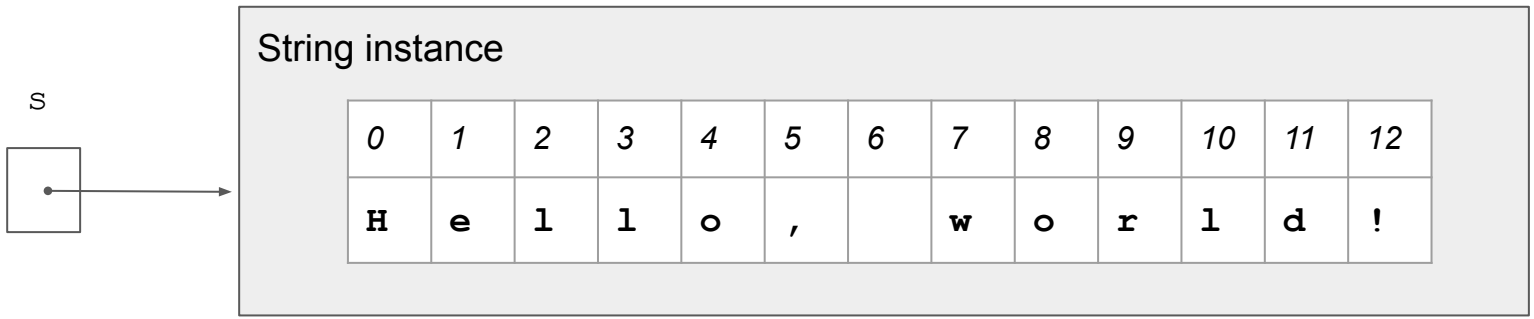
2.6: Strings

Playing around with words

Strings

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

```
String s = “Hello, world!”;
```



Strings are made up of chars

The `char` data type is a single 16-bit Unicode character.

It ranges from `'\u0000'` (or 0) to `'\uffff'` (65,535, or 2^{16}-1)

(`\u` is the backslash escape sequence for a Unicode code point in hexadecimal.)

Unicode covers all the characters for most writing systems of the world, modern and ancient (including proposals for Elvish and Klingon). 144,697 total characters today.

To the right is just the “Basic Latin” code points.


U+1F911


U+1F3B2


U+1F9E9


U+1F490

跑

U+8DD1

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
0010	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
0020		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0030	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0040	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0050	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0060	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0070	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
0080	XXX	XXX	BPH	NBH	IND	NEL	SSA	ESA	HTS	HTJ	VTS	PLD	PLU	RI	SS2	SS3
0090	DCS	PU1	PU2	STS	CCH	MW	SPA	EPA	SOS	XXX	SCI	CSI	ST	OSC	PM	APC
00A0	NB SP	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬		®	¯
00B0	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
00C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
00D0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
00E0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
00F0	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Creating Strings

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

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

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

```
String s = “Hello, world!”;
```

These do… *mostly* the same thing…

Main.java

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

Shell × Console ×

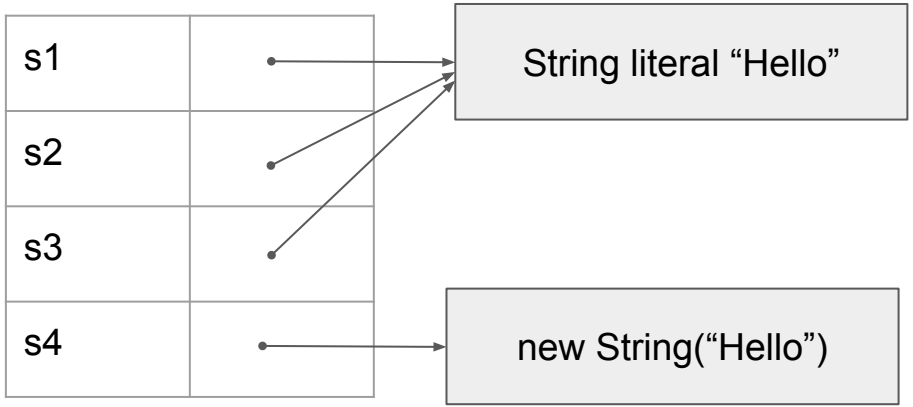
```
> sh -c javac -classpath .:target/dependency/* -d . $(find . -type f -name '*.java')
> java -classpath .:target/dependency/* Main
s1 == s2: true
s2 == s3: true
s1 == s4: false
s1.equals(s4): true
>
```

Main.java

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

Shell × Console ×

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



Java Language Standard 15.29:

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

String comparison in Java

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

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

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

String concatenation

Strings can be appended to each other to create a new string using the `+` or `+=` operator. This is also called **concatenation**.

In the expression `x + y`, `x` and `y` are the operands and `+` is the operator.

If x or y is a String, the other operand will be converted to String. That's why

```
System.out.println("Temp: " + 43 + " Frozen: " + false);
```

works. What does this print out?

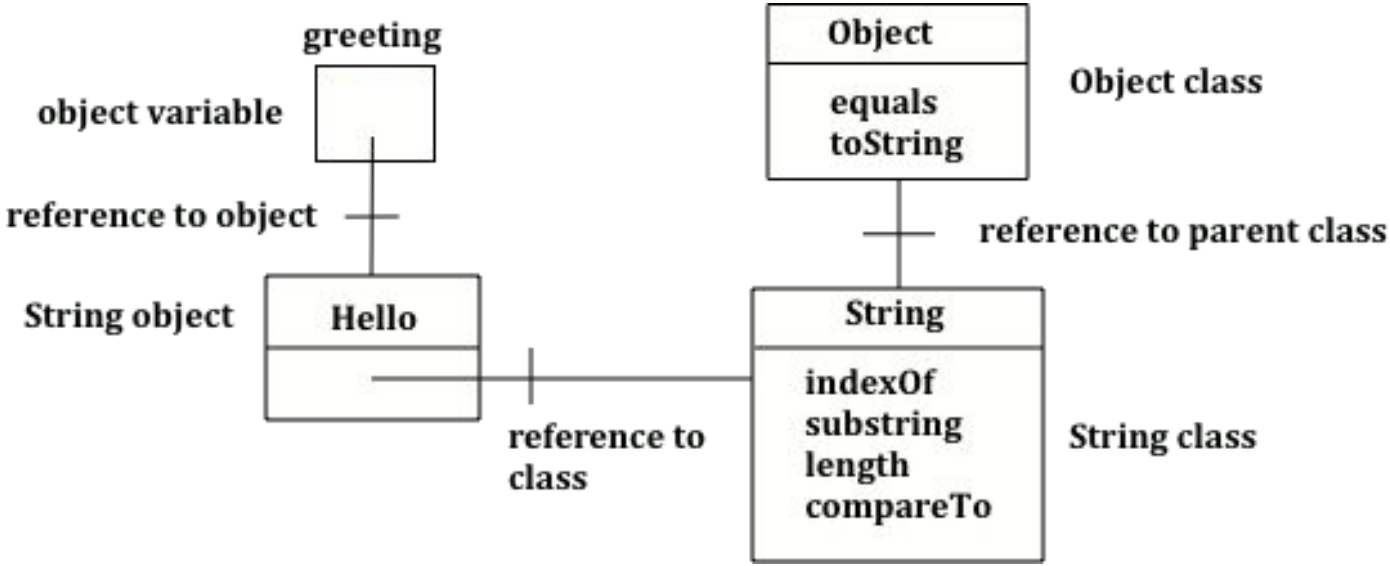
```
System.out.println("Age: " + 1 + 2);
```

Strings are Objects

Java classes have a parent class. The parent class of `java.lang.String` is `java.lang.Object`, which is the “ancestor” of all Java classes.

There is also a `Class` class! Every class has a `getClass` method which returns its `Class`.

Java programs can inspect their own classes and even change them in some ways. This is called ***reflection*** *or* ***metaprogramming****.*



2.7: String Methods

Playing around with words

String Methods

These are some of the most commonly used methods by programmers in industry.

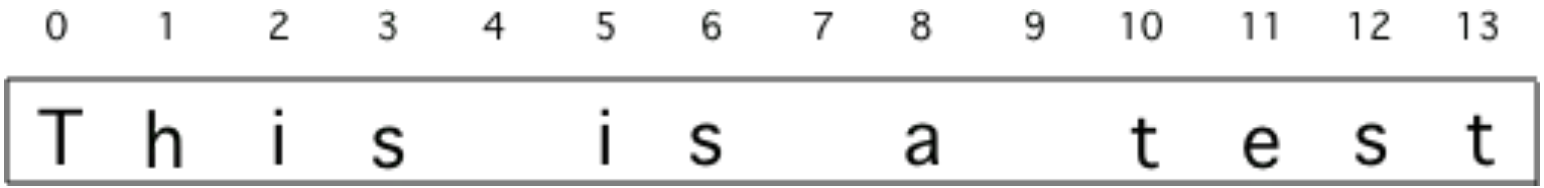
Definitions for these will be provided to you in the AP CS Exam Java Reference Sheet.

(Even professional Java developers still look these up from time to time!)

It's still important to understand these methods though, as you'll be using them a lot.

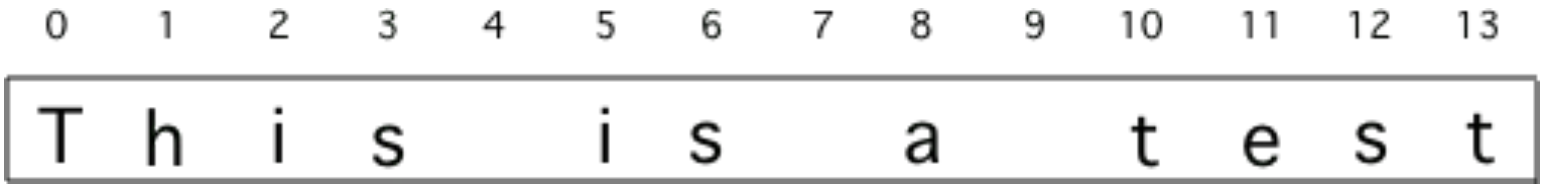
String Methods

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



String Methods

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



null

If you declare a String variable without initializing it, its value is `null`.

```
String s;
```
← Will contain `null`

Just like any other class, if you invoke a method of `null`, like `s.length()` here, Java will throw a `NullPointerException`.

`null` has its uses. It can represent the absence of a thing. For example, some people don't have a middle name:

```
class Person(String firstName, String middleName, String lastName) {...}
Person person = new Person("John", null, "Middlenameless");
```

Mutable vs Immutable

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

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

Why are strings immutable?

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

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

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

(but is stylistically better)

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

Practice!

Repl.it : stringsMethods1

Practice!

Repl.it : stringsMethods2