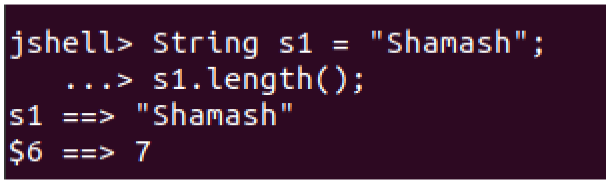
Modifier and Type	Method	Description
`char`	`charAt(int index)`	Returns the `char` value at the specified index.
`IntStream`	`chars()`	Returns a stream of `int` zero-extending the `char` values from this sequence.
`int`	`codePointAt(int index)`	Returns the character (Unicode code point) at the specified 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.
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.

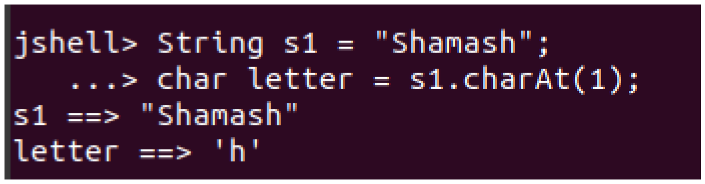
Demonstration of the most common String operations. We operate on objects by first creating them, and then sending them messages in the form of ***method calls***. Strings are ***objects*** that contain ***methods*** that operate on those objects. You should notice the use of ***dot notation*** here. You prefix the method call with the name of the object. This tells Java which object you would like to manipulate. This is another significant difference from Python, which is a mix of functional and object-based programming.

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



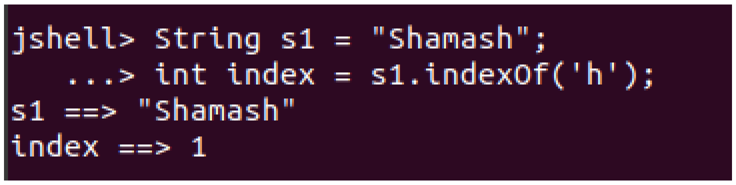
```
jshell> String s1 = "Shamash";
   ...> s1.length();
s1 ==> "Shamash"
$6 ==> 7
```

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***.



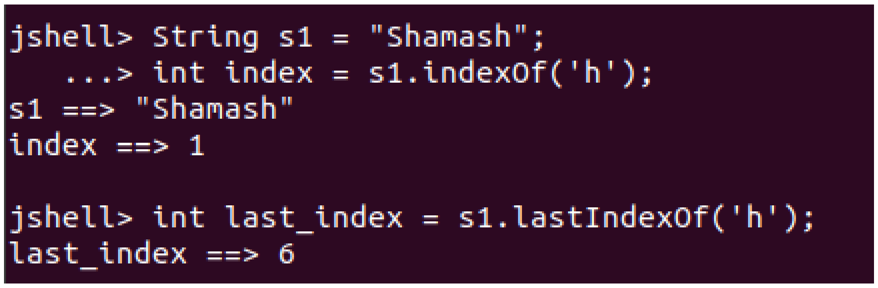
```
jshell> String s1 = "Shamash";
   ...> char letter = s1.charAt(1);
s1 ==> "Shamash"
letter ==> 'h'
```

3. Find the position of a character in a String. Use the **indexOf()** method. This will return the first index of the character if present. If not, it will return -1. ***indexOf*** acts just like the find() method in Python. The important distinction is the the explicit usage of data types. In Python single quotes and double quotes essentially mean the same thing. In Java they do not. Double quotes in Java are used for String objects and single quotes are used for character primitives. This is an important distinction.



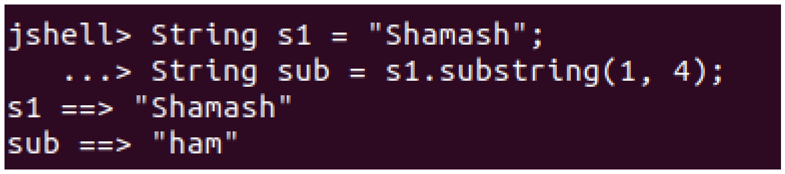
```
jshell> String s1 = "Shamash";
   ...> int index = s1.indexOf('h');
s1 ==> "Shamash"
index ==> 1
```

4. Find the last position of a character in a String. Use **lastIndexOf().** This will return the last index of the character if present. If not, it will return -1



```
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. This is similar to Python slicing where you provide the ***start index*** and the non-inclusive ***ending index.***



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

Research the API for all of the details. **You will here me say this hundreds of times this semester.**

Viewing API details in a programming environment

Integrated Development Environments like ***VS Code*** and ***Eclipse*** provide ways to locally view the details of the Java libraries.

VS Code

VS Code operates on the concept of a ***workspace.*** In Visual Studio Code, a "Workspace" means a collection of one or more filesystem folders (and their children) and all of the VS Code configurations that take effect when that "Workspace" is open in VS Code. There are two kinds of "Workspaces" in VS Code, "folder workspaces" and "multi-root workspaces".

A "folder workspace" is presented by VS Code when you open a filesystem folder (directory) in VS Code. All code shown in the course materials is shown as a folder workspace.

If you haven't already, install VS Code. Most of the screen shots that you see in the course material will be illustrating VS Code.

My favorite aspect of VS Code is the integrated terminal that allows you to perform command line processes like compiling and executing Java code and file management, without having to have a separate window open.

Installation: https://code.visualstudio.com/

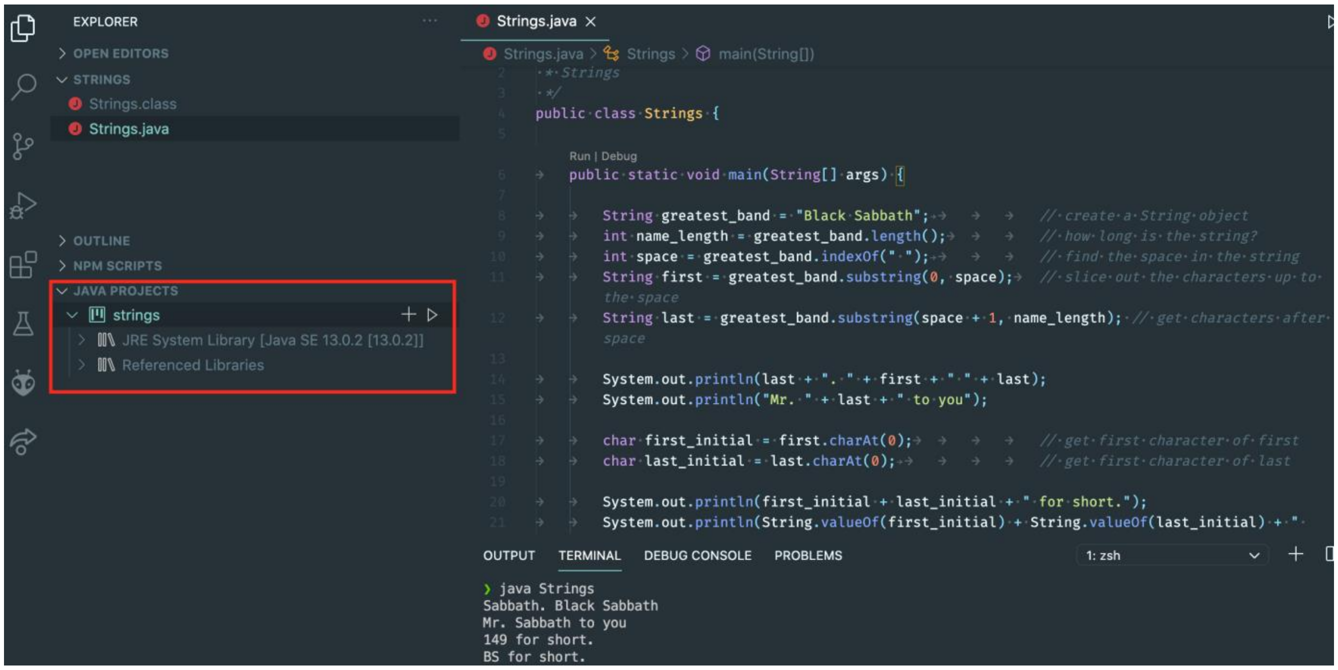
Java Configuration: https://code.visualstudio.com/docs/languages/java

Video Step by Step: https://youtu.be/ClU9N4ub_Ko

There are many videos available for these concepts. I chose the one above for two important reasons

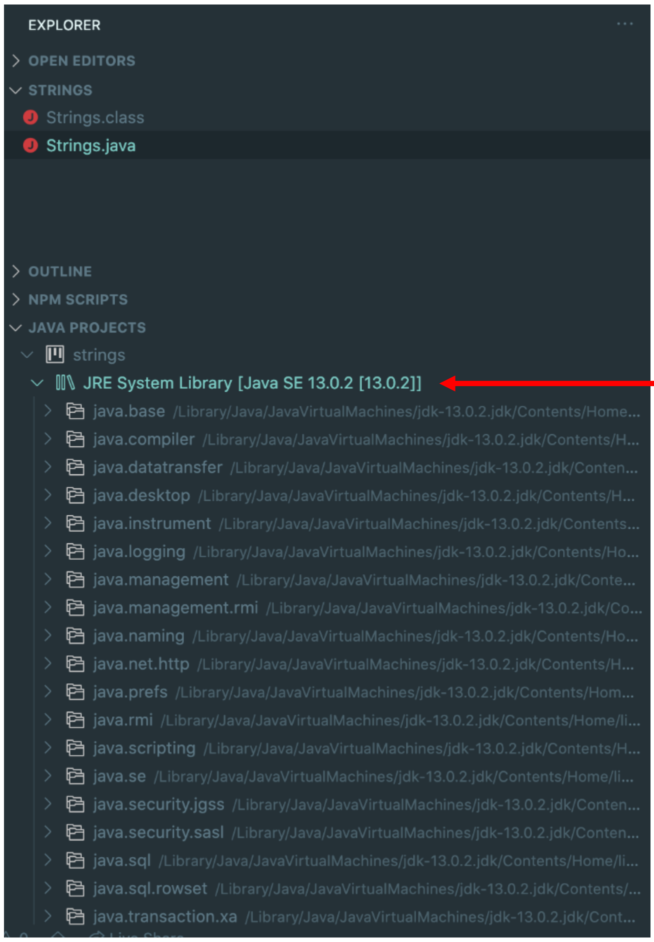
1. It illustrates the simple task of creating a VS Code workspace by typing ***code .*** from a terminal. Pay close attention to this. VS Code works ***much more reliably*** when you work with workspaces, instead of simply opening files. This will be my first question when asked about configuration issues or VS Code acting funny. **VERY IMPORTANT**
2. It gives a brief introduction to running Java programs in ***debug mode***. This is also incredibly important!! Expect much more on this subject and be sure that you can successfully run your code in debug mode. This will be my very first question when asked about code not working properly. If we participate in a screen sharing session or a live programming session I will expect you to be able to use the debugger.

With a VS Code workspace opened in a Java context (with a ***.java*** file present) you can find the JRE System Library at your fingertips. Make sure that the ***Explorer*** is open.



```java
 * Strings
 */
public class Strings {

    public static void main(String[] args) {

        String greatest_band = "Black Sabbath";          // create a String object
        int name_length = greatest_band.length();        // how long is the string?
        int space = greatest_band.indexOf(" ");          // find the space in the string
        String first = greatest_band.substring(0, space);   // slice out the characters up to the space
        String last = greatest_band.substring(space + 1, name_length); // get characters after space

        System.out.println(last + ". " + first + " " + last);
        System.out.println("Mr. " + last + " to you");

        char first_initial = first.charAt(0);            // get first character of first
        char last_initial = last.charAt(0);              // get first character of last

        System.out.println(first_initial + last_initial + " for short.");
        System.out.println(String.valueOf(first_initial) + String.valueOf(last_initial) + "
```

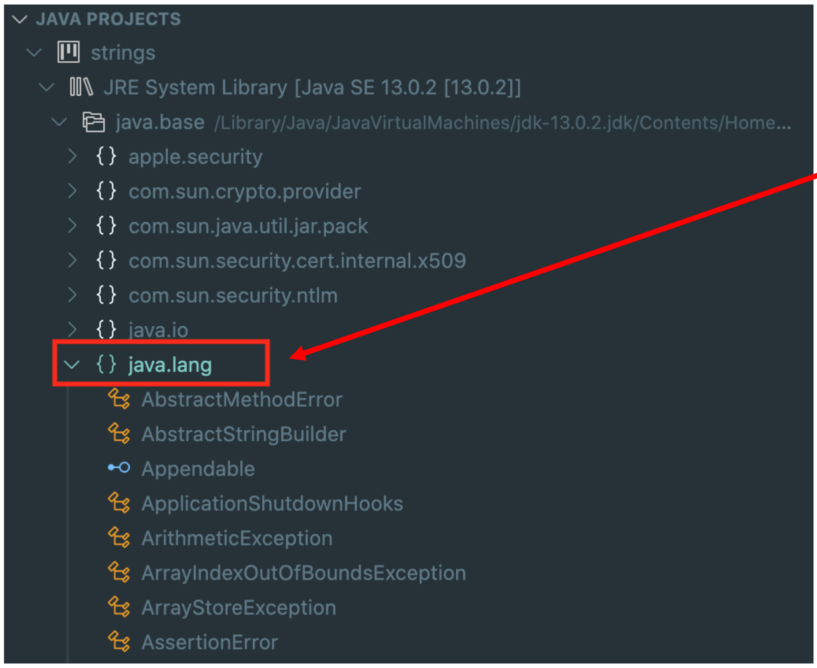
```
❯ java Strings
Sabbath. Black Sabbath
Mr. Sabbath to you
149 for short.
BS for short.
```


EXPLORER
OPEN EDITORS
STRINGS
Strings.class
Strings.java
OUTLINE
NPM SCRIPTS
JAVA PROJECTS
strings
JRE System Library [Java SE 13.0.2 [13.0.2]]
java.base /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents/Home...
java.compiler /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents/H...
java.datatransfer /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Conten...
java.desktop /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents/H...
java.instrument /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents...
java.logging /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents/Ho...
java.management /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Conte...
java.management.rmi /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Co...
java.naming /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents/Ho...
java.net.http /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents/H...
java.prefs /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents/Hom...
java.rmi /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents/Home/li...
java.scripting /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents/H...
java.se /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents/Home/li...
java.security.jgss /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Conten...
java.security.sasl /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Conten...
java.sql /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents/Home/li...
java.sql.rowset /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents/...
java.transaction.xa /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Cont...

Drop down **Java Projects → workspace name → JRE System Library**

This will expose the foundational modules of the Java API. These can be drilled into to find Class specific listings

Let’s expose the details of the String class



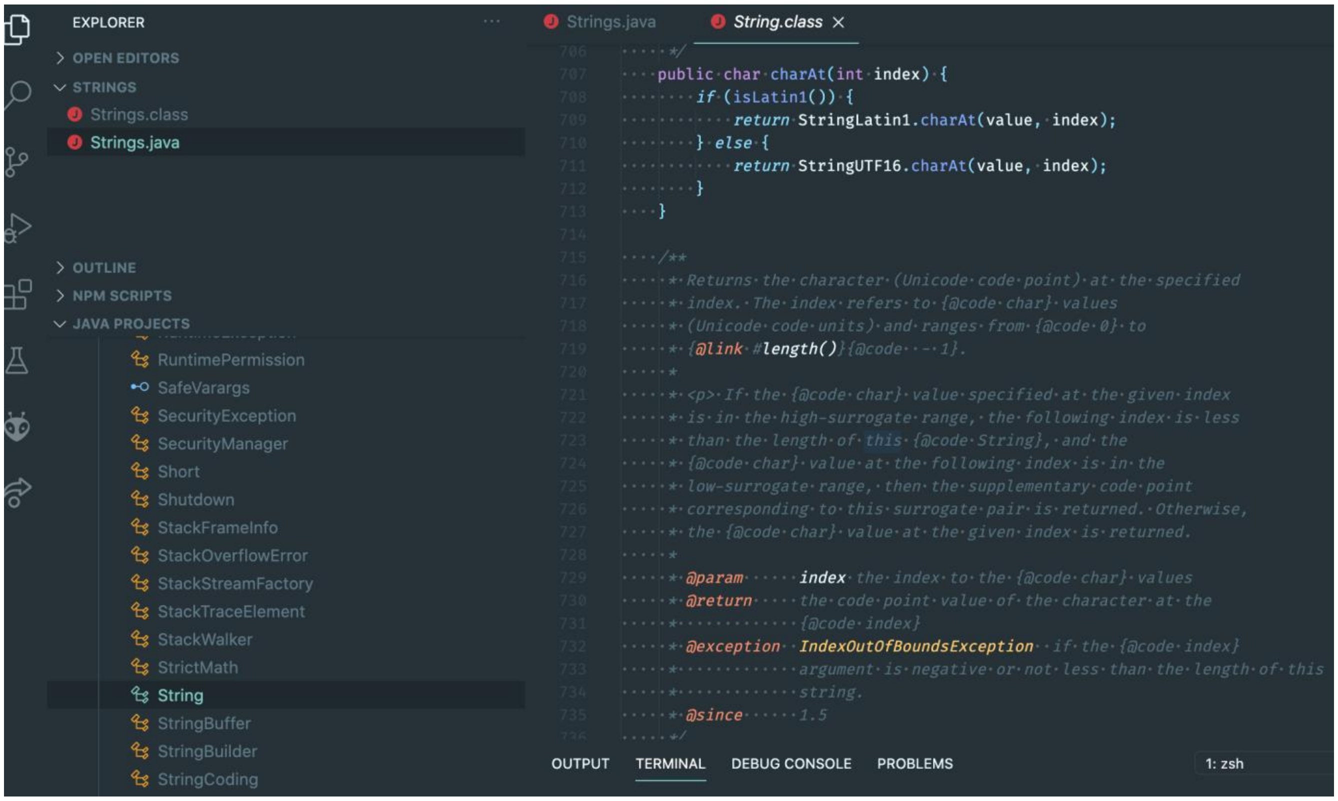
```
JAVA PROJECTS
  strings
    JRE System Library [Java SE 13.0.2 [13.0.2]]
      java.base /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents/Home...
        {} apple.security
        {} com.sun.crypto.provider
        {} com.sun.java.util.jar.pack
        {} com.sun.security.cert.internal.x509
        {} com.sun.security.ntlm
        {} java.io
        {} java.lang
          AbstractMethodError
          AbstractStringBuilder
          Appendable
          ApplicationShutdownHooks
          ArithmeticException
          ArrayIndexOutOfBoundsException
          ArrayStoreException
          AssertionError
```

The String class is in the ***java.lang*** package which is contained in the ***java.base*** module.

Drill down into that

Now you can have a look at the Java code that implements a String, including documentation strings.

The screen shot below shows the ***charAt*** method.



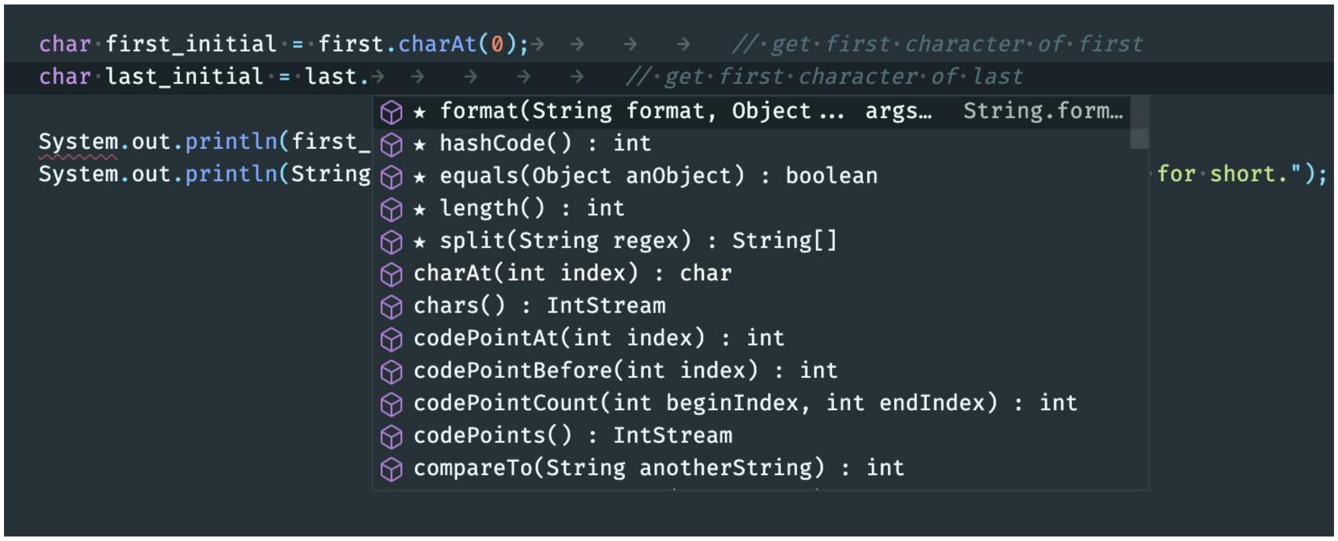
```
EXPLORER
> OPEN EDITORS
v STRINGS
  Strings.class
  Strings.java
> OUTLINE
> NPM SCRIPTS
v JAVA PROJECTS
    RuntimePermission
    SafeVarargs
    SecurityException
    SecurityManager
    Short
    Shutdown
    StackFrameInfo
    StackOverflowError
    StackStreamFactory
    StackTraceElement
    StackWalker
    StrictMath
    String
    StringBuffer
    StringBuilder
    StringCoding
```

Strings.java | String.class

```java
     */
    public char charAt(int index) {
        if (isLatin1()) {
            return StringLatin1.charAt(value, index);
        } else {
            return StringUTF16.charAt(value, index);
        }
    }

    /**
     * Returns the character (Unicode code point) at the specified
     * index. The index refers to {@code char} values
     * (Unicode code units) and ranges from {@code 0} to
     * {@link #length()}{@code  - 1}.
     *
     * <p> If the {@code char} value specified at the given index
     * is in the high-surrogate range, the following index is less
     * than the length of this {@code String}, and the
     * {@code char} value at the following index is in the
     * low-surrogate range, then the supplementary code point
     * corresponding to this surrogate pair is returned. Otherwise,
     * the {@code char} value at the given index is returned.
     *
     * @param      index the index to the {@code char} values
     * @return     the code point value of the character at the
     *             {@code index}
     * @exception  IndexOutOfBoundsException  if the {@code index}
     *             argument is negative or not less than the length of this
     *             string.
     * @since      1.5
     */
```

OUTPUT | TERMINAL | DEBUG CONSOLE | PROBLEMS | 1: zsh

This convenient organization allows you to discover details of methods that are provided by the ***intellisense*** feature of an IDE. Intellisense will show you available methods, parameter types and order and return types. All of this ***offline.*** Incredibly useful! Don't be intimidated by the overwhelming amount of data . . . just pick and choose what you need and look around.



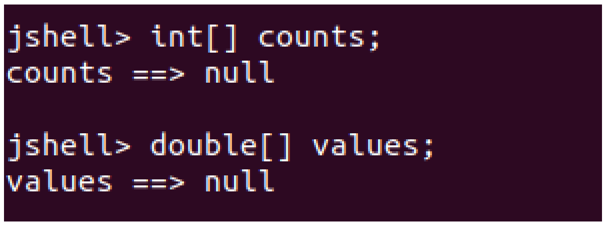
```
char first_initial = first.charAt(0);        // get first character of first
char last_initial = last.                    // get first character of last

System.out.println(first_
System.out.println(String                                          for short.");
```

```
★ format(String format, Object ... args…   String.form…
★ hashCode() : int
★ equals(Object anObject) : boolean
★ length() : int
★ split(String regex) : String[]
charAt(int index) : char
chars() : IntStream
codePointAt(int index) : int
codePointBefore(int index) : int
codePointCount(int beginIndex, int endIndex) : int
codePoints() : IntStream
compareTo(String anotherString) : int
```

Java Arrays

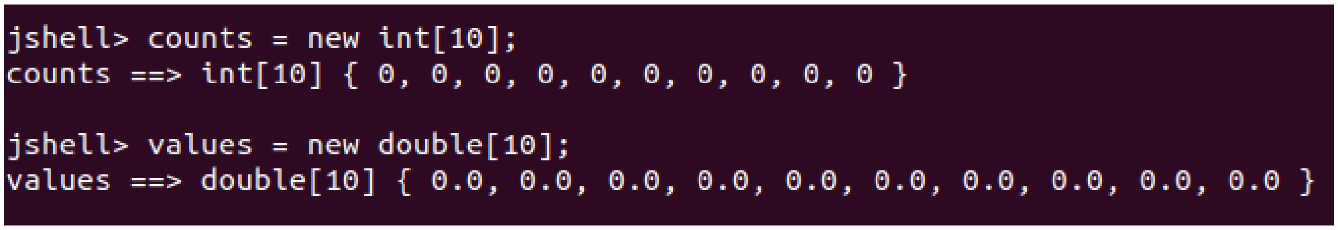
An array is a sequence of values; the values in the array are called elements. For folks with only Python experience, an array in Java is analogous to a List in Python. **There are some very important differences though**. Pay close attention to this section.

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 the variable ***counts*** is an "integer array" and the variable ***values*** is a "double array":



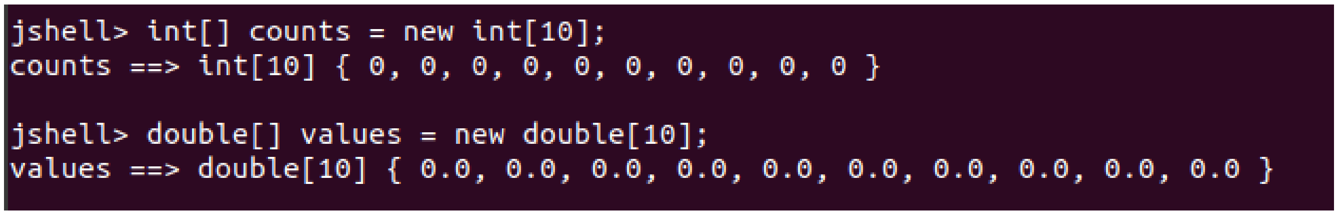
```
jshell> int[] counts;
counts ==> null

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

Notice above that the variables ***counts and values*** have been created with an array type but are initially **null**. The keyword **null** is an important concept 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 }
```

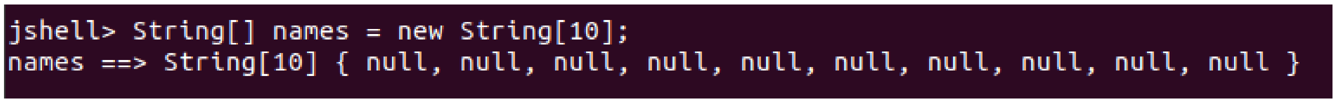
These can be combined into a single line.



```
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. You cannot continually append to a Java array like you can a Python list.** 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.

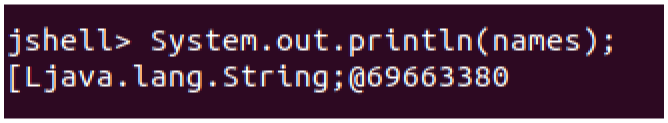
Check out this array of type String. Remember, ***strings are objects***. Because this array is of an object type, the initial values are **null.** This is a different approach from a language like C++. The C++ compiler will actually call the **default constructor** on each of the objects in an array.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

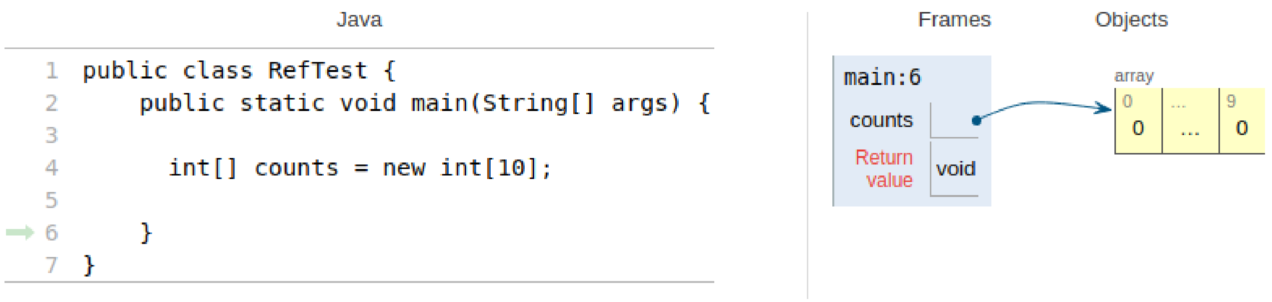


```
jshell> System.out.println(names);
[Ljava.lang.String;@69663380
```

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 things **references.**

Rule: Object variables ***refer*** to the actual object, they are not the object themselves. Become familiar with the term ***reference*** as it is an important concept in OOP.

Moral of the Story: Object variables do not actually **store** the values they **"contain".** They simply refer to them. 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**



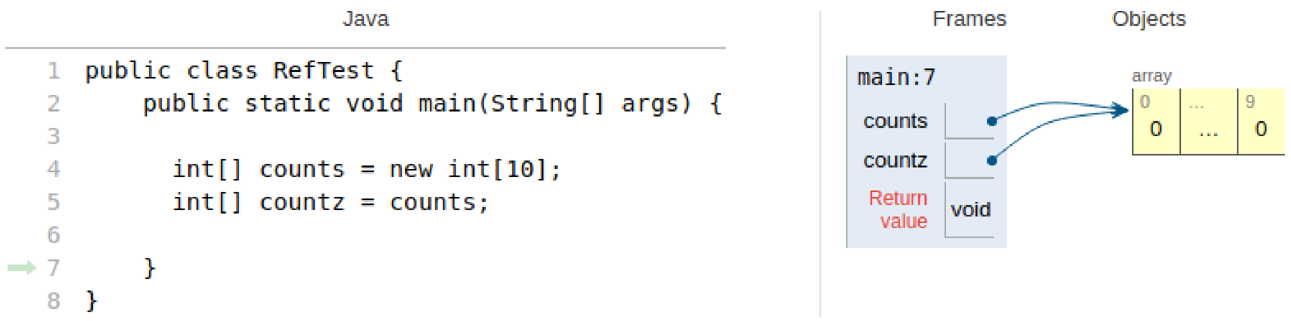
```
public class RefTest {
    public static void main(String[] args) {

      int[] counts = new int[10];

    }
}
```

Notice what happens when we assign one object variable to another object variable.

```
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
```

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**

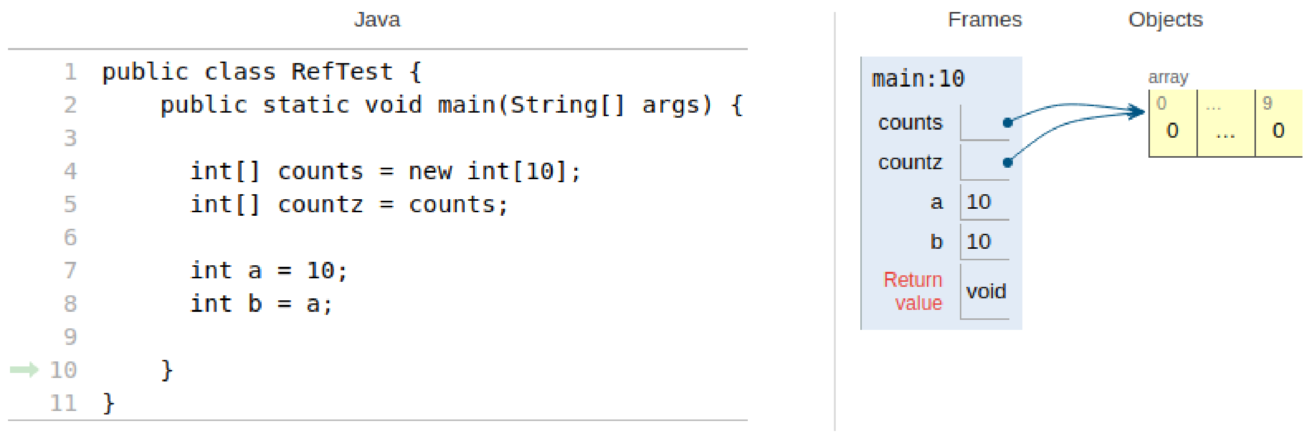
JavaTutor makes this clear . . . there is only a single object, but there are two references to this object.



```
public class RefTest {
    public static void main(String[] args) {

      int[] counts = new int[10];
      int[] countz = counts;

    }
}
```

One array, two references 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



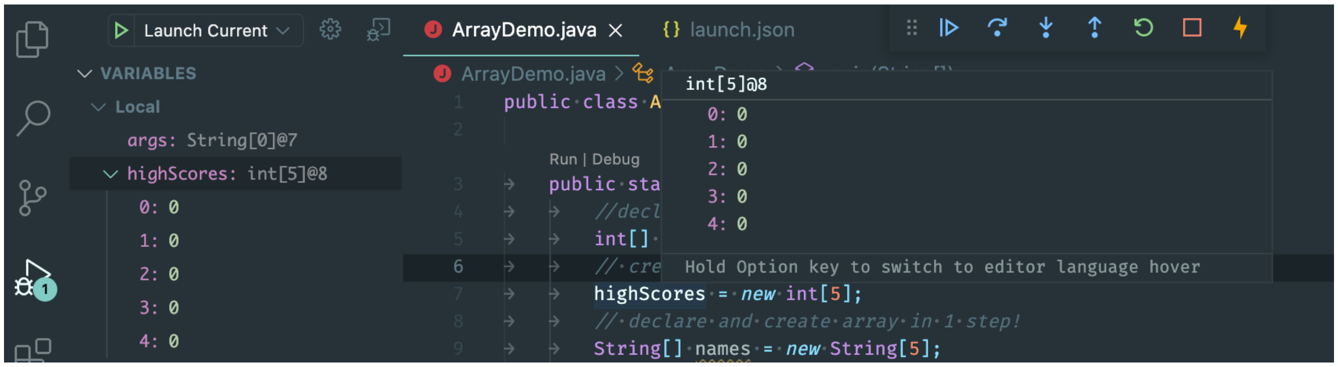
Java

```
 1  public class RefTest {
 2      public static void main(String[] args) {
 3
 4          int[] counts = new int[10];
 5          int[] countz = counts;
 6
 7          int a = 10;
 8          int b = a;
 9
10      }
11  }
```

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.**

Experiment with the VS Code debugger and see how it allows you to view the contents of arrays. There are two ways to see array contents in debug mode

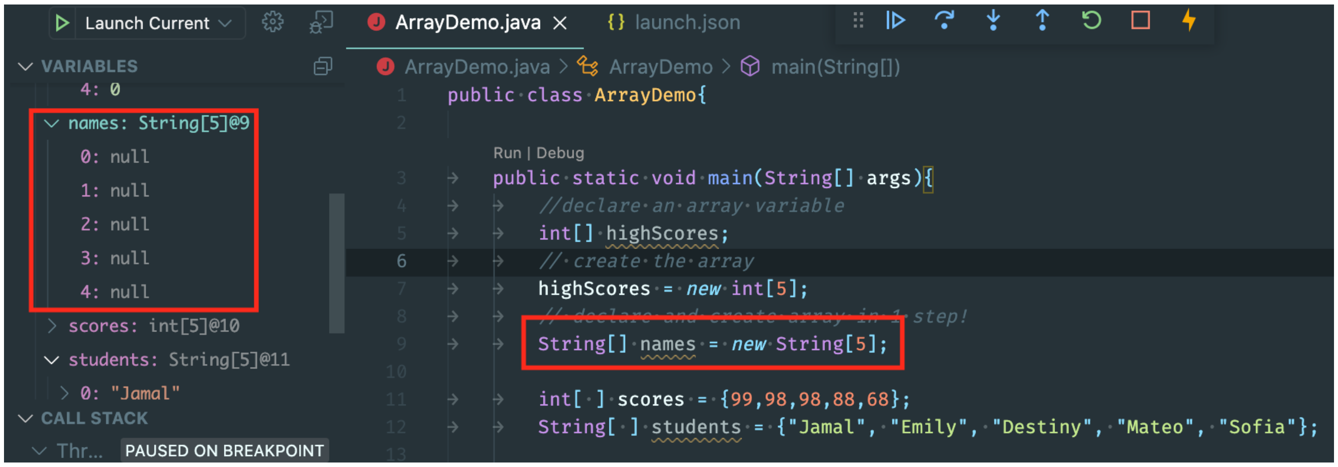
1. All local variables will be shown under **VARIABLES → Local**. Drill down into an array variable and you can see the indices and the data values.
2. In the ***editor pane*** you can hover over a variable and also see its value.



```
args: String[0]@7
highScores: int[5]@8
  0: 0
  1: 0
  2: 0
  3: 0
  4: 0
```

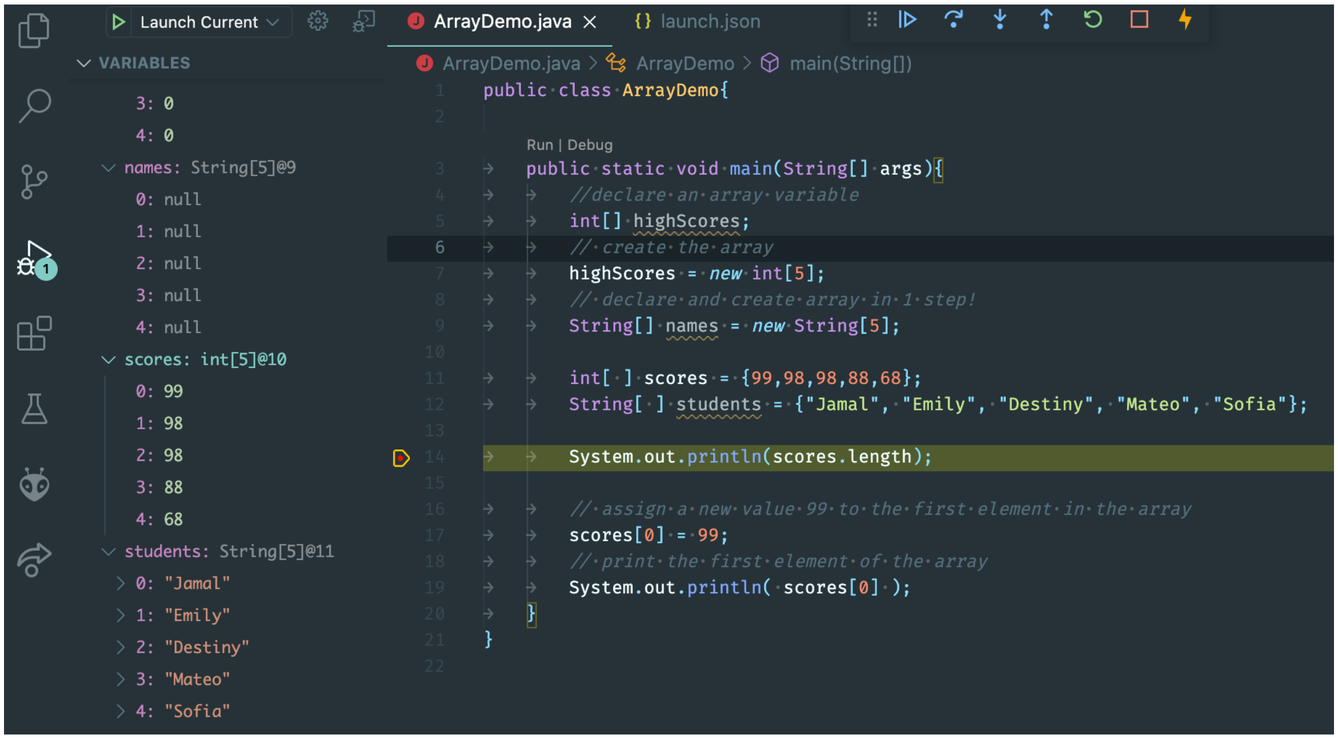
```
public class A
public sta
  //decl
  int[]
  //cre
  highScores = new int[5];
  //declare and create array in 1 step!
  String[] names = new String[5];
```

```
int[5]@8
  0: 0
  1: 0
  2: 0
  3: 0
  4: 0
Hold Option key to switch to editor language hover
```



```
    4: 0
names: String[5]@9
    0: null
    1: null
    2: null
    3: null
    4: null
scores: int[5]@10
students: String[5]@11
    0: "Jamal"
```

```
public class ArrayDemo{

    public static void main(String[] args){
        //declare an array variable
        int[] highScores;
        // create the array
        highScores = new int[5];
        // declare and create array in 1 step!
        String[] names = new String[5];

        int[ ] scores = {99,98,98,88,68};
        String[ ] students = {"Jamal", "Emily", "Destiny", "Mateo", "Sofia"};
```



```
    3: 0
    4: 0
  names: String[5]@9
    0: null
    1: null
    2: null
    3: null
    4: null
  scores: int[5]@10
    0: 99
    1: 98
    2: 98
    3: 88
    4: 68
  students: String[5]@11
  > 0: "Jamal"
  > 1: "Emily"
  > 2: "Destiny"
  > 3: "Mateo"
  > 4: "Sofia"
```

```java
public class ArrayDemo{

    Run | Debug
    public static void main(String[] args){
        //declare an array variable
        int[] highScores;
        //create the array
        highScores = new int[5];
        //declare and create array in 1 step!
        String[] names = new String[5];

        int[ ] scores = {99,98,98,88,68};
        String[ ] students = {"Jamal", "Emily", "Destiny", "Mateo", "Sofia"};

        System.out.println(scores.length);

        //assign a new value 99 to the first element in the array
        scores[0] = 99;
        //print the first element of the array
        System.out.println( scores[0] );
    }
}
```