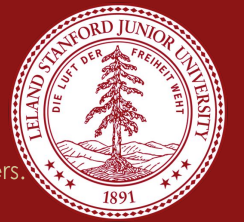
Characters and Strings

Lecture 13

CS106A, Summer 2019

Sarai Gould && Laura Cruz-Albrecht

With inspiration from slides created by Keith Schwarz, Mehran Sahami, Eric Roberts, Stuart Reges, Chris Piech, Brahm Capoor and others.



Announcements: Midterm

- **Midterm: Monday July 22nd, 7-9PM** in Bishop Auditorium
- Review session: Friday July 19th, 10:30AM in Gates B01

- If you have an academic or University conflict, email both instructors and fill out the following form **by tonight** 7/16
  - http://bit.ly/CS106AMidtermConflicts

Announcements: Midterm

- Download BlueBook software on website
- Make sure you have two-factor authentication on Duo Mobile with **passcodes** set up; will need for submission
  - Two-Factor Authentication: Duo Mobile
  - How to Use the Duo Mobile Passcode for Two-Step Authentication

- Check out Website for other relevant Midterm information

Plan for Today

- Review: Memory
- Characters
- Strings

Plan for Today

- Review: Memory
- Characters
- Strings

Review: Memory

Primitives

passed by *value*

stored on the ***stack***

Objects

passed by *reference*

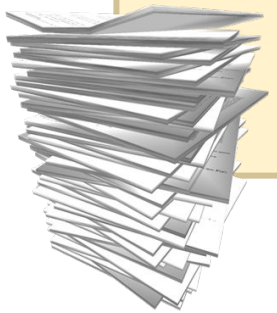
stored on the ***heap***,
referred to from the ***stack***

Review: Stack vs. Heap

The **Stack** is more temporary memory. Things on the Stack are added and removed as methods are called and returned.

The **Heap** is more permanent memory. Things on the Heap don't disappear as methods are called or returned.

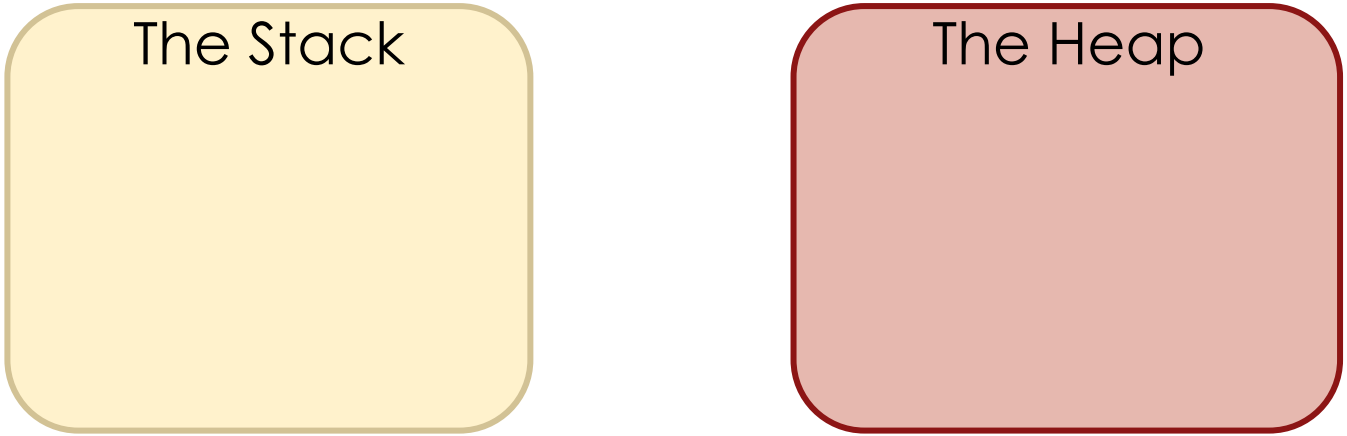
The Stack



The Heap

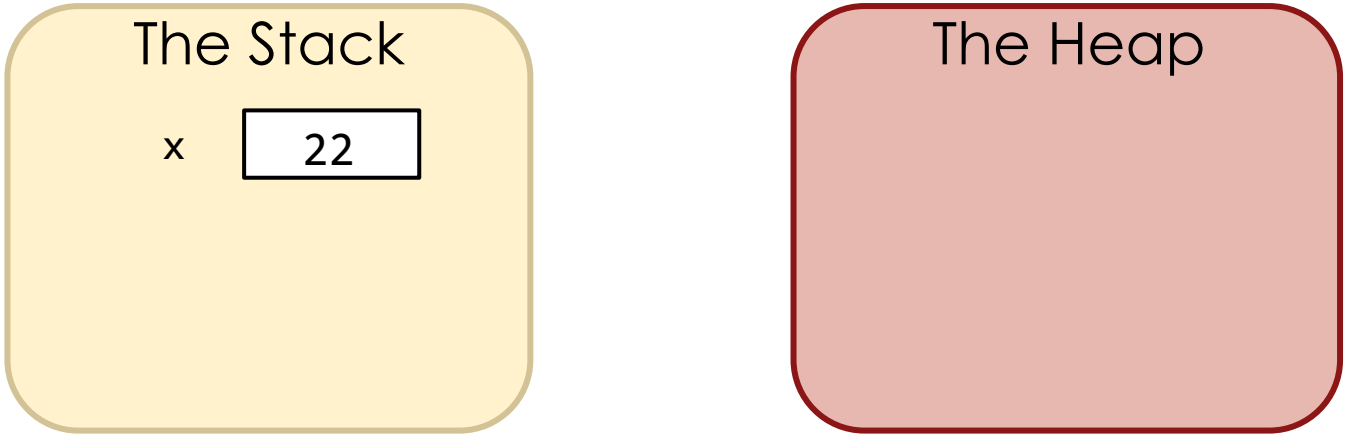
Review: Stack vs. Heap

```
int x = 22;
```



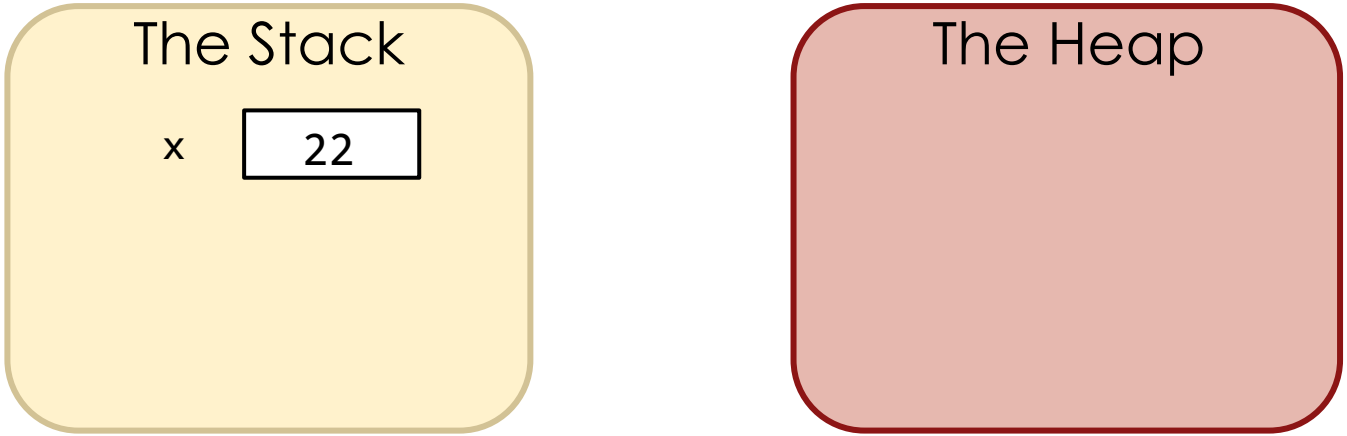
Review: Stack vs. Heap

```
int x = 22;
```



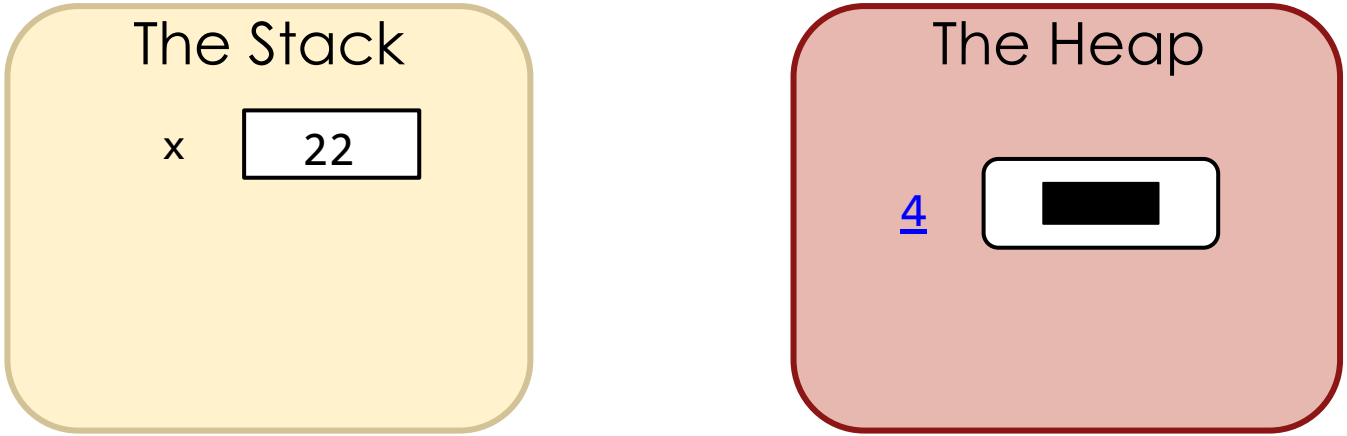
Review: Stack vs. Heap

```
int x = 22;
GRect rect1 = new GRect(30, 10);
```



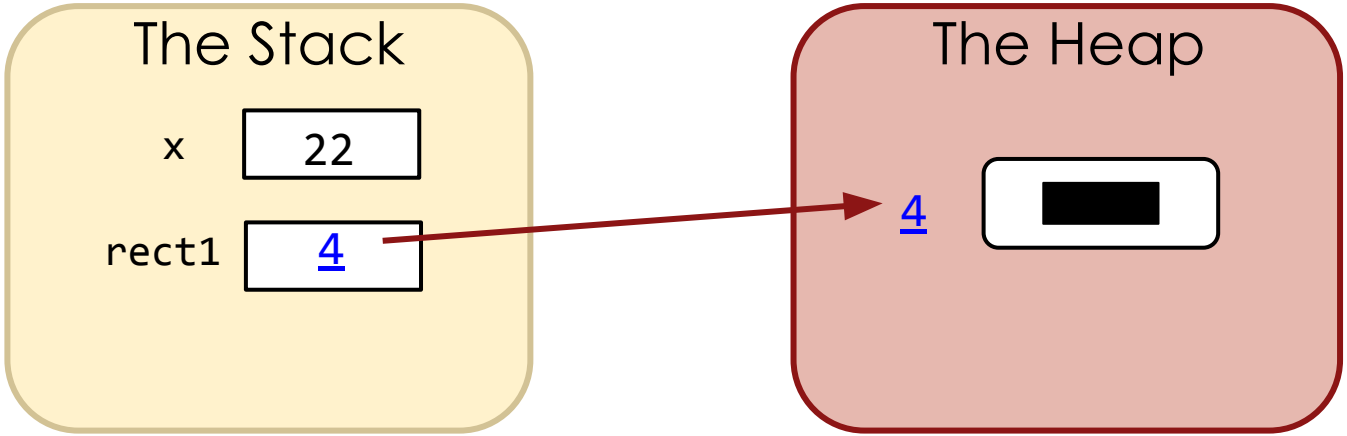
Review: Stack vs. Heap

```
int x = 22;
GRect rect1 = new GRect(30, 10);
```



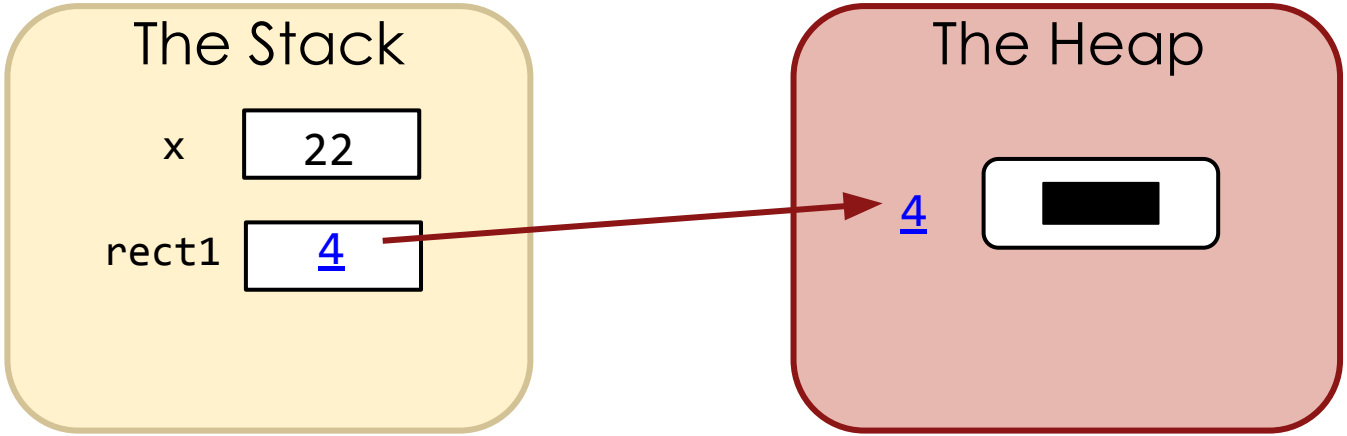
Review: Stack vs. Heap

```
int x = 22;
GRect rect1 = new GRect(30, 10);
```



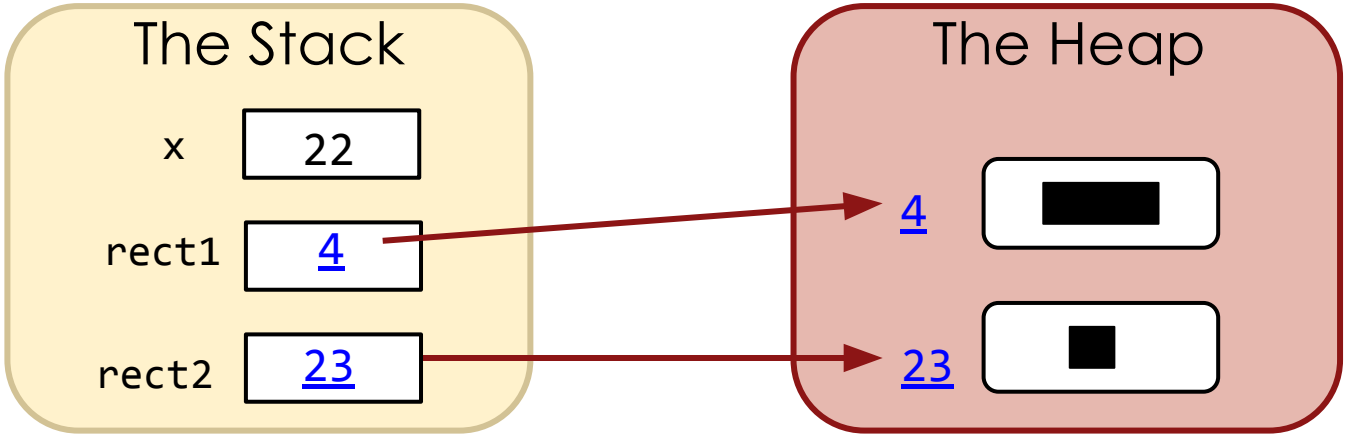
Review: Stack vs. Heap

```
int x = 22;
GRect rect1 = new GRect(30, 10);
GRect rect2 = new GRect(10, 10);
```



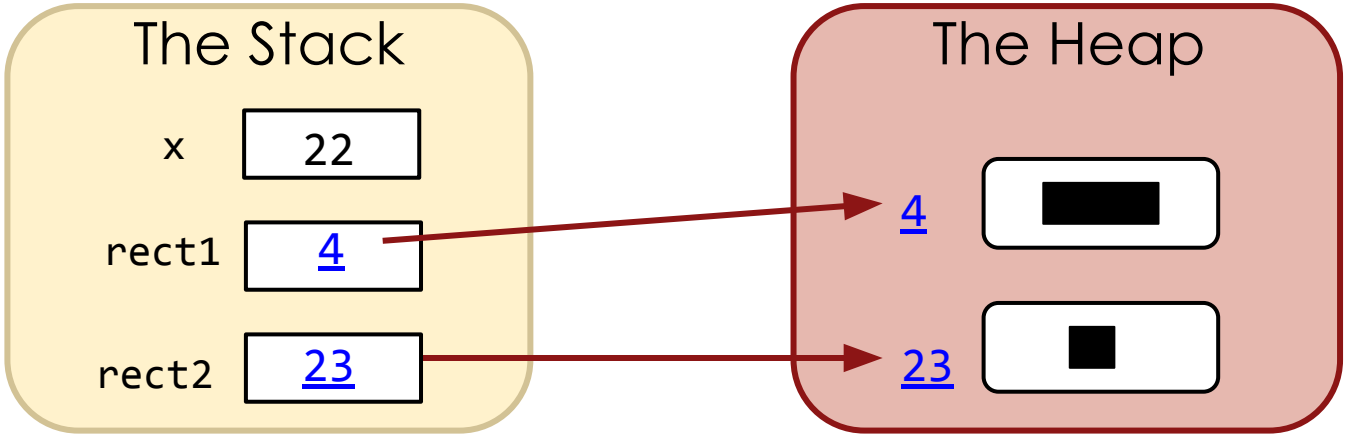
Review: Stack vs. Heap

```
int x = 22;
GRect rect1 = new GRect(30, 10);
GRect rect2 = new GRect(10, 10);
```



Review: Stack vs. Heap

```
int x = 22;
GRect rect1 = new GRect(30, 10);
GRect rect2 = new GRect(10, 10);
```



== compares values in the Stack

`.equals()` compares objects on the Heap

Review: Are They Equal? #1

```
public void run(){

     GRect rect1 = new GRect(100, 100); // rect1 = location 4
     GRect rect2 = new GRect(100, 100); // rect2 = location 5

     // is the memory address of rect1 equal to the memory address of rect2?
     if(rect1 == rect2){
          println("These rectangles are equal!");
     } else {
          // no! They're at different locations in memory!
          println("Actually, these rectangles are not equal.");
     }
}
```

We cannot create two different GRects in the same location in memory.

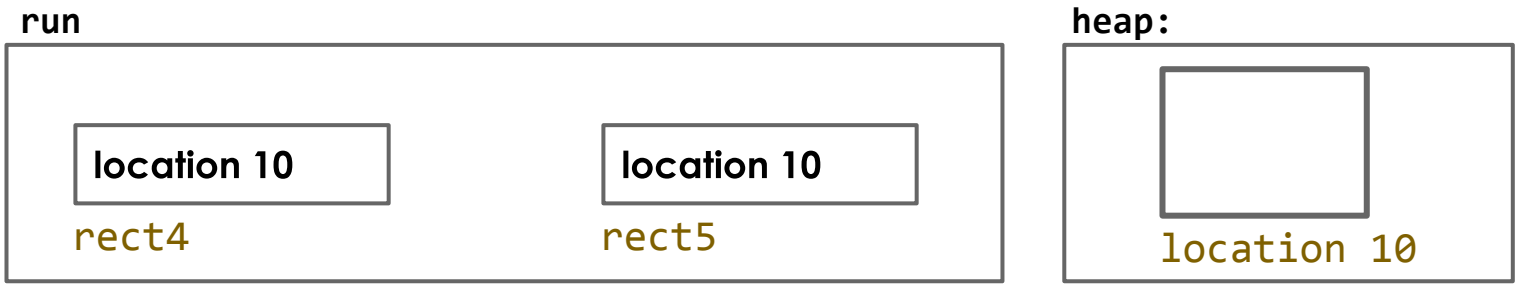
Review: Are They Equal? #2

```
public void run(){

    GRect rect3 = new GRect(100, 100);

    if(rect3 == rect3){
        // This is true!
        println("This rectangle is equal to itself!");
    } else {
        println("Actually, this rectangle is not equal to itself.");
    }
}
```

Remember: This will only evaluate to true if it's the exact same rectangle! **This has something to do with how objects are stored in memory.**

Review: Are They Equal? #3

```
public void run(){

      GRect rect4 = new GRect(100, 100); // rect4 = location 10
      GRect rect5 = rect4;               // rect5 also stores location 10

      // is the memory address of rect4 equal to the memory address of rect4?
      if(rect4 == rect5){
            // They are equal! They both point to the same place in memory!
            println("These rectangles are equal!");
      } else {
            println("Actually, these rectangles are not equal.");
      }
}
```



Pass by Reference vs Value

If something is passed by ***reference***, it ***can*** be altered simply by passing it into a method. This is because we are passing in a **reference to its location in memory**, not a copy of the object.

If something is passed by ***value***, it ***cannot*** be altered simply by passing it into a method. This is because we are a passing in a **copy of its value**.

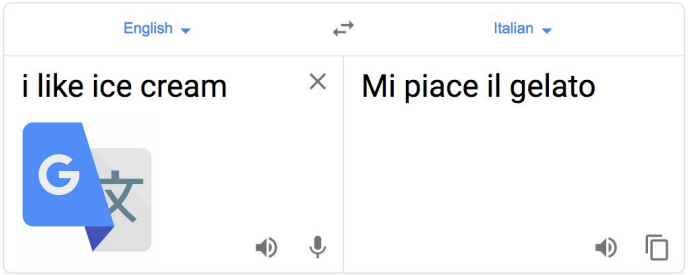
Plan for Today

- Review: Memory
- Characters
- Strings

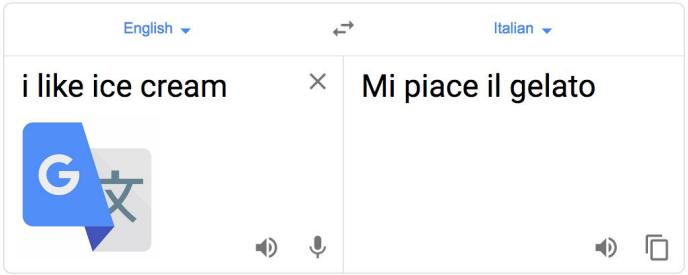
Text Processing

Text Processing

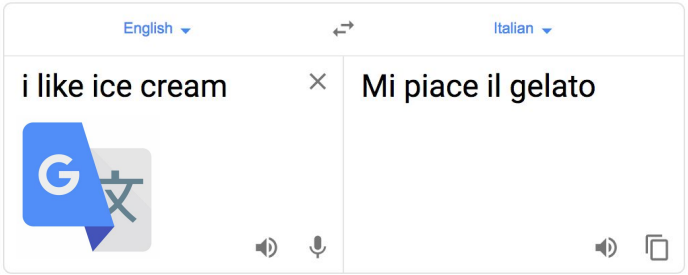
Text Processing



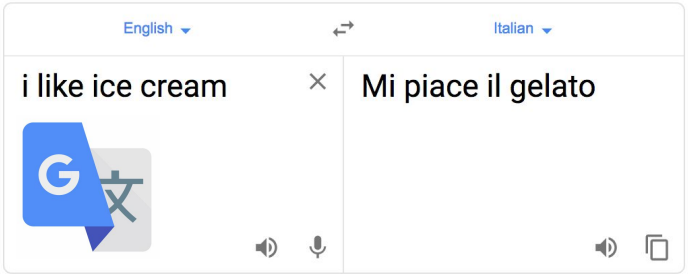
Text Processing



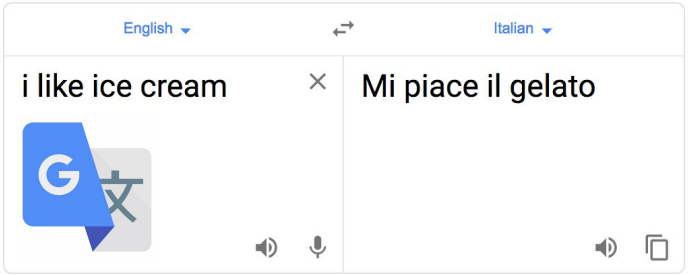
Text Processing

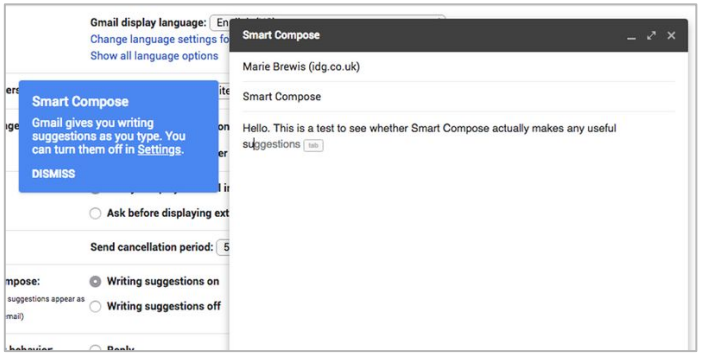


Text Processing



Text Processing





Plan for Today

- Review: Memory
- Characters
- Strings

Char

A **char** is a variable type that represents a single character or “glyph”.

```
char letterA = 'A';
```

Char

A **char** is a variable type that represents a single character or "glyph".

Char

A **char** is a variable type that represents a single character or “glyph”.

```
char letterA = 'A';
char plus = '+';
char zero = '0';
char space = ' ';
```

Char

A **char** is a variable type that represents a single character or “glyph”.

```
char letterA = 'A';
char plus = '+';
char zero = '0';
char space = ' ';

// special characters
char newLine = '\n';
char tab = '\t';
char singleQuote = '\'';
char backSlash = '\\';
```

Char

Under the hood, Java represents each **`char`** as an *integer*.

This integer is its "ASCII" value.

ASCII

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32	[space]	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	[backspace]

* This is only the first half of the table

** ASCII: American Standard Code for Information Interchange

ASCII

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32	[space]	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	[backspace]

* This is only the first half of the table

```
char uppercaseA = 'A';     // Actually 65
```

ASCII

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32	[space]	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	[backspace]

* This is only the first half of the table

```
char uppercaseA = 'A';     // Actually 65
char lowercaseA = 'a';     // Actually 97
```

ASCII

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32	[space]	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	[backspace]

* This is only the first half of the table

```
char uppercaseA = 'A';     // Actually 65
char lowercaseA = 'a';     // Actually 97
char zeroDigit = '0';      // Actually 48
```

Other Useful ASCII Properties

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32	[space]	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	[backspace]

- Uppercase letters are sequential (`'A' -> 'Z'`)

Other Useful ASCII Properties

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32	[space]	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	[backspace]

- Uppercase letters are sequential (`'A' -> 'Z'`)
- Lowercase letters are sequential (`'a' -> 'z'`)

Other Useful ASCII Properties

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32	[space]	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	[backspace]

- Uppercase letters are sequential (`'A' -> 'Z'`)
- Lowercase letters are sequential (`'a' -> 'z'`)
- Digits are sequential (`'0' -> '9'`)

Char Math

We can take advantage of Java representing each char as an integer!

Char Math

```
boolean areEqual = 'A' == 'A';      // true
```

Char Math

```
boolean areEqual = 'A' == 'A';      // true
boolean earlierLetter = 'f' < 'c'; // false
```

Char Math

```
boolean areEqual = 'A' == 'A';      // true
boolean earlierLetter = 'f' < 'c'; // false
char uppercaseB = 'A' + 1;          // 'B'
```

Char Math

```
boolean areEqual = 'A' == 'A';      // true
boolean earlierLetter = 'f' < 'c'; // false
char uppercaseB = 'A' + 1;          // 'B'
int diff = 'c' - 'a';               // 2
```

Char Math

```
boolean areEqual = 'A' == 'A';      // true
boolean earlierLetter = 'f' < 'c'; // false
char uppercaseB = 'A' + 1;          // 'B'
int diff = 'c' - 'a';               // 2

int alphabetSize = 'z' - 'a' + 1;
// or
int alphabetSize = 'Z' - 'A' + 1;
```

Revisiting our Friend, For Loops

```
// prints the numbers 1 to 20
for (int i = 1; i <= 20; i++) {
    println(i);
}
```

If chars are ints under the hood can we use chars for the loop variable?

Char Math

```
// prints the characters a to z
for (char ch = 'a'; ch <= 'z'; ch++) {
    println(ch);
}
```

Char Math

Not every integer maps to a character. So when you have an expression with `int`s and `char`s, Java picks `int` as the *most expressive type*.

Char Math

Not every integer maps to a character. So when you have an expression with **`int`**s and **`char`**s, Java picks **`int`** as the *most expressive type*.

Recall the expressive hierarchy:

`String` > `double` > **`int`** > **`char`** > `boolean`

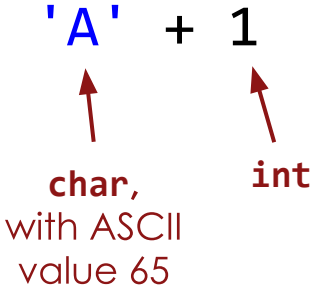
Char Math

Not every integer maps to a character. So when you have an expression with **`int`**s and **`char`**s, Java picks **`int`** as the *most expressive type*.

```
'A' + 1
```

Char Math

Not every integer maps to a character. So when you have an expression with **`int`**s and **`char`**s, Java picks **`int`** as the *most expressive type*.



Char Math

Not every integer maps to a character. So when you have an expression with **`int`**s and **`char`**s, Java picks **`int`** as the *most expressive type*.

```
'A' + 1
```

`char`, with ASCII value 65

`int`

```
// evaluates to 66 (int)
```

Answer will be an **`int`**

Char Math

Not every integer maps to a character. So when you have an expression with **`int`**s and **`char`**s, Java picks **`int`** as the *most expressive type*.

```
'A' + 1                 // evaluates to 66 (int)
'c' + (2 * 5) - 1       // evaluates to 108
```

Char Math

Not every integer maps to a character. So when you have an expression with **`int`**s and **`char`**s, Java picks **`int`** as the *most expressive type*.

```
'A' + 1                 // evaluates to 66 (int)
'c' + (2 * 5) - 1       // evaluates to 108
```

We can make it a **`char`** by putting it in a **`char`** variable.

Char Math

Not every integer maps to a character. So when you have an expression with `int`s and `char`s, Java picks `int` as the *most expressive type*.

```
'A' + 1                 // evaluates to 66 (int)
'c' + (2 * 5) - 1       // evaluates to 108
```

We can make it a `char` by putting it in a `char` variable.

```
char uppercaseB = 'A' + 1;
char alsoUppercaseB = 66;
```

Type-Casting

If we want to force Java to treat an expression as a particular type, we can also *cast it* to that type.

Type-Casting

If we want to force Java to treat an expression as a particular type, we can also *cast it* to that type.

```
'A' + 1                 // evaluates to 66 (int)
```

Type-Casting

If we want to force Java to treat an expression as a particular type, we can also *cast it* to that type.

```
'A' + 1              // evaluates to 66 (int)
(char)('A' + 1)      // evaluates to 'B' (char)
```

Type-Casting

If we want to force Java to treat an expression as a particular type, we can also *cast it* to that type.

```
'A' + 1              // evaluates to 66 (int)
(char)('A' + 1)      // evaluates to 'B' (char)
(char)'A' + 1        // evaluates to 66 (int)
```

Type-Casting

If we want to force Java to treat an expression as a particular type, we can also *cast it* to that type.

```
'A' + 1              // evaluates to 66 (int)
(char)('A' + 1)      // evaluates to 'B' (char)
(char)'A' + 1        // evaluates to 66 (int)

1 / 2                // evaluates to 0 (int)
```

Type-Casting

If we want to force Java to treat an expression as a particular type, we can also *cast it* to that type.

```
'A' + 1              // evaluates to 66 (int)
(char)('A' + 1)      // evaluates to 'B' (char)
(char)'A' + 1        // evaluates to 66 (int)

1 / 2                // evaluates to 0 (int)
(double)1 / 2        // evaluates to 0.5 (double)
```

Type-Casting

If we want to force Java to treat an expression as a particular type, we can also *cast it* to that type.

```
'A' + 1              // evaluates to 66 (int)
(char)('A' + 1)      // evaluates to 'B' (char)
(char)'A' + 1        // evaluates to 66 (int)

1 / 2                // evaluates to 0 (int)
(double)1 / 2        // evaluates to 0.5 (double)
1 / (double)2        // evaluates to 0.5 (double)
```

Type-Casting

If we want to force Java to treat an expression as a particular type, we can also *cast it* to that type.

```
'A' + 1              // evaluates to 66 (int)
(char)('A' + 1)      // evaluates to 'B' (char)
(char)'A' + 1        // evaluates to 66 (int)

1 / 2                // evaluates to 0 (int)
(double)1 / 2        // evaluates to 0.5 (double)
1 / (double)2        // evaluates to 0.5 (double)
(double)(1 / 2)      // eek! evaluates to 0.0 (double)
```

Character Methods

There are several helpful built-in Java methods to manipulate **chars**.

Character Methods

There are several helpful built-in Java methods to manipulate **chars**.

```
char lowercaseA = 'a';
char uppercaseA = Character.toUpperCase(lowercaseA);
```

Character Methods

There are several helpful built-in Java methods to manipulate **chars**.

```
char lowercaseA = 'a';
char uppercaseA = Character.toUpperCase(lowercaseA);

char plus = '+';
if (Character.isLetter(plus)) {
    ... // Does not execute: + is not a letter
}
```

Character Methods

`boolean Character.isDigit(char ch)` Determines if the specified character is a digit.
`boolean Character.isLetter(char ch)` Determines if the specified character is a letter.
`boolean Character.isLetterOrDigit(char ch)` Determines if the specified character is a letter or a digit.
`boolean Character.isLowerCase(char ch)` Determines if the specified character is a lowercase letter.
`boolean Character.isUpperCase(char ch)` Determines if the specified character is an uppercase letter.
`boolean Character.isWhitespace(char ch)` Determines if the specified character is **whitespace** (spaces and tabs).
`char Character.toLowerCase(char ch)` Converts **`ch`** to its lowercase equivalent, if any. If not, **`ch`** is returned unchanged.
`char Character.toUpperCase(char ch)` Converts **`ch`** to its uppercase equivalent, if any. If not, **`ch`** is returned unchanged.

Character Methods

`toLowerCase` and `toUpperCase` return the new char; they cannot modify an existing char!

Always save the return value of Character methods!

Character Methods

`toLowerCase` and `toUpperCase` return the new char; they cannot modify an existing char!

Always save the return value of Character methods!

```
char letter = 'a';
Character.toUpperCase(letter); // Does nothing!
println(letter);               // prints 'a'
```

Character Methods

`toLowerCase` and `toUpperCase` return the new char; they cannot modify an existing char!

Always save the return value of Character methods!

```
char letter = 'a';
Character.toUpperCase(letter); // Does nothing!
println(letter);               // prints 'a'

char letter = 'a';
char newLetter = Character.toUpperCase(letter);
println(newLetter);            // prints 'A' :)
```

Plan for Today

- Review: Memory
- Characters
- Strings

Strings

String

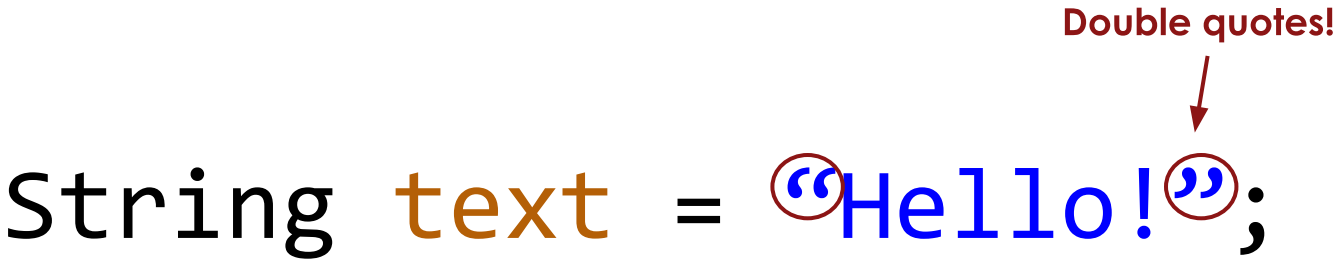
Text is stored using the variable type `String`.
A `String` is a sequence of characters!

```
String text = “Hello!”;
```

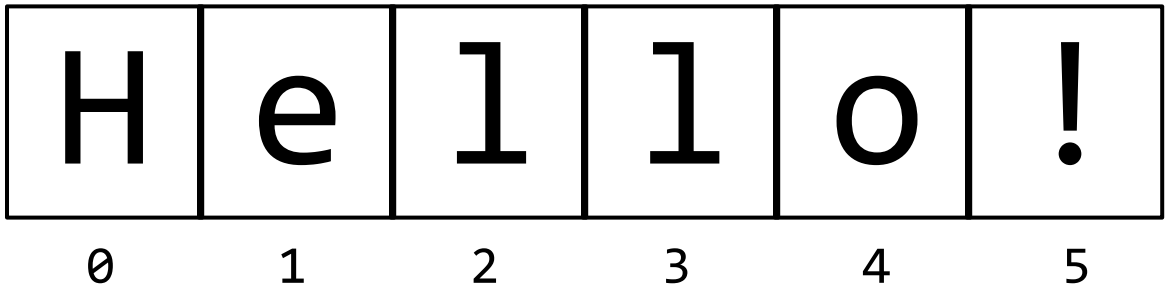
String

Text is stored using the variable type `String`.

A `String` is a sequence of characters!

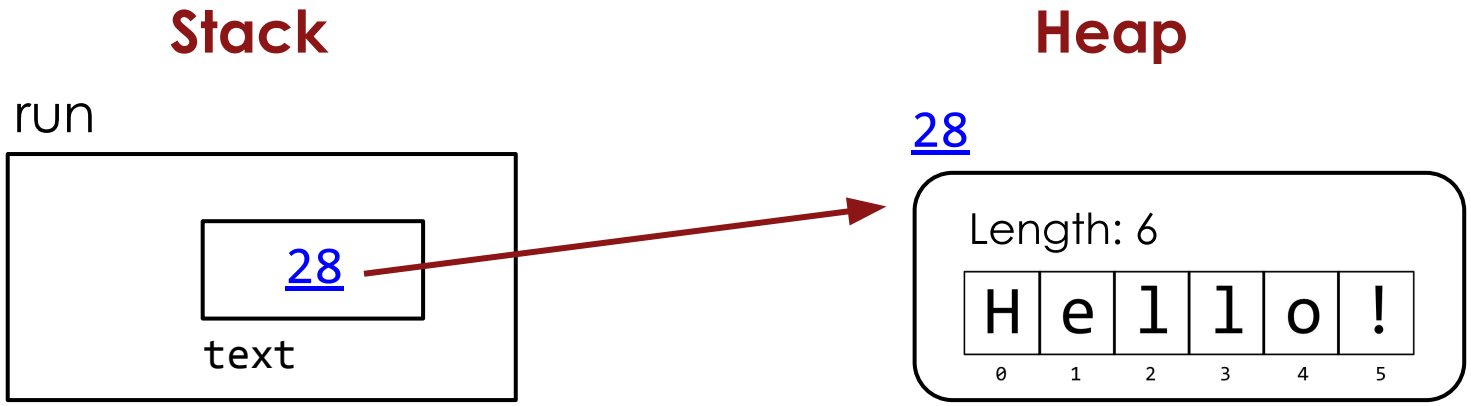


H e l l o !


H e l l o !
0 1 2 3 4 5

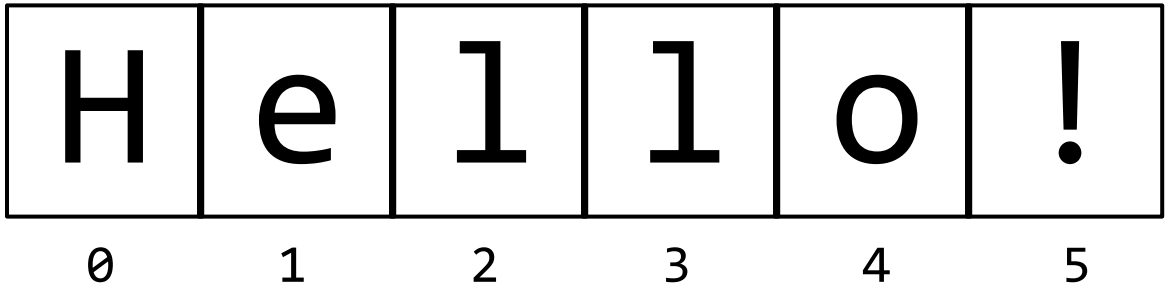
How it's actually stored

```
public void run() {
    String text = "hello!";
}
```



String Methods: length

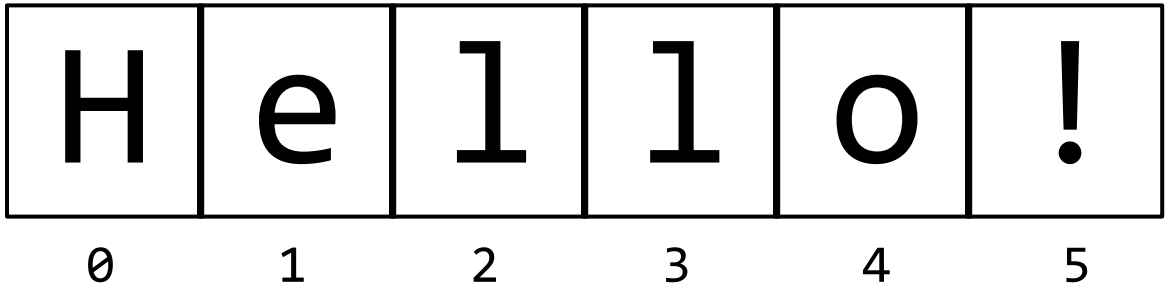
The *string*`.length()` method returns the number of characters in the string. This is one larger than the last valid index in the string.




```
int strLen = text.length();  // 6
```

String Methods: charAt

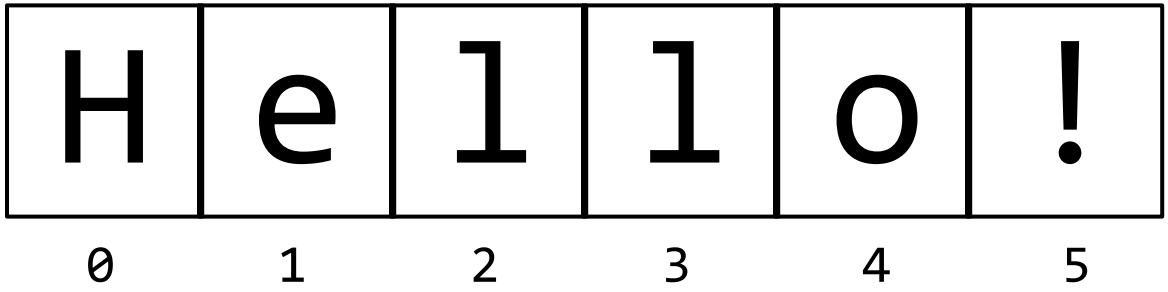
The *string*.charAt(*index*) method returns the character at a given index.




```
char first = text.charAt(0);                // 'H'
```

String Methods: charAt

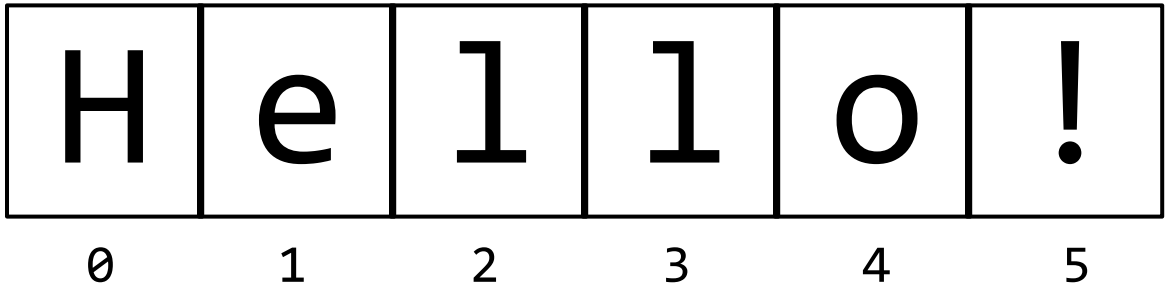
The *string*.charAt(*index*) method returns the character at a given index. The index must be between 0 and length - 1.




```
char first = text.charAt(0);           // 'H'
char last = text.charAt(strLen - 1);   // '!'
```

String Methods: charAt

The *string*.charAt(*index*) method returns the character at a given index.




```
char first = text.charAt(0);            // 'H'
char last = text.charAt(strLen - 1);    // '!'
char bad = text.charAt(strLen);         // error
```

Exercise

```
String q =
"Can you think of my favorite hobby?"
```

Exercise

```
String q =
"Can you think of my favorite hobby?"
  0   2   4   6    8   10  12  14  16  18   20  22  24   26  28  30   32  34
```

Exercise

```
String q =
"Can you think of my favorite hobby?"
  0   2   4   6    8   10  12  14  16  18   20  22  24   26  28  30   32  34

"" + q.charAt(12) + q.charAt(2) + q.charAt(10) + q.charAt(26)
```

Exercise

```
String q =
"Can you think of my favorite hobby?"
  0  2  4  6   8  10 12 14 16 18  20 22 24  26 28 30  32 34

"" + q.charAt(12) + q.charAt(2) + q.charAt(10) + q.charAt(26)
```

knit

Exercise

```
String q =
"Can you think of my favorite hobby?"
  0  2  4  6   8  10 12 14 16 18  20 22 24  26 28 30  32 34

"" + q.charAt(12) + q.charAt(2) + q.charAt(10) + q.charAt(26)
```

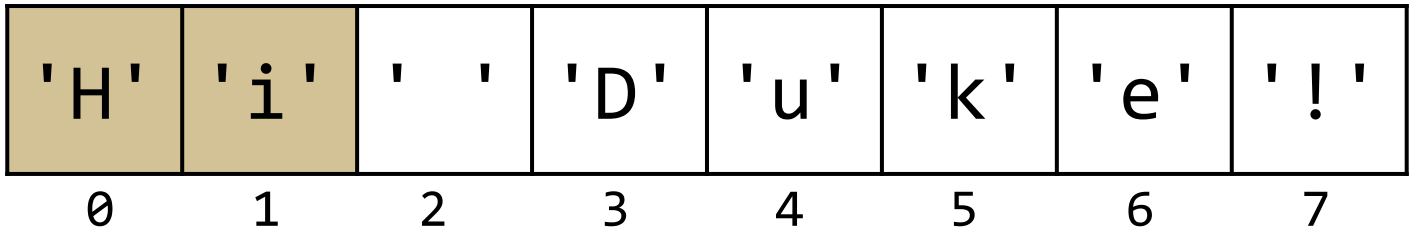
knit

* fun fact: also uses Strings

Substrings

A ***substring*** is a subset of a string.

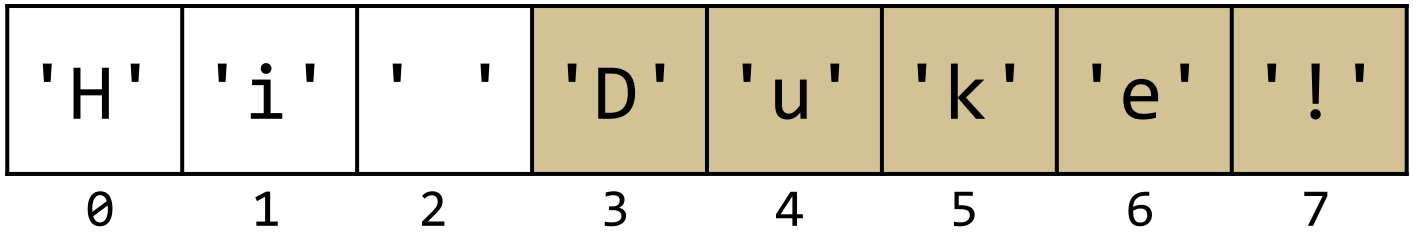
```
String str = “Hi Duke!”;
String hi = str.substring(0, 2);
```



Substrings

A ***substring*** is a subset of a string.

```
String str = “Hi Duke!”;
String dukeExclm = str.substring(3);  // to end
```



Useful String Methods

`int length()` Returns the length of the string
`char charAt(int index)` Returns the character at the specified index. Note: Strings indexed starting at 0.
`String substring(int p1, int p2)` Returns the substring beginning at **`p1`** and extending up to but not including **`p2`**
`String substring(int p1)` Returns substring beginning at **`p1`** and extending through end of string.
`boolean equals(String s2)` Returns true if string **`s2`** is equal to the receiver string. This is case sensitive.
`int compareTo(String s2)` Returns integer whose sign indicates how strings compare in lexicographic order
`int indexOf(char ch)` *or* **`int indexOf(String s)`** Returns index of first occurrence of the character or the string, or -1 if not found
`String toLowerCase()` *or* **`String toUpperCase()`** Returns a lowercase or uppercase version of the receiver string

* remember, called using **dot notation**: *`myString`*`.length()`

Creating Strings

```
String str = "Hello, world!";
```

Creating Strings

```
String str = "Hello, world!";
String empty = "";
```

Creating Strings

```
String str = "Hello, world!";
String empty = "";

// Read in text from the user
String name = readLine("What is your name? ");
```

Creating Strings

```
String str = "Hello, world!";
String empty = "";

// Read in text from the user
String name = readLine("What is your name? ");

// String concatenation (using “+”)
String message = 2 + " be or not " + 2 + " be";
```

Creating Strings

```
String str = "Hello, world!";
String empty = "";

// Read in text from the user
String name = readLine("What is your name? ");

// String concatenation (using “+”)
String message = 2 + " be or not " + 2 + " be";

int x = 2;
println("x has the value " + x);
```

Strings are Immutable

Java strings are **immutable**: once you create a String, its contents cannot be changed.

```
// Cannot change individual chars in the string
String typo = "Hello, warld!”;
typo.charAt(8) = ‘o’;   // Error! Will not run.
```

Strings are Immutable

Java strings are **immutable**: once you create a String, its contents cannot be changed.

```
// Cannot change individual chars in the string
String typo = "Hello, warld!”;
typo.charAt(8) = ‘o’;   // Error! Will not run.
```

To change a String, you must create a *new* String containing the value you want (e.g. using String methods).

```
String corrected = "Hello, world!”;
```

Strings are Immutable

Important consequence: if you pass a String into a method, that method cannot modify that string.

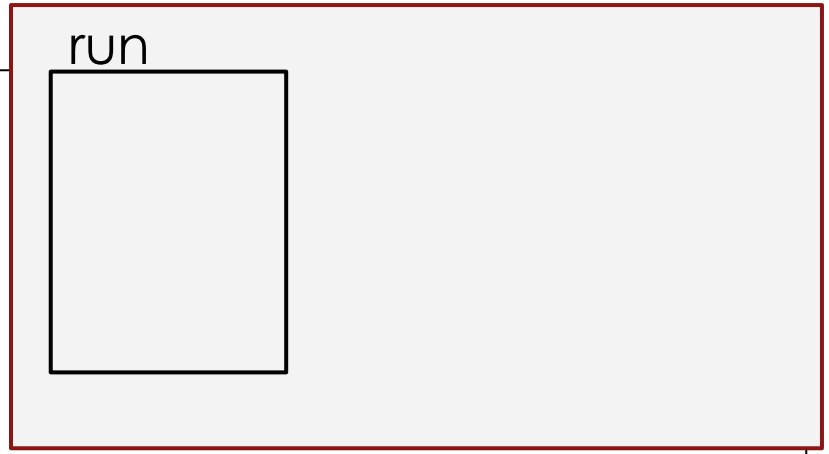
```
String className = "cs 106a";
className.toUpperCase(); // does nothing!

className = className.toUpperCase();
println(className);      // CS 106A
```

Are They Equal?

```
public void run(){

    String str1 = “Hello!”;
    String str2 = “Hello!”;

    if(str1 == str2){
        println(“These Strings are equal!”);
    } else {
        println(“Actually, these Strings are not equal.”);
    }
}
```

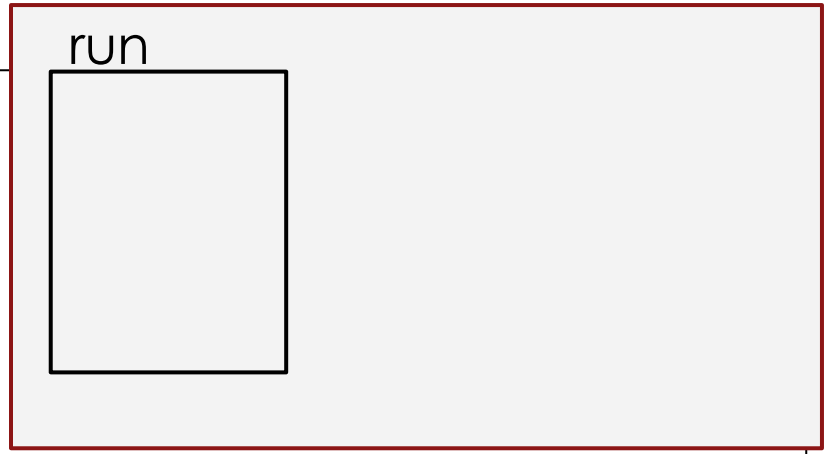
Comparing Strings

```
public void run(){

    String str1 = "Hello!";
    String str2 = "Hello!";

    if(str1 == str2){
        println("These Strings are equal!");
    } else {
        println("Actually, these Strings are not equal.");
    }
}
```



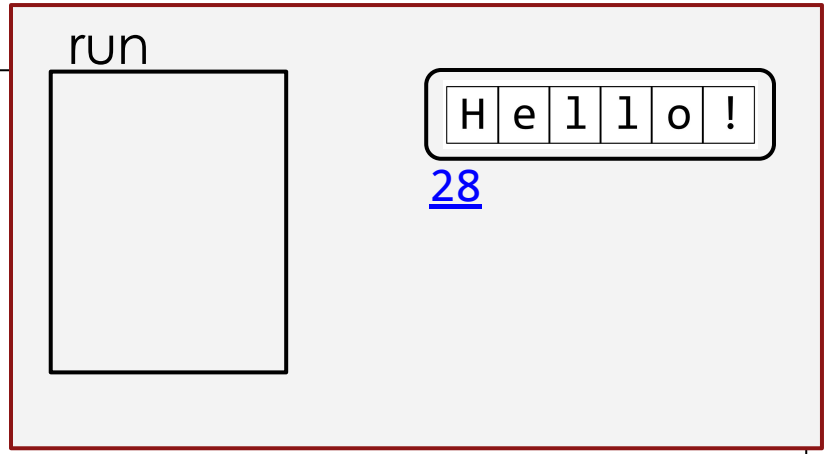
Comparing Strings

```
public void run(){

    String str1 = "Hello!";
    String str2 = "Hello!";

    if(str1 == str2){
        println("These Strings are equal!");
    } else {
        println("Actually, these Strings are not equal.");
    }
}
```



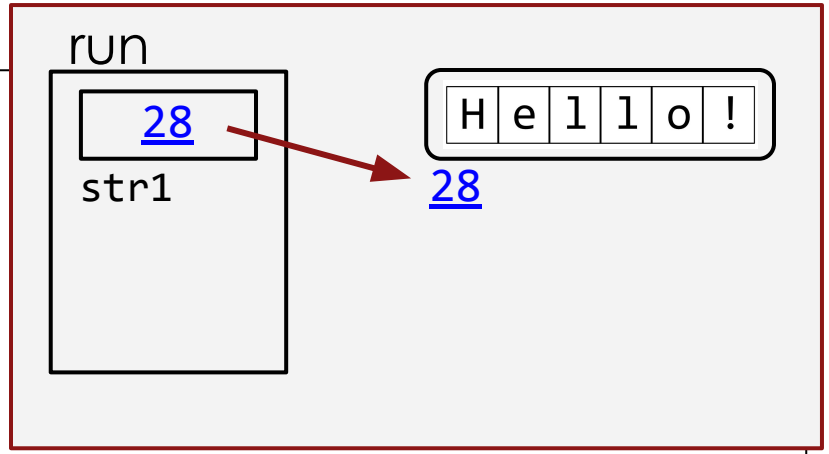
Comparing Strings

```
public void run(){

    String str1 = "Hello!";
    String str2 = "Hello!";

    if(str1 == str2){
        println("These Strings are equal!");
    } else {
        println("Actually, these Strings are not equal.");
    }
}
```



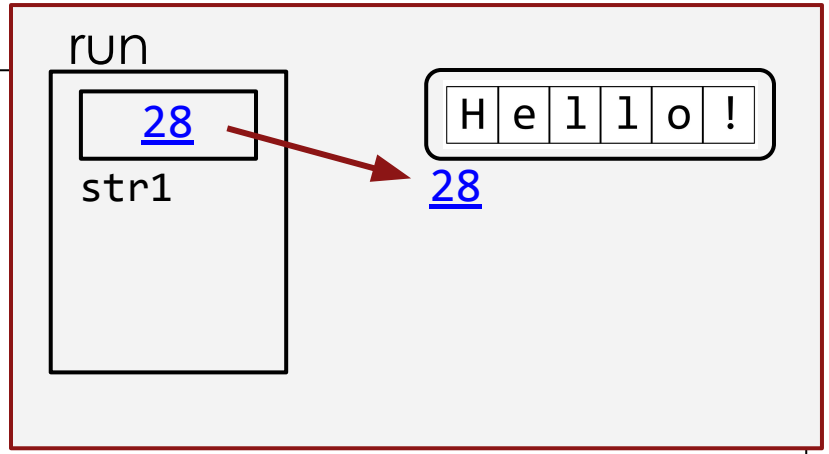
Comparing Strings

```
public void run(){

    String str1 = "Hello!";
    String str2 = "Hello!";

    if(str1 == str2){
        println("These Strings are equal!");
    } else {
        println("Actually, these Strings are not equal.");
    }
}
```



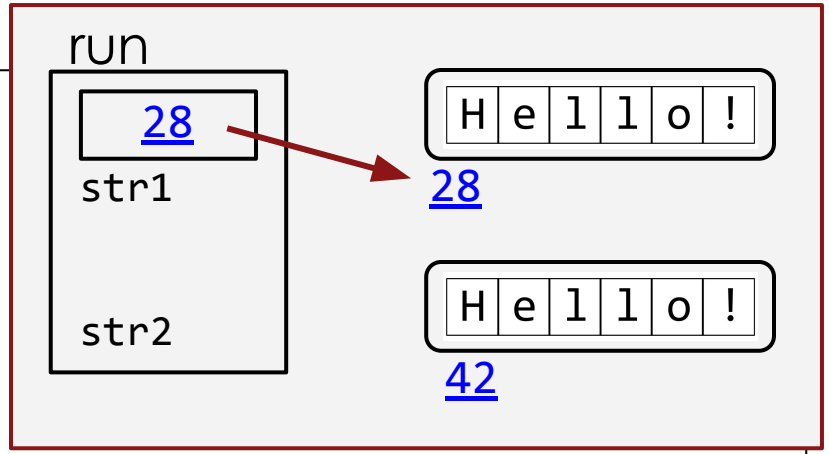
Comparing Strings

```
public void run(){

    String str1 = "Hello!";
    String str2 = "Hello!";

    if(str1 == str2){
        println("These Strings are equal!");
    } else {
        println("Actually, these Strings are not equal.");
    }
}
```



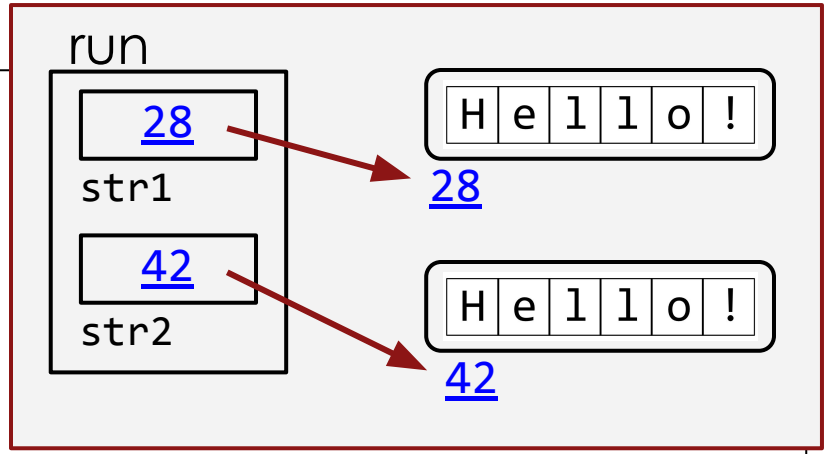
Comparing Strings

```
public void run(){

    String str1 = "Hello!";
    String str2 = "Hello!";

    if(str1 == str2){
        println("These Strings are equal!");
    } else {
        println("Actually, these Strings are not equal.");
    }
}
```



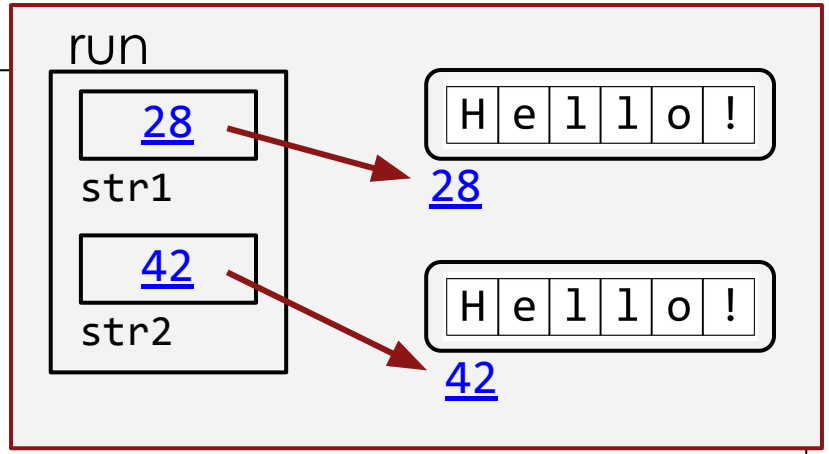
Comparing Strings

```
public void run(){

    String str1 = "Hello!";
    String str2 = "Hello!";

    if(str1 == str2){
        println("These Strings are equal!");
    } else {
        println("Actually, these Strings are not equal.");
    }
}
```



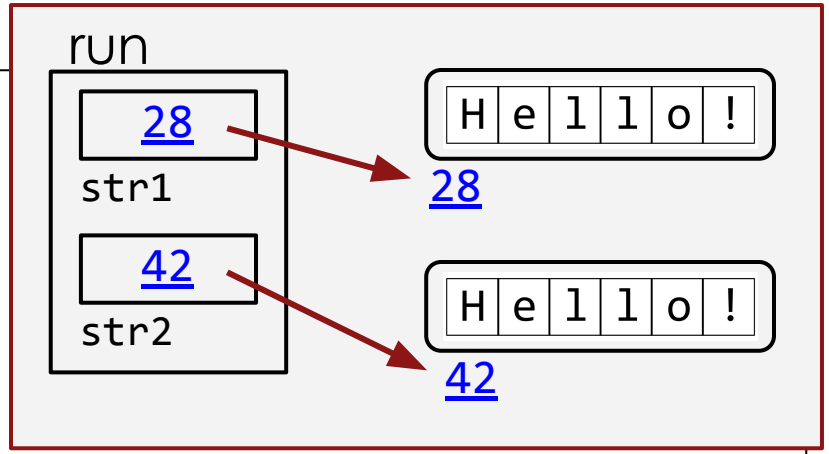
Comparing Strings

```
public void run(){

    String str1 = "Hello!";
    String str2 = "Hello!";

    if(str1 == str2){
        println("These Strings are equal!");
    } else {
        println("Actually, these Strings are not equal.");
    }
}
```



Comparing Strings

```
public void run(){

    String str1 = "Hello!";
    String str2 = "Hello!";

    if(str1 == str2){
        println("These Strings are equal!");
    } else {
        println("Actually, these Strings are not equal.");
    }
}
```



Instead Use...

```
str1.equals(str2)
```

Comparing Strings

```
public void run(){

    String str1 = “Hello!”;
    String str2 = “Hello!”;

    if(str1.equals(str2)){
        println(“These Strings have the same text!”);
    } else {
        println(“These Strings do NOT have the same text.”);
    }
}
```

Comparing Strings

Method	Description
`s1.equals(s2)`	whether two strings contain the same characters
`s1.equalsIgnoreCase(s2)`	whether two strings contain the same characters, ignoring upper vs. lower case
`s1.startsWith(s2)`	whether **s1** contains **s2**'s characters at start
`s1.endsWith(s2)`	whether **s1** contains **s2**'s characters at end
`s1.contains(s2)`	whether **s2** is found within **s1**

Looping over Strings

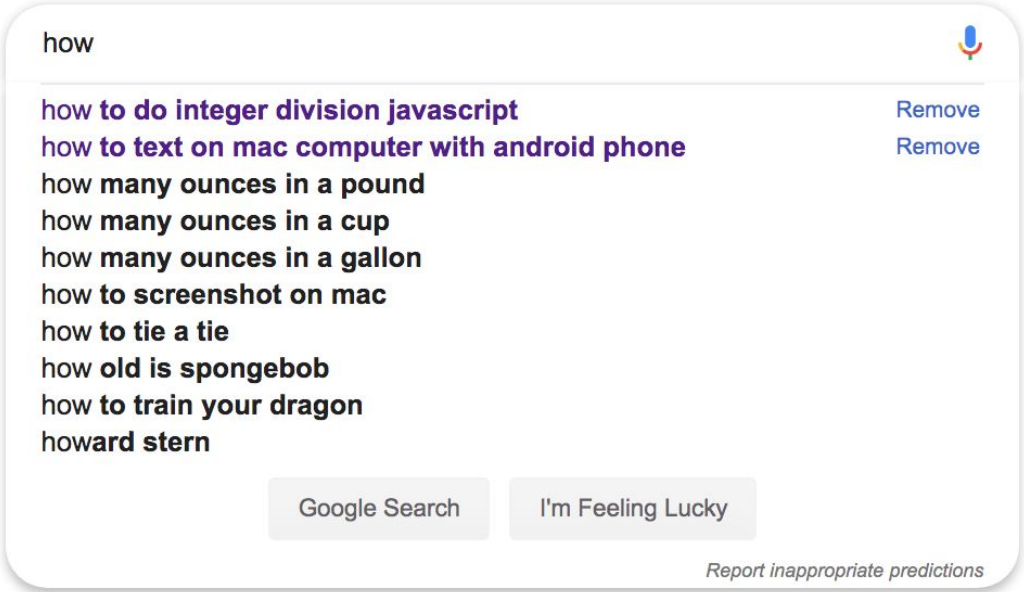
A common String programming pattern is looping over a String and operating on each character.

```
for (int i = 0; i < str.length(); i++) {
    char ch = str.charAt(i);
    // do something
}
```

Looping over Strings

```
// Creates a new String in all caps
String str = "Hello!";
String newStr = "";

for (int i = 0; i < str.length(); i++) {
    char ch = str.charAt(i);
    newStr += Character.toUpperCase(ch);
}

println(newStr);  // HELLO
```

Starts With



Given two strings, how can we check if one starts with the other?

Starts With

```
/*
 * startsWith
 * ----------
 * This method returns whether the String s1 starts with
 * the String s2. Can assume s1 is as long as or longer than s2.
 */
private boolean startsWith(String s1, String s2) {
    for (int i = 0; i < s2.length(); i++) {
        if (s1.charAt(i) != s2.charAt(i)) {
            return false;
        }
    }
    return true;
}
```

Plan for Today

- Review: Memory
- Characters
- Strings

Reminder: Check out Midterm Info page on website
→ Download Bluebook
→ Set up Two-Factor with Passcodes

Next time: Problem Solving with Strings!