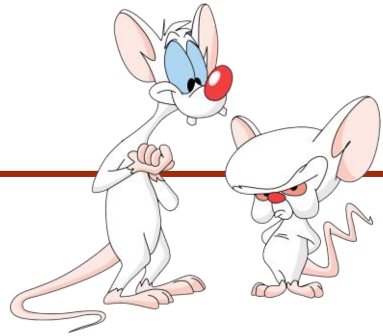
COMP 250

INTRODUCTION TO COMPUTER SCIENCE

Week 2-1: Primitive Data Types and Strings

Giulia Alberini, Fall 2020

WHAT ARE WE GOING TO DO IN THIS VIDEO?



- Primitive data types
- `char`
- `String`
- type conversion

PRIMITIVE DATA TYPES

PRIMITIVE TYPES

A **primitive** type is

- predefined by the language, and

- named by a reserved keyword

Java supports 8 primitive data types.

THE 8 TYPES SUPPORTED

byte	Integer values
short	
int	
long	
float	Real Numbers
double	
boolean	`true` or `false`
char	One character

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

HOW MANY VALUES?

How many values can you represent with:

- 1 bit?
- 2 bits?
- 3 bits?
- And what about n bits?

$$\boxed{2^n}$$

HOW MANY BITS?

And how many bits do you need to represent:

- 2 different values?
- 4 different values?
- 5 different values?
- And what about x different values?

$$\boxed{\lceil \log_2 x \rceil}$$

➢ So, how many bits do you need to store a boolean?

HOW MANY BITS N DO WE NEED TO REPRESENT A POSITIVE INTEGER m?

$$m = \sum_{i=0}^{N-1} b_i \, 2^i$$

What is the relationship between m and N ?

GEOMETRIC SERIES

Recall that,

$$\sum_{i=0}^{N-1} x^i = 1 + x + x^2 + x^3 + \cdots + x^{N-1} = \frac{x^N - 1}{x - 1}$$

That is, if $x = 2$,

$$\sum_{i=0}^{N-1} 2^i = 2^N - 1$$

HOW MANY BITS N DO WE NEED TO REPRESENT A POSITIVE INTEGER m?

$$
\begin{aligned}
m &= \sum_{i=0}^{N-1} b_i \cdot 2^i \\
&\le \sum_{i=0}^{N-1} 1 \cdot 2^i \\
&= 2^N - 1 \\
&< 2^N
\end{aligned}
$$

Thus,

$$m < 2^N$$

To solve for N, we take the log (base 2) of both sides and obtain the following equation:

$$\boxed{N > \log_2 m}$$

Lower bound

HOW MANY BITS *N* DO WE NEED TO REPRESENT A POSITIVE INTEGER *m*?

Now, let's assume that $N-1$ is the index i of the leftmost bit b_i such that $b_i = 1$.

e.g. We ignore leftmost 0's of the binary representation of m, $(\ldots 00000010011)_2$

Then,

$$m = \sum_{i=0}^{N-1} b_i 2^i = 1 \cdot 2^{N-1} + \sum_{i=0}^{N-2} b_i 2^i \geq 2^{N-1}$$

Taking the log (base 2) of both sides,

$$\log_2 m \geq N-1 \quad \Rightarrow \quad \boxed{N \leq (\log_2 m) + 1}$$

Upper Bound

HOW MANY BITS N DO WE NEED TO REPRESENT A POSITIVE INTEGER m?

We proved that,

$$\log_2 m < N \le (\log_2 m) + 1$$

Thus, N must be equal to the largest integer less than or equal to $(\log_2 m) + 1$.
We write,

$$N = floor\big((\log_2 m) + 1\big) = \lfloor(\log_2 m) + 1\rfloor$$

where $floor$ means "round down to the nearest integer".

WHY DIFFERENT TYPES?

It turns out that the difference between the types storing integer values and real numbers is the number of bits reserved for those values. For more info: COMP 273

Description	Keyword	Size	Values
Very Small Integer	`byte`	8-bits	$[-128, 127]$
Small Integer	`short`	16-bits	$[-2^{15}, 2^{15}-1]$
Integer	`int`	32-bits	$[-2^{31}, 2^{31}-1]$
Big Integer	`long`	64-bits	$[-2^{63}, 2^{63}-1]$
Low Precision Reals	`float`	32-bits	-
High Precision Reals	`double`	64-bits	-
True/False	`boolean`	1-bit	[true, false]
One character	`char`	16-bits	-

OVERFLOW AND UNDERFLOW

- Variables of type `int` store values between $2^{31}-1$ and -2^{31}.
  - $2^{31}-1 = 2147483647$ (`Integer.MAX_VALUE`)
  - $-2^{31} = -2147483648$ (`Integer.MIN_VALUE`)

- What happens if:

```
int x = 2147483647;
System.out.println(x+1);
```

```
int y = -2147483648;
System.out.println(y-1);
```

```
Output:-2147483648
```

```
Output 2147483647
```

STORING INTEGER AND OVERFLOW

Let's pretend that we only have 8 bits.

7 bits are used to store the number, and the left most bit for the sign.

0 means positive and 1 means negative.

0111 1111 = 127

What happens if we add 1?

STORING INTEGER AND OVERFLOW

Let's pretend that we only have 8 bits.

7 bits are used to store the number, and the left most bit for the sign.

0 means positive and 1 means negative.

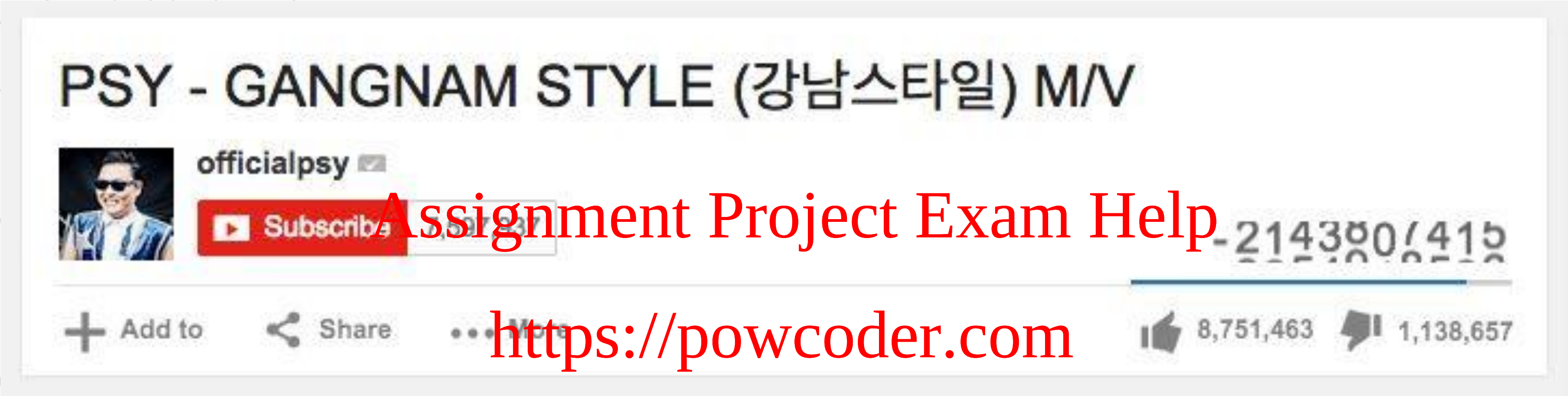
$$0111\ 1111 = 127$$

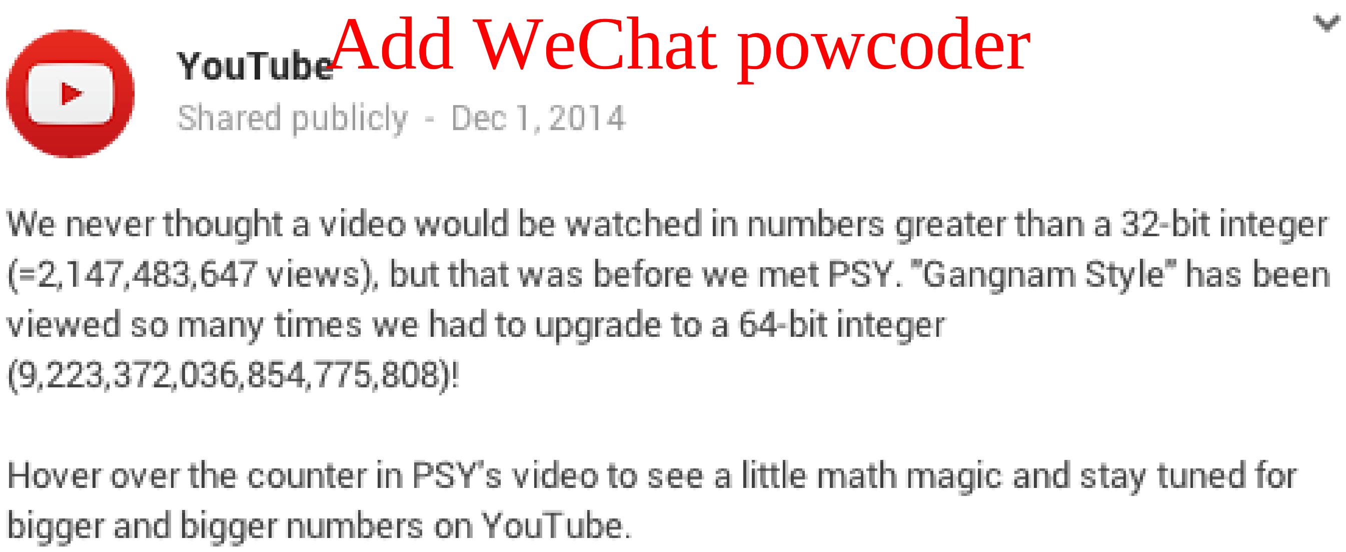
What happens if we add 1?

$$1000\ 0000 = -128$$

Note that negative numbers are stored a little bit differently. For more info see: https://en.wikipedia.org/wiki/Two's_complement

EXAMPLES





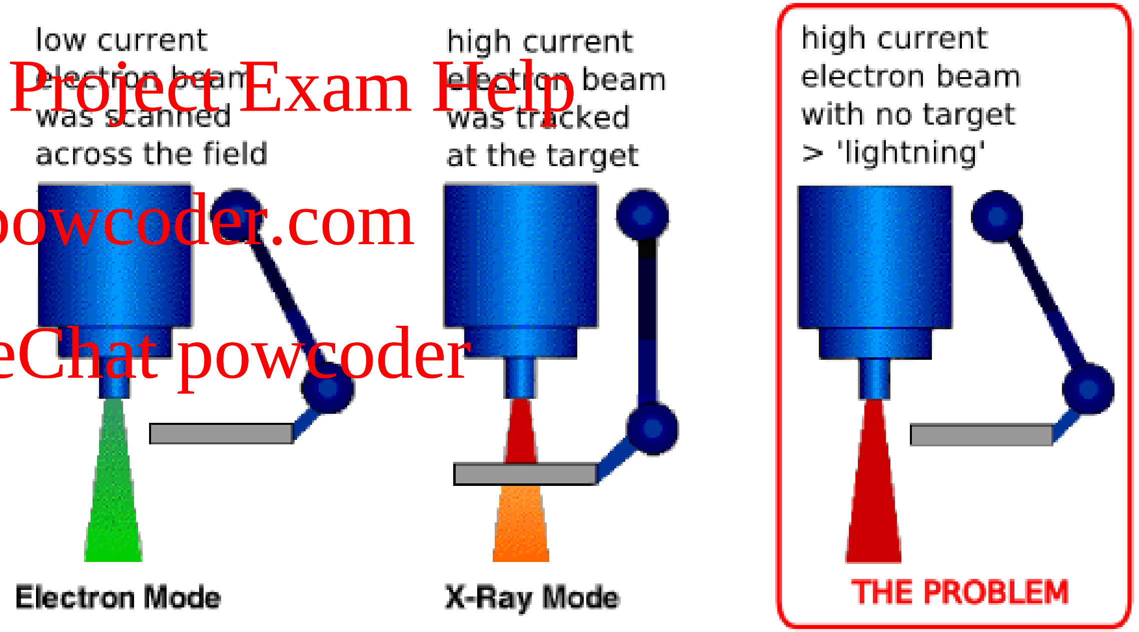
We never thought a video would be watched in numbers greater than a 32-bit integer (=2,147,483,647 views), but that was before we met PSY. "Gangnam Style" has been viewed so many times we had to upgrade to a 64-bit integer (9,223,372,036,854,775,808)!

Hover over the counter in PSY's video to see a little math magic and stay tuned for bigger and bigger numbers on YouTube.

EXAMPLES

Therac-25, radiation machine

- overflow during safety checks
- metal target would not be moved into place.
- result: beams 100 times higher than intended were fired into patients.
- 6 known cases causing the death of 4 patients.


low current electron beam was scanned across the field
high current electron beam was tracked at the target
high current electron beam with no target > 'lightning'
Electron Mode
X-Ray Mode
THE PROBLEM

tray including the target, a flattening filter, the collimator jaws and an ion chamber was moved OUT for "electron" mode, and IN for "photon" mode.

FLOATING POINT

- In java the default floating point type is `double`.
- All standard arithmetic operations can be done on floating point.
- NOTE: Java distinguishes between `1` and `1.0`.

  If you write .0 after an integer, it will be considered to be a `double`.

```
int x = 3.0;      ✗
int x = 3;        ✓
double x = 3.0;   ✓
```

BE CAREFUL!

- Java automatically converts one type to the other (e.g. int to double) if need be AND if no loss of information would occur.

```
double x = 1;  // legal, but bad style!
```

- If the mathematical operators are used with at least one operand of type double, then java will convert the other operands to double and it will output a values of type double. BUT, if all the operands are integers, the output of the operator will also be an integer!!

```
int x = 1.0/2;   // compiler error!
double y = 1/4; // no compiler error, but is it correct?
```

CHAR and UNICODE

CHAR DATA TYPE

We have seen `char` as one of the primitive data types that we have in Java.

- We can declare and initialize a variable of type `char` as follows:

```
char letter = 'a';
```

- Character literals appears in <u>single quotes</u>
- Character literals can only contain a single character

ESCAPE SEQUENCES

- Escape sequence: a sequence of characters that represents a special character.
- Examples:
  - `\n` represents the character newline
  - `\"` or `\ '` represent quotation marks
  - `\t` represents a tab.
- Escape sequences are legal characters because they represent a single character

```
char nl = '\n';
```

UNICODE

- A character set is an ordered list of character, where each character corresponds to a unique number.

- **Unicode** is an international character set. Java uses Unicode to represent characters.

- Variables of type char have 16 bits reserved in the memory to store a value.

- Each character is represented by an integer.
  Note: not every integer represent a character!

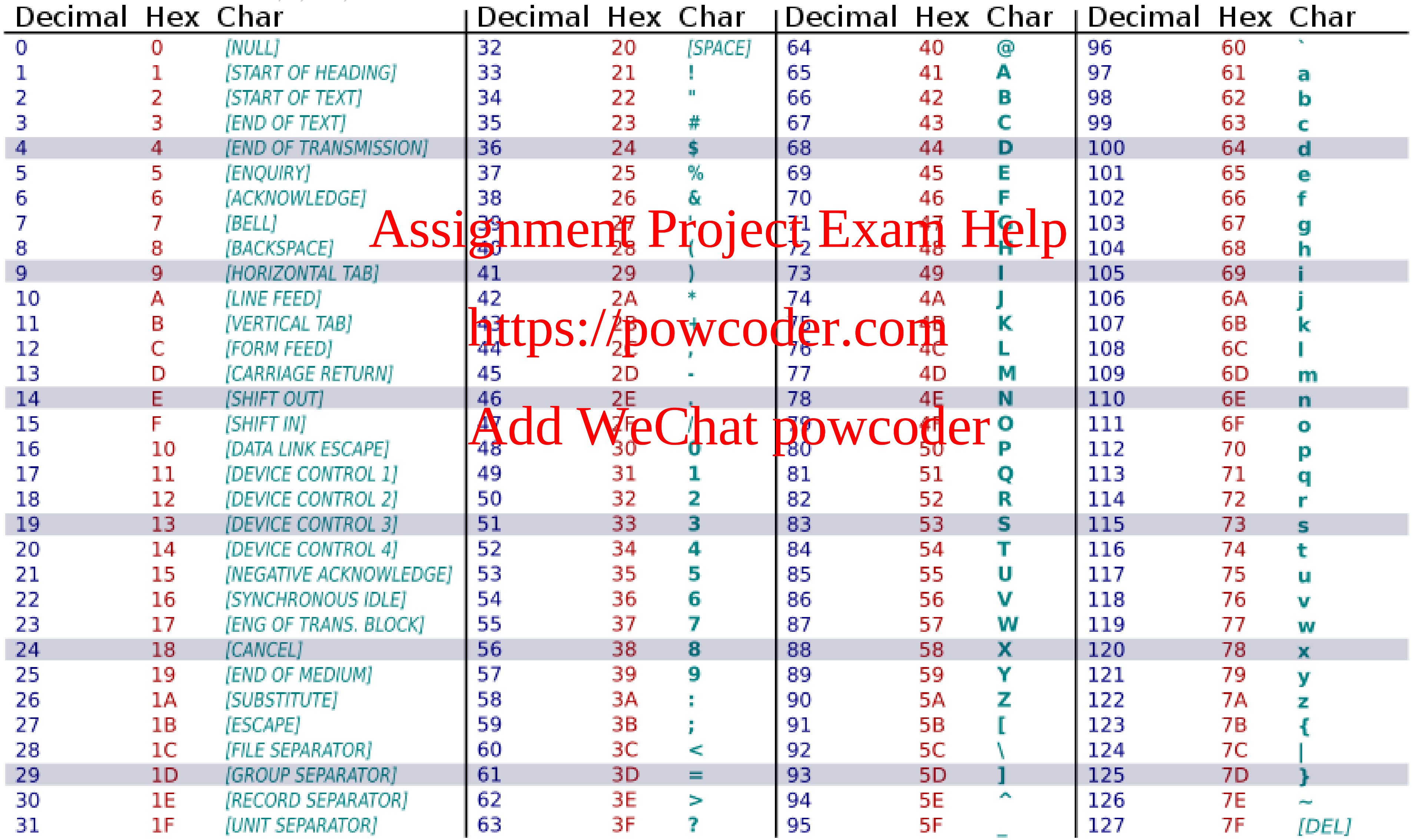
ASCII VS UNICODE

- ASCII: 7 bits. → It can represent 128 characters.

- UNICODE: 16 bits → 65536 characters.

  - It is a superset of ASCII: the numbers 0-127 map to the same characters both in ASCII and Unicode.

ASCII TABLE



Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

CHARACTER ARITHMETIC

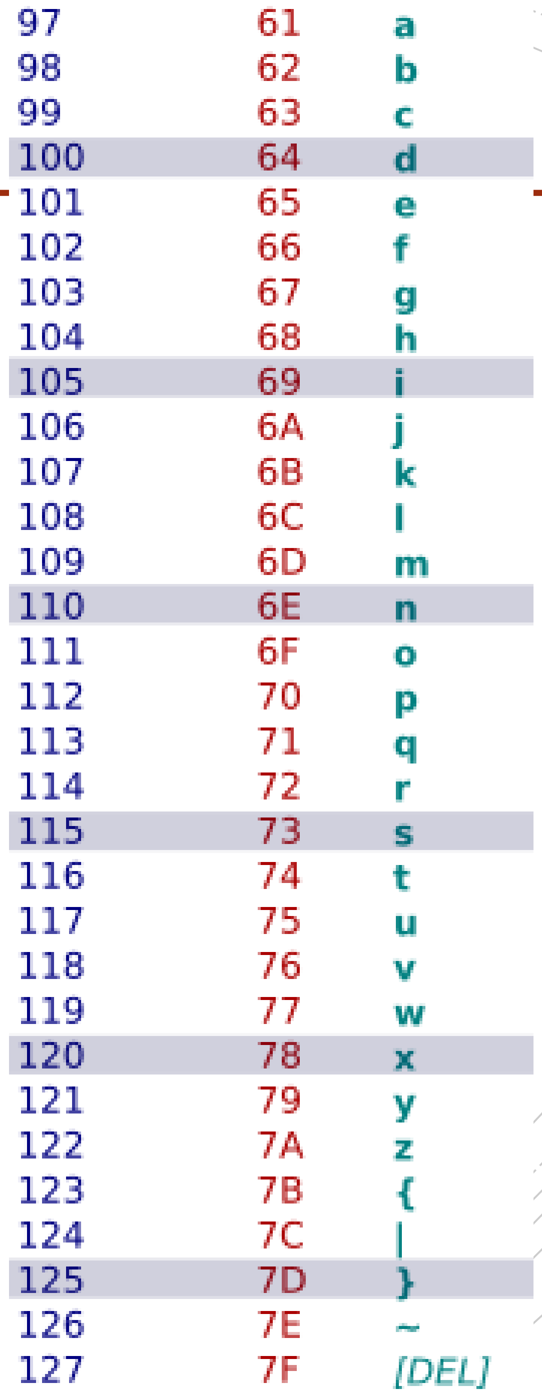
- Since every character is practically an integer, we can perform arithmetic operations on variables of type char.

```
char first = 'a';
char second = (char) (first + 1);
```

- What is the value of `second`?
  - `'b'`
- Note the typecasting!

  `first` is automatically converted into an integer, and `first + 1` evaluates to 98.

  Then the typecasting converts the `int` into a `char`, and stores `'b'` in `second`.



97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	[DEL]

COMPARING CHARS

```
char letter = 'g';
if(letter == 'a') {
   System.out.println("First letter of the alphabet");
} else if (letter == 'z') {
   System.out.println("Last letter of the alphabet");
} else if (letter > 'a' && letter < 'z') {
   System.out.println("Another letter of the alphabet");
} else {
   System.out.println("Not a lower case letter of the alphabet");
}
```

What prints?

- `Another letter of the alphabet`

TRY IT! - `charRightShift`

Write a method called `charRightShift` which takes a character and an integer `n` as inputs, and returns a character. If the character received as input is a lower case letter of the English alphabet, the method returns the letter of the alphabet which is `n` positions to the right on the alphabet. If the character received as input is not a lower case letter of the English alphabet, then the method returns the character itself with no modifications.

For example:

- `charRightShift('g', 2 )` returns `'i'`,
- `charRightShift('#', 2 )` returns `'#'`
- `charRightShift('z', 27 )` returns `'a'`

TYPE CASTING

TYPECASTING

- We can convert back and forth between variables of different types using **typecasting**. (or casting, for short)

```
int x = 3;
double y = 4.56;
int n = (int) y;
double m = (double) x;
```

- What are the values of `x, y, n`, and `m`?
  - `x = 3, y = 4.56, n = 4, m = 3.0`

PRIMITIVE TYPE CONVERSION – INT ↔ DOUBLE

- When going from `int` to `double`, an explicit cast is NOT necessary.

- When going from `double` to `int`, you will get a compile-time error if you don't have an explicit cast.

PRIMITIVE TYPE CONVERSION – IN GENERAL

type	*number of bits*
double	64
float	32
long	64
int	32
char	16
short	16
byte	8

wider

narrower

Here, wider usually (but not always) means more bytes.

NOTE: *char is "special"... see the following slides.*

EXAMPLES

```
int i = 3;
double d = 4.2;
d = i; // widening (implicit casting)
```

EXAMPLES

```
int i = 3;
double d = 4.2;
d = i; // widening (implicit casting)

d = 5.3 * i; //widening(by "promotion")
```

EXAMPLES

```
int i = 3;
double d = 4.2;
d = i; // widening (implicit casting)

d = 5.3 * i; // widening(by "promotion")

i = (int)d; // narrowing (by casting)
float f = (float) d; // narrowing   (by casting)
```

EXAMPLES

```
int i = 3;
double d = 4.2;
d = i;  // widening (implicit casting)

d = 5.3 * i;  // widening(by "promotion")

i = (int)d;  // narrowing (by casting)
float f = (float) d;  // narrowing   (by casting)
```

- For primitive types, both widening and narrowing change the bit representation. (See COMP 273.)

- For narrowing conversions, you get a compiler error if you don't cast.

EXAMPLES WITH CHAR

```
char c = 'q';
int x = c // widening
```

EXAMPLES WITH CHAR

```
char c = 'q';
int x = c // widening

c = (char) x; // narrowing
```

EXAMPLES WITH CHAR

```
char c = 'q';
int x = c // widening

c = (char) x; // narrowing

short y = 12;
c = y;  // compile time error!! (need explicit casting)
y = c;  // compile time error!! Narrowing → need explicit casting
```

STRINGS

STRING

- Recall that a `String` is sequence of characters.

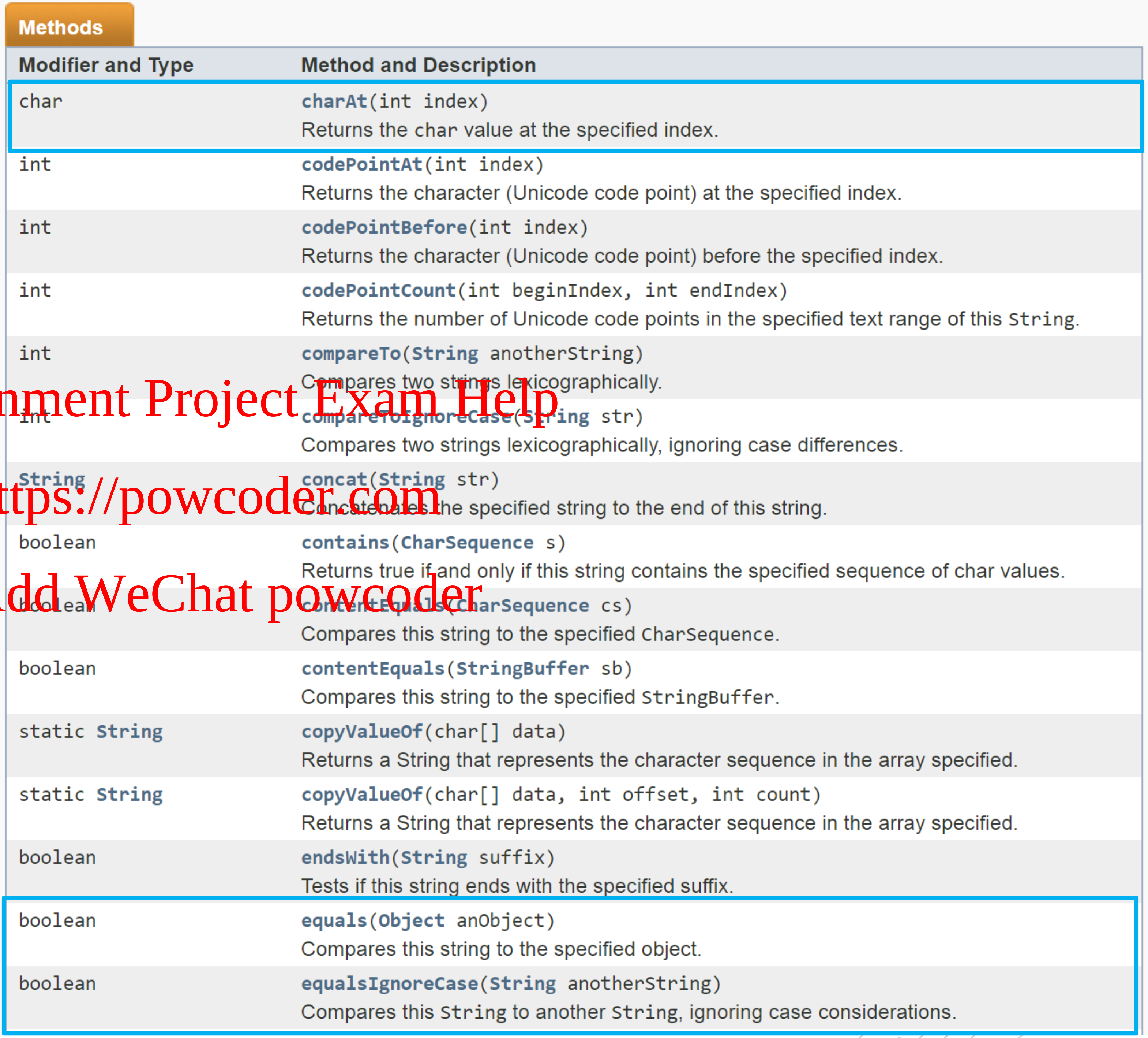
- `String` is a **Class** and a string literal is an **Object**.
  (more on classes and objects in the following weeks)

- We cannot use on Strings the same operators we use on primitive data types.

- There's a **set of methods** provided to manipulate characters and they can be called **on** values of type `String`.

DOCUMENTATION

You can find it here:
https://docs.oracle.com/javase/7/docs/api/java/lang/String.html

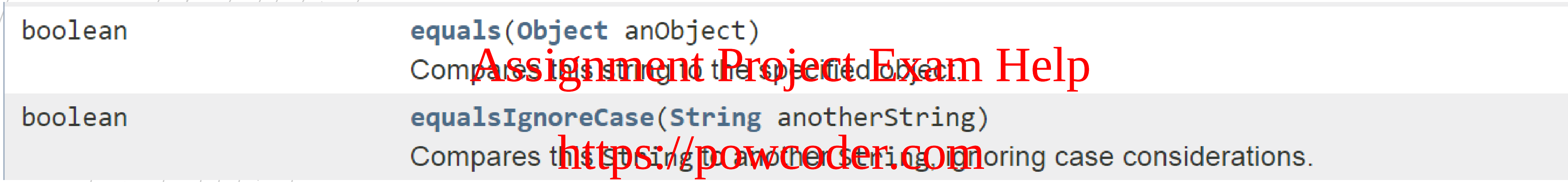


Methods

Modifier and Type	Method and Description
char	charAt(int index) Returns the char value at the specified index.
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.
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.
boolean	contains(CharSequence s) Returns true if and only if this string contains the specified sequence of char values.
boolean	contentEquals(CharSequence cs) Compares this string to the specified CharSequence.
boolean	contentEquals(StringBuffer sb) Compares this string to the specified StringBuffer.
static String	copyValueOf(char[] data) Returns a String that represents the character sequence in the array specified.
static String	copyValueOf(char[] data, int offset, int count) Returns a String that represents the character sequence in the array specified.
boolean	endsWith(String suffix) Tests if this string ends with the specified suffix.
boolean	equals(Object anObject) Compares this string to the specified object.
boolean	equalsIgnoreCase(String anotherString) Compares this String to another String, ignoring case considerations.

COMPARING STRINGS

- To compare two strings you can use one of the following methods



boolean	**equals**(**Object** anObject) Compares this string to the specified object.
boolean	**equalsIgnoreCase**(**String** anotherString) Compares this String to another String, ignoring case considerations.

- `equals` is case sensitive, use `equalsIgnoreCase` if you don't want to distinguish between upper and lower case.
- Note that there's no keyword `static`!
  This means that the methods need to be called on a specific value/variable of type `String` and not on the name of the class (like, for instance, the method `abs` from the `Math` library).

EXAMPLES

```
String course = "COMP 250";
String course2 = "comp 250";
boolean a = course.equals(course2);
boolean b = course.equalsIgnoreCase(course2);
```

- **The value of** `a` **is** `false`
- **The value of** `b` **is** `true`

BE CAREFUL!

- If you try to use == or != on Strings your program will compile and run.
  It is **not** doing what you think it's doing though.

- **Always use** `equals` **or** `equalsIgnoreCase` **if you want to compare strings.**

OTHER METHODS

Let `s` be a variable of type `String`. Then some useful methods include:

- `s.length()`

  It takes no inputs and returns the number of characters in the `String s`.

- `s.charAt(i)`

  It takes an integer as input and returns the character in the `String s` which has index equal to `i`. The index determines the position of the character in the String. Note that the first character is in position 0.

  If in the `String s` there's no character with index `i`, then we will get a run-time error. (`StringIndexOutOfBoundsException`)

EXAMPLE

```
String s = "Another string";
System.out.println(s.length());
```

What prints?

- 14

EXAMPLE

```
String s = "Another string";

System.out.println(s.charAt(2));
```

What prints?

- o

EXAMPLE

```
String s = "Another string";

System.out.println(s.charAt(0)=='a');
```

What prints?

- `false`

REVIEW – METHODS FROM THE STRING CLASS

```
String s = "Review";
```

Example – method call	Input type	Return type	Return value
`s.equals("review")`	String	boolean	false
`s.equalsIgnoreCase("review")`	String	boolean	true
`s.length()`	none	int	6
`s.charAt(2)`	int	char	'v'
`s.toLowerCase()`	none	String	"review"
`s.toUpperCase()`	none	String	"REVIEW"

CONVERTING TYPES WITH STRINGS

You cannot use a cast when converting from a String.

- To convert from `int/double` to a `String`, just concatenate the number with the empty String ("").

```
String s = "" + 4;
```

- To convert from a `String` to an `int`, use:

```
int x = Integer.parseInt("54");
String s = "5";
int y = Integer.parseInt(s);
```

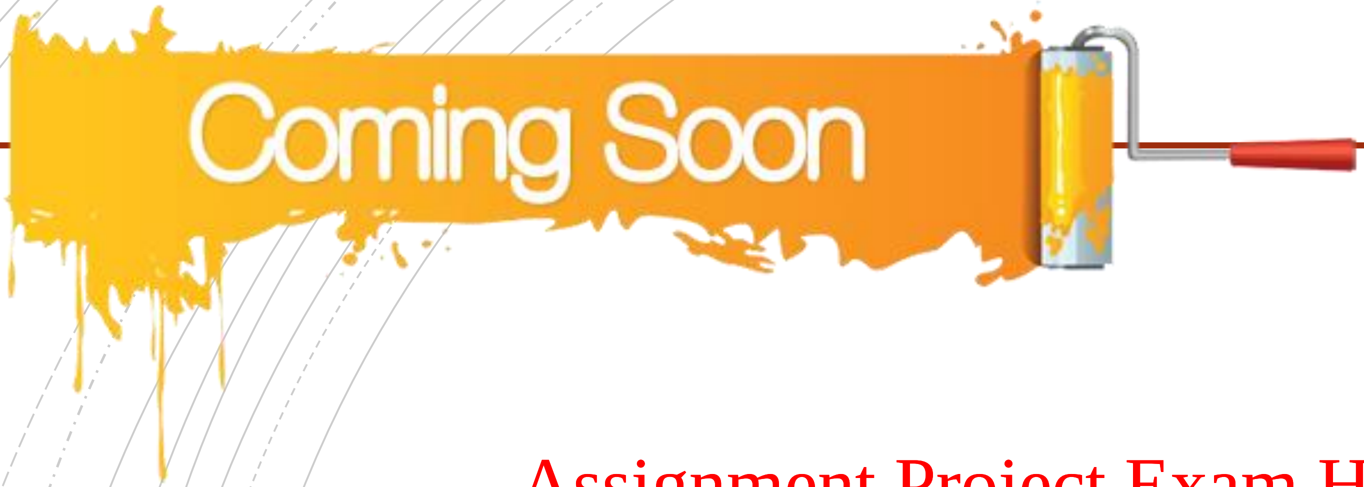
- To convert from a `String` to a `double`, use:

```
double z = Double.parseDouble("5.4");
```

TRY IT!

1. Write a method that takes a `String` as input and prints `true` if the String received is equal to a password (you, the programmer, can choose the password). The method should print `false` otherwise.

2. Write a method that takes a `String s` and an `int i` as input. The method should return `true` if the character at index `i` is a vowel, `false` otherwise.



In the next video we will be talking about arrays and reference types.