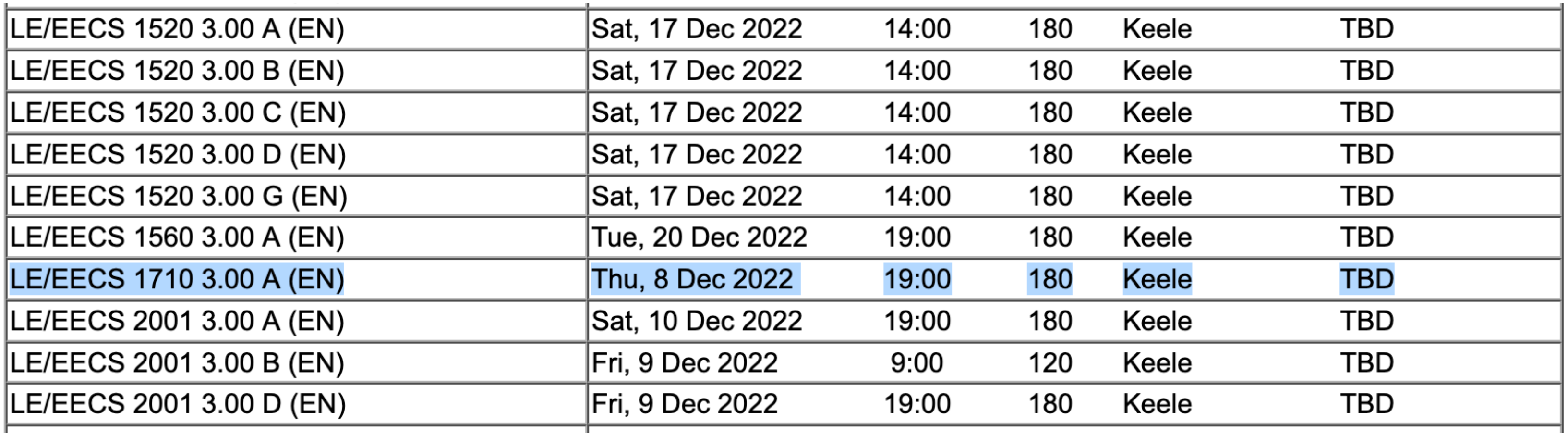
EECS 1710
Programming for Digital Media

Lecture 13 :: Objects I

Announcements

- Final EXAM schedule (date) is set
  - https://w2prod.sis.yorku.ca/Apps/WebObjects/cdm.woa/wa/curexam
  - location TBD still

- EECS1710 final exam is scheduled for:
  - Thursday December 8, 2022 (7pm start)



LE/EECS 1520 3.00 A (EN)	Sat, 17 Dec 2022	14:00	180	Keele	TBD
LE/EECS 1520 3.00 B (EN)	Sat, 17 Dec 2022	14:00	180	Keele	TBD
LE/EECS 1520 3.00 C (EN)	Sat, 17 Dec 2022	14:00	180	Keele	TBD
LE/EECS 1520 3.00 D (EN)	Sat, 17 Dec 2022	14:00	180	Keele	TBD
LE/EECS 1520 3.00 G (EN)	Sat, 17 Dec 2022	14:00	180	Keele	TBD
LE/EECS 1560 3.00 A (EN)	Tue, 20 Dec 2022	19:00	180	Keele	TBD
LE/EECS 1710 3.00 A (EN)	Thu, 8 Dec 2022	19:00	180	Keele	TBD
LE/EECS 2001 3.00 A (EN)	Sat, 10 Dec 2022	19:00	180	Keele	TBD
LE/EECS 2001 3.00 B (EN)	Fri, 9 Dec 2022	9:00	120	Keele	TBD
LE/EECS 2001 3.00 D (EN)	Fri, 9 Dec 2022	19:00	180	Keele	TBD

Recall:

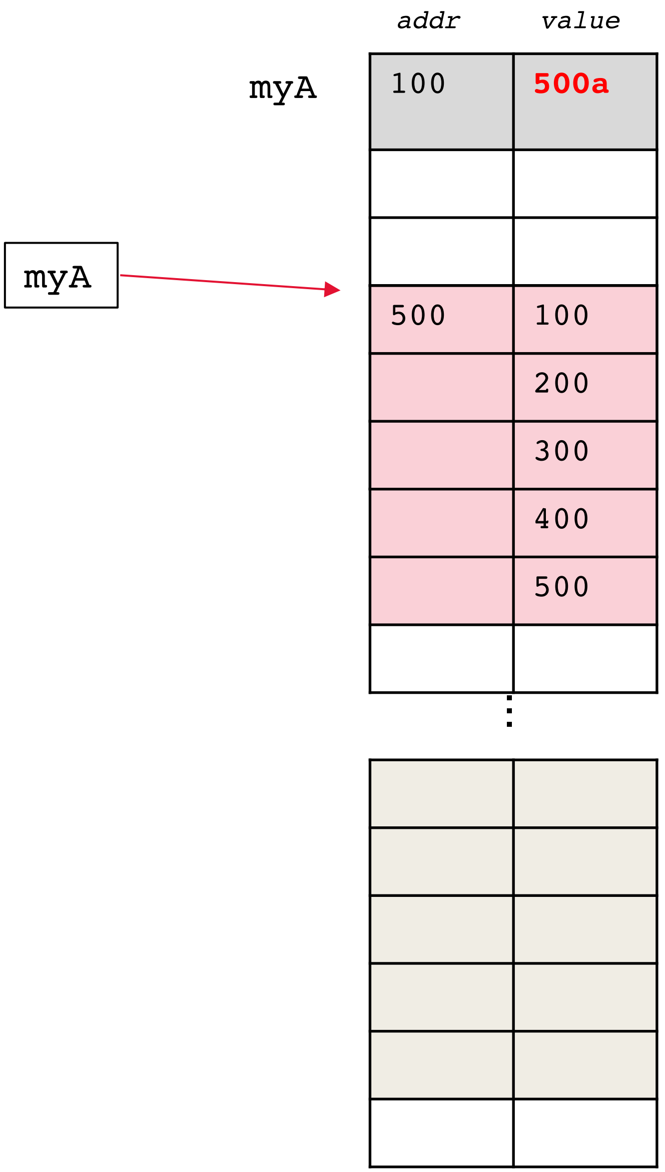
- what is a reference type (as opposed to primitive type)?
  - E.g. Arrays:

```
int[] myA = { 100,200,300,400,500 };
print(myA);

myA = new int[5];
// assign same values as above
print(myA);
```

  - E.g. Strings:

```
String str1 = “Bob”;
String str2 = “Jane”;
String str3 = “Bob”;

println(str1 == str2);
println(str1 == str3);
```

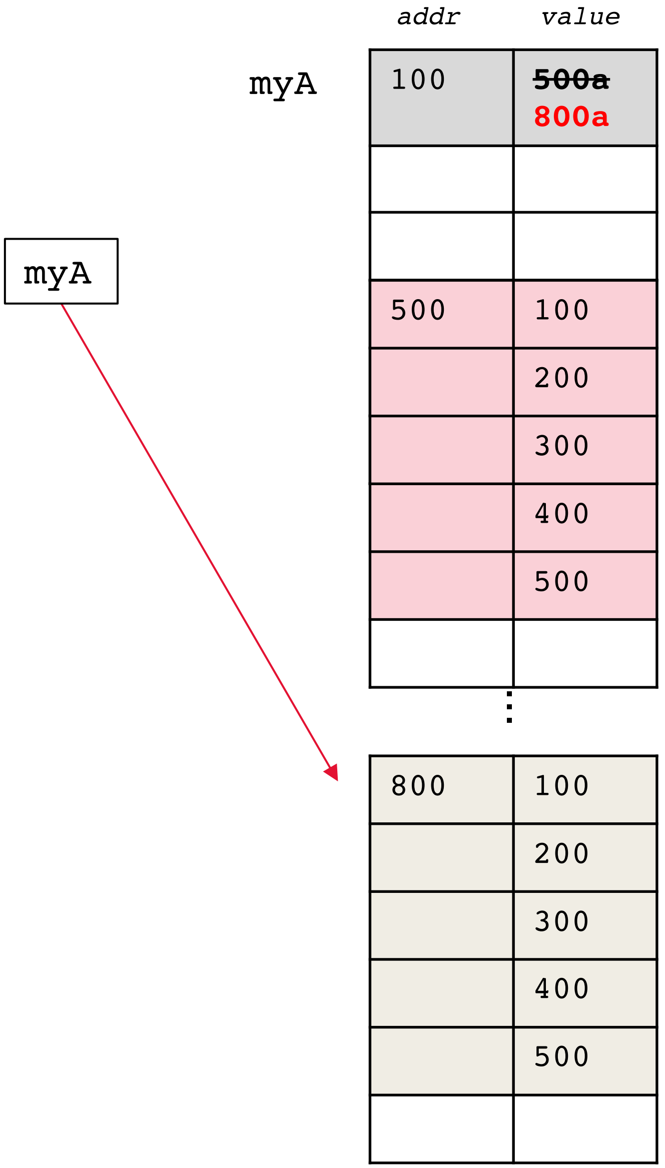
```
int[] myA = { 100,200,300,400,500 };
print(myA);
for (int i=0; i<myA.length; i++ ) {
  println("\tmyA[" + i + "] = " + myA[i] );
}

myA = new int[5];
// assume we set values similarly
for (int i=0; i<myA.length; i++ ) {
  myA[i] = i*100 + 100;
}

print(myA);
for (int i=0; i<myA.length; i++ ) {
  println("\tmyA[" + i + "] = " + myA[i] );
}
```



	addr	value
myA	100	500a
	500	100
		200
		300
		400
		500
	⋮	

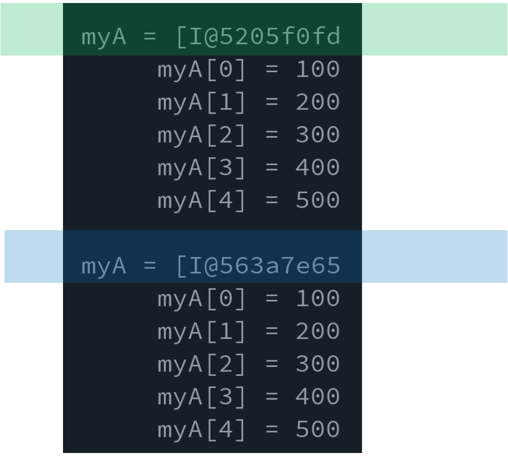
```
int[] myA = { 100,200,300,400,500 };
print(myA);
for (int i=0; i<myA.length; i++ ) {
  println("\tmyA[" + i + "] = " + myA[i] );
}

myA = new int[5];
// assume we set values similarly
for (int i=0; i<myA.length; i++ ) {
  myA[i] = i*100 + 100;
}

print(myA);
for (int i=0; i<myA.length; i++ ) {
  println("\tmyA[" + i + "] = " + myA[i] );
}
```

Should these print statements be the same?

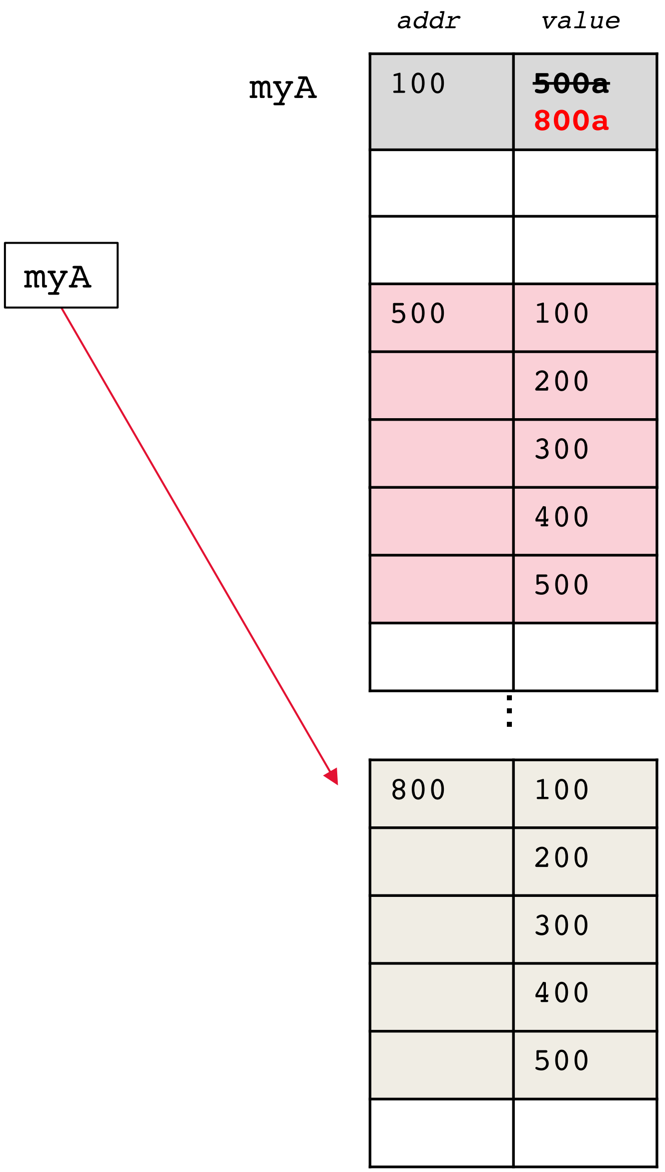


	addr	*value*
myA	100	~~500a~~ 800a
	500	100
		200
		300
		400
		500
	⋮	
	800	100
		200
		300
		400
		500

```
int[] myA = { 100,200,300,400,500 };
print(myA);
for (int i=0; i<myA.length; i++ ) {
  println("\tmyA[" + i + "] = " + myA[i] );
}

myA = new int[5];
// assume we set values similarly
for (int i=0; i<myA.length; i++ ) {
  myA[i] = i*100 + 100;
}

print(myA);
for (int i=0; i<myA.length; i++ ) {
  println("\tmyA[" + i + "] = " + myA[i] );
}
```



```
myA = [I@5205f0fd
        myA[0] = 100
        myA[1] = 200
        myA[2] = 300
        myA[3] = 400
        myA[4] = 500

myA = [I@563a7e65
        myA[0] = 100
        myA[1] = 200
        myA[2] = 300
        myA[3] = 400
        myA[4] = 500
```



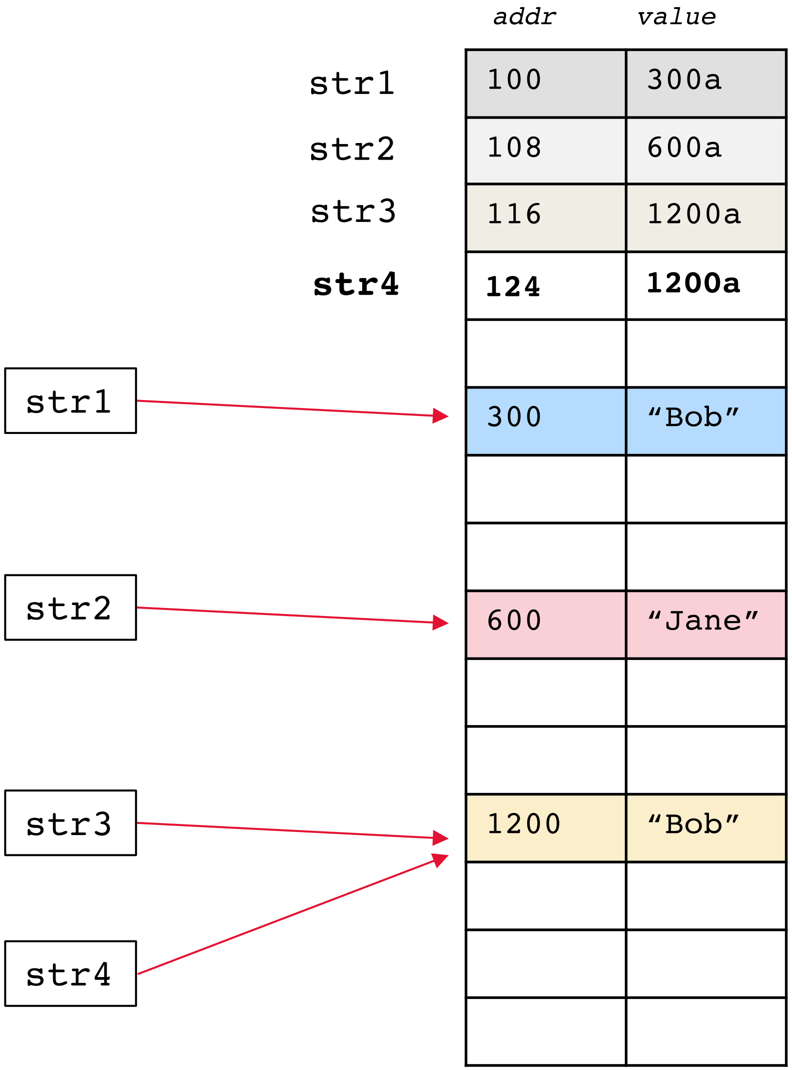
	addr	value
myA	100	~~500a~~ 800a
	500	100
		200
		300
		400
		500
	⋮	
	800	100
		200
		300
		400
		500

```
String str1 = “Bob”;
String str2 = “Jane”;
String str3 = “Bob”;

println(str1 == str2);
println(str1 == str3);
```

```
false
false
```

```
// how about this?
String str4 = str3;


// prints?
println(str3);
println(str4);
println(str4 == str3);
```

```
Bob
Bob
true
```



	addr	*value*
str1	100	300a
str2	108	600a
str3	116	1200a
str4	**124**	**1200a**
	300	“Bob”
	600	“Jane”
	1200	“Bob”

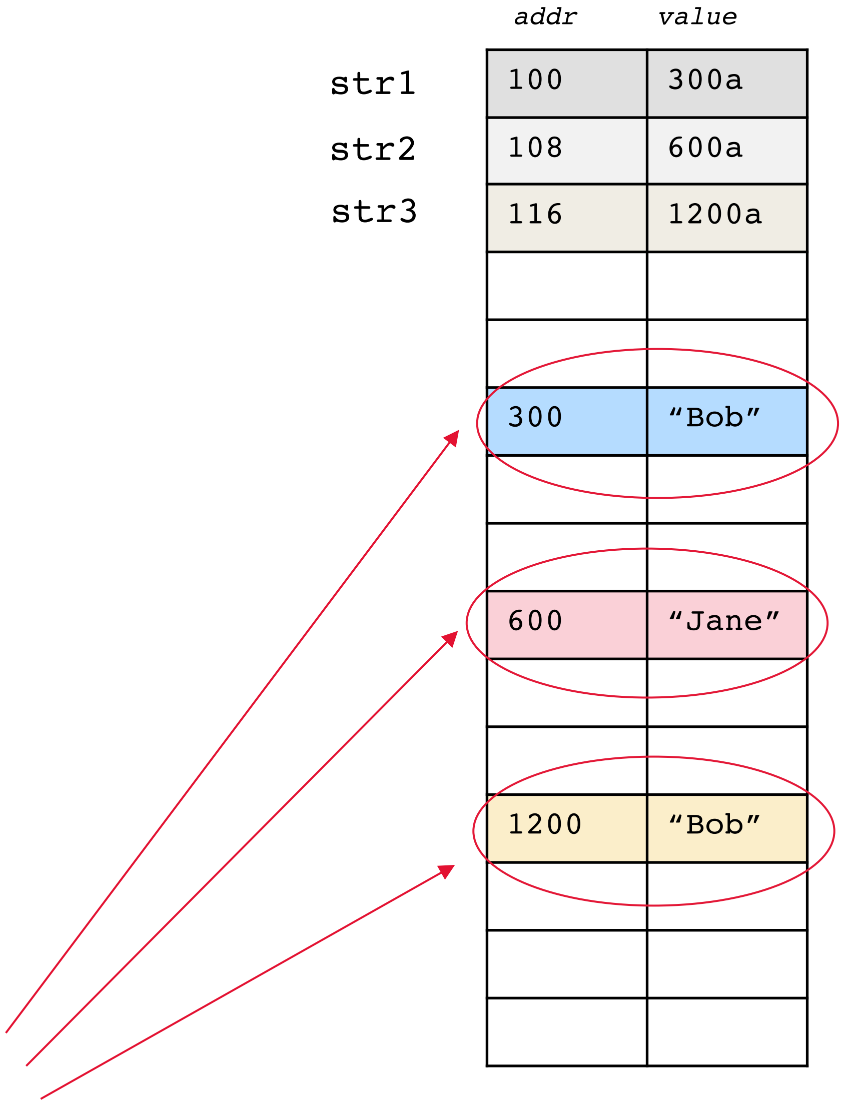
Reference types

- Point to locations in memory
    - i.e. references to address locations

- Even if two types are the same, and contain the same data values, they aren't necessarily in the same place in memory!
    - But if I assign one reference type to another?
        - String str4 = str3;   // then they both hold same address
    - They both point to the same thing in memory? [yes]

- A reference type holds an address
    - of a type of thing in memory
    - The "thing" in memory, is not a type per se… its an object!

```
String str1 = "Bob";
String str2 = "Jane";
String str3 = "Bob";
```



	addr	value
str1	100	300a
str2	108	600a
str3	116	1200a
	300	"Bob"
	600	"Jane"
	1200	"Bob"

String objects!

Does == have meaning for reference types?

- Generally, it does not mean equality…
- It means… same position (address) in memory!

- OK… so how to test if two strings are the same??
  - One way: convert to char[ ] and test all characters are same
    - Not convenient really.. A bit annoying
  - Another way: utilize some of the built in methods for available for the String type!
    - More convenient
    - In fact, reference types like String usually include a host of other methods that make life more convenient (think of them as a collection of useful methods that help us work with String types)

Introduction to Classes & Objects

A quick introduction to classes & objects

- Objects are possible if there are variable features in a class (i.e. variable attributes/fields)

- E.g. Strings
  - (recall: these, like arrays are a reference type)

```
String str1 = “Bob”;
String str2 = “Jane”;
```

- `str1` and `str2` each refer to individual `String` “objects”
  - They are each entirely separate, but they are similar!
    - they each have the same definition (data types/underlying config)
    - they each have different state (data values)
    - they each have special features (properties & methods)

A quick introduction to objects

- The “String” type defines:
  - The structure of a string (i.e. a sequence of characters)
  - This makes a “String” variable different from a “char” variable

- A String variable uses this definition to refer to the value of a specific String

```
String str1 = “Bob”;
```

  - We say that **`str1`** refers to an <u>instance</u> of a String type
  - “Bob” is a specific instance of a String
  - An instance is also called an “object”

Class vs Object

class → defines a reference type!

- Class
  - A **category** of a thing (a ***type*** of thing)
    - A thing is defined by a set of attributes/properties
      - Multiple attributes (a composite of different types)
      - Attributes are usually encoded by one/more variables
    - A thing has an associated set of behaviours
      - These define what can be done on/with that thing
  - In Java – a *class* is a formal structure that defines a *type*
    - organizes/ collates several attributes and behaviours together
- Object
  - A particular version (**instance**) of a ***type*** of thing

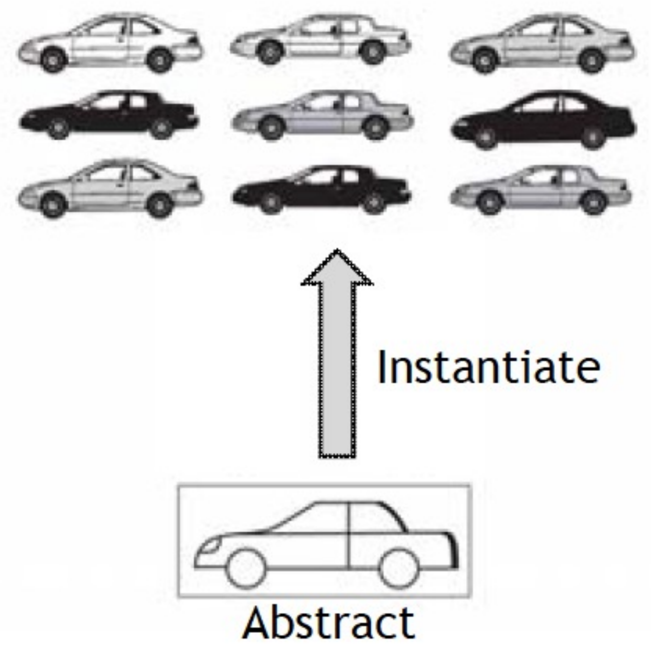
object → instantiates a reference type!

- Examples:
  - `"hello"` and `"how are you?"` are both instances of the type: String
      - They are each a sequence of characters (so they are the same *type* of thing)
      - They are each unique (they are each different versions of a String)
  - Imagine a class of object called "Human" → we are all instances of the type: **Human**
      - Human is the class/type of thing we are
      - we are EACH a unique version of a Human (we are each unique instances of a Human)

Objects (specific) vs. Classes (abstract)

- An object has attributes[1], methods, an identity, and a state
- A class has attributes[1] and methods
- Objects with the same attributes and methods can be represented by a class that abstracts them:

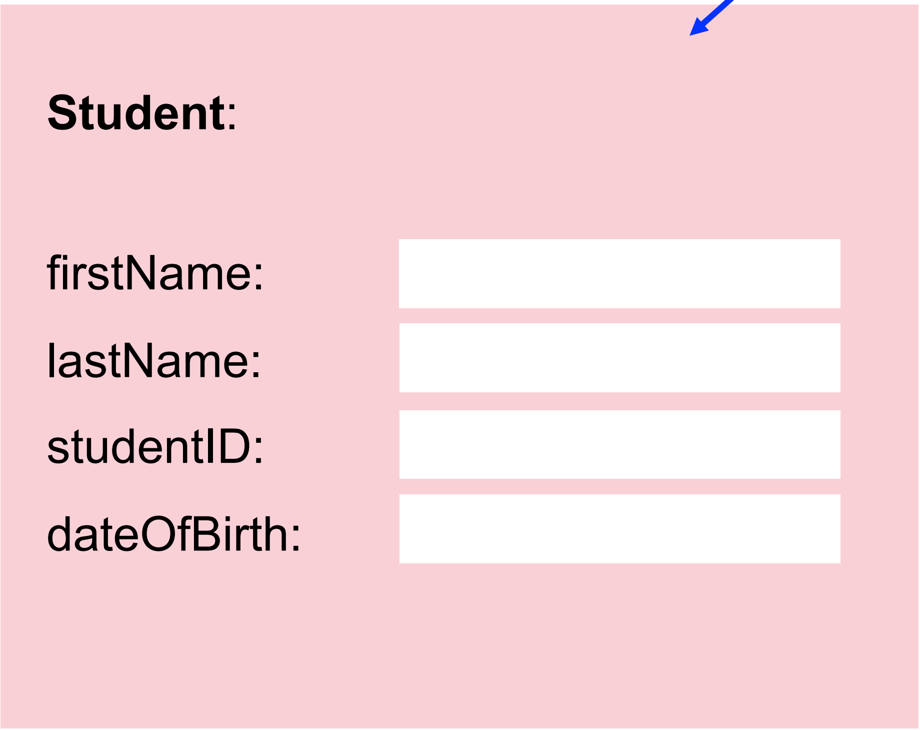
fields (variables)



Analogy:

- Think of a class as way of defining a form

empty form =
blueprint / template for a
generic Student



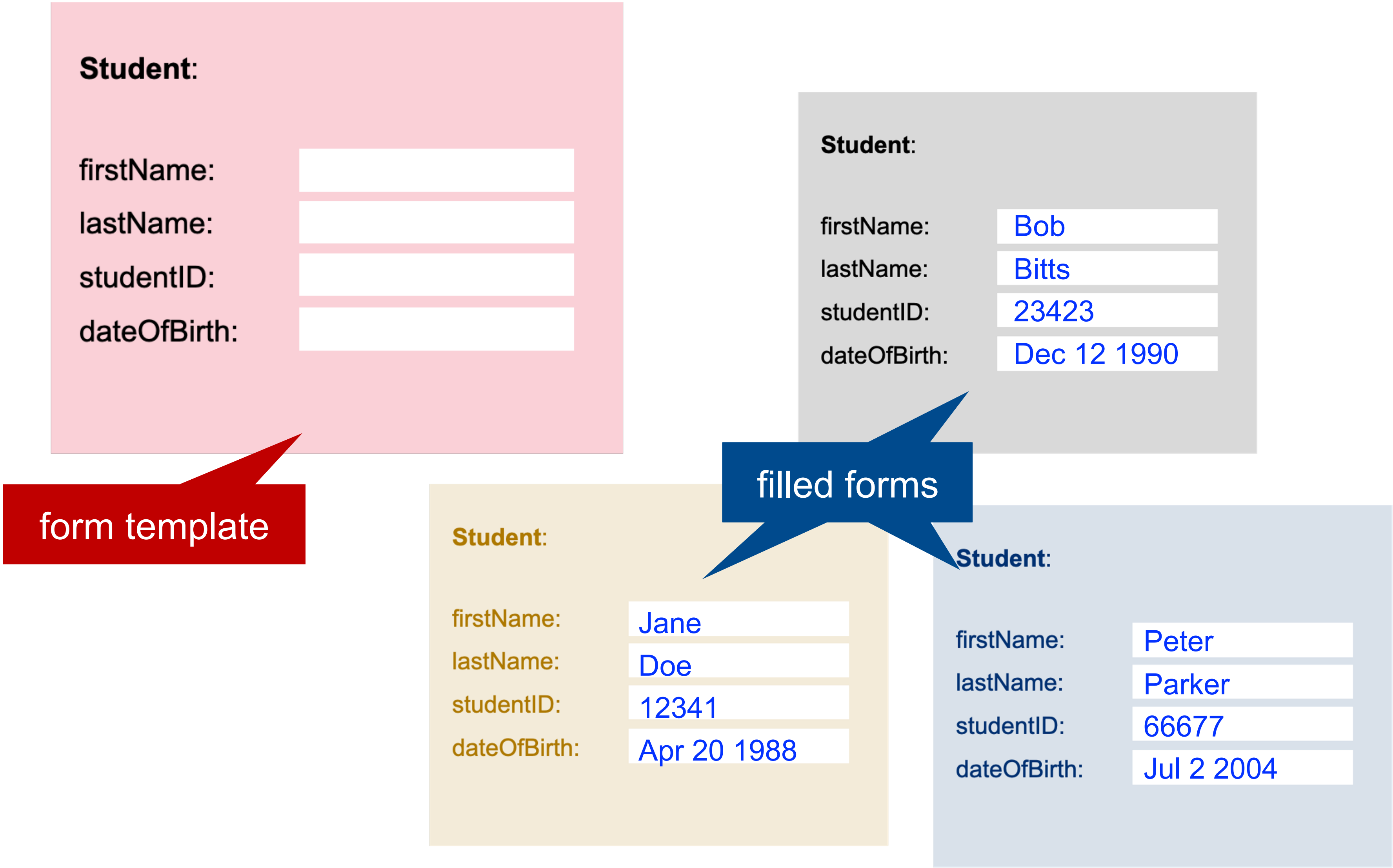
defines data relevant to a Student

template acts as an “abstraction” of a Student

“abstraction” in that a Student is defined by common attributes

Instantiating a form (making objects)?

make copies & fill out many times



Student:

firstName:	
lastName:	
studentID:	
dateOfBirth:	

form template

Student:

firstName:	Bob
lastName:	Bitts
studentID:	23423
dateOfBirth:	Dec 12 1990

filled forms

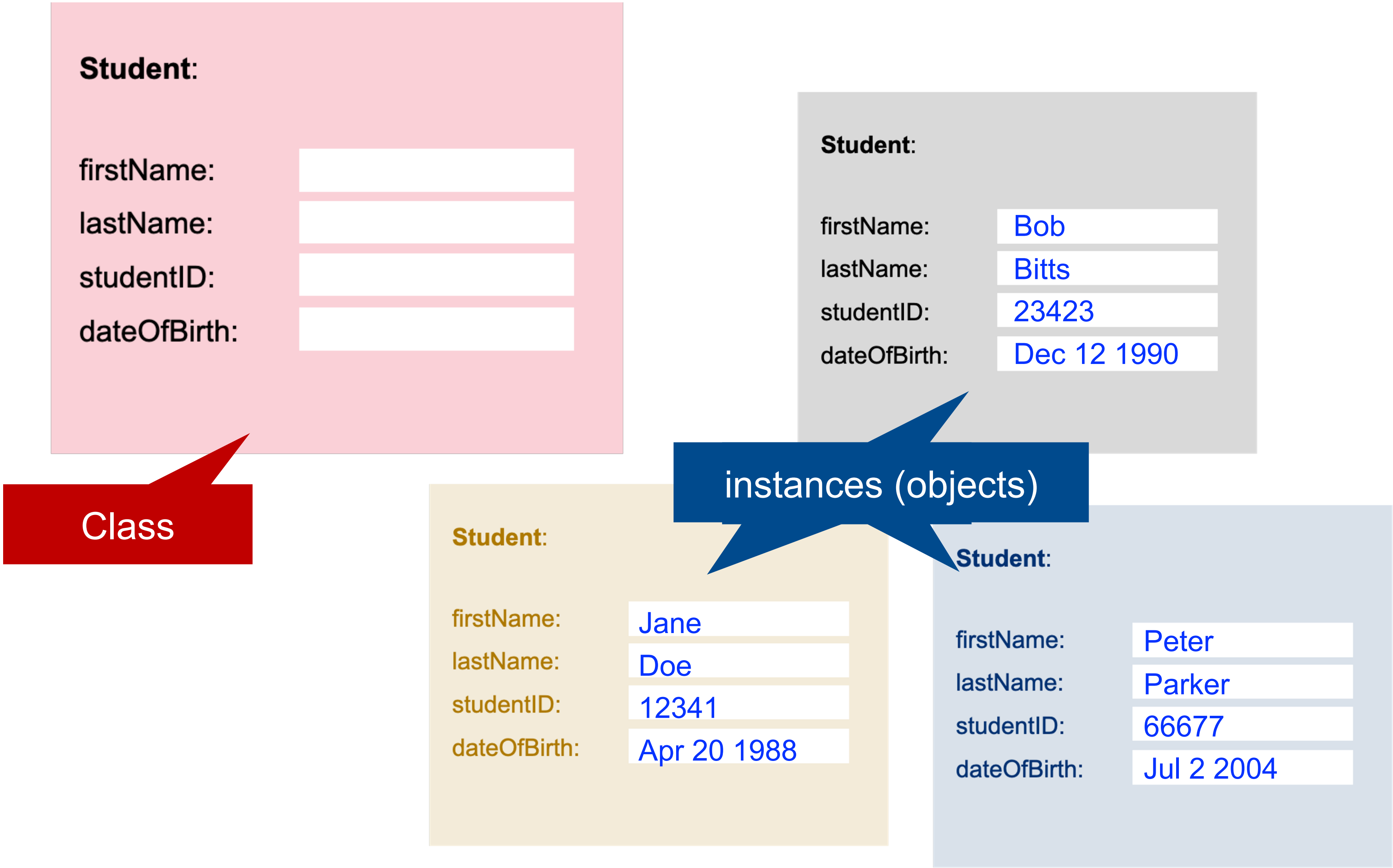
Student:

firstName:	Jane
lastName:	Doe
studentID:	12341
dateOfBirth:	Apr 20 1988

Student:

firstName:	Peter
lastName:	Parker
studentID:	66677
dateOfBirth:	Jul 2 2004

Class acts as template for an Object
An object is an "instance" of a Class



Student:

firstName:	
lastName:	
studentID:	
dateOfBirth:	

Class

Student:

firstName:	Bob
lastName:	Bitts
studentID:	23423
dateOfBirth:	Dec 12 1990

instances (objects)

Student:

firstName:	Jane
lastName:	Doe
studentID:	12341
dateOfBirth:	Apr 20 1988

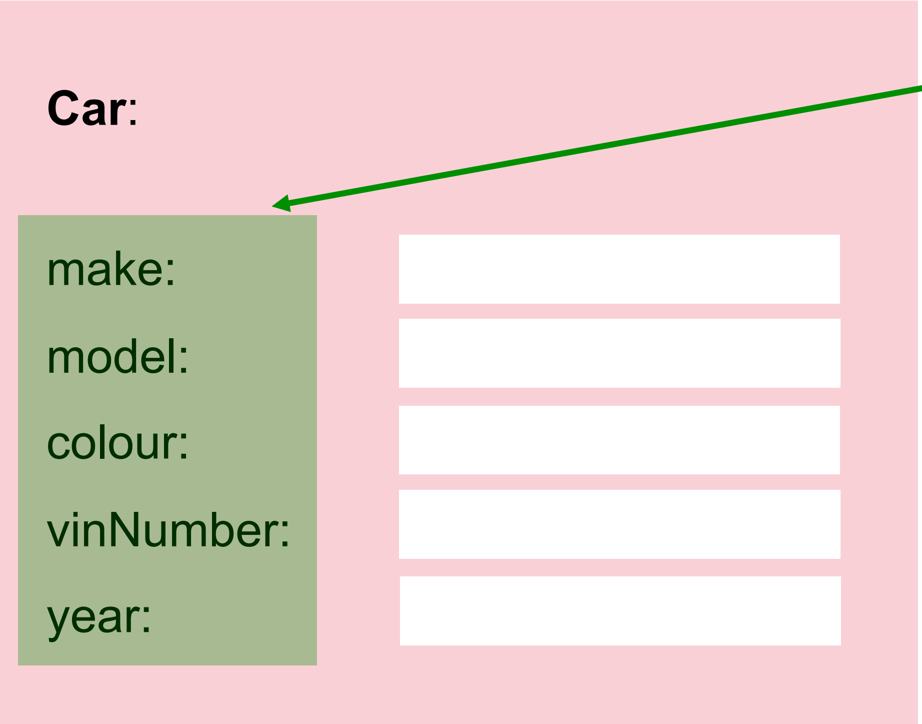
Student:

firstName:	Peter
lastName:	Parker
studentID:	66677
dateOfBirth:	Jul 2 2004

Analogy:

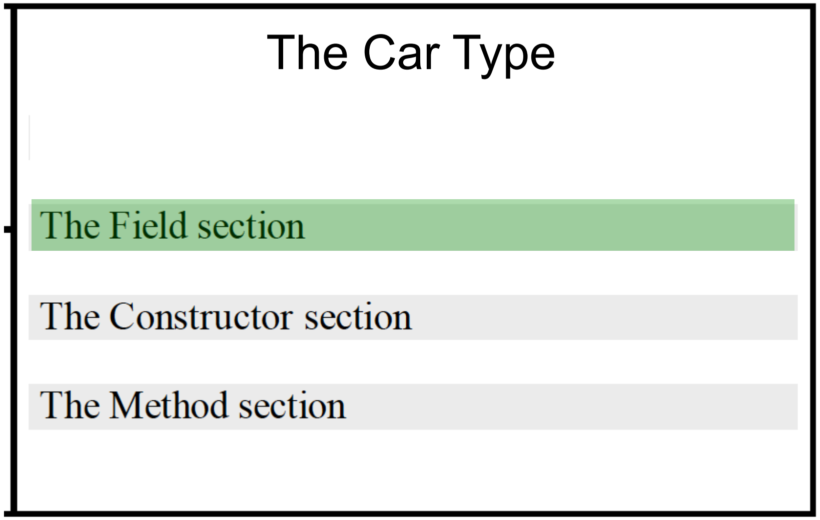
- Car type

Car class = blueprint / template for a generic car



fields (attributes)

```
e.g.     String make;
         String model;
         Color colour;
         int vinNumber;
         int year;
```


The Car Type

The Field section

The Constructor section

The Method section

A class is an abstract view of a type

- An object has attributes[1], methods, an identity, and a state
- A class has attributes[1] and methods
- Objects with the same attributes and methods can be represented by a class that abstracts them:

Car:

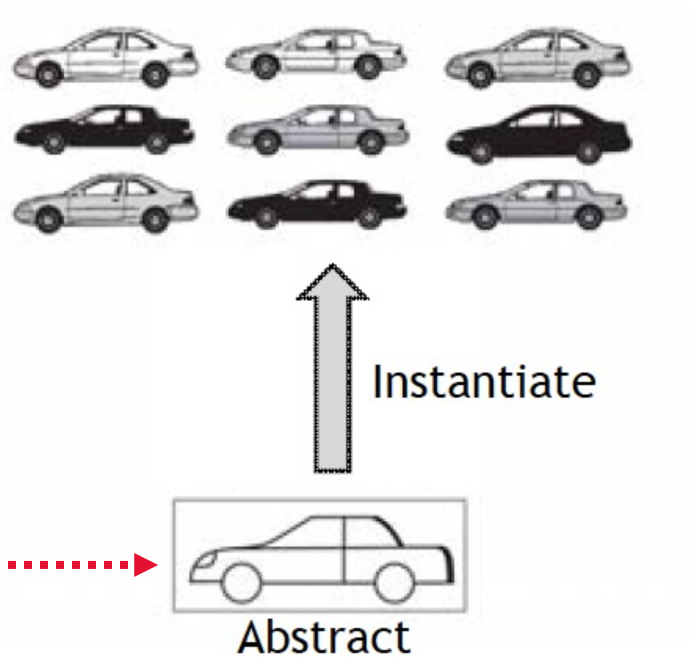
make:

model:

colour:

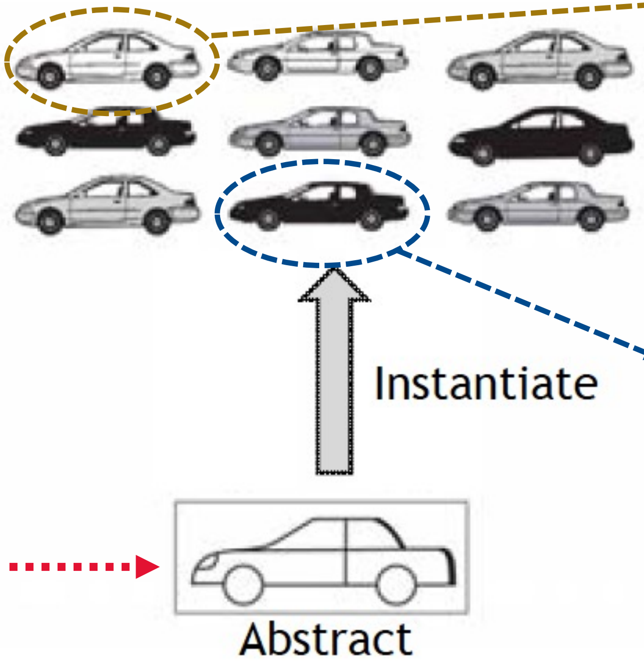
vinNumber:

year:



An object is a concrete view of a type

- An object has attributes[1], methods, an identity, and a state
- A class has attributes[1] and methods
- Objects with the same attributes and methods can be represented by a class that abstracts them:



Car:

make:	Mazda
model:	CX5
colour:	blue
vinNumber:	R121FI23100
year:	2015

Car:

make:	
model:	
colour:	
vinNumber:	
year:	

Car:

make:	Toyota
model:	Corolla
colour:	grey
vinNumber:	Z023BI44913
year:	2018

back to String objects …

```
String str1 = “Bob”;
String str2 = “Jane”;
```

- `str1` and `str2` are variable names
  - we consider them as *references* to two separate String objects
  - String objects “Bob” and ”Jane” represent individual “instances” of a String
  - These objects exist in separate memory locations

There is another way to create a String…

- Using a “constructor”

```
String str1 = “Bob”;
String str2 = new String(“Jane”);
```

- What is a constructor?
  - Called at the same time we ***instantiate*** an object (create a new object)
  - A special method (with same name as the type), used to ***initialize*** an object at instantiation
  - Can **only** be called alongside “new” keyword
    - as memory is allocated, this method is used to initialize all the fields associated with the new object

There is another way to create a String…

- Using a “constructor”

```
String str1 = “Bob”;
String str2 = new String(“Jane”);
```

instantiation
- uses keyword “new”
(create a new object)

initialization
(set fields of the new object)

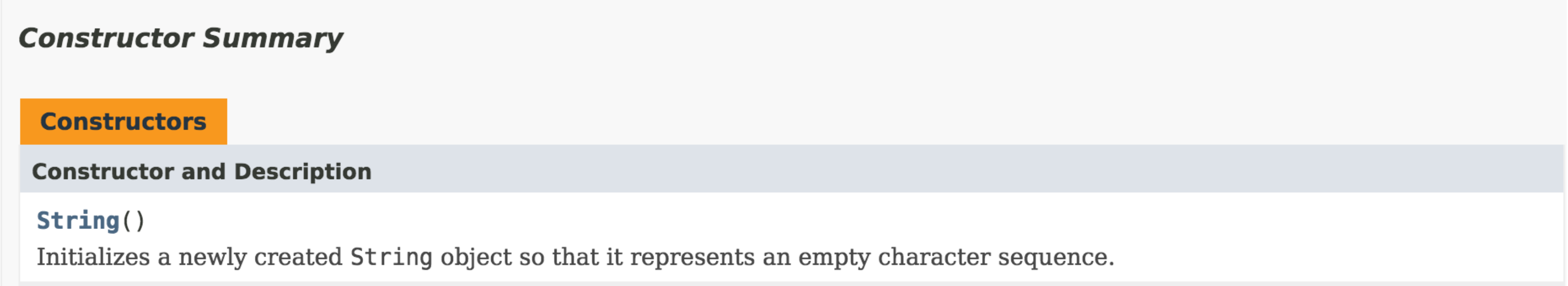
new + String(…) → create new String object

General way to construct objects
- is to use a constructor!

- Constructor is a special type of method that has the same name as the class
- It is used ONLY to create an instance of an object
- It may also allow for some input arguments (used to initialize the object. I.e. set its initial state)

- String has several constructors!

String (constructors) – Java

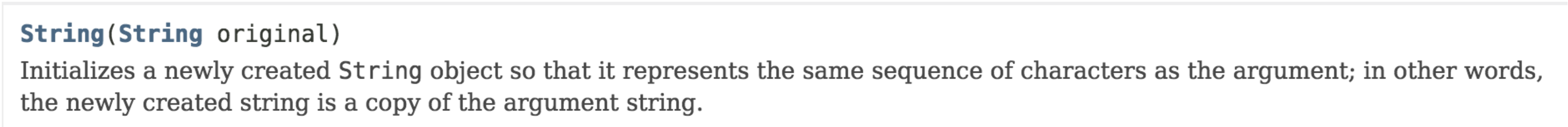


Constructor Summary

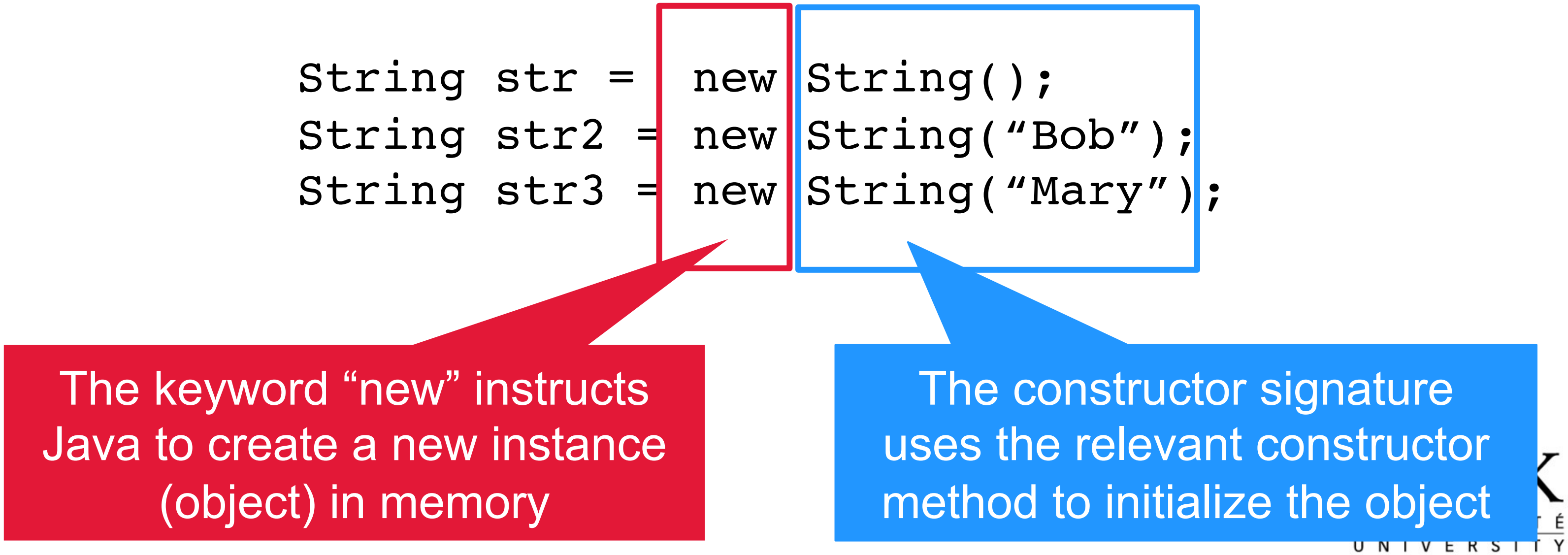
Constructors

Constructor and Description

`String()`
Initializes a newly created `String` object so that it represents an empty character sequence.



`String(String original)`
Initializes a newly created `String` object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.



```
String str  = new String();
String str2 = new String(“Bob”);
String str3 = new String(“Mary”);
```

The keyword “new” instructs Java to create a new instance (object) in memory

The constructor signature uses the relevant constructor method to initialize the object

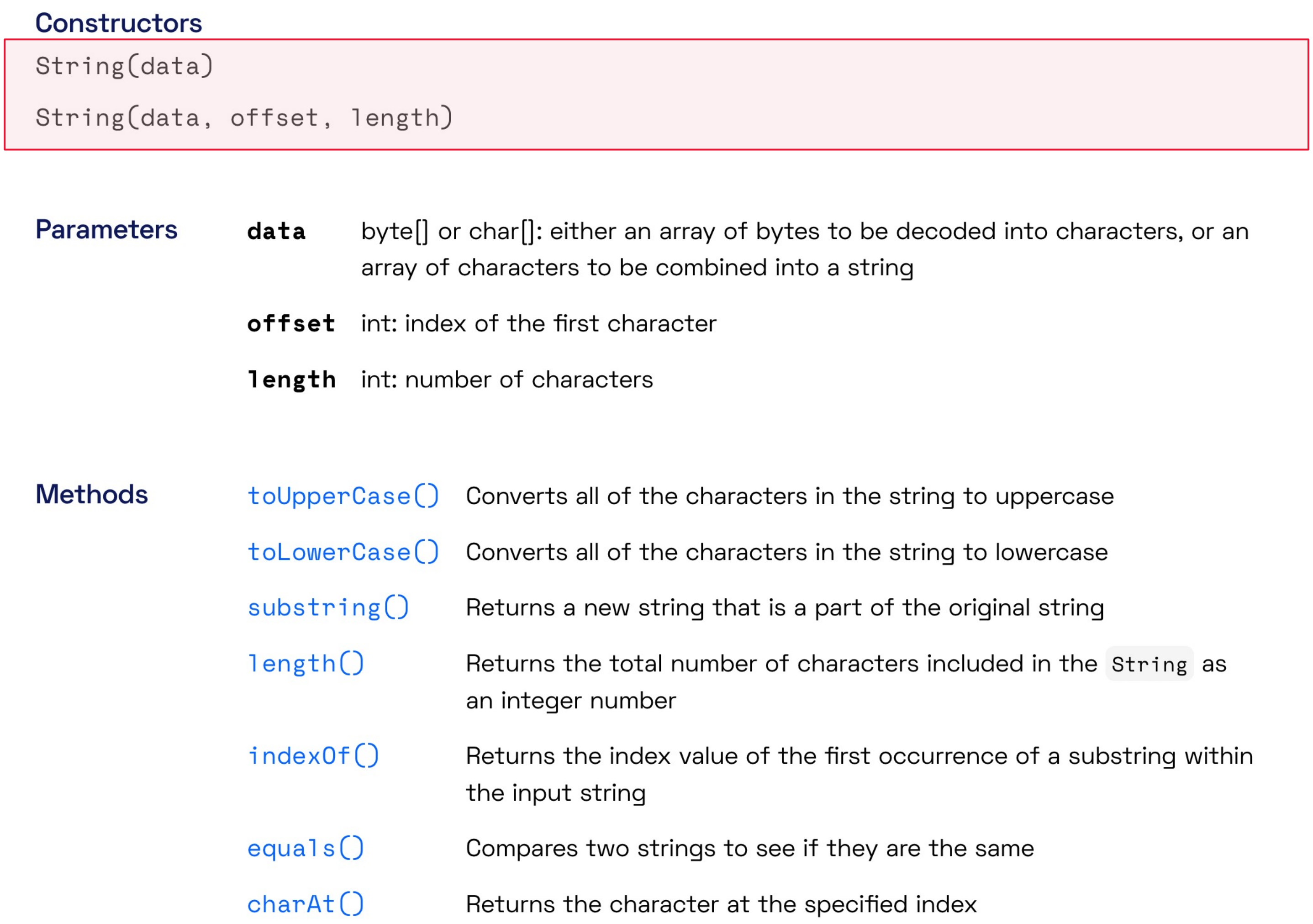
Notes

- Strings are a special case as far as classes go
    - Since they are commonly used (almost like a primitive type) we can declare them as we declare primitive types, OR by using the constructors:

```
String str1 = “Bob”;
String str2 = new String(“Mary”);
```

- For most classes, you need to create instances of objects for use in your program using the constructor approach

String (constructors) – Processing reference



Constructors

```
String(data)

String(data, offset, length)
```

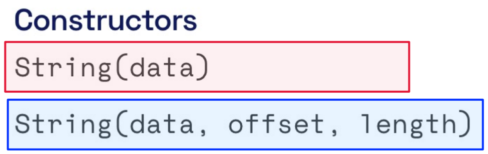
Parameters

data	byte[] or char[]: either an array of bytes to be decoded into characters, or an array of characters to be combined into a string
offset	int: index of the first character
length	int: number of characters

Methods

`toUpperCase()`	Converts all of the characters in the string to uppercase
`toLowerCase()`	Converts all of the characters in the string to lowercase
`substring()`	Returns a new string that is a part of the original string
`length()`	Returns the total number of characters included in the `String` as an integer number
`indexOf()`	Returns the index value of the first occurrence of a substring within the input string
`equals()`	Compares two strings to see if they are the same
`charAt()`	Returns the character at the specified index

Notes



Constructors

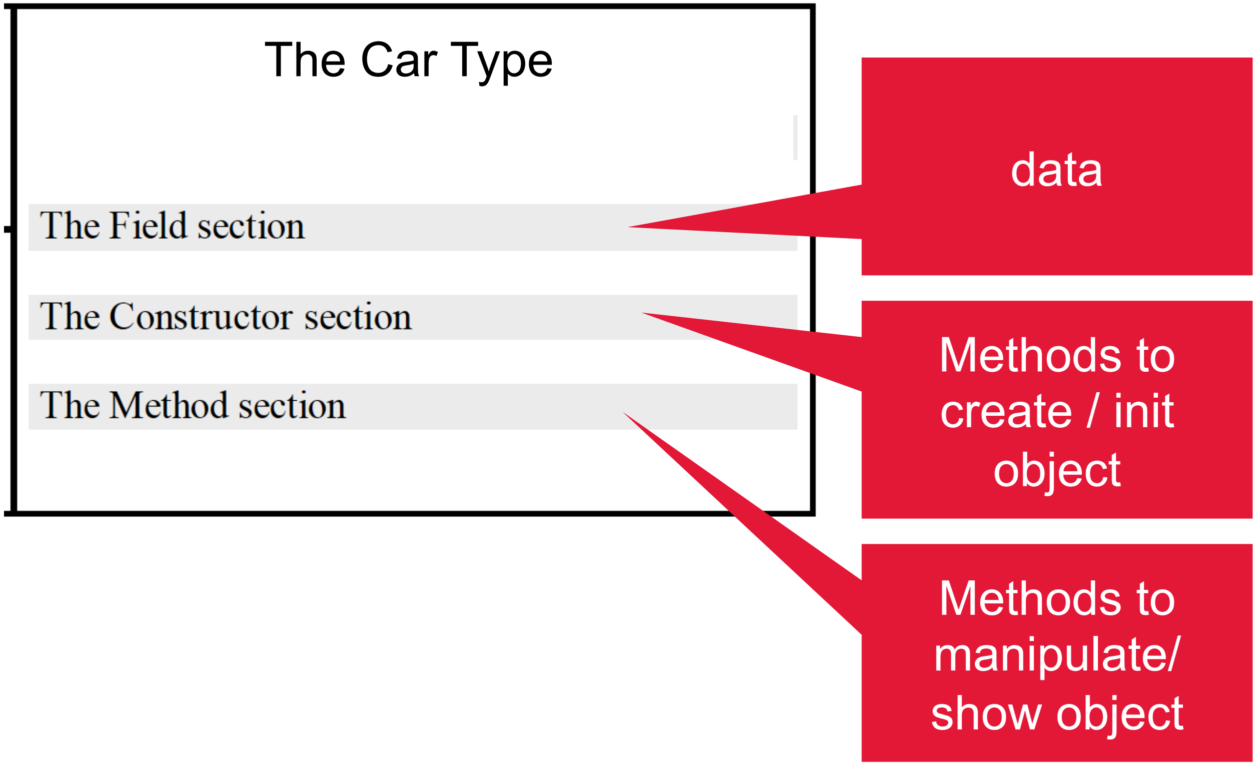
String(data)

String(data, offset, length)

- Other constructor examples (in processing)

```
String str1 = “Bob”;
String str2 = new String(“Mary”);

char[] charArray = {‘J’,’a’,’n’,’e’};

String str3 = new String(charArray);          // “Jane”

String str4 = new String(charArray,1,2);      // “an”
```

Objects typically define: fields + constructors + methods



The Car Type

The Field section

The Constructor section

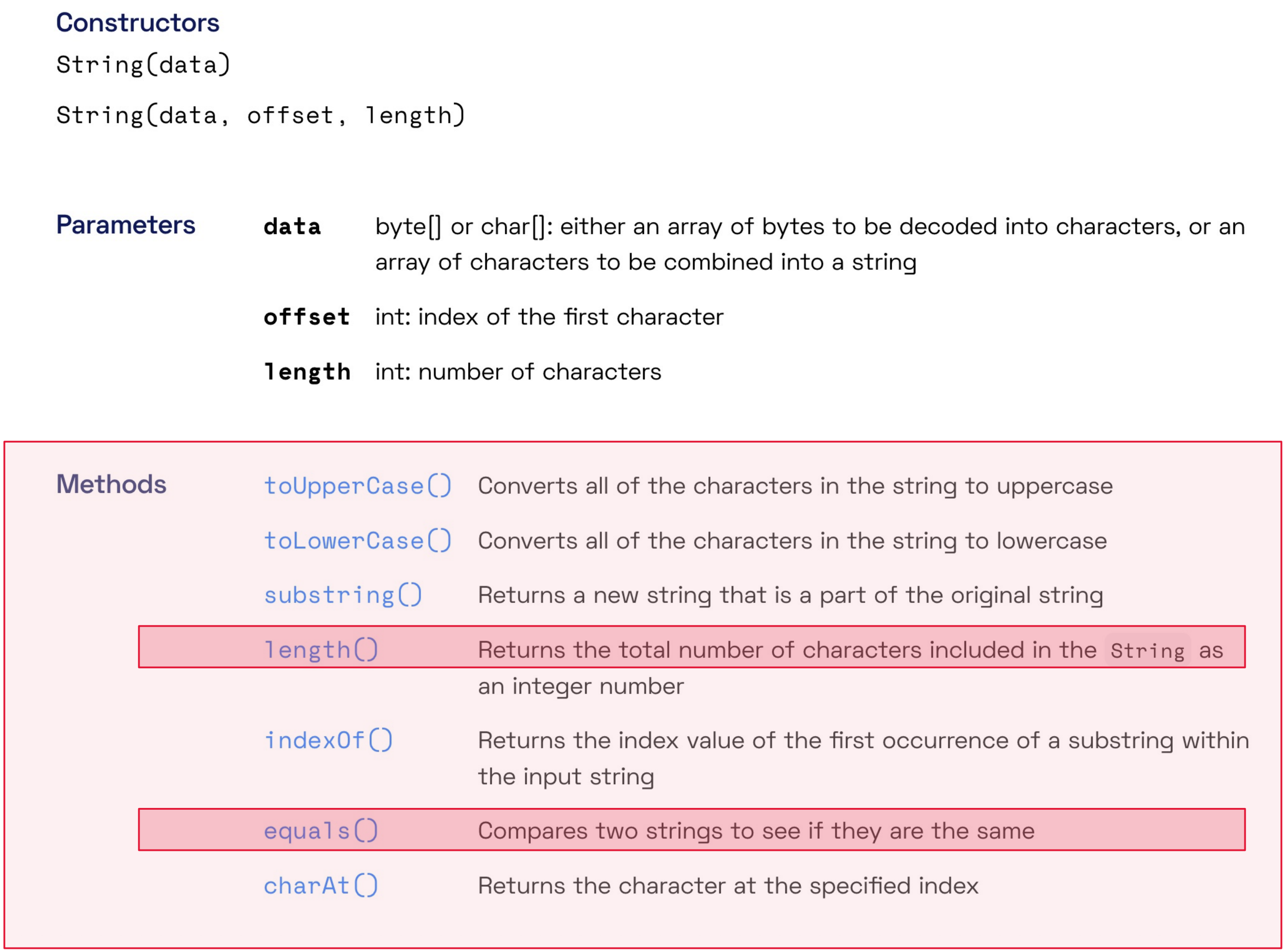
The Method section

data

Methods to create / init object

Methods to manipulate/ show object

String (methods) – Processing reference



Constructors

```
String(data)

String(data, offset, length)
```

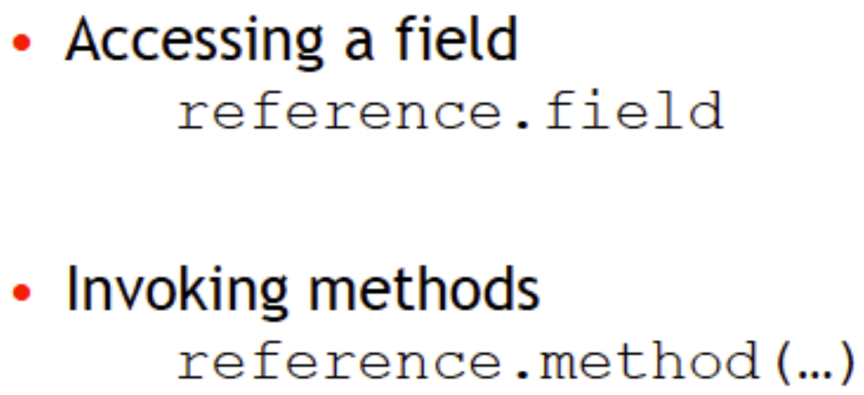
Parameters

`data`	byte[] or char[]: either an array of bytes to be decoded into characters, or an array of characters to be combined into a string
`offset`	int: index of the first character
`length`	int: number of characters

Methods

`toUpperCase()`	Converts all of the characters in the string to uppercase
`toLowerCase()`	Converts all of the characters in the string to lowercase
`substring()`	Returns a new string that is a part of the original string
`length()`	Returns the total number of characters included in the `String` as an integer number
`indexOf()`	Returns the index value of the first occurrence of a substring within the input string
`equals()`	Compares two strings to see if they are the same
`charAt()`	Returns the character at the specified index

We work through the variable to access fields or to invoke methods



- Accessing a field

  `reference.field`

- Invoking methods

  `reference.method(…)`

```
String str1 = “Bob”;
String str2 = new String(“Mary”);

// prints 3 (3 chars in Bob)
println(str1.length());

// this is how we test equality => false (“Bob” not same as “Mary”)
println(str1.equals(str2));
```

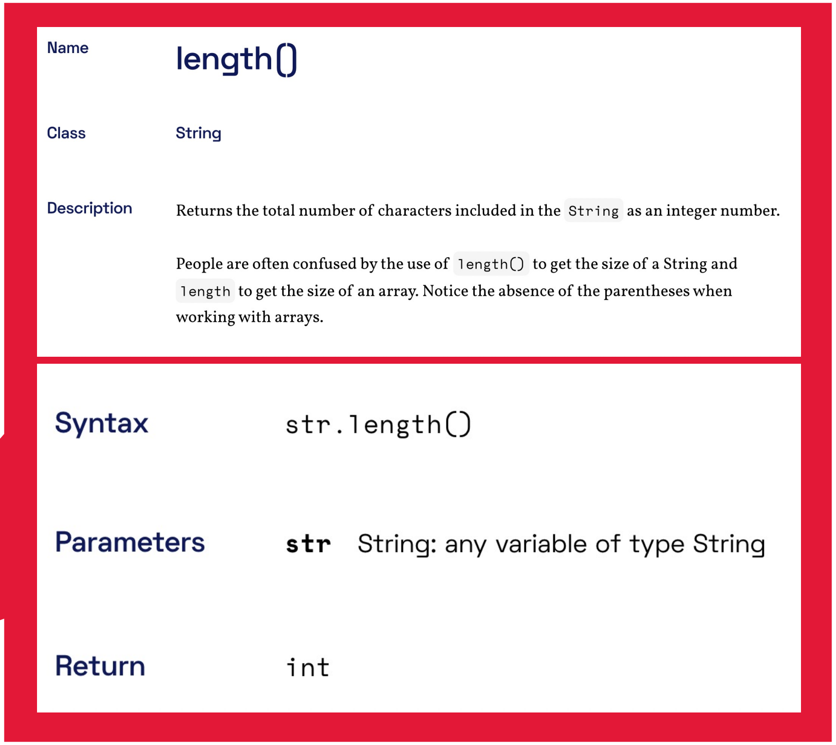
We work through the variable to access fields or to invoke methods

- Accessing a field

  `reference.field`

- Invoking methods

  `reference.method(...)`



Name	length()
Class	String
Description	Returns the total number of characters included in the `String` as an integer number. People are often confused by the use of `length()` to get the size of a String and `length` to get the size of an array. Notice the absence of the parentheses when working with arrays.

Syntax	`str.length()`	
Parameters	**str**	String: any variable of type String
Return	int	

```
String str1 = “Bob”;
String str2 = new String(“Mary”)

// prints 3 (3 chars in Bob)
println(str1.length());

// this is how we test equality => false (“Bob” not same as “Mary”)
println(str1.equals(str2));
```

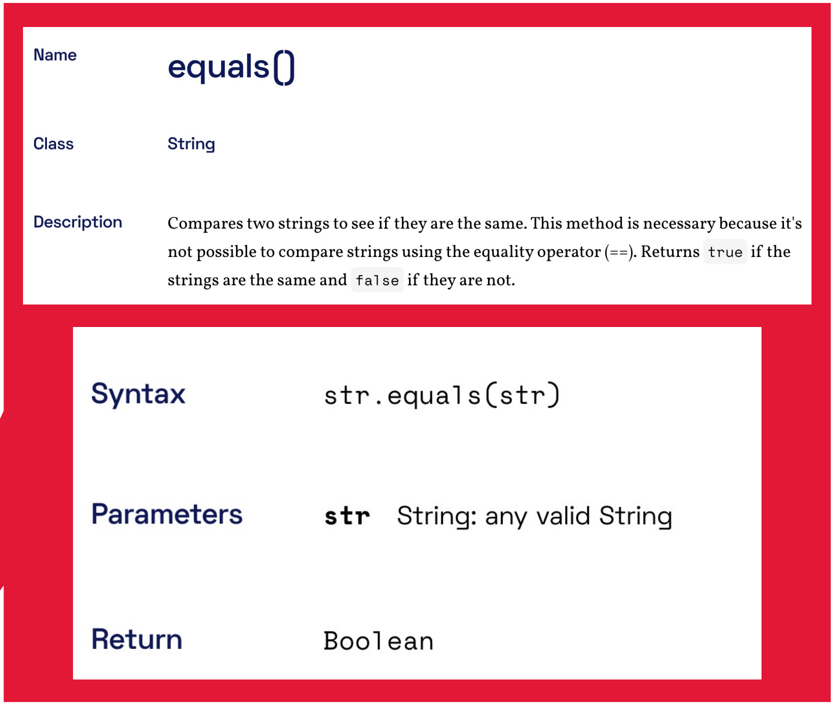
We work through the variable to access fields or to invoke methods

- Accessing a field

  `reference.field`

- Invoking methods

  `reference.method(...)`



Name equals()

Class String

Description Compares two strings to see if they are the same. This method is necessary because it's not possible to compare strings using the equality operator (==). Returns `true` if the strings are the same and `false` if they are not.

Syntax `str.equals(str)`

Parameters **`str`** String: any valid String

Return `Boolean`

```
String str1 = “Bob”;
String str2 = new String(“Mary”)

// prints 3 (3 chars in Bob)
println(str1.length());

// this is how we test equality => false (“Bob” not same as “Mary”)
println(str1.equals(str2));
```

So… why have reference types?

- convenient way to aggregate together many attributes
  - E.g. a String aggregates characters
- we can have other types that aggregate other attributes
- we can aggregate attributes **together** with methods that operate on those attributes