EECS 1720
Building Interactive Systems

Lecture 2 :: Java Classes & Objects (review)

YORK UNIVERSITÉ UNIVERSITY

IMPORTANT: email protocol

- If you need to email me about anything related to the course, *please use "**EECS1720_W2023**" in the subject when emailing*

ZOOM LINKS

- Must use yu-passport login to access
  - login to eclass first, then access zoom links there, or
  - be logged into yu-passport then open the following zoom links

- **Lectures**
  - All lectures are streamed/recorded
  - Tuesday 2:30-4pm (in-person VH-A => will also stream)
  - Thursday 2:30-4pm (remote/zoom/hybrid => stream only **)

  - EECS1720 - W2023 - LECT
  - https://yorku.zoom.us/j/98286340595
  - Meeting ID: 982 8634 0595

*** Thursday lecture will be remote only (no in-person lecture in LAS-B)*

YORK UNIVERSITÉ UNIVERSITY

ZOOM LINKS

- **Office Hours**
  - Office hours will be held immediately following the lab session
  - Location: WSC 105, Wednesday's 2-3pm **

  - EECS1720 - W2023 – OFFICE HOURS
  - https://yorku.zoom.us/j/96128552701
  - Meeting ID: 961 2855 2701

*** Additional office hours can be made by appointment*

Brief Review
(EECS1710 concepts – high level)

see appendix for more detailed version

YORK UNIVERSITÉ UNIVERSITY

what you should know:

- Java Compilation, Bytecode and the Java VM (JVM)
- Know how to find your way around Eclipse

- Elements of a Java program
  - Data => need to store information (for processing)
  - Methods => need a recipe to process data (algorithm)

- Data Types
  - primitive vs reference types
  - variables vs. types
  - different types take up different amounts of space in memory
  - Basic Memory Model

YORK UNIVERSITÉ UNIVERSITY

what you should know …

- Primitive Types & Literals
  - types that hold data only (no methods)
  - variables of these types hold a value directly in memory
    - Numeric (int, long, float, double);
    - Char (char); Boolean (boolean);
    - Declaration vs. Assignment?
    - Literals (values assigned to a variable or used directly in an expression)
    - Magic Numbers? (arbitrary literals in expressions/statements)

YORK U
UNIVERSITÉ
UNIVERSITY

Basic Memory Model

- With the declaration

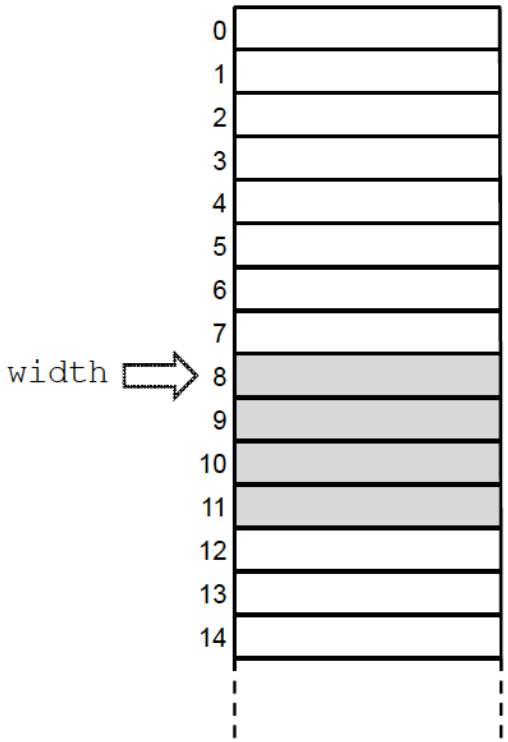
```
int width;
```

  the compiler will set aside a 4-byte (32-bit) block of memory (see right)

- The compiler has a symbol table, which will have an entry such as

Identifier	Type	Block Address
`width`	`int`	8

- *Note*: No initialization is involved; there is only an association of a name with an address.



what you should know …

- Expressions
  - Operators vs. Operands
  - Numeric Operators (+,-,/,*,%) -> output is a numeric type
  - Operator Precedence & Promotion/Demotion/Casting
  - Numeric Expressions

- Making Decisions (conditional logic)
  - Relational Operators (==, >, <, >=, <=, !=, !, ?)
  - Conjunctive Operators (&&, ||)
  - If, else
  - switch, case

YORK UNIVERSITÉ UNIVERSITY

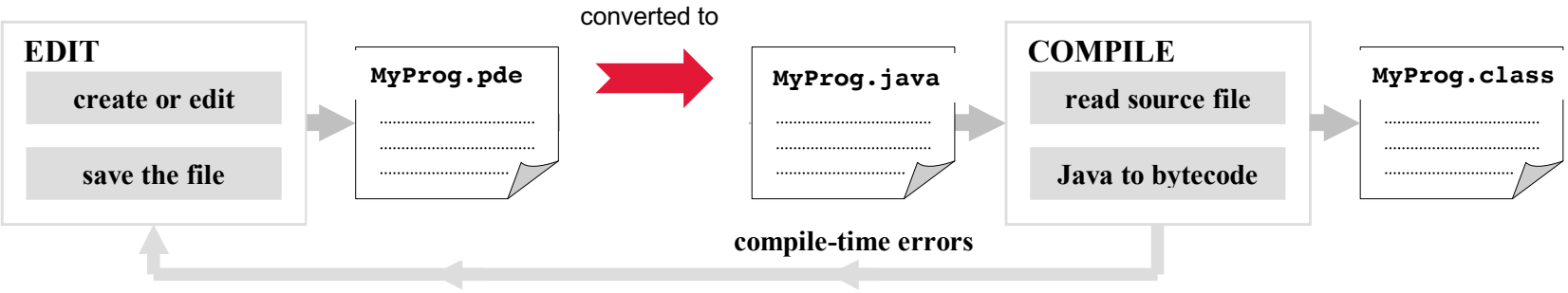
what you should know …

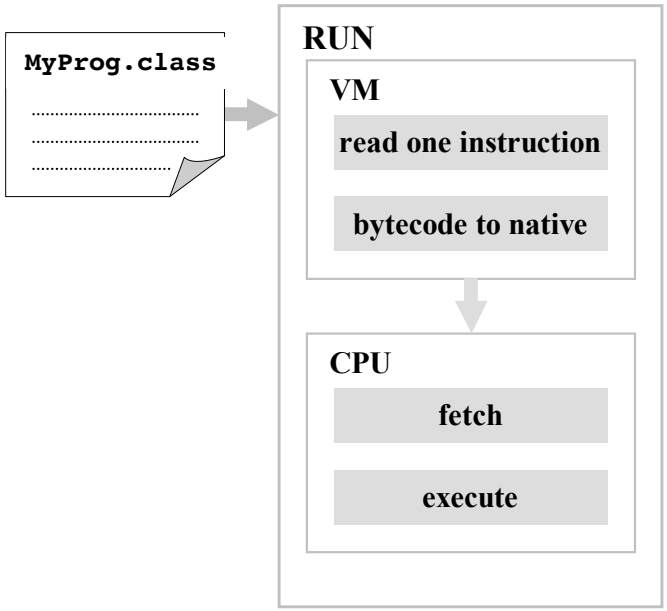
- Arrays & Loops
  - Numeric, Char and Boolean Arrays
  - for and while loops

- Reference Types & Objects
  - A type that contains both data + methods
  - Variables of these types hold a memory address (of an object location in memory after it is instantiated and assigned)
  - Because these variables only hold an address they are considered a reference to an object (hence called Reference Type)
    - Instantiating Objects
    - Object Death
    - the null reference (or null pointer)
    - Garbage Collection

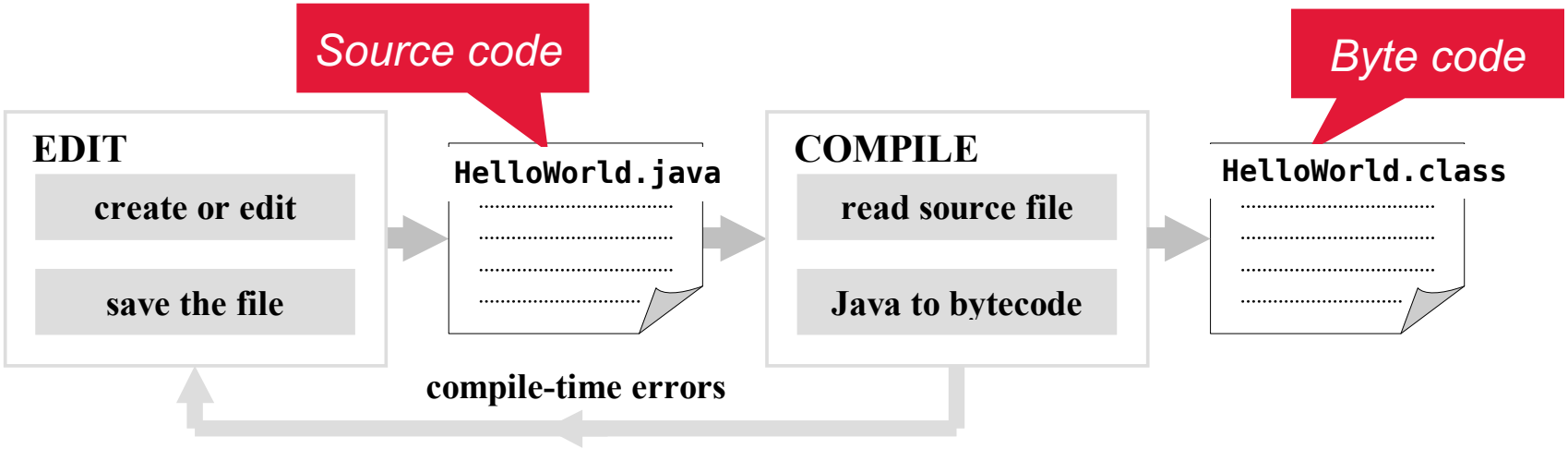
YORK UNIVERSITÉ UNIVERSITY

Previously… (processing converted to java)

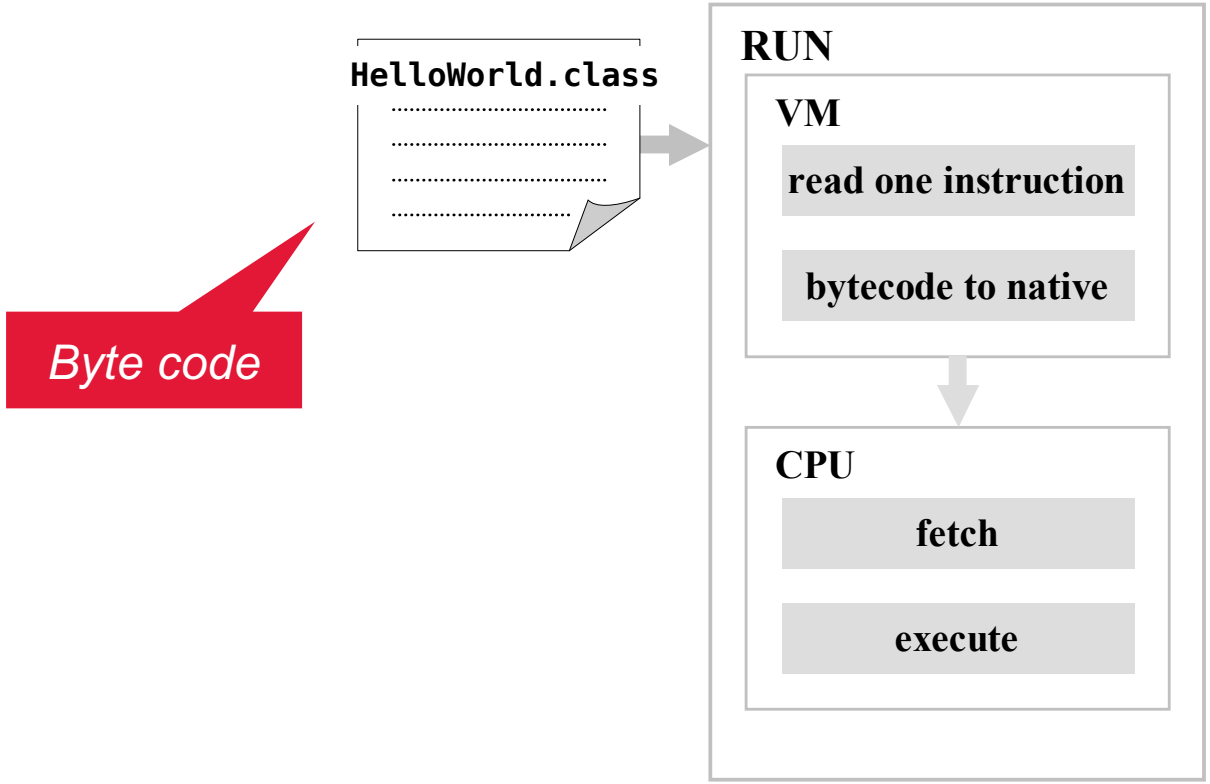




Now (we will skip the conversion step)



Java Virtual Machine JVM (review)

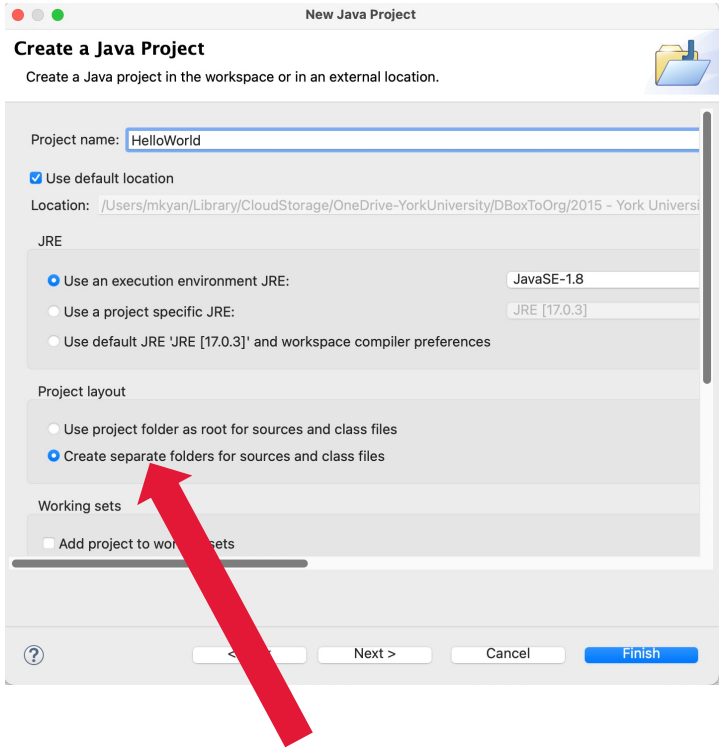


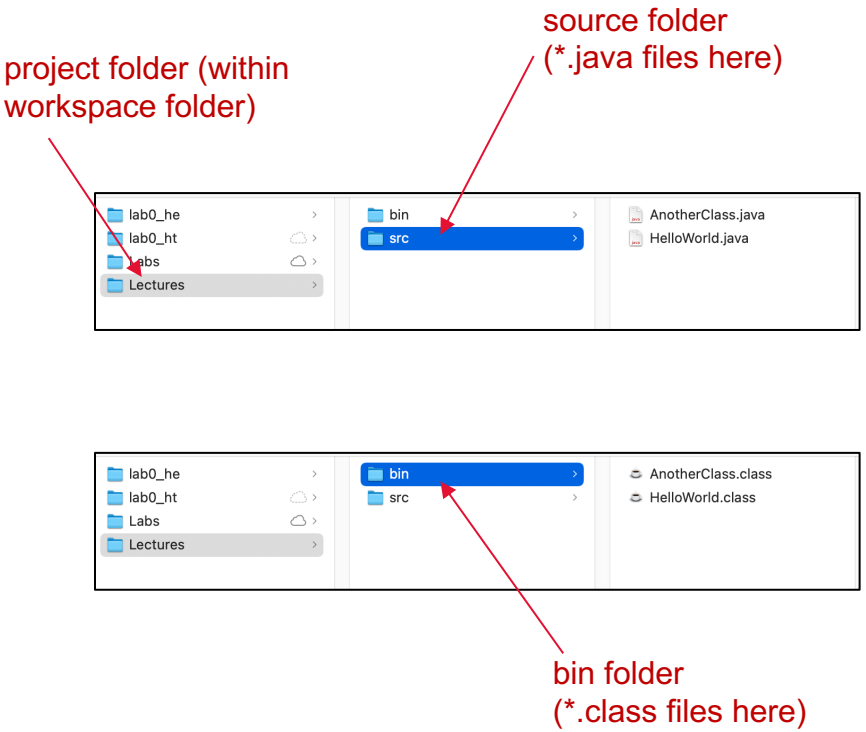
HelloWorld

```
public class HelloWorld {

    public static void main(String[] args) {

        // comment in line

        /* multiline comment
        */

        System.out.println("Hello EECS1720 World");

    }
}
```

Creating a Project in Eclipse

- File →New → Java Project





Use this option when creating projects from scratch
(then can run *.class files in terminal from bin folder)
e.g. “java HelloWorld” (see lab0)

Where are the files?

- Go to the workspace directory → project directory:
  - If you used option 2 (separate class and source files), there should be 2 subfolders: “bin” and “src”
  - “src” stores the source files (*.java files)
  - “bin” stores the binary (executable) files (*.class files)

YORK U
UNIVERSITÉ
UNIVERSITY

Also note (in this simple HelloWorld example)

- Methods are usually invoked through a classname or an object (so this is a little different from processing)… e.g. in this simple HelloWorld program, the println() method is invoked through the System class

- This is because methods are always associated with a *.java file, and hence a java class!
- So there are some differences in the way we access/invoke things in pure java.

YORK U
UNIVERSITÉ
UNIVERSITY

HelloWorld

What is this?

```
public class HelloWorld {

    public static void main(String[] args) {

        // comment in line

        /* multiline comment
        */

        System.out.println("Hello EECS1720 World");

    }
}
```

What is this?

Recall: Anatomy of an API

Packages	Details
	The Class section
	The Field section
Classes	The Constructor section
	The Method section

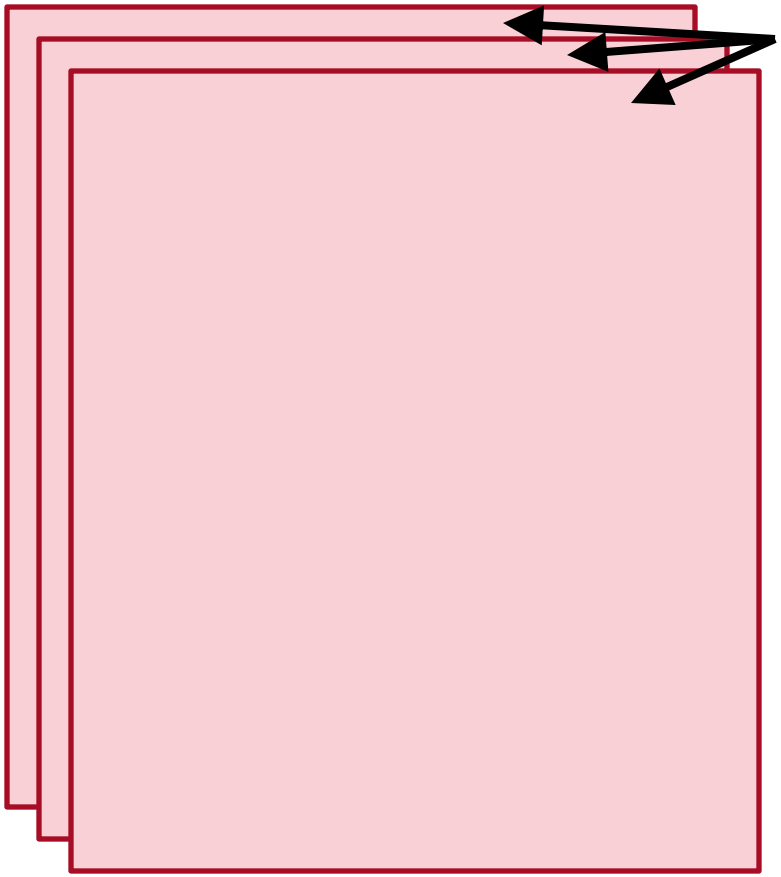
API ~ exposes public structure of a class
(how is this organized internally?)

YORK UNIVERSITÉ UNIVERSITY

Organization of a Java Program

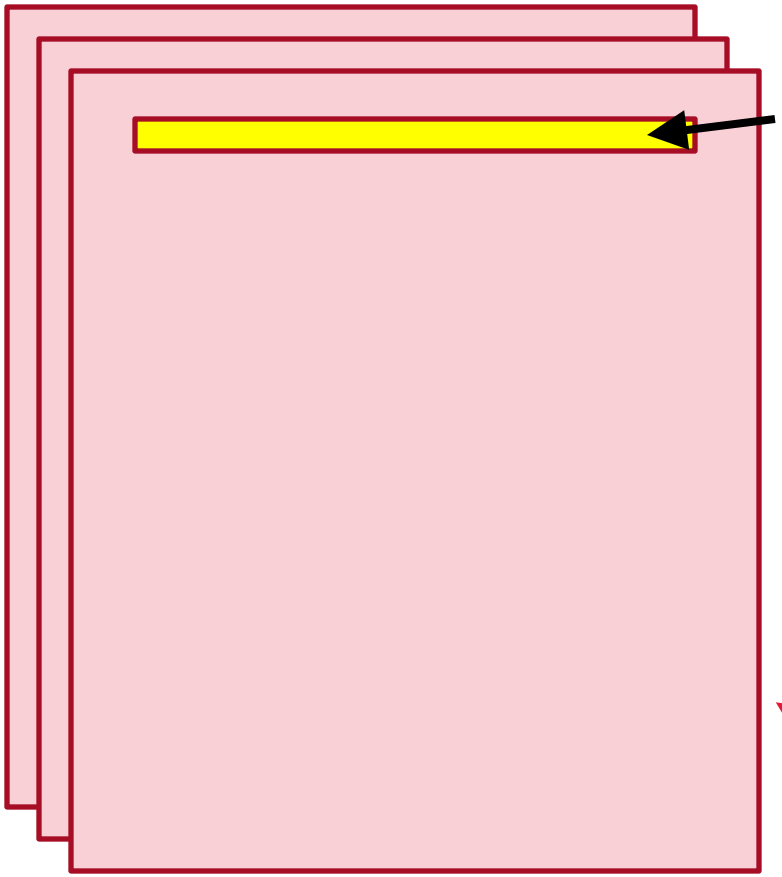
Packages, classes, fields, and methods

Organization of a Typical Java Program



- one or more files

Organization of a Typical Java Program

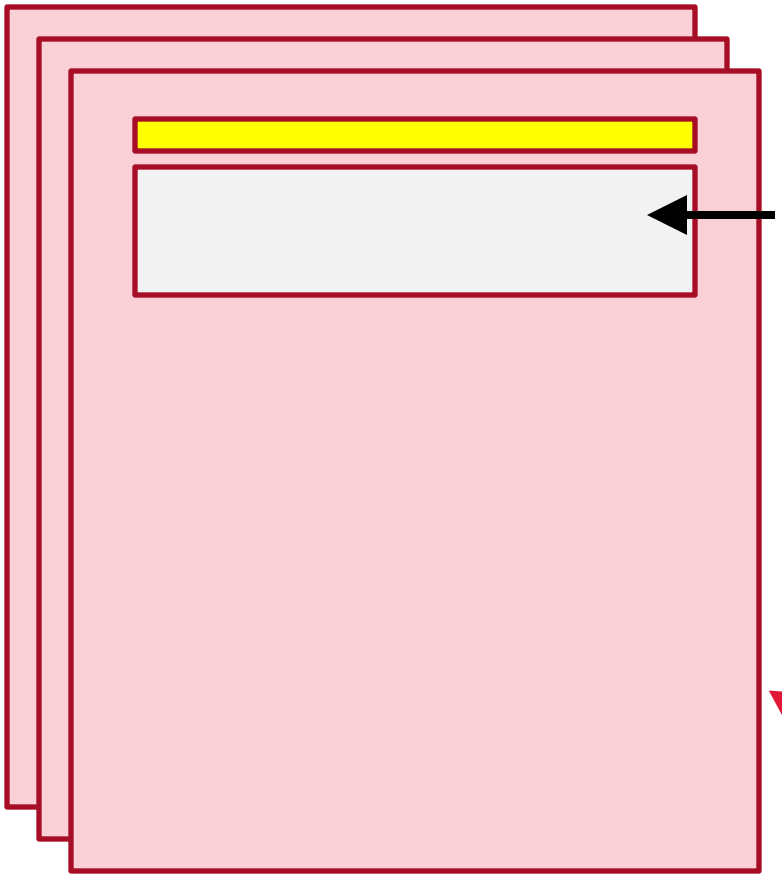


- one or more files
- zero or one package name

Example:

package => processing.sound

Organization of a Typical Java Program



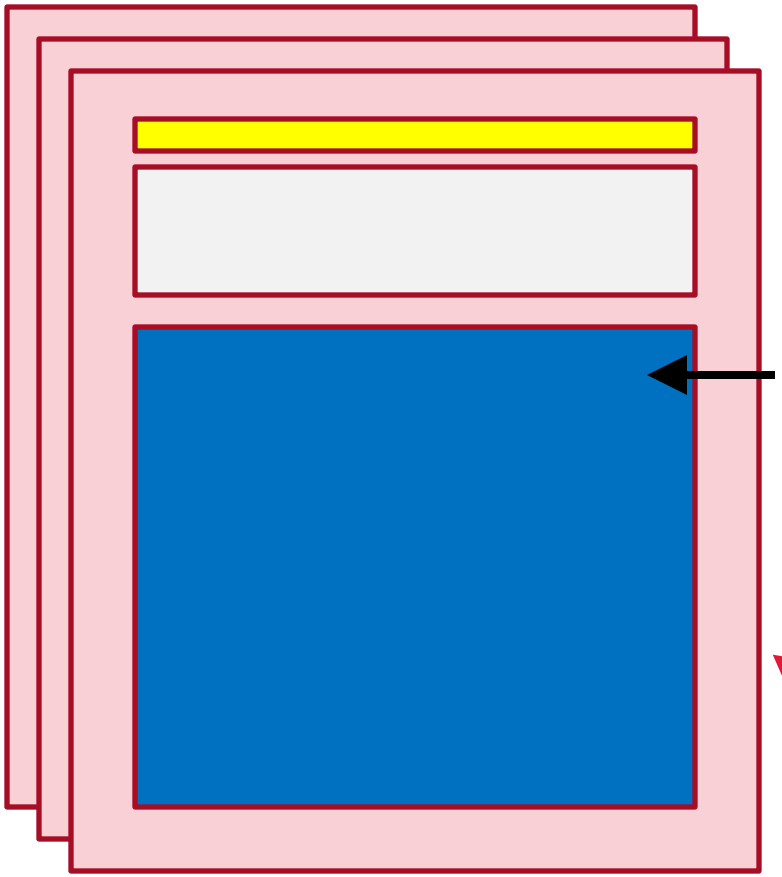
- one or more files
- zero or one package name
- zero or more import statements

Example:

import (all classes or single classes) =>

```
import processing.sound.*;  // imports all classes
import processing.sound.AudioSample;
```

Organization of a Typical Java Program



- one or more files
- zero or one package name
- zero or more import statements
- one class

Example:

processing.sound classes
(stored in separate files) =>

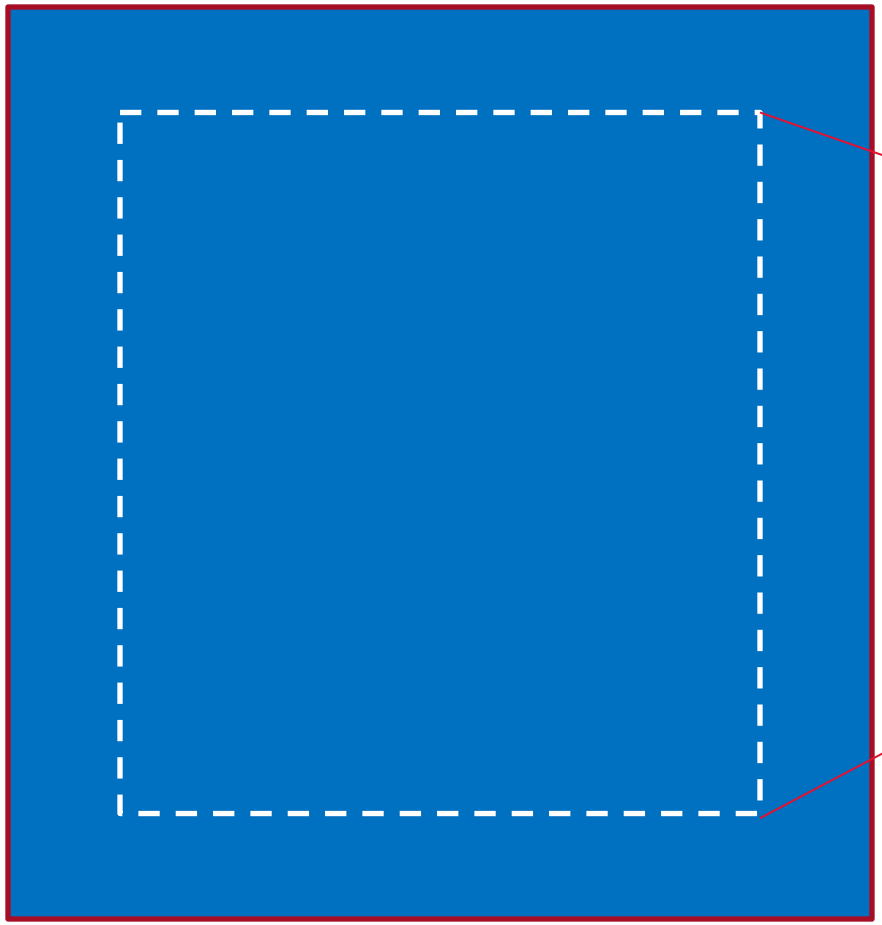
AudioSample (AudioSample.java)
SoundFile (SoundFile.java)
etc,

Recall: Java Class

- a class is a model of a thing or concept
- CLASS → "BLUEPRINT" for a *type*
  - fields (or attributes)
    - the structure of an object: its components and the information (data) contained by the object
  - methods
    - the behaviour of an object; what an object can do
    - Includes constructors (for creating objects) + other methods

YORK UNIVERSITÉ UNIVERSITY

Anatomy of a Class



The Class section

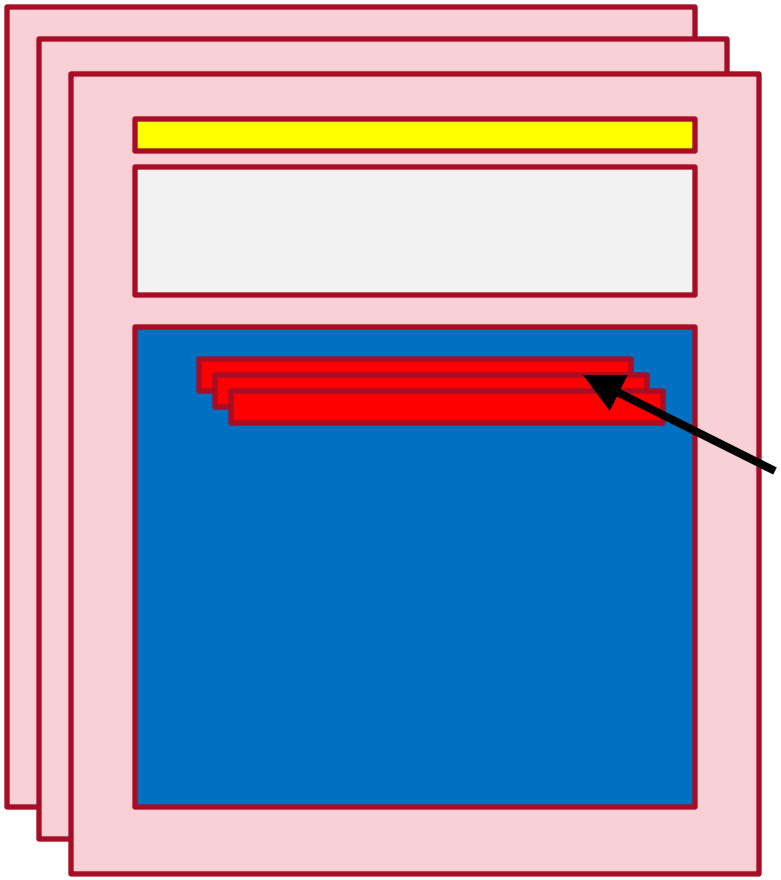
The Field section

The Constructor section

The Method section

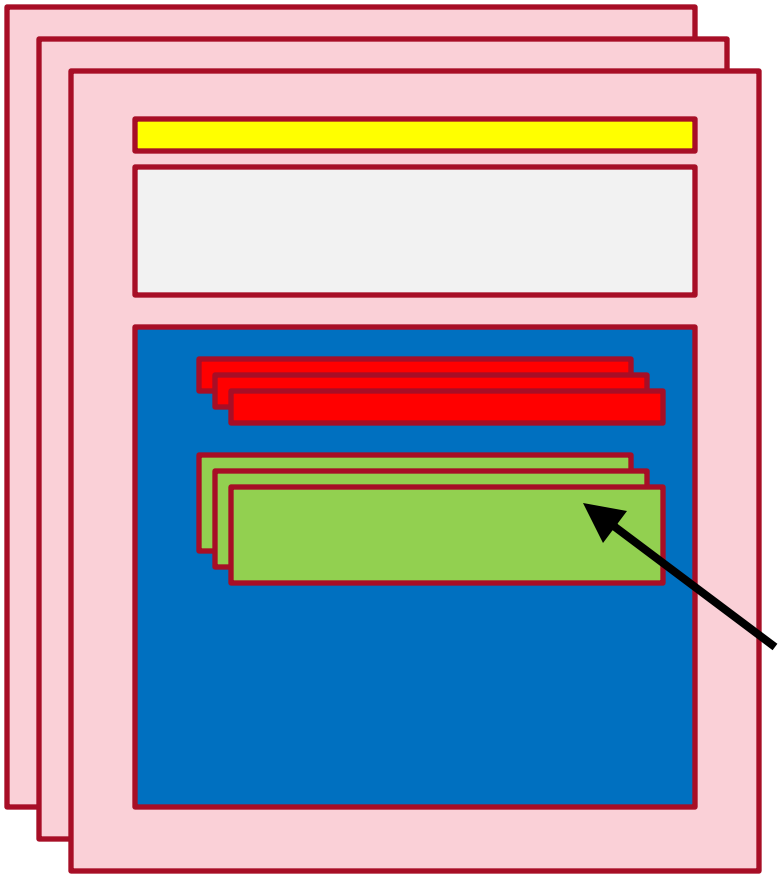
YORK UNIVERSITÉ UNIVERSITY

Organization of a Typical Java Program



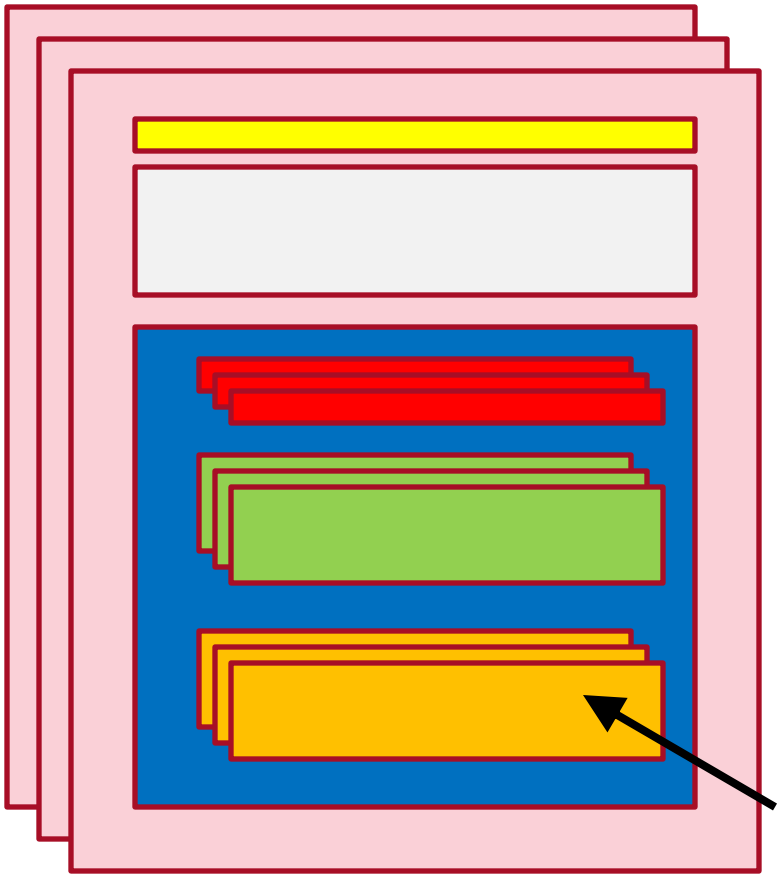
- one or more files
- zero or one package name
- zero or more import statements
- one class
- one or more fields (class variables)

Organization of a Typical Java Program



- one or more files
- zero or one package name
- zero or more import statements
- one class
- zero or more fields (class variables)
- zero or more more constructors

Organization of a Typical Java Program



- one or more files
- zero or one package name
- zero or more import statements
- one class
- zero or more fields (class variables)
- zero or more more constructors
- zero or more methods

The Java API (full reference)

https://docs.oracle.com/javase/8/docs/api/

YORK
UNIVERSITÉ
UNIVERSITY

Recall: Anatomy of an API

- General layout

Packages	Details
java.lang	The Class section
	The Field section
Classes	The Constructor section
Math String **System**	The Method section

YORK U
UNIVERSITÉ
UNIVERSITY

Everything in Java is a class!

- 2 types of classes
  - UTILITY (all methods and components are “static”)
  - DYNAMIC (has constructors, not static)
    - More on this next week

YORK U
UNIVERSITÉ
UNIVERSITY

Basic I/O (input/output)

- System: a class containing several key fields

java.lang

Class System

java.lang.Object
    java.lang.System

```
public final class System
extends Object
```

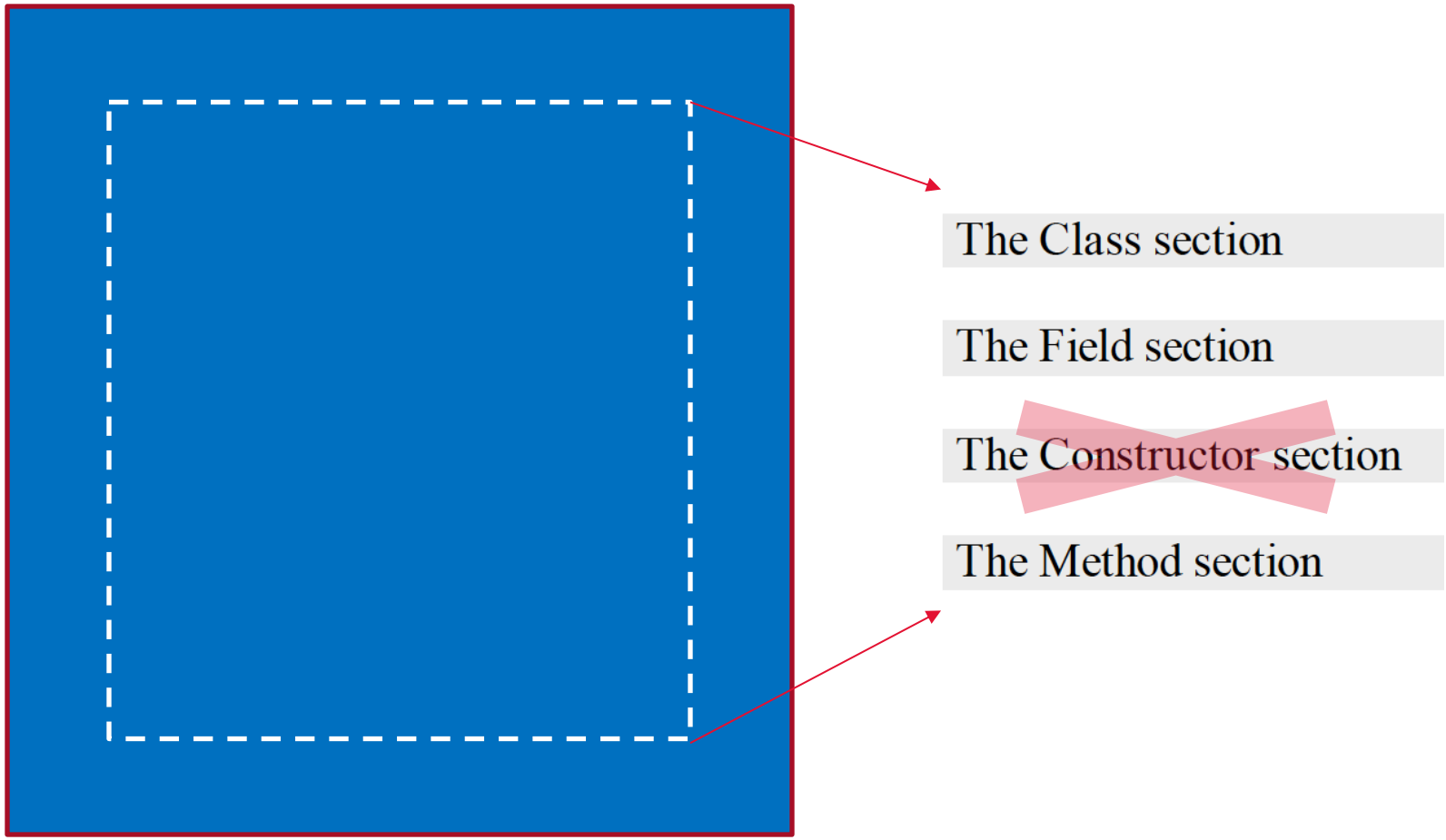
The System class contains several useful class fields and methods. It cannot be instantiated.

Among the facilities provided by the System class are standard input, standard output, and error output streams; access to externally defined properties and environment variables; a means of loading files and libraries; and a utility method for quickly copying a portion of an array.

cannot create instances of System:
(i.e. cannot make System objects)

YORK UNIVERSITÉ UNIVERSITY

System Class → "UTILITY CLASS"



utility classes (like "Math" only have fields/methods)

Basic I/O (input/output)

- System: a class containing several key fields

Field Summary

Fields

Modifier and Type	Field and Description
static **PrintStream**	**err** The "standard" error output stream.
static **InputStream**	**in** The "standard" input stream.
static **PrintStream**	**out** The "standard" output stream.

  - ”err” and “out” are <u>references</u> to a **`PrintStream`** type
  - “in” is a <u>reference</u> to an **`InputStream`** type

- InputStream objects are connected to input sources
  - the one in System class is connected to the keyboard device (considered the standard input device)
    - Generally we connect a *Scanner* object to this so that we can use simple methods to get individual characters/numbers/strings as they are typed into the console

- PrintStream objects are connected to output sources
  - err is an output device that captures errors from programs (we can ignore this for now)
  - out is connected to the screen (considered the standard output device)
    - PrintStream objects include the print methods!

```
System.out.println("HelloWorld");
System.out.printf("%s", "HelloWorld");
```

YORK UNIVERSITÉ UNIVERSITY

System class

- System is a class that we have delegated to…
  - we use this class to invoke print(), println(), printf() methods

Field Summary

Fields

Modifier and Type	Field and Description
static `PrintStream`	`err` The "standard" error output stream.
static `InputStream`	`in` The "standard" input stream.
static `PrintStream`	`out` The "standard" output stream.

Method Summary

All Methods | **Static Methods** | **Concrete Methods** | **Deprecated Methods**

Modifier and Type	Method and Description
static void	`arraycopy(Object src, int srcPos, Object dest, int destPos, int length)` Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array.
static `String`	`clearProperty(String key)` Removes the system property indicated by the specified key.
static `Console`	`console()` Returns the unique `Console` object associated with the current Java virtual machine, if any.
static long	`currentTimeMillis()` Returns the current time in milliseconds.
static void	`exit(int status)` Terminates the currently running Java Virtual Machine.

System.out // accesses variable "out"
// from System class

System.out // "out" is a variable of type
// "PrintStream"

System.out → a PrintStream object

Constructors

Constructor and Description
`PrintStream(File file)` Creates a new print stream, without automatic line flushing, with
`PrintStream(File file, String csn)` Creates a new print stream, without automatic line flushing, with
`PrintStream(OutputStream out)` Creates a new print stream.
`PrintStream(OutputStream out, boolean autoFlush)` Creates a new print stream.
`PrintStream(OutputStream out, boolean autoFlush, String` Creates a new print stream.
`PrintStream(String fileName)` Creates a new print stream, without automatic line flushing, with
`PrintStream(String fileName, String csn)` Creates a new print stream, without automatic line flushing, with

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
`void`	`print(double d)` Prints a double-precision floating-point number.
`void`	`print(float f)` Prints a floating-point number.
`void`	`print(int i)` Prints an integer.
`void`	`print(long l)` Prints a long integer.
`void`	`print(Object obj)` Prints an object.
`void`	`print(String s)` Prints a string.
`PrintStream`	`printf(Locale l, String format, Object... args)` A convenience method to write a formatted string to this output stream using the specified format string and arguments.
`PrintStream`	`printf(String format, Object... args)` A convenience method to write a formatted string to this output stream using the specified format string and arguments.
`void`	`println()` Terminates the current line by writing the line separator string.
`void`	`println(boolean x)` Prints a boolean and then terminate the line.
`void`	`println(char x)` Prints a character and then terminate the line.

System

- System.out → connected to screen (console)

  screen is a PrintStream object

- System.in → connected to keyboard

  keyboard is an InputStream object

YORK UNIVERSITÉ UNIVERSITY

Takeaways

- Understanding how classes are organized (both individual class files, and collections of them)
- Understand how java documentation (API) is organized (into packages, classes, and the breakdown of a class)
- Understand that there are 2 types of class
  - Utility
    - Cannot create objects from these… used to collate related constants & methods usually
    - Does **not** have usable constructors
    - E.g. Math, System classes
  - Dynamic
    - Can create objects from these (so some of the fields at least will be variable and hold different values for different objects)
    - Has usable constructors (you can see in the API if it does or not)
    - E.g. String, (*or from processing libs: PVector, PImage, AudioSample, etc*)

YORK UNIVERSITÉ UNIVERSITY

For next week

- Accept invite to join discord server for 1720 (posted on 1710 discord and on eclass)

- Review lab0
- Review appendix of this slide deck (what you should know from eecs1710)
    - Section is setup in discord to ask questions on any concepts/topics you need clarification on from last course 1710

YORK UNIVERSITÉ UNIVERSITY

APPENDIX

Quick overview of EECS1710 concepts (Java Specific Version)

YORK UNIVERSITÉ UNIVERSITY

what you should know:

- Java Compilation, Bytecode and the Java VM (JVM)
- Know how to find your way around Eclipse

- Elements of a Java program
  - Data => need to store information (for processing)
  - Methods => need a recipe to process data (algorithm)

- Data Types
  - primitive vs reference types
  - variables vs. types
  - different types take up different amounts of space in memory
  - Basic Memory Model

YORK UNIVERSITÉ UNIVERSITY

Relative sizes of primitive data types..

PRIMITIVE TYPES			Type	Size (bytes)	Approximate Range min	max	S.D.
NUMBER	INTEGER	SIGNED	byte	1	-128	+127	-
			short	2	-32,768	+32,767	-
			int	4	-2×10^{9}	$+2\times10^{9}$	-
			long	8	-9×10^{18}	$+9\times10^{18}$	-
		UNSIGNED	char	2	0	65,535	-
	REAL	SINGLE	float	4	$+3.4\times10^{38}$	$+3.4\times10^{38}$	7
		DOUBLE	double	8	-1.7×10^{308}	$+1.7\times10^{308}$	15
BOOLEAN			boolean	1	true/false		-

YORK UNIVERSITÉ UNIVERSITY

Basic Memory Model

- With the declaration

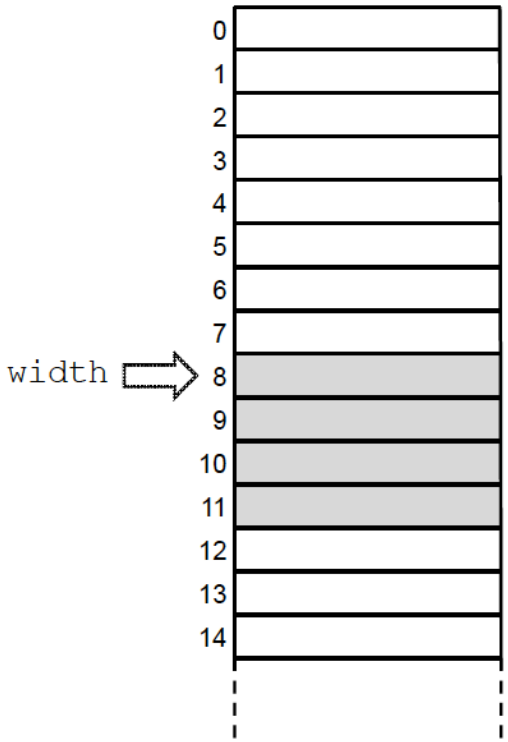
```
int width;
```

  the compiler will set aside a 4-byte (32-bit) block of memory (see right)

- The compiler has a symbol table, which will have an entry such as

Identifier	Type	Block Address
`width`	`int`	8

- *Note*: No initialization is involved; there is only an association of a name with an address.



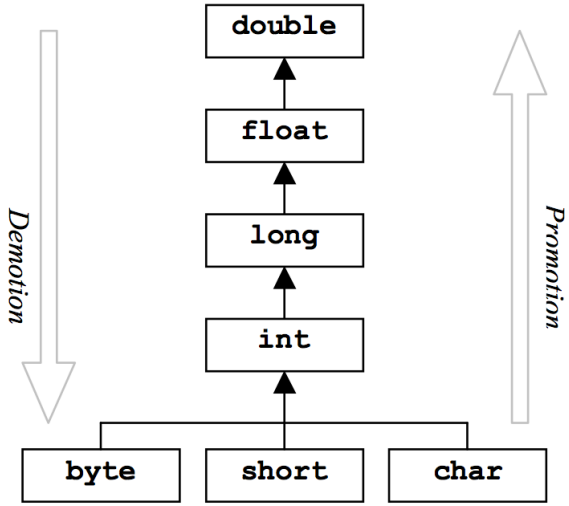
what you should know …

- Primitive Types & Literals
  - types that hold data only (no methods)
  - variables of these types hold a value directly in memory
    - Numeric (int, long, float, double);
    - Char (char); Boolean (boolean);
    - Declaration vs. Assignment?
    - Literals (values assigned to a variable or used directly in an expression)
    - Magic Numbers? (arbitrary literals in expressions/statements)

YORK U
UNIVERSITÉ
UNIVERSITY

Primitive types; Promotion/Demotion

PRIMITIVE TYPES			Type	Size (bytes)	Approximate Range min	max	S.D.
NUMBER	INTEGER	SIGNED	byte	1	-128	+127	-
			short	2	-32,768	+32,767	-
			int	4	-2×10^{9}	$+2\times10^{9}$	-
			long	8	-9×10^{18}	$+9\times10^{18}$	-
		UNSIGNED	char	2	0	65,535	-
	REAL	SINGLE	float	4	$+3.4\times10^{38}$	$+3.4\times10^{38}$	7
		DOUBLE	double	8	-1.7×10^{308}	$+1.7\times10^{308}$	15
BOOLEAN			boolean	1	true/false		-



what you should know …

- Expressions
  - Operators vs. Operands
  - Numeric Operators (+,-,/,*,%) -> output is a numeric type
  - Operator Precedence & Promotion/Demotion/Casting
  - Numeric Expressions

- Making Decisions (conditional logic)
  - Relational Operators (==, >, <, >=, <=, !=, !, ?)
  - Conjunctive Operators (&&, ||)
  - If, else
  - switch, case

YORK UNIVERSITÉ UNIVERSITY

Numeric Operators

Precedence	Operator	Kind	Syntax	Operation
-5 ➔	+	infix	x + y	add y to x
	-	infix	x - y	subtract y from x
-4 ➔	*	infix	x * y	multiply x by y
	/	infix	x / y	divide x by y
	%	infix	x % y	remainder of x / y
-2 ⬅	+	prefix	+x	identity
	-	prefix	-x	negate x
	++	prefix	++x	x = x + 1; result = x
	--	prefix	--x	x = x - 1; result = x
-1 ➔	++	postfix	x++	result = x; x = x + 1
	--	postfix	x--	result = x; x = x - 1

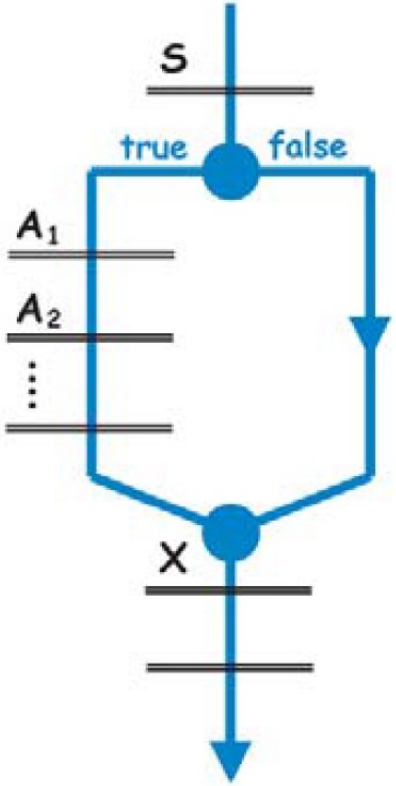
YORK UNIVERSITÉ UNIVERSITY

Relational Operators

Precedence	Operator	Operands	Syntax	true if
-7 ➔	<	numeric	x < y	x is less than y
	<=	numeric	x <= y	x is less than or equal to y
	>	numeric	x > y	x is greater than y
	>=	numeric	x >= y	x is greater than or equal to y
	instanceof	x instanceof C is true if object reference x points to an instance of class C or a subclass of C		
-8 ➔	==	any type	x == y	x is equal to y
	!=	any type	x != y	x is not equal to y

YORK U
UNIVERSITÉ
UNIVERSITY

IF-ELSE

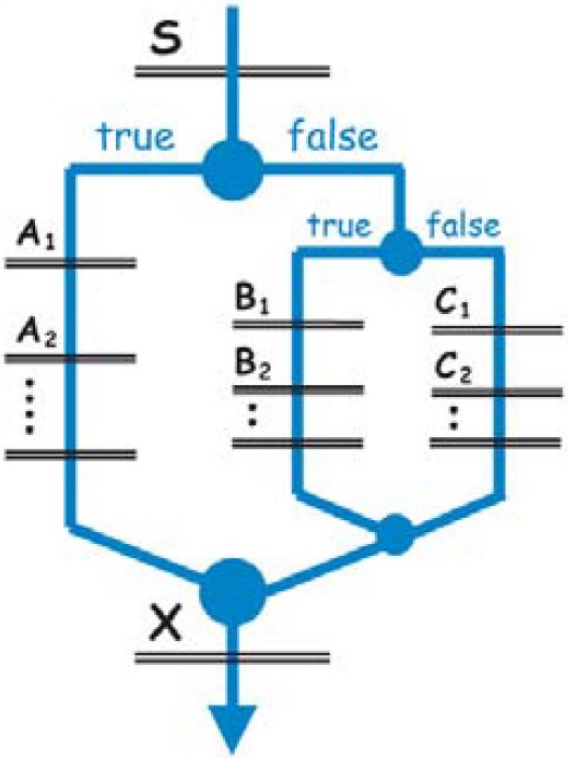



```
Statement-S
if (condition-1)
{   Statement-A1
    Statement-A2
    ...
} else
{   Statement-B1
    Statement-B2
    ...
}

Statement-X
```

execute if condition-1 true

execute if condition-1 false

Multiway (nested) branching

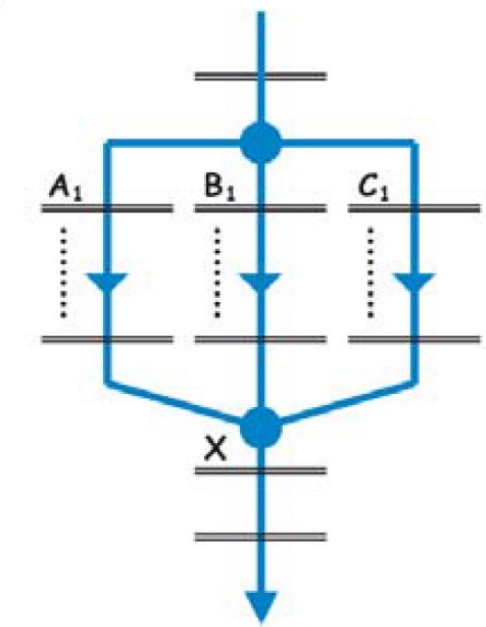



```
Statement-S
if (condition-1)
{   Statement-A1
    Statement-A2
    ...
} else if (condition-2)
{   Statement-B1
    Statement-B2
    ...
} else
{   Statement-C1
    Statement-C2
    ...
}
Statement-X
```

More Advanced Branching

- SWITCH STATEMENTS
- (not required, but can make code neater! Especially when there are a large number of branches to consider for a given outcome)

- Usually used when we branch on several alternative VALUES of a given variable/expression



switch vs if

```
void setup() {
  showMenuOptions();
}

void draw() {
}

void keyPressed() {

  if (key == '0') shapeToDraw=0;
  if (key == '1') shapeToDraw=1;
  if (key == '2') shapeToDraw=2;
  if (key == '3') shapeToDraw=3;



  println("You chose to draw a "
                 + whichShape()   );

}
```

```
void setup() {
  showMenuOptions();
}

void draw() {
}

void keyPressed() {

  switch (key) {
  case '0': shapeToDraw=0; break;
  case '1': shapeToDraw=1; break;
  case '2': shapeToDraw=2; break;
  case '3': shapeToDraw=3; break;
  }

  println("You chose to draw a "
                 + whichShape()   );

}
```

YORK U
UNIVERSITÉ
UNIVERSITY

what you should know …

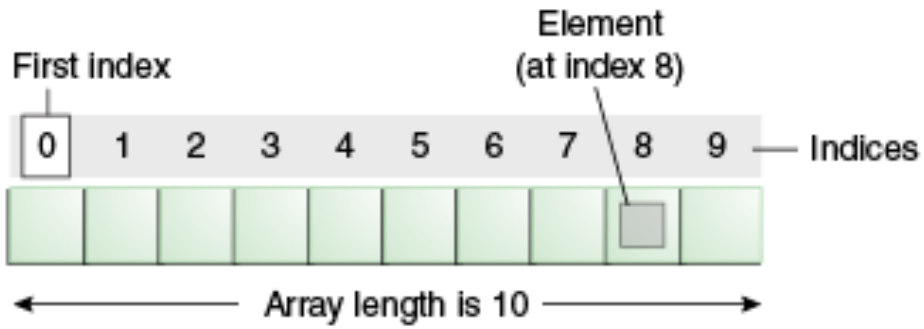
- Arrays & Loops
  - Numeric, Char and Boolean Arrays
  - for and while loops

- Reference Types & Objects
  - A type that contains both data + methods
  - Variables of these types hold a memory address (of an object location in memory after it is instantiated and assigned)
  - Because these variables only hold an address they are considered a reference to an object (hence called Reference Type)
    - Instantiating Objects
    - Object Death
    - the null reference (or null pointer)
    - Garbage Collection

YORK UNIVERSITÉ UNIVERSITY

Arrays

- the values in an array are called elements
- the elements can be accessed using a zero-based index




```
// set all elements
// to equal 100.0

collection[0] = 100.0;
collection[1] = 100.0;
collection[2] = 100.0;
collection[3] = 100.0;
collection[4] = 100.0;
collection[5] = 100.0;
collection[6] = 100.0;
collection[7] = 100.0;
collection[8] = 100.0;
collection[9] = 100.0;
```

```
int n = collection.length;
// all arrays automatically have a public field length = size of array
```

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

Recap: Arrays of primitive types

```
byte[]        anArrayOfBytes;
short[]       anArrayOfShorts;
long[]        anArrayOfLongs;
float[]       anArrayOfFloats;
double[]      anArrayOfDoubles;
boolean[]     anArrayOfBooleans;
char[]        anArrayOfChars;
```

declaration

YORK U
UNIVERSITÉ
UNIVERSITY

Recap: Arrays of primitive types

```
byte[]        anArrayOfBytes;
short[]       anArrayOfShorts;
long[]        anArrayOfLongs;
float[]       anArrayOfFloats;
double[]      myArray = new double[100];
boolean[]     anArrayOfBooleans;
char[]        anArrayOfChars;
```

declaration + creation

YORK UNIVERSITÉ UNIVERSITY

Recap: Arrays of primitive types

```
byte[]          anArrayOfBytes;
short[]         anArrayOfShorts;
long[]          anArrayOfLongs;
float[]         anArrayOfFloats;
double[] myArray = { 1.0, 2.0, …
                        … , 100.0 };
boolean[]       anArrayOfBooleans;
char[]          anArrayOfChars;
```

declaration + creation + initialization

YORK UNIVERSITÉ UNIVERSITY

Recap: Arrays of primitive types

```
byte[]          anArrayOfBytes;
short[]         anArrayOfShorts;
long[]          anArrayOfLongs;
float[]         anArrayOfFloats;
double[] myArray = { 1.0, 2.0, …
                        … , 100.0 };
boolean[]       anArrayOfBooleans;
char[]          anArrayOfChars;

int idx = 1;          // set index variable "idx"
myArray[0]            // first element of array
myArray[1]            // second element
myArray[idx]          // (idx-1)th element
myArray[idx++]        // next element (idx+1)
myArray[-1] ??        // ERROR (causes an exception)
```

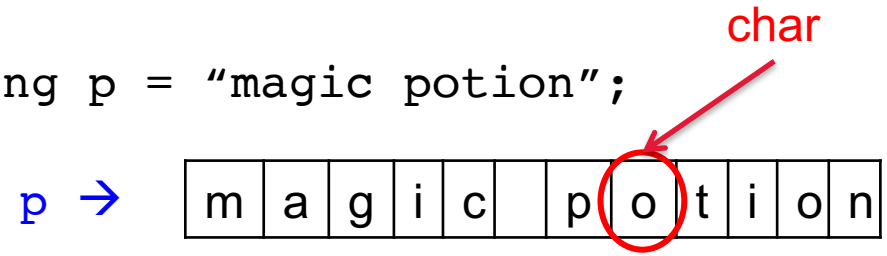
declaration + creation + initialization

YORK UNIVERSITÉ UNIVERSITY

Recap: String stores array of chars

```
byte[]          anArrayOfBytes;
short[]         anArrayOfShorts;
long[]          anArrayOfLongs;
float[]         anArrayOfFloats;
double[]        anArrayOfDoubles;
boolean[]       anArrayOfBooleans;
char[]          anArrayOfChars;
```

```
String p = "magic potion";
```



Recap: String stores array of chars + methods

```
byte[]          anArrayOfBytes;
short[]         anArrayOfShorts;
long[]          anArrayOfLongs;
float[]         anArrayOfFloats;
double[]        anArrayOfDoubles;
boolean[]       anArrayOfBooleans;
char[]          anArrayOfChars;
```

A String is not a char[ ] (it is its own reference type)

char

```
String p = "magic potion";
```

p →	m	a	g	i	c		p	o	t	i	o	n

p.substring(3,8) →	i	c		p	o

p.charAt(9) → 'i'

p.length() → 12

YORK UNIVERSITÉ UNIVERSITY

We have seen arrays of primitive types…
Can we have arrays of non-primitive (reference) types?

- Yes

- Examples:

```
String[] anArrayOfStrings;
PVector[] anArrayOfVectors;
AudioSample[] anArrayOfAudioSamples;
PImage[] anArrayOfImages;

anArrayOfStrings = new String[5];
anArrayOfVectors = new PVector[10];

anArrayOfStrings[0] = new String("some sentence"); //first
anArrayOfVectors[9] = new PVector(1.5,3.8);        // last
```

declaration

creation

initialization (partial)

YORK UNIVERSITÉ UNIVERSITY

Arrays

- Very useful for storing/managing ***state*** (data) in a program
  - Examples:
    - a set of number guesses (number guessing game)
    - the letters in a hangman game
    - a set of shapes/lines drawn to the screen
    - inventory (items collected in a RPG = role playing game)
    - where you are currently on a game board
    - storing moves made by a PShape object (for undo)

YORK U
UNIVERSITÉ
UNIVERSITY

API / Objects

- Non-Primitive (Reference) Types (objects)
- API of a class (fields, constructors, methods)
  - Fields
    - variables/constants defined outside a method but within a class block - have scope for all the methods in a class
  - Method signatures
    - Signature is the method name + list of types used (in order) as parameters for the method, eg:
  - Constructors
    - Special version of method, used with “new” when creating an object only, has same name as class, no return type, and is only used to initialize an instance of an object when created at runtime.
    - Obeys same signature rule as methods (above)
  - How do we invoke methods/access fields?
    - Through the class name (if method is static), or a variable of that type (if the method is not static and we can make objects from the class – i.e. it has constructors)
    - Passing arguments to methods?

YORK U
UNIVERSITÉ
UNIVERSITY

ArrayLists

- Like arrays, but a lot more convenient!
  - keeps elements in a sequence (like arrays) but can grow
  - includes methods to add, sort, find max, reverse, shuffle etc…

Don't exist in pure java, everything uses ArrayList

- *IntList → dynamic/resizable array of ints*
- *FloatList → dynamic/resizable array of floats*
- *StringList → dynamic/resizable array of Strings*

- ArrayList → use if you want a list of any type of object
  (e.g. like an ArrayList of PVector)

YORK UNIVERSITÉ UNIVERSITY

e.g. ArrayList< > of PImage types..

- ArrayList<PImage>
  - ArrayList of a generic type (have to specify)
  - `ArrayList<PImage> myAList = new ArrayList<PImage>();`
    - `myAList.add(new Pimage( … ) );`

no argument **constructor** for ArrayList<PImage>

** ArrayLists are reference types (with **methods**) & can grow/shrink (unlike arrays which are fixed in size when created)

YORK UNIVERSITÉ UNIVERSITY

What happens in MEMORY? (primitive vs. reference types)

- Variables of primitive types hold **values directly**
- Variables of reference types hold **addresses** of objects (not the objects themselves)

- A class may be **instantiated** using the ***new*** operator along with a ***constructor***
  - The object exists at **runtime only** (not compile time), and is allocated an available slot in memory at runtime
  - If a reference is never assigned an instantiated object, then the program will cause a compile time error

YORK UNIVERSITÉ UNIVERSITY

Primitive types (in memory)

YORK
UNIVERSITÉ
UNIVERSITY

Memory model

```
import java.lang.System;

public class Area
{
      public static void main(String[] args)
      {
            int width = 8;
            int height = 3;
            int area = width * height;
            System.out.println(area);
      }
}
```

memory block

address	value	
…		
300		← args
…		
320		← width
…		
324		← height
…		
328		← area
…		

symbol table

Identifier	Type	Block Address
args		300
width	int	320
height	int	324
area	int	328

starting addresses for each variable are 4 bytes apart

YORK UNIVERSITÉ UNIVERSITY

Memory model

```
import java.lang.System;

public class Area
{
      public static void main(String[] args)
      {
            int width = 8;
            int height = 3;
            int area = width * height;
            System.out.println(area);
      }
}
```

memory block

address	value
…	
300	
…	
320	
…	
324	
…	
328	
…	

← height

← area

What if width, height and area were all of type **double**?

		Block Address
		300
		320
		324
area	int	328

YORK UNIVERSITÉ UNIVERSITY

Memory model

```
import java.lang.System;

public class Area
{
      public static void main(String[] args)
      {
            int width = 8;
            int height = 3;
            int area = width * height;
            System.out.println(area);
      }
}
```

memory block

address	value	
…		
300		← args
…		
320		← width
…		
328		← height
…		
336		← area
…		

symbol table

Identifier	Type	Block Address
args		300
width	**double**	**320**
height	**double**	**328**
area	**double**	**336**

double takes up 8 bytes
starting addresses now 8 bytes apart

YORK UNIVERSITÉ UNIVERSITY

Objects (in memory) ??

YORK
UNIVERSITÉ
UNIVERSITY

Birth of an object (happens at runtime)

Four steps

- Locate the class

  `(import PVector) => done automatically in Processing`

- Declare a reference

  ```
  PVector v1;
  ```

- Instantiate the class

  ```
  new PVector(150, 150);
  ```

- Assign the reference

  ```
  v1 = new PVector(150, 150);
  ```

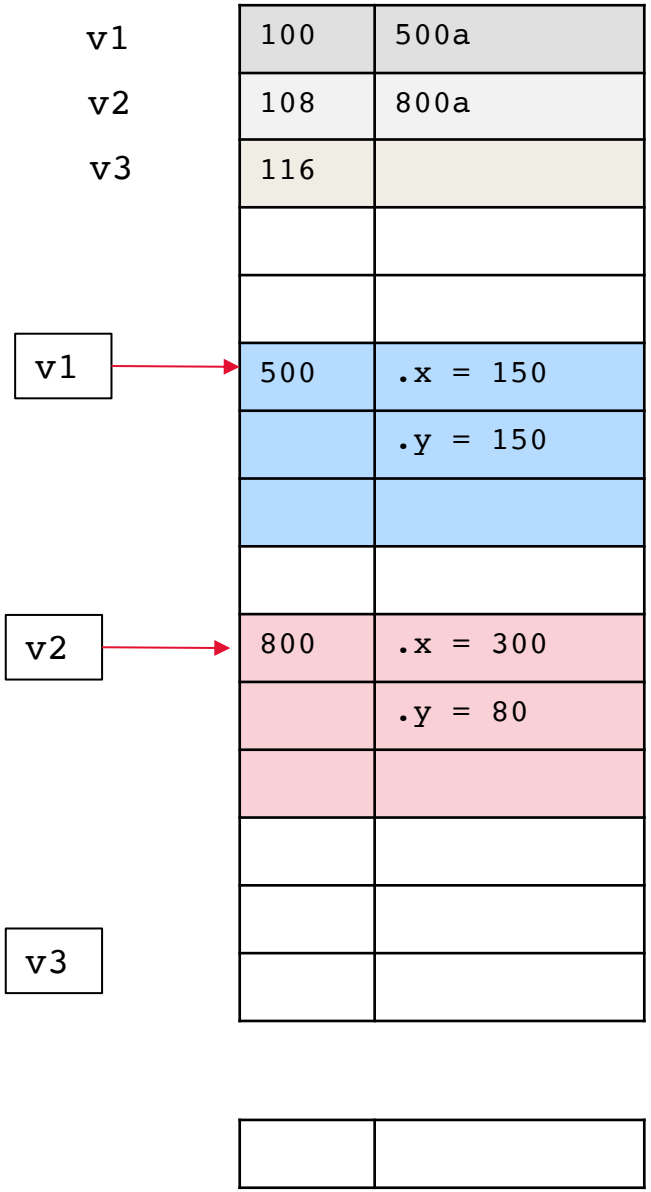
Lets assume we are using PVector objects..

Declaring a PVector variable only creates a reference (not the object itself)

Usually combined

PVector objects have several fields: .x, .y, .z (2D => .z==0)

```
final int WHITE = color(255,255,255);
final int GREEN = color(0,255,0);
final int PURPLE = color(255,0,255);

PVector v1;
PVector v2;
PVector v3;

void setup() {

  size(600, 600);
  background(0,0,0);

  v1 = new PVector(150, 150);
  v2 = new PVector(300, 80);



}
```



Aliases

- Many variables can point at the same object:

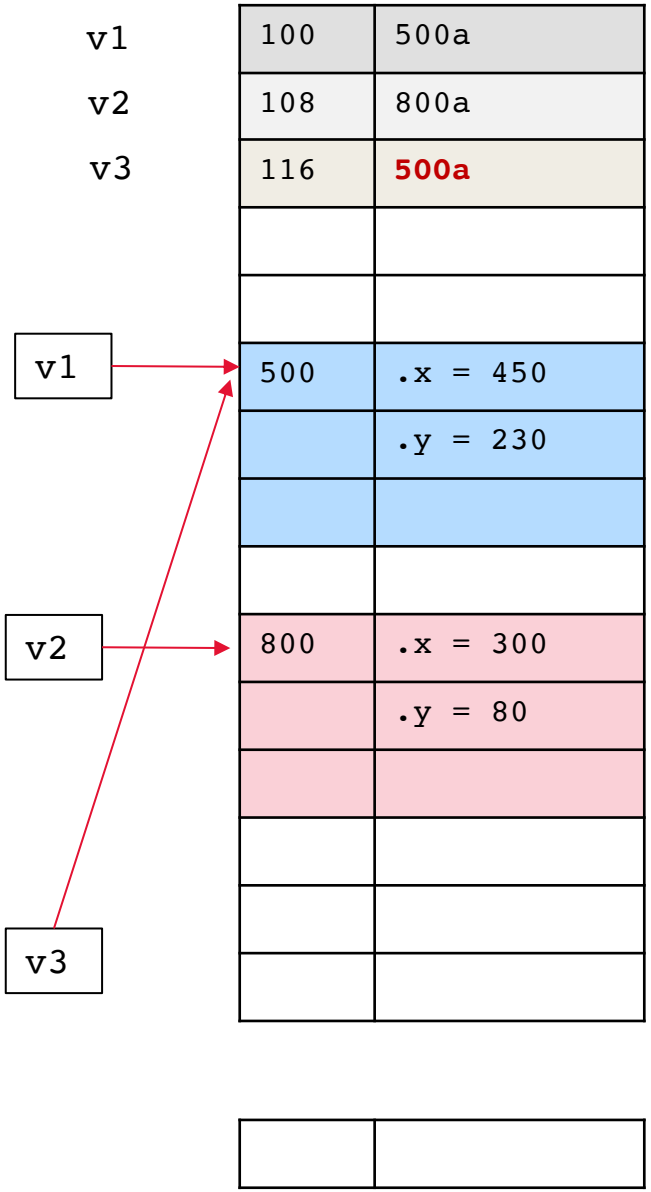
```
v1 = new PVector(150, 150);
v2 = new PVector(300, 80);
v1.add(v2);
PVector v3 = v1;            // v3 and v1 both
                            // reference same object
                            // v1 & v3 are ALIASES
```

- If the object is changed through `v1` the change will be seen by `v3`

*** similarly if v3 changed, v1 will see v3*

YORK U
UNIVERSITÉ
UNIVERSITY

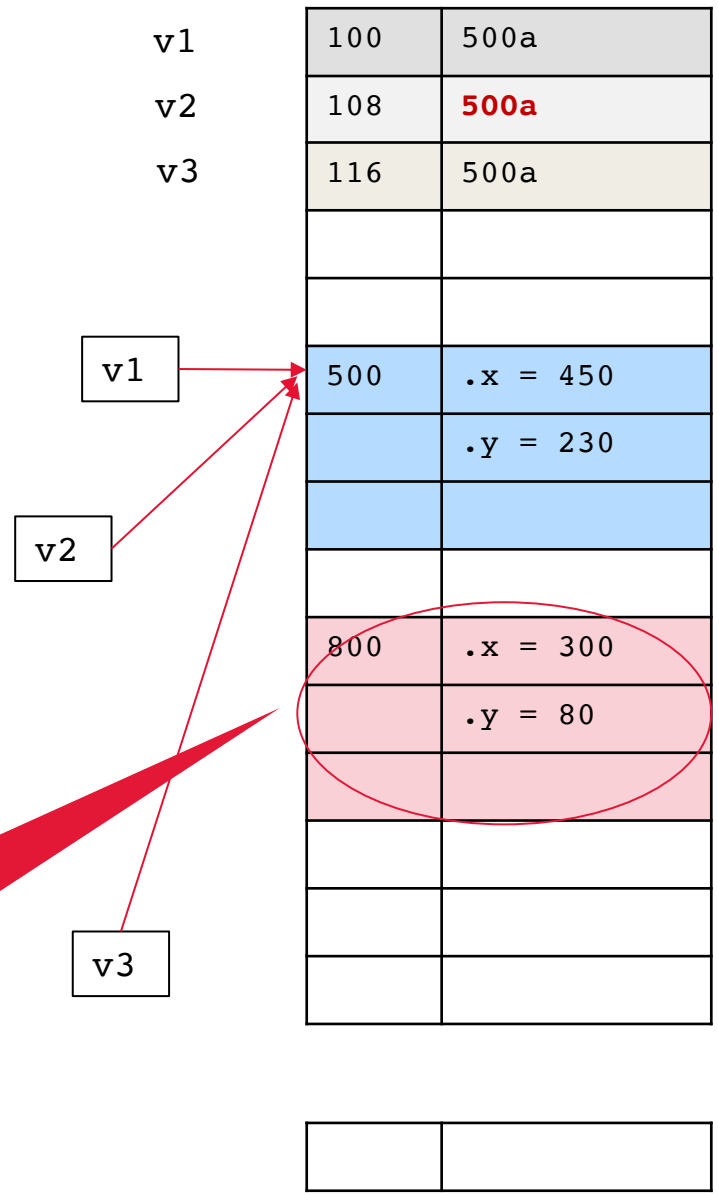
```
final int WHITE = color(255,255,255);
final int GREEN = color(0,255,0);
final int PURPLE = color(255,0,255);

PVector v1;
PVector v2;
PVector v3;

void setup() {

  size(600, 600);
  background(0,0,0);

  v1 = new PVector(150, 150);
  v2 = new PVector(300, 80);
  v1.add(v2);
  v3 = v1;



}
```



What if an object suddenly has no reference to it?

```
v1 = new PVector(150, 150);
v2 = new PVector(300, 80);
v1.add(v2);
v3 = v1;
v2 = v1;
```

```
final int WHITE = color(255,255,255);
final int GREEN = color(0,255,0);
final int PURPLE = color(255,0,255);

PVector v1;
PVector v2;
PVector v3;

void setup() {

  size(600, 600);
  background(0,0,0);

  v1 = new PVector(150, 150);
  v2 = new PVector(300, 80);
  v1.add(v2);
  v3 = v1;
  v2 = v1;


}
```



orphaned object
(no reference to it –
i.e. not accessible)

No reference has a record
of the address 800a
anymore

Death of an Object [1]

- The object-reference connection can be destroyed by
  1. No more references to the object (orphaned)
  2. Exiting the scope of the reference variable
  3. Setting the reference to **null**

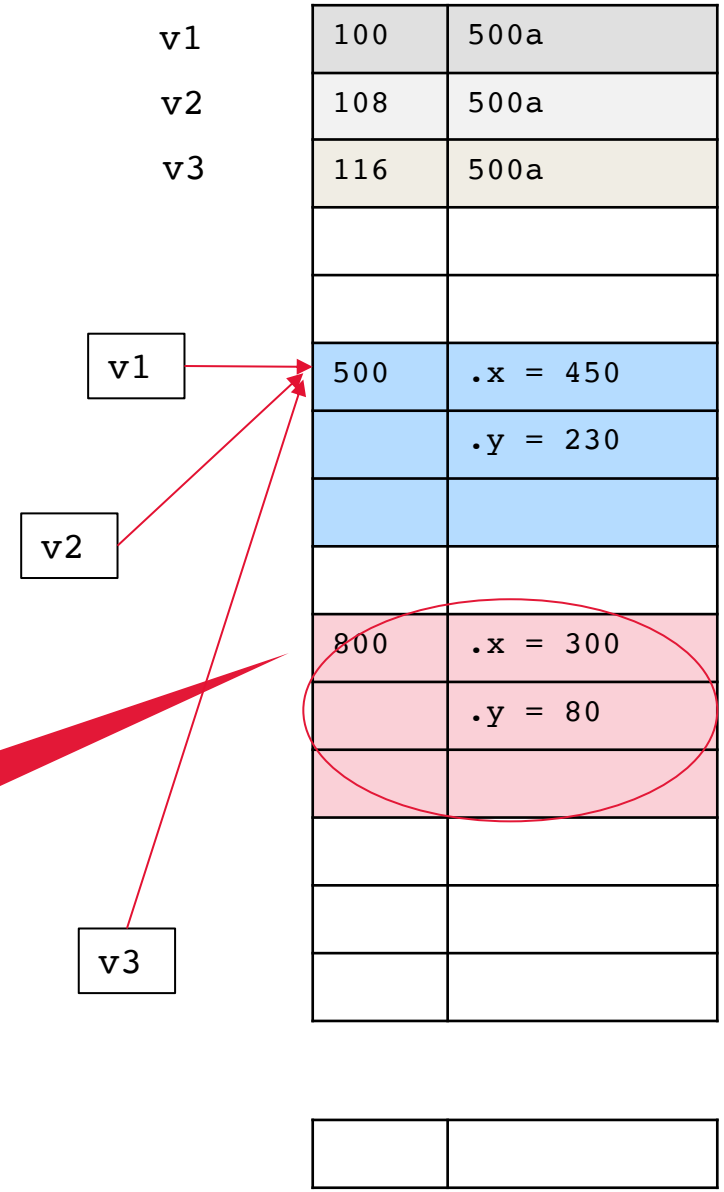
```
v1 = new PVector(150, 150);
v2 = new PVector(300, 80);



v2 = v1;
{   PVector v3 = new Pvector(1,1); }
v1 = null;
```

1
2
3

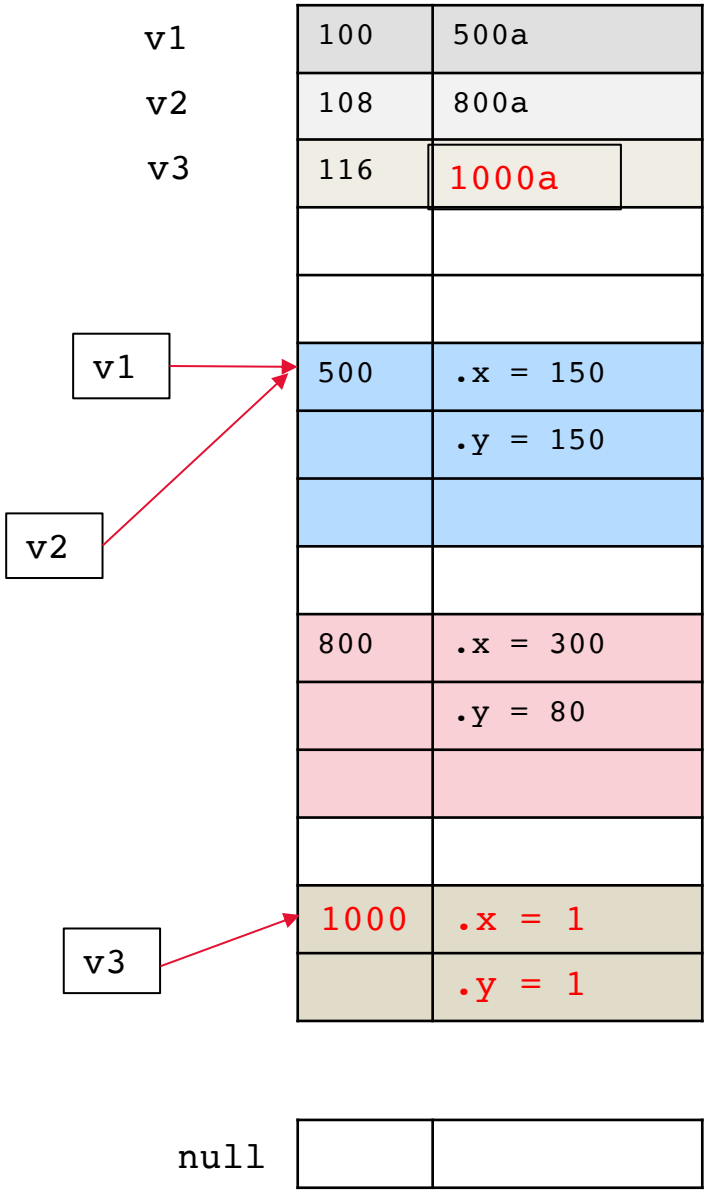
YORK UNIVERSITÉ UNIVERSITY

```
final int WHITE = color(255,255,255);
final int GREEN = color(0,255,0);
final int PURPLE = color(255,0,255);

PVector v1;
PVector v2;
PVector v3;

void setup() {

  size(600, 600);
  background(0,0,0);

  v1 = new PVector(150, 150);
  v2 = new PVector(300, 80);
  v1.add(v2);
  v3 = v1;
  v2 = v1;


}
```

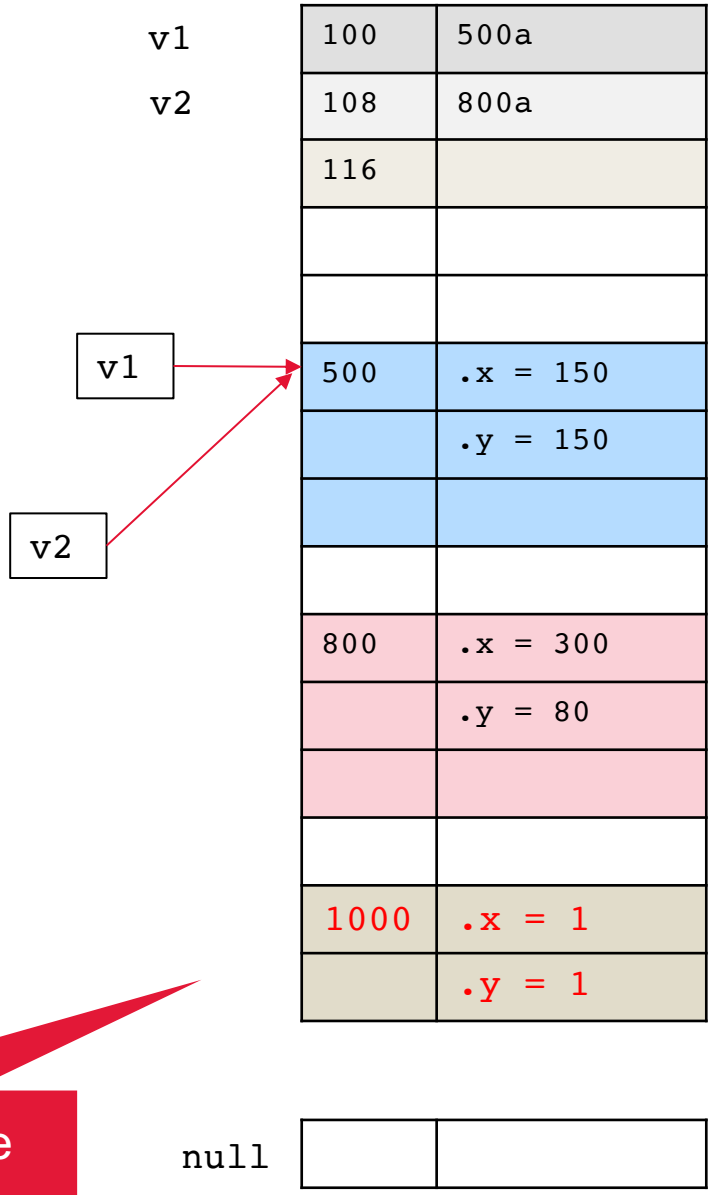
(1) orphaned object
(no more references to it – i.e. not accessible)



```
final int WHITE = color(255,255,255);
final int GREEN = color(0,255,0);
final int PURPLE = color(255,0,255);

PVector v1;
PVector v2;
// PVector v3;

void setup() {

  size(600, 600);
  background(0,0,0);

  v1 = new PVector(150, 150);
  v2 = new PVector(300, 80);
  // v1.add(v2);
  // v3 = v1;
  v2 = v1;

  {
    PVector v3 = new PVector(1,1);
  }


}
```

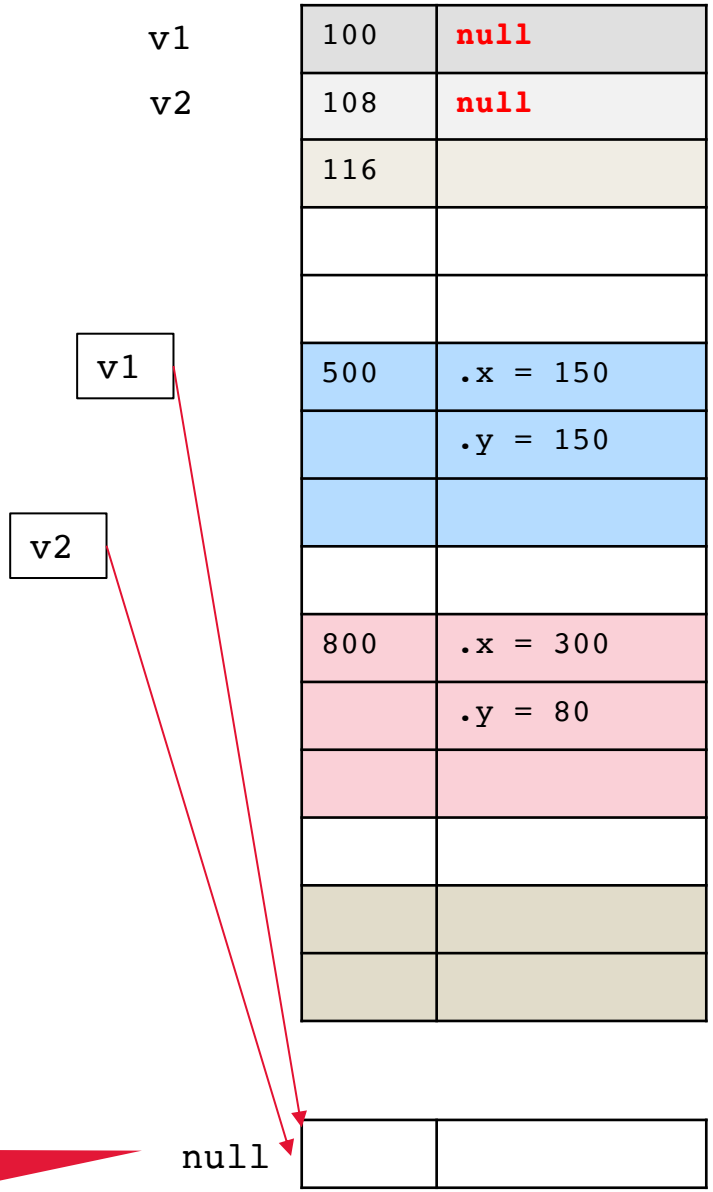


```
final int WHITE = color(255,255,255);
final int GREEN = color(0,255,0);
final int PURPLE = color(255,0,255);

PVector v1;
PVector v2;
// PVector v3;

void setup() {

  size(600, 600);
  background(0,0,0);

  v1 = new PVector(150, 150);
  v2 = new PVector(300, 80);
  // v1.add(v2);
  // v3 = v1;
  v2 = v1;

  {
    PVector v3 = new PVector(1,1);
  }

  // you are here

}
```



(2) v3 went out of scope
(no longer exists)

```
final int WHITE = color(255,255,255);
final int GREEN = color(0,255,0);
final int PURPLE = color(255,0,255);

PVector v1;
PVector v2;
// PVector v3;

void setup() {

  size(600, 600);
  background(0,0,0);

  v1 = new PVector(150, 150);
  v2 = new PVector(300, 80);
  // v1.add(v2);
  // v3 = v1;
  v2 = v1;

  {
    PVector v3 = new PVector(1,1);
  }

  v1 = null;
  v2 = null;


}
```



(3) v1 & v2 set to null

Death of an object [2]

- What is **`null`**?
  - A special (keyword) of an address that refers to “no object”
  - Any **reference** type may be assigned `null`
  - May test/output the value of `null`, but not access any object methods/fields (as there are none)
    - `null` is a literal (just like `true` and `false`) whose type is compatible with any non-primitive type.
    - It is OK to print a `null` reference
    - It is *not* OK to invoke methods on a null reference
  - Attempting to access a field/method of an object reference that is currently pointing at **`null`** will cause an exception (resulting in a program crash!)

YORK UNIVERSITÉ UNIVERSITY