AN INTRODUCTION TO JAVA

AY19/20 Sem 2

School of Computing

ACKNOWLEDGEMENTS

2 slides were “inspired” by the Java Confidence Course.

Thanks to the CS2030 teaching teem for their feedback.

SCOPE

- About Java
- Using UNIX CLI and Vim
- Syntactical Comparison
- Classes, Instances, Methods and Attributes
- Understanding Objects
- Java API

ABOUT JAVA

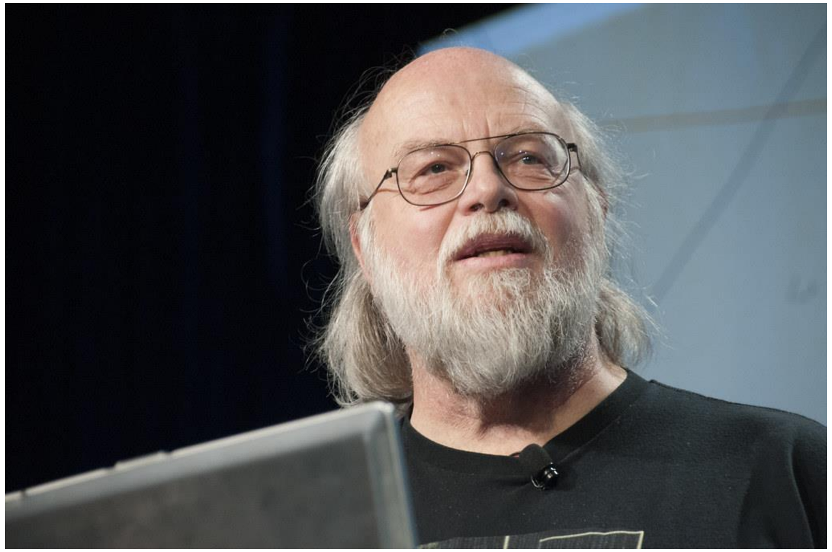
The high level, multi-threaded, Just-In-Time (JIT) or Ahead-Of-Time (AoT) compiled, statically typed, Object-Oriented, class-based inheritance, General Purpose programming language with Parallel Workers, Assembly Line and Functional Parallelism as its concurrency models.

JAVA

Invented by James Gosling in 1991, originally codenamed 'Oak'

Was initially created **cross-platform desktop applications** and rich internet applications (RIAs)

Currently serves as a general purpose, backend language.



JAVA

Compiled Java code can be highly portable, provided some care is taken

Very verbose, making it easy to learn and use, though sacrificing some expressiveness

WRITE ONCE,
RUN ANYWHERE

JAVA

Java is a **high level**, multi-threaded, Just-In-Time (JIT) or Ahead-Of-Time (AoT) **compiled**, **statically typed**, **Object-Oriented**, **class-based inheritance**, General Purpose programming language with Parallel Workers, Assembly Line and Functional Parallelism as its concurrency models.

NUS National University of Singapore

Withdrawal from the University (Undergraduate Students)
Please read the instructions and general notes on this form.

To: Dean of Faculty/School

I wish to withdraw from the University for the reason below (please tick only 1 of the following boxes):

- ☐ Unable to cope with studies
- ☐ English language difficulties
- ☐ Financial difficulties
- ☐ Overseas study (self-financing)
- ☐ Overseas study (scholarship)
- ☐ Others (please specify):
- ☐ National Service commitment
- ☐ Obtained employment
- ☐ Medical reasons
- ☐ Not interested in the course
- ☐ Personal difficulties

Name:
Student Number:
Programme of Study:
Current Year of Study:
Citizenship:
Mailing Address:

NUS email:
Personal email:
Telephone (Home) :
Telephone (Mobile) :

Please tick where appropriate:
☐ I am a scholarship holder and understand that I am required to obtain acknowledgement from the Office of Financial Aid (within Office of Admissions) before submitting the withdrawal form to the Faculty/School.
Name of scholarship:

☐ I am not a scholarship holder.

✓ I declare the above information is true.

HIGH-LEVEL LANGUAGES

Machines read and process data / instructions in binary (0s and 1s)

Will it be feasible for us to provide instructions to a machine in binary?

```
0111011000001101110110000011011101100000110111011111
0111011011111101110110111111011101101111110111011111
0000011000001100000110000011000001100000110000011111
0111011011111101110110111111011101101111110111011111
0111011000001101110110000011011101100000110111011111
```

HIGH-LEVEL LANGUAGES

High-level programming languages **abstract** these away so that we can code faster.

As such, Java faces the Abstraction Penalty as certain low-level components (like memory management) are inaccessible to us.

However, the ease of development of Java programs are much higher compared to its low-level counterparts (like Assembly).

COMPILED VS INTERPRETED

Generally speaking, among programming languages, they can be split into two types:

COMPILED

INTERPRETED

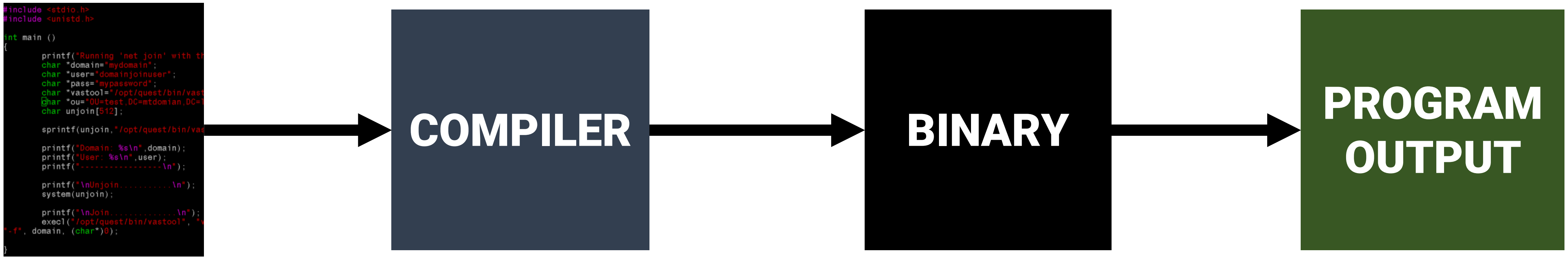
COMPILED LANGUAGES

Source code is stored as text.

```c
#include <stdio.h>
#include <unistd.h>

int main ()
{
	printf("Running 'net join' with the following parameters: \n");
	char *domain="mydomain";
	char *user="domainjoinuser";
	char *pass="mypassword";
	char *vastool="/opt/quest/bin/vastool";
	char *ou="OU=test,DC=mtdomian,DC=local";
	char unjoin[512];

	sprintf(unjoin,"/opt/quest/bin/vastool -u %s -w '%s' unjoin -f",user,pass);

	printf("Domain: %s\n",domain);
	printf("User: %s\n",user);
	printf("-----------------\n");

	printf("\nUnjoin...........\n");
	system(unjoin);

	printf("\nJoin..............\n");
	execl("/opt/quest/bin/vastool", "vastool", "-u", user, "-w", pass, "join", "-c", ou,
"-f", domain, (char*)0);

}
```

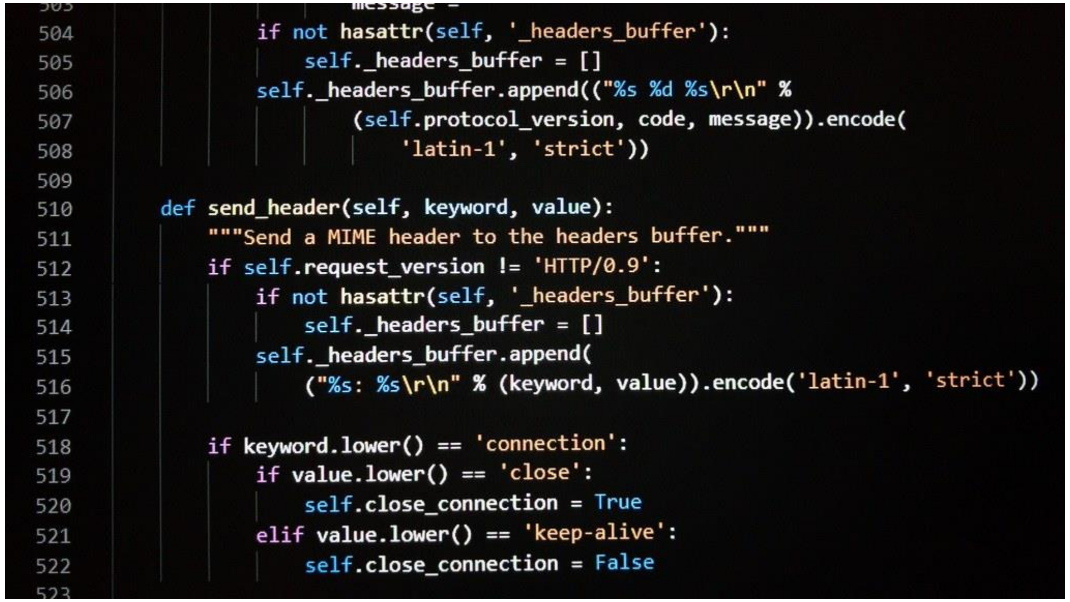
COMPILED LANGUAGES

The source code is first compiled by a compiler into binary before it can be natively run.



INTERPRETED LANGUAGES

Source code is also stored as text.



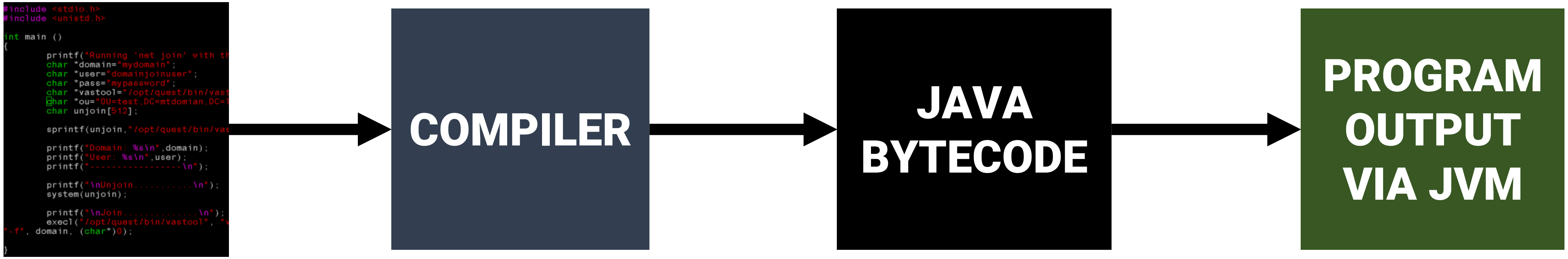
INTERPRETED LANGUAGES

The source code is interpreted by an interpreter and executed.

JAVA

Java is also a compiled language.

However, instead of compiling to native binary, it is compiled to **Java Bytecode**. Right before program execution, this bytecode is compiled **Just-In-Time** by the Java Virtual Machine (JVM) to be run.



JAVA

This allows the **same Java Bytecode** to be run on all platforms.

JAVA BYTECODE

STATIC TYPING

Those who are familiar with JavaScript / Python would be familiar with this variable assignment statement:

Name of new / existing variable

```
[var/let/const] x = 5
```

New value of variable

This is a feature of **dynamically typed** languages—there is no need to determine the type of the variable beforehand.

STATIC TYPING

In Java and most other compiled languages however, there is a need to determine the type of the variable when it is **declared**, such as by doing:

Type of newly declared variable

Name of newly declared variable

```
int x;
```

In this scenario, we are **declaring** a new variable x of type/class `int`. All variables must be declared **once** before any operations on it are performed.

STATIC TYPING

We may also choose to initialise the variable when it is first declared, such as by doing

```
int x = 2;
```

Value of new variable

In this scenario, we are declaring and **initialising** a new variable `x` of type/class `int` which has an initial value of `2`.

STATIC TYPING

We may then continue to assign values to this variable. Aside from Polymorphism, all new values that the variable takes must be the **same as the type** defined in its declaration.

```
x = 3; // allowed
x = “Hello World!”; // compilation error
```

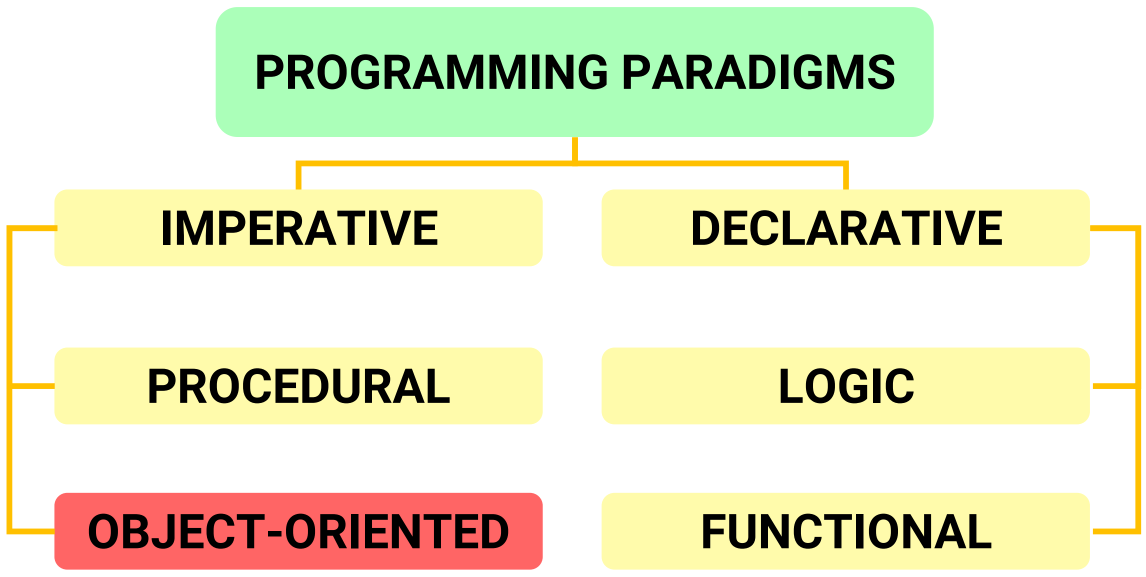
Because `x` was declared to be an `int`, it cannot take on the value of `“Hello World”`, which is a `String`.

PROGRAMMING PARADIGMS

Programming paradigms are schools of thought—**different approaches** to solving problems.

Java is heavily object-oriented, where **Object-Oriented Programming (OOP)** is a well-known imperative programming paradigm.

Most of you would be familiar with the procedural programming paradigm in previous courses.



UNIX & VIM

The right way to code

UNIX

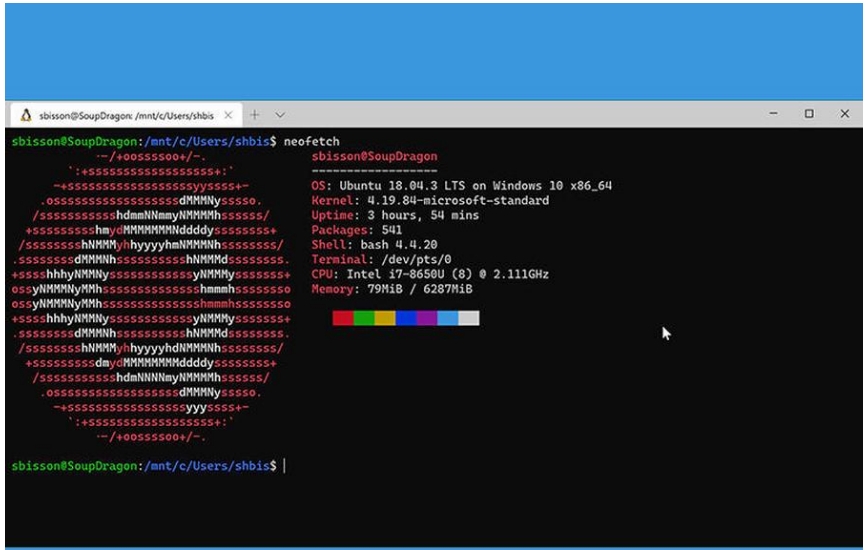
UNIX is an operating system developed in the 1970s by AT&T's Bell Labs, by Ken Thompson, Dennis Ritchie and others.

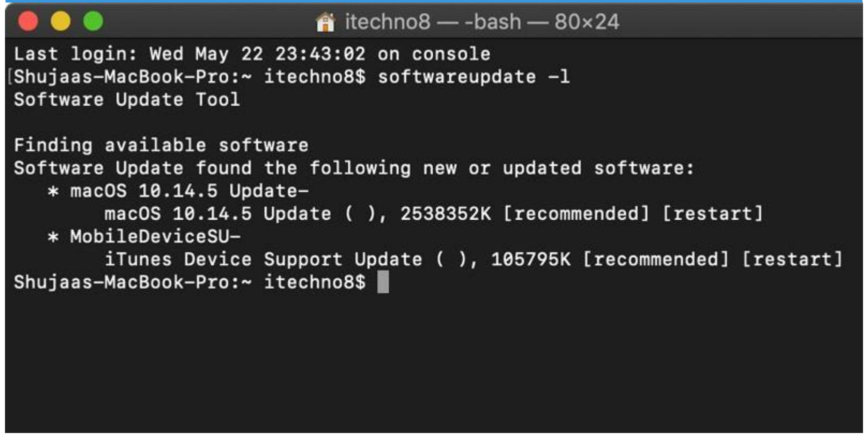
What we are interested in is using the **Command-Line Interface (CLI)** in MacOS / Linux environment which are similar to the UNIX CLI.

UNIX

For Windows users, you should have the Windows Subsystem for Linux (WSL) installed and ready to go.

Launch your WSL. For MacOS / Linux users, launch your terminal.





UNIX COMMANDS

Command	Function
`ls`	List all files and directories in current working directory
`cd directoryname`	Change directory to `directoryname`
`rm filename`	Remove (delete) `filename`
`mv filename directoryname`	Move `filename` into `directoryname`
`mkdir directoryname`	Make a directory called `directoryname`
`clear`	Clear the screen
`exit`	Logout of SSH

UNIX COMMANDS

Command	Function
`vim filename.java`	Open `filename.java` in Vim
`vim -p file1.java file2.java`	Open `file1` and `file2` in separate tabs in Vim
`javac codename.java`	Compile `codename.java` as Java classes
`javac -d . codename.java`	Compile `codename.java` with packages in correct directories
`java ClassName`	Run `ClassName.class`
`cp filename newfilename`	Copy `filename` and paste as `newfilename`
`fg number`	Resume job `number`

UNIX COMMANDS

Command	Function
`javadoc filename.java`	Create Javadocs for `filename.java`
`checkstyle fileName.java`	Java Checkstyle for `fileName.java`
`java ClassName < inputTextFile | diff - outputTextFile`	Run `ClassName`, using `inputTextFile` as its input. Then compare the respective output with `outputTextFile`

VIM

Vim is the text editor of choice for CS2030.

It is a command-line editor; there is a slight learning curve to it, but one of the best parts about it is that you **don’t have to use your mouse**.

VIM COMMANDS

You need to be in Command Mode first. Press Esc to get there.

Command	**Function**
`i`	Enter **I**nsert Mode
`v`	Enter **V**isual Mode
`:w`	**W**rite file
`:wq`	**W**rite file and **q**uit
`:q!`	**Q**uit without writing since last change (proceed with caution!)
`:123456`	Jump to line 123456
`/foo`	Find the first instance of `foo` in your code (press `n` to cycle)

VIM COMMANDS

You need to be in Command Mode first. Press Esc to get there.

Command	Function
`gg=G`	Auto-indent code
`gg`	Go to first line
`G`	Go to last line
`:%s/foo/bar/g`	Replace all instances of `foo` with `bar`
`y` (in Visual Mode)	Yank (copy) selected line(s)
`x` (in Visual Mode)	Cut selected line(s)
`p` (in Visual Mode)	Paste line(s) in clipboard
Ctrl + Z	Stop job

VIM COMMANDS

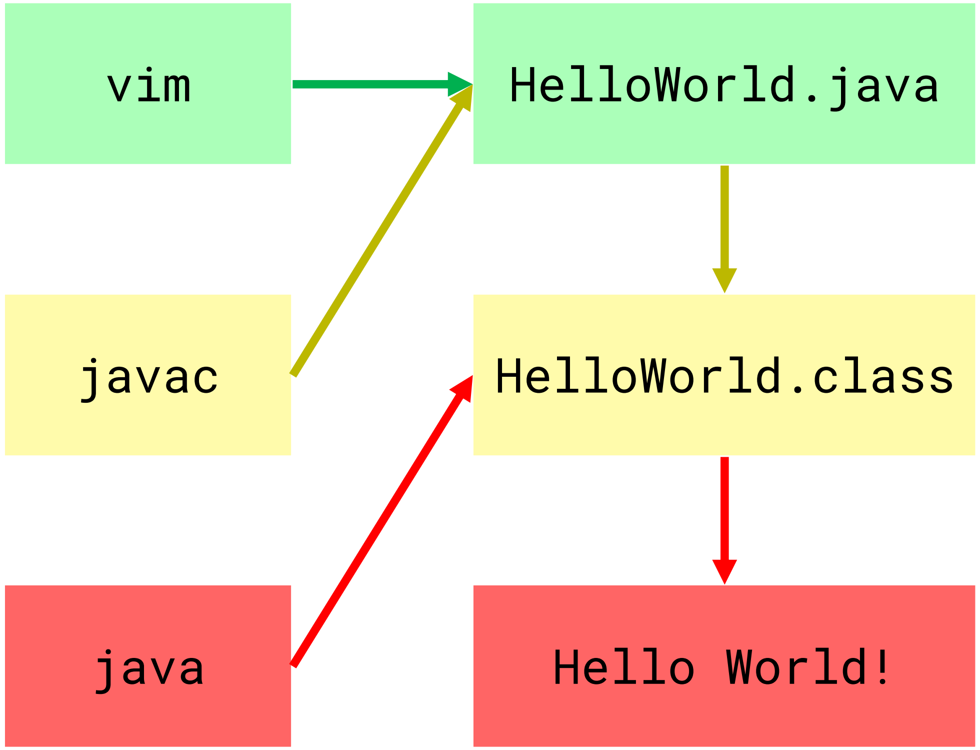
You need to be in Command Mode first. Press Esc to get there.

Command	**Function**
`:tabedit fileName`	Open `fileName` in another tab
`gt / gT`	Cycle tab
`:split fileName`	Split screen and open `fileName`
`:e $MYVIMRC`	Edit your `.vimrc` file
Ctrl + N	Auto-complete

QUESTIONS?

EDIT-COMPILE-RUN

Writing programs in Java follow the Edit-Compile-Run process:



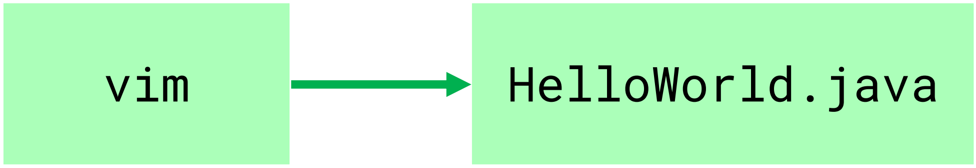
Edit: Editing your code

Compile: Compiling your code into Java bytecode

Run: Running your code on the Java Virtual Machine (JVM)

EDIT-COMPILE-RUN

Edit



`vim HelloWorld.java`

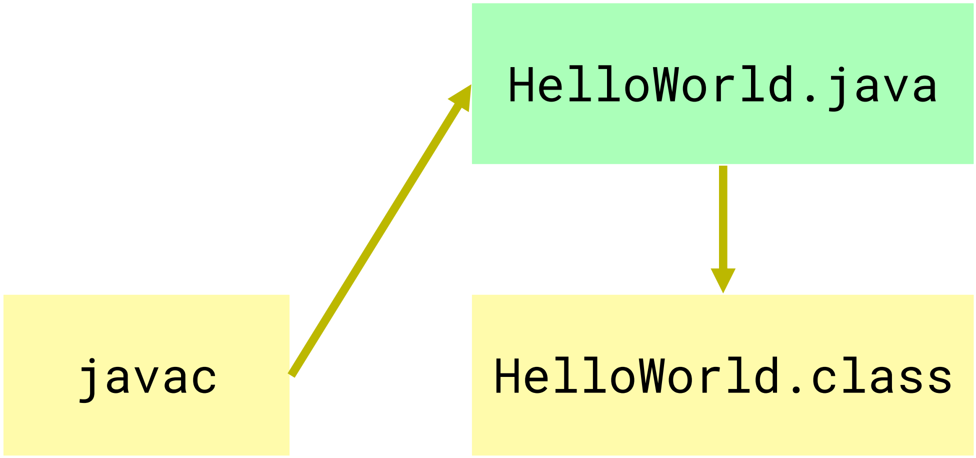
Allows us to edit our
`HelloWorld.java` file

EDIT-COMPILE-RUN

Compile

`javac HelloWorld.java`

Compiles `HelloWorld.java` into Java bytecode as `HelloWorld.class`

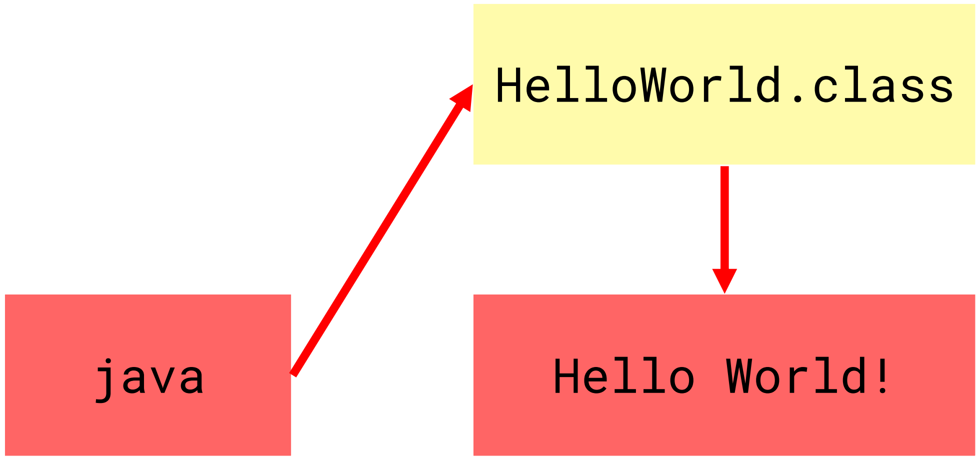


EDIT-COMPILE-RUN

Run

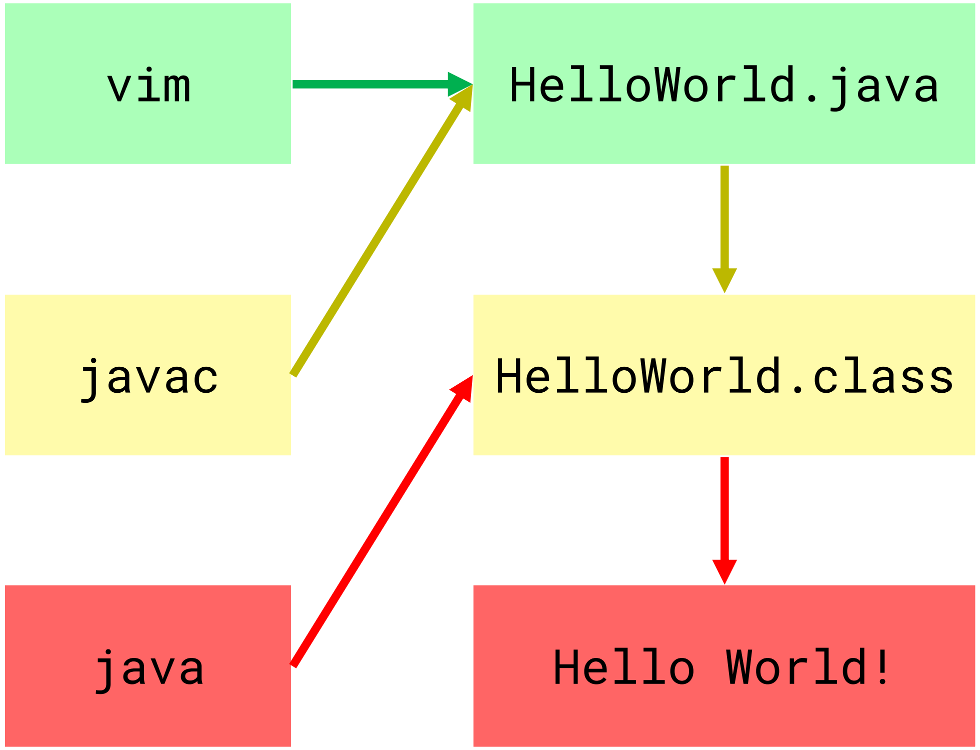
`java HelloWorld`

Executes the `HelloWorld.class` bytecode on the JVM.



EDIT-COMPILE-RUN

If we want to make changes to the behaviour of our programs, we need to follow this cycle again.



HELLO WORLD HANDS-ON

HELLO WORLD

Write and run the program following the edit-compile-run cycle.

`HelloWorld.java:`

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println(“Hello World!”);
    }
}
```

QUESTIONS?

JAVA SYNTAX

A comparison of basic syntax across popular languages

SYNTACTICAL COMPARISON

Java has quite a unique syntax which you need to know in order to write programs in Java.

The following slides will show the difference in syntax between Java and C / Python / JavaScript (ES6).

SYNTACTICAL COMPARISON

Empty programs:

C:

```c
#include <stdio.h>

int main(void) {
    // Code here
    return 0;
}
```

Python:

```python
# Code here
```

JavaScript:

```javascript
// Code here
```

SYNTACTICAL COMPARISON

Empty programs:

Java:

```java
public class DriverClass {
    public static void main(String[] args) {
        // Code here
    }
}
```

SYNTACTICAL COMPARISON

Console output:

C:

```
printf(“string”);
```

Python:

```
print(‘string’)
```

JavaScript:

```
console.log(‘string’)
```

SYNTACTICAL COMPARISON

Console output:

Java:

```
System.out.println(“string”);
System.out.print(“string”); // no line feed
```

SYNTACTICAL COMPARISON

Console input:

C:

```
scanf("%d", &var1);
scanf("%lf", &var1);
```

Python:

```
var1 = input()
var1 = int(input())
var1 = float(input())
```

SYNTACTICAL COMPARISON

Console input:

Java:

```java
import java.util.Scanner;

public class DriverClass {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int i = sc.nextInt();
        double j = sc.nextDouble();
        String k = sc.next();
    }
}
```

SYNTACTICAL COMPARISON

Console input:

String	next()	Finds and returns the next complete token from this scanner.
String	next(String pattern)	Returns the next token if it matches the pattern constructed from the specified string.
String	next(Pattern pattern)	Returns the next token if it matches the specified pattern.
BigDecimal	nextBigDecimal()	Scans the next token of the input as a BigDecimal.
BigInteger	nextBigInteger()	Scans the next token of the input as a BigInteger.
BigInteger	nextBigInteger(int radix)	Scans the next token of the input as a BigInteger.
boolean	nextBoolean()	Scans the next token of the input into a boolean value and returns that value.
byte	nextByte()	Scans the next token of the input as a byte.
byte	nextByte(int radix)	Scans the next token of the input as a byte.
double	nextDouble()	Scans the next token of the input as a double.
float	nextFloat()	Scans the next token of the input as a float.
int	nextInt()	Scans the next token of the input as an int.
int	nextInt(int radix)	Scans the next token of the input as an int.
String	nextLine()	Advances this scanner past the current line and returns the input that was skipped.
long	nextLong()	Scans the next token of the input as a long.
long	nextLong(int radix)	Scans the next token of the input as a long.
short	nextShort()	Scans the next token of the input as a short.
short	nextShort(int radix)	Scans the next token of the input as a short.

SYNTACTICAL COMPARISON

Strings with format specifiers / Template Literals:

C:

```
printf("%d %f\n", var1, var2);
```

Python:

```
print("%d %f" % (var1, var2))
```

JavaScript (ES6):

```
console.log(`${var1} ${var2}`)
```

SYNTACTICAL COMPARISON

Strings with format specifiers:

Java:

```
// Print with format specifier
System.out.printf("%d %f", var1, var2);

// String.format constructs a new string
String s = String.format("%d\n", var1);
```

SYNTACTICAL COMPARISON

Comments

C / JavaScript:

```
// single line comment

/* Multi
Line
Comment
*/
```

Python:

```
# single line comment

‘’’ multi
line string but we pretend
it is a comment
‘’’
```

SYNTACTICAL COMPARISON

Comments

Java:

```
// single line comment

/* Multi
 * Line
 * Comment
 */
```

SYNTACTICAL COMPARISON

Declaration of variables:

C:

```c
int i;
double j;
bool k;
int m[5];
```

Python:

```python
m = []
```

JavaScript (ES6):

```javascript
var i
let j
const k = false
var m = []
```

SYNTACTICAL COMPARISON

Declaration of variables:

Java:

```
int i;
double j;
ClassName k;
int[] m;
```

SYNTACTICAL COMPARISON

Initialisation of variables:

C:

```c
int i = 1;
double j = 4.0;
bool k = true;
int m[5] = {1, 2, 3, 4, 5};
```

JavaScript:

```javascript
var i = 1
let j = 4.0
const k = true
var m = [1, 2, 3, 4, 5]
```

Python:

```python
i = 1
j = 4.0
k = True
m = [1, 2, 3, 4, 5]
```

SYNTACTICAL COMPARISON

Initialisation of variables / instantiating objects:

Java:

```
int i = 1;
double j = 4.0;
boolean k = true;
int[] m = new int[5];
int[] n = new int[] {1, 2, 3, 4, 5};
ClassName n = new ClassName(arg1, arg2);
```

SYNTACTICAL COMPARISON

Type casting:

Python:

```
int(i)
float(i)
```

C & Java:

```
(int) i;
(double) i;
```

SYNTACTICAL COMPARISON

Mathematical operators:

Java:

```
1 + 1 // == 2
2 - 1 // == 1
3 * 4 // == 12
4 / 3 // == 1
4.0 / 3 // == 1.333...
4 % 3 // == 1
Math.pow(2, 3) // == 8
```

SYNTACTICAL COMPARISON

Pre/Post incrementation/decrementation:

Java / C / JavaScript:

```
i++;
++i;
i--;
--i;
```

Python:

```
i += 1
i -= 1
```

System.out.print(i++) will print 1, then i change to 2;
System.out.print(++i) i will change to 2, then print 2;

SYNTACTICAL COMPARISON

Control structures

Python:

```
if i > 1 and i > 2 or j != 3:
    func()
elif not isEmpty():
    someOtherfunc()
else:
    return
```

SYNTACTICAL COMPARISON

Control structures

Java / C / JavaScript:

```
if (i > 1 && i > 2 || j != 3) {
    func();
} else if (!isEmpty()) {
    someOtherfunc();
} else {
    return;
}
```

OPERATOR	SIGNIFICANCE
==	Equals
!=	Not equals
&&	Logical AND
||	Logical OR
!	Logical NOT

SYNTACTICAL COMPARISON

Loops

Python:

```python
for i in range(10):
    ...

while condition():
    ...
```

SYNTACTICAL COMPARISON

Loops

Java / C / JavaScript:

```
for (i = 0; i < 10; ++i) {
    ...
}

while (i <= 2) {
    ...
}
```

```
do {
    ...
} while (i <= 2);
```

SYNTACTICAL COMPARISON

Enhanced for loops

Java:

```
for (Point p : listOfPoints) {
      ...
}
```

LOOP MECHANISM

For loops:

1

```
for (i = 0; i < 5; i++){
    statements;
}
```

1 – initialise loop variable(s)

Loop sequence:

1

LOOP MECHANISM

For loops:

1 2

```
for (i = 0; i < 5; i++){
   statements;
}
```

2 – check loop condition.
If true:

Loop sequence:

1

2

LOOP MECHANISM

For loops:

1 2

```
for (i = 0; i < 5; i++){
    statements;
}
```

3

3 – run statement block

Loop sequence:

1

2

3

LOOP MECHANISM

For loops:

```
for (i = 0; i < 5; i++){
    statements;
}
```

4 – do something

Loop sequence:

1
2
3
4

LOOP MECHANISM

For loops:

```
          1          2        4
for (i = 0; i < 5; i++){
    statements;   3
}
```

Loop sequence:

1

2 2

3

4

2 – check loop condition again.
If true:

LOOP MECHANISM

For loops:

```
        1       2      4
for (i = 0; i < 5; i++){
    statements;   3
}
```

Loop will continue to run until…

Loop sequence:

1

2 2

3 . . .

4

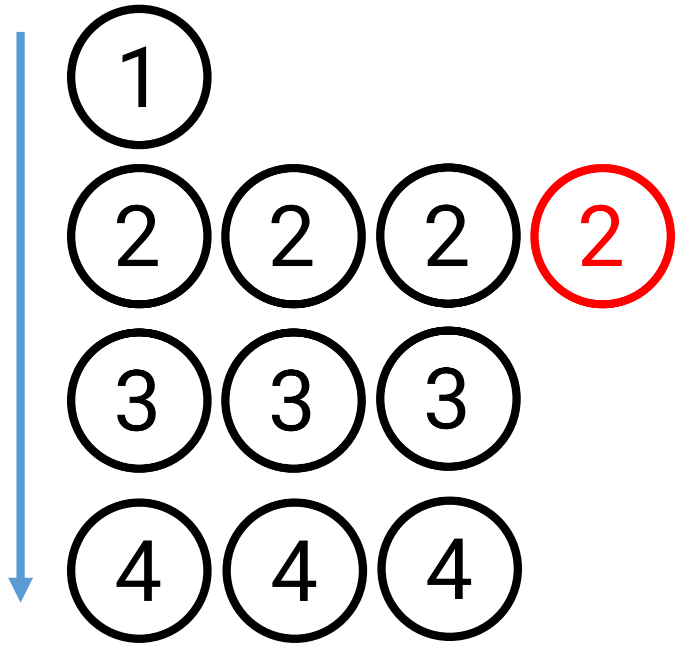
LOOP MECHANISM

For loops:

```
for (i = 0; i < 5; i++){
    statements;
}
```

Loop condition returns `false`.
Loop terminates

Loop sequence:



LOOP MECHANISM

While loops:

1

```
while (i == 0){
    statements;
}
```

1 – check loop condition
If true:

Loop sequence:

1

LOOP MECHANISM

While loops:

1

```
while (i == 0){
    statements;
}
```

2

2 – run statement block

Loop sequence:

1

2

LOOP MECHANISM

While loops:

```
while (i == 0){
    statements;
}
```

1 – check loop condition
If true:

Loop sequence:

1

2

1

LOOP MECHANISM

While loops:

1

```
while (i == 0){
    statements;
}
```

2

Loop will continue until…

Loop sequence:

1

2

1

. . .

LOOP MECHANISM

While loops:

```
                1
while (i == 0){
    statements;
}
                    2
```

Loop condition returns `false`.
Loop terminates

Loop sequence:

1 1 1 1

2 2 2

1 1 1

2 2 2

LOOP MECHANISM

Do while loops:

```
do {                1
    statements;
} while (i == 0);
```

1 – run statement block

Loop sequence:

1

LOOP MECHANISM

Do while loops:

1

```
do {
    statements;
} while (i == 0);
```

2

2 – check loop condition
If true:

Loop sequence:

1

2

LOOP MECHANISM

Do while loops:

1

```
do {
    statements;
} while (i == 0);
```

2

1 – run statement block

Loop sequence:

1

2

1

LOOP MECHANISM

Do while loops:

1

```
do {
    statements;
} while (i == 0);
```

2

Loop condition returns `false`.
Loop terminates

Loop sequence:

1	1	1	1
2	2	2	2
1	1	1	
2	2	2	

LOOP MECHANISM

Enhanced for loops:

Point p =

```
for (Point p : points) {
    System.out.println(p);
}
```

Output:

points list:

1 2 3 4 5

LOOP MECHANISM

Enhanced for loops:

```
for (Point p : points) {
    System.out.println(p);
}
```

points list:

1 2 3 4 5

Point p = 1

Output:

```
point 1
```

LOOP MECHANISM

Enhanced for loops:

```
for (Point p : points) {
    System.out.println(p);
}
```

points list:

1 2 3 4 5

Point p = 2

Output:

```
point 1
point 2
```

LOOP MECHANISM

Enhanced for loops:

```
for (Point p : points) {
    System.out.println(p);
}
```

Point p = 3

points list:

1 2 3 4 5

Output:

```
point 1
point 2
point 3
```

LOOP MECHANISM

Enhanced for loops:

Point p = 4

```
for (Point p : points) {
    System.out.println(p);
}
```

points list:

1 2 3 4 5

Output:

```
point 1
point 2
point 3
point 4
```

LOOP MECHANISM

Enhanced for loops:

Point p = 5

```
for (Point p : points) {
    System.out.println(p);
}
```

points list:

1 2 3 4 5

Output:

```
point 1
point 2
point 3
point 4
point 5
```

LOOP MECHANISM

Enhanced for loops:

Point p = 5

```
for (Point p : points) {
    System.out.println(p);
}
```

points list:

1 2 3 4 5

Loop has reached end of list and thus terminates.

Output:

```
point 1
point 2
point 3
point 4
point 5
```

SYNTACTICAL COMPARISON

Functions

C:

```c
int myFunction(int var1, double var2) {
    ...
    return 1;
}
```

Python:

```python
def myFunction(var1, var2):
    ...
    return 1
```

SYNTACTICAL COMPARISON

Functions

JavaScript:

```
const myFunction = function (var1, var2) {
    ...
}

function myFunction2 (var1, var2) {
    ...
}
```

SYNTACTICAL COMPARISON

Methods

Java:

```
int myFunction(int var1, double var2) {
    ...
    return 1;
}
```

SYNTACTICAL COMPARISON

Attributes / Method Calls

Java / JavaScript / Python:

```
s.myAttribute = 1
x = t.anotherAttribute
y = u.methodCall(arg1, arg2)
```

QUESTIONS?

SYNTAX EXERCISE

Write a program that takes in an `int` and a `double`, a and b, and output:

a^b,

$\gcd(a, \text{floor}(b))$

smallest $\text{floor}(b)$ consecutive Fibonacci numbers greater than a

Do not print anything if either a or b are negative.

SYNTAX EXERCISE

Example run (user input is underlined):

```
~ $ java Main
6 4.3
2218.45373779
2
[8, 13, 21, 34]
~ $
```

OOP BASICS

Classes, Instances, Methods and Attributes

WHAT IS OOP

As the name suggests, OOP is all about Objects.

In Java, pretty much everything is an object.

CLASSES

Before we jump in and create objects, we need to define classes first.

Classes define the blueprint of its objects. You can think of them as “types”.

CLASSES

Objects primarily have data and behaviour.

To group objects of a certain type having similar data and behaviours, we create a class.

Let's create a `Student` class.

CLASSES

It is good practice to have each class definition stored in its own file.

Let's create a `Student.java` in our working directory.

In our terminal, enter (exclude everything before and including `$`)

```
~ $ vim Student.java
```

CLASSES

To get started, we need to state that we are creating a new class in the new file.

```
class Student {

}
```

The definition of the `Student` class falls within the curly braces.

CLASSES

We may add access modifiers and/or abstract/final to this class as well, for example,

```
[public/private] [abstract/final] class Student {

}
```

We will stick to `class Student` for now.

CLASSES

Within our class, we may define attributes (data) and methods (behaviour). Let's say all students have an `id` attribute as an `int`.

```
class Student {
    int id;
}
```

Now, all `Students` have an `int id`.

CLASSES

Let's also define that all Students have a name. Because it is unlikely that they will change names, let's define it as a constant using the `final` keyword.

```
class Student {
    int id;
    final String name;
}
```

When an attribute/variable is `final`, its value **cannot change after initialisation**.

CLASSES

Let's say all students have a crush on another student. Let's add that in.

```
class Student {
    int id;
    final String name;
    Student crush;
}
```

Notice that the attribute types can be classes as well.

CLASSES

Now that we have defined the attributes that all Students have, let's define its **method**s (behaviours).

Let's define a hello world method that simply prints "<<name>> says hello world!"

CLASSES

A simple method requires certain keywords:

```
<return type> <method name>(<arg1, arg2, ...>) {
    // method definition
}
```

CLASSES

In our case, we are defining a method called `sayHello` that returns `void`. Let's add that in.

```
void sayHello() {

}
```

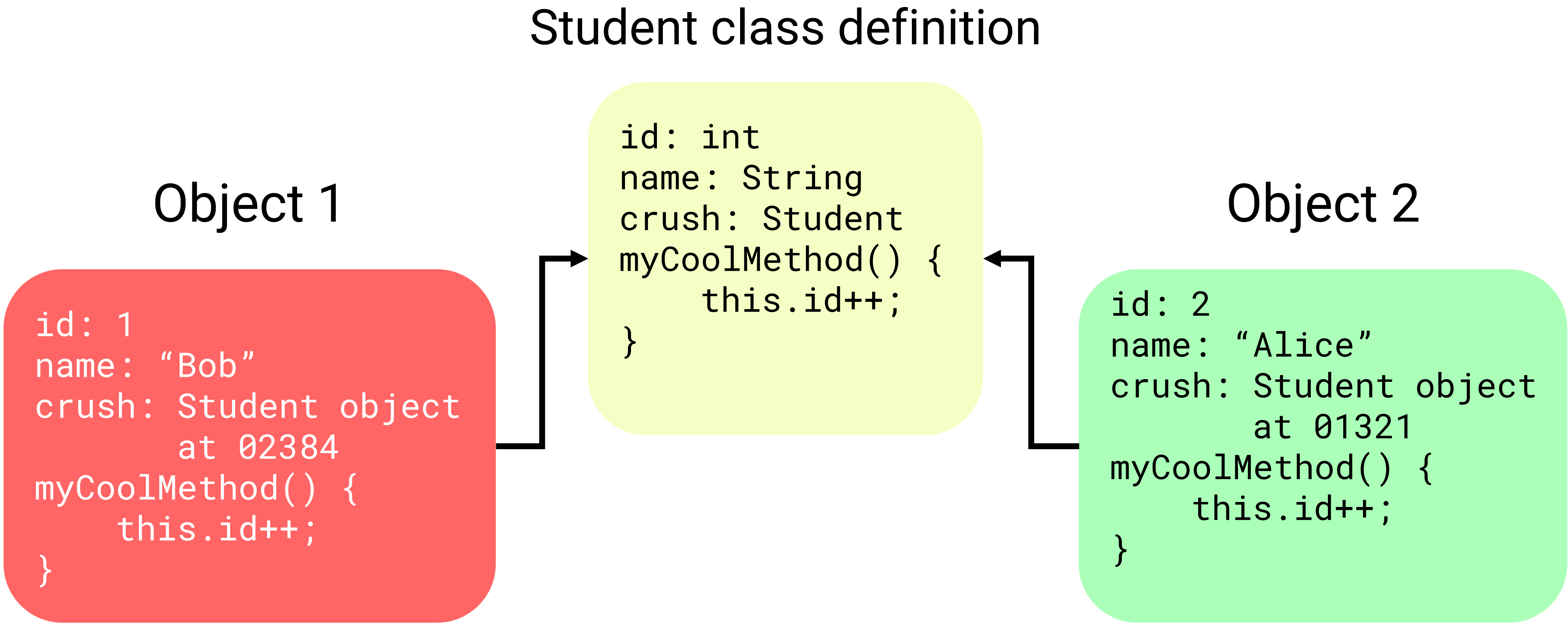
CLASSES

For a student to `sayHello`, the method requires knowledge of that particular student's `name`. We can use the `this` keyword.

How does `this` work?

CLASSES

The `this` keyword refers to the object / context from which it is called.



CLASSES

Assume we are calling the `myCoolMethod` method from a `Student` object whose name is `"Bob"`. In the **context** of the object in red, `this` refers to itself.

```
id: 1
name: "Bob"
crush: Student object
       at 02384
```

CLASSES

When we call some method of this object, within the method, `this.attributeName` would refer to the value stored in `attributeName` for that particular object.

That is, `this.id` is 1, `this.name` is "Bob" and so on.

```
id: 1
name: "Bob"
crush: Student object
       at 02384
```

call `myCoolMethod()` →

```
this.id      // 1
this.name    // "Bob"
this.crush   // Student@02384
```

CLASSES

Conversely, if we called the **same method** from a **different object**, the values for those attributes will match accordingly.

```
id: 2
name: “Alice”
crush: Student object
       at 01321
```

call `myCoolMethod()`

```
this.id     // 2
this.name   // “Alice”
this.crush  // Student@01321
```

CLASSES

Let’s make use of that. Our method definition now embodies the desired behaviour.

```
void sayHello() {
    System.out.println(this.name +
            “ says hello world!”);
}
```

CLASSES

Let's add that in.

```
class Student {
    // ...
    void sayHello() {
        System.out.println(this.name +
                " says hello world!");
    }
}
```

INSTANCES

Now, we are almost ready to create Student objects.

First, let's talk about instances.

INSTANCES

An instance of a class is a particular object that was created from the definition of that class.

That is to say, if some object is an **instance** of the `Student` class, then that object is a `Student`.

INSTANCES

To **instantiate** (create) objects of a particular class, we use the `new` keyword.

```
new Student();
```

Notice that there are parentheses “`()`” after `Student`. Is that a method?

INSTANCES

```
new Student();
```

Yes! This is what is called a **constructor**. It is the method(s) of a class that defines how objects of that class are instantiated.

When you use the `new` keyword, it is most likely followed by a constructor.

INSTANCES

“But we have not defined any constructor!”

If no constructors are defined, the default empty constructor is added in automatically. However, we should always define our own constructor for our convenience.

INSTANCES

Let's define our constructor. A constructor looks something like this:

```
<<className>>(arg1, arg2, ...) {
    // constructor definition
}
```

INSTANCES

Since we are defining a constructor of the `Student` class, we shall write it as such:

```
Student() {
    // constructor definition
}
```

INSTANCES

To make our lives easier, we want to immediately define the id and name of each student when we instantiate it.

We can do so by passing them as arguments to our constructor.

```
Student(int id, String name) {
    // constructor definition
}
```

INSTANCES

Now we can set these attributes in the constructor definition.

```
Student(int id, String name) {
    this.id = id;
    this.name = name;
}
```

Notice the distinction between `this.id` and `id`, `this.id` refers to the attribute of the object itself, while `id` is the argument passed into the method. Likewise for `name`.

INSTANCES

Let's add that into our class definition.

```
class Student {
    // ...

    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    // ...
}
```

INSTANCES

Now, whenever we want to instantiate an object of the `Student` class, we just need to call its constructor:

```
new Student(123, “Charlie”);
```

We have our first object!

UNDERSTANDING OBJECTS

When we say

```
int i = 4;
```

What do we actually mean?

UNDERSTANDING OBJECTS

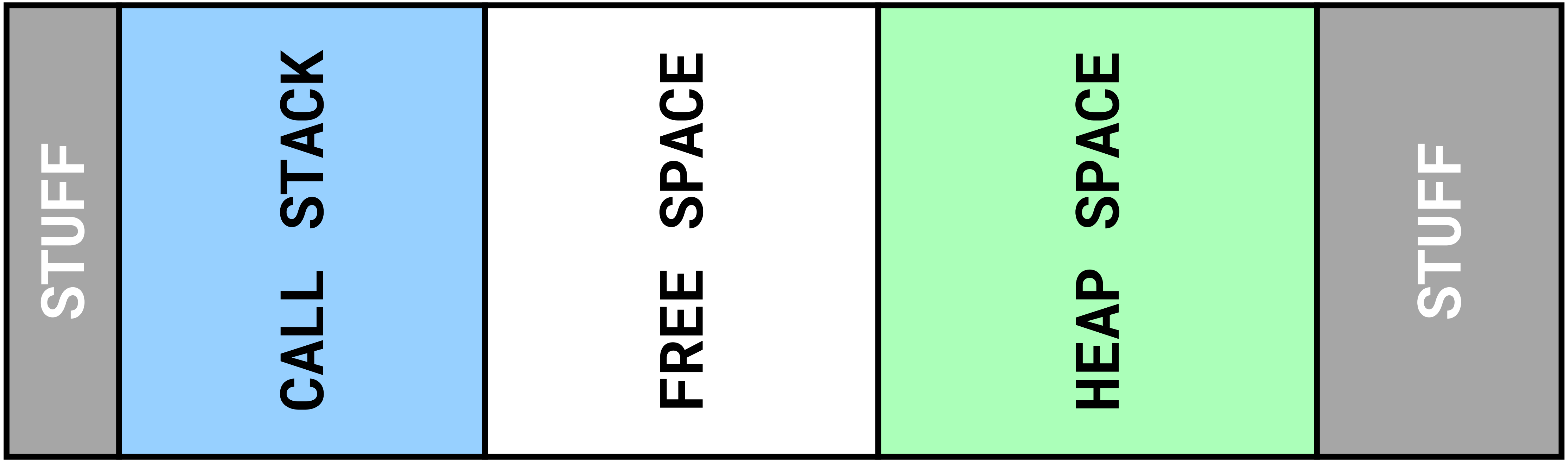
```
int i = 4;
```

in plain English it translates to:

reserve some memory to store the value of `4` as an `int`. This block of memory holds our variable, and we are calling it `i`.

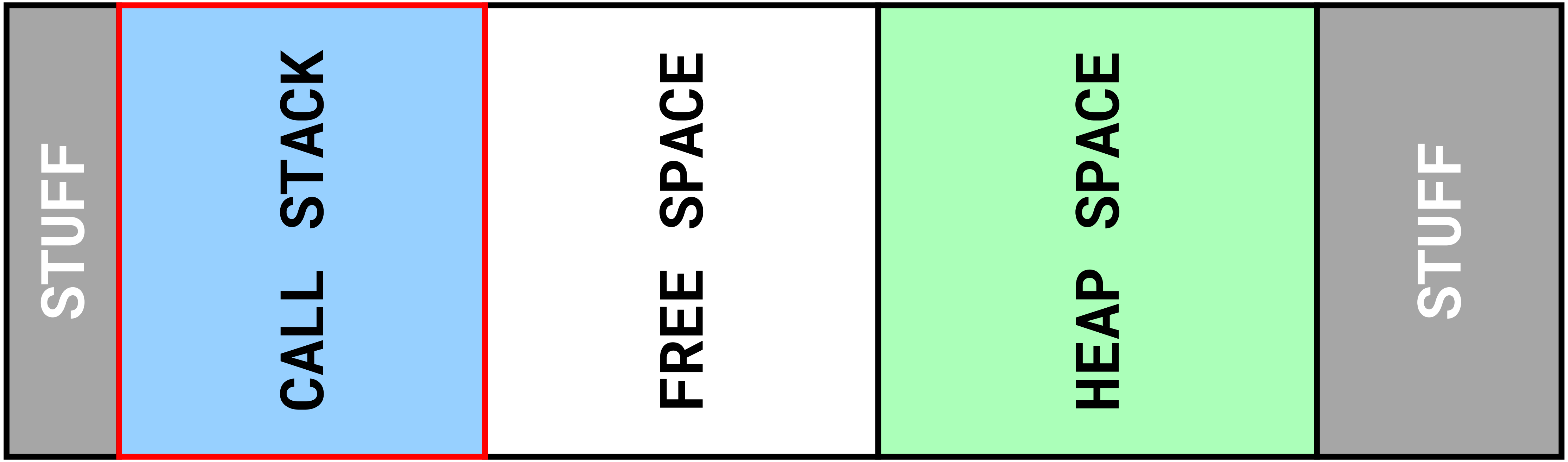
UNDERSTANDING OBJECTS

The statement will make some changes in memory. We will illustrate roughly what happens to help you understand better.



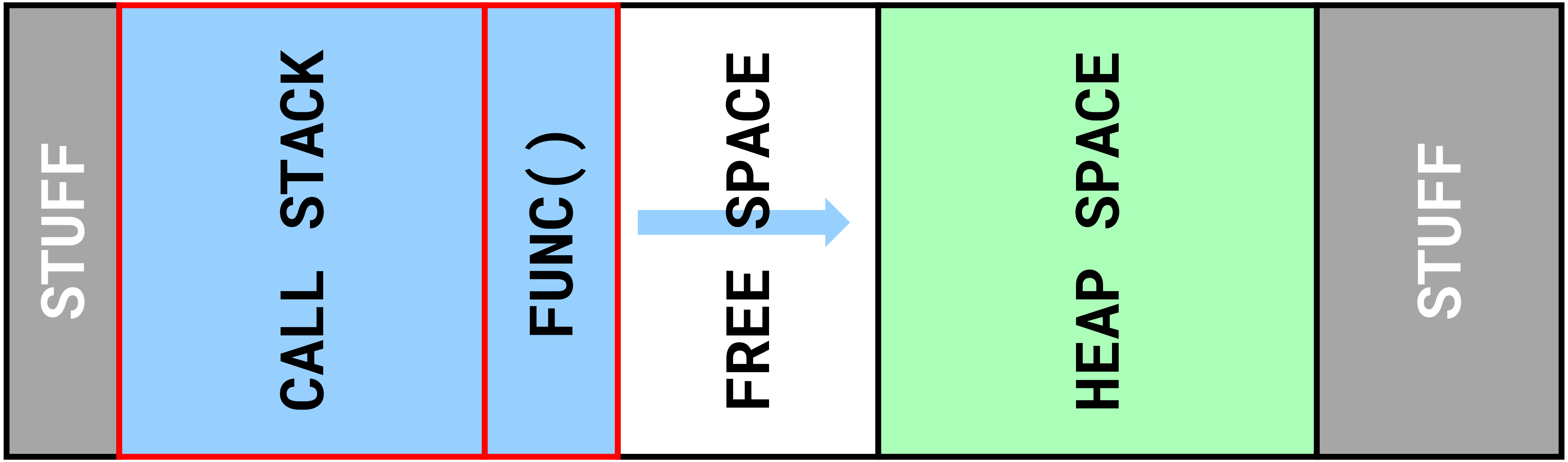
UNDERSTANDING OBJECTS

This is what our program looks like in memory. The call stack is used for local variables in method/function calls.



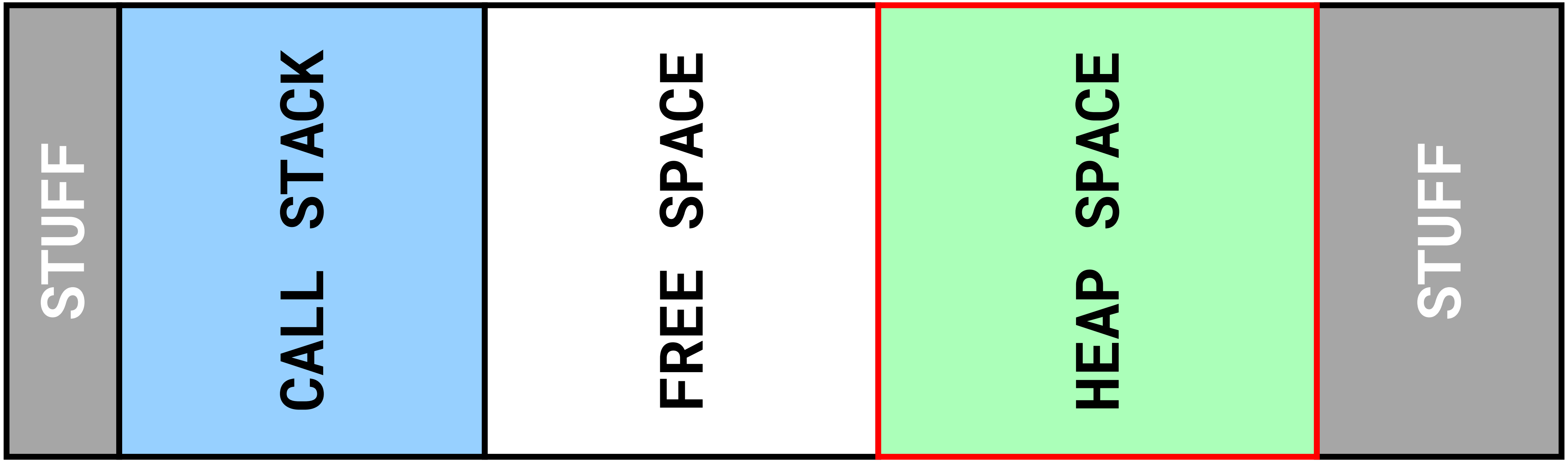
UNDERSTANDING OBJECTS

When another method/function is called, a stack frame is pushed. That frame is popped off from the stack once it is returned.



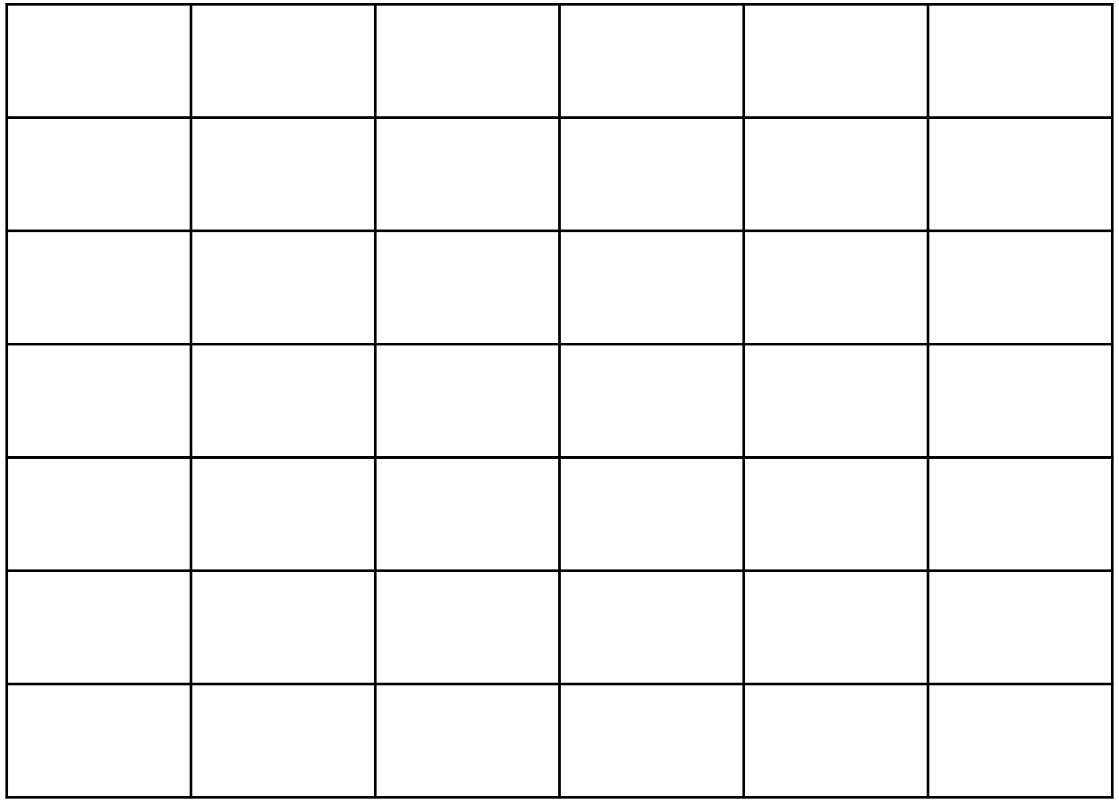
UNDERSTANDING OBJECTS

The heap is used to store global variables and in our case, objects and instance variables. We will touch more on objects later.



UNDERSTANDING OBJECTS

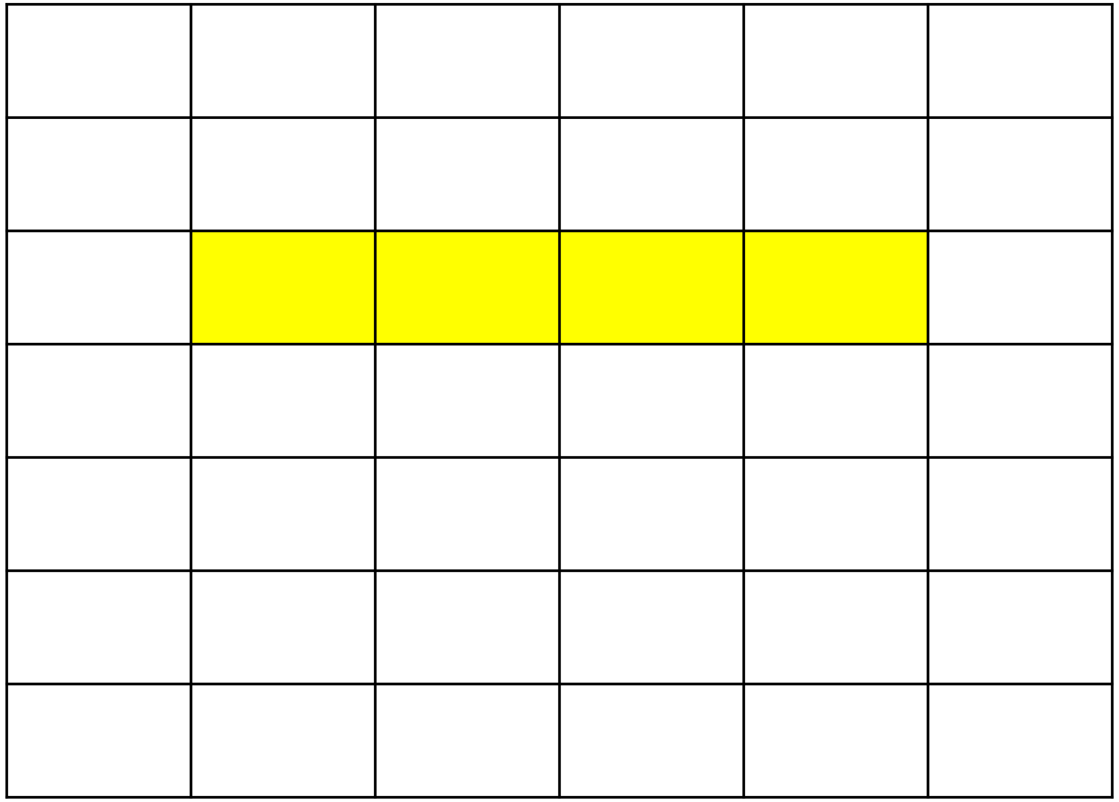
For now, imagine a single stack frame in the call stack, with each block representing 8 bits.



I know this is not very accurate but for the sake of understanding let's use this analogy first.

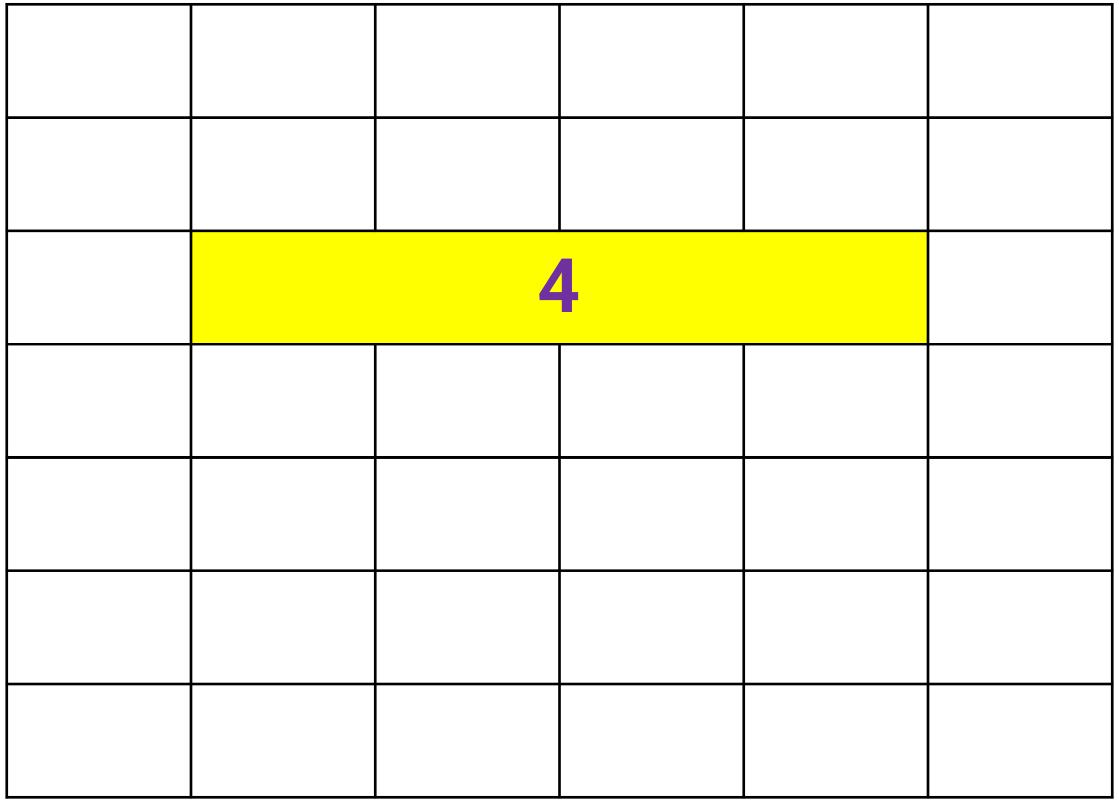
UNDERSTANDING OBJECTS

When we say `int i`, we reserve some memory to hold an integer.



UNDERSTANDING OBJECTS

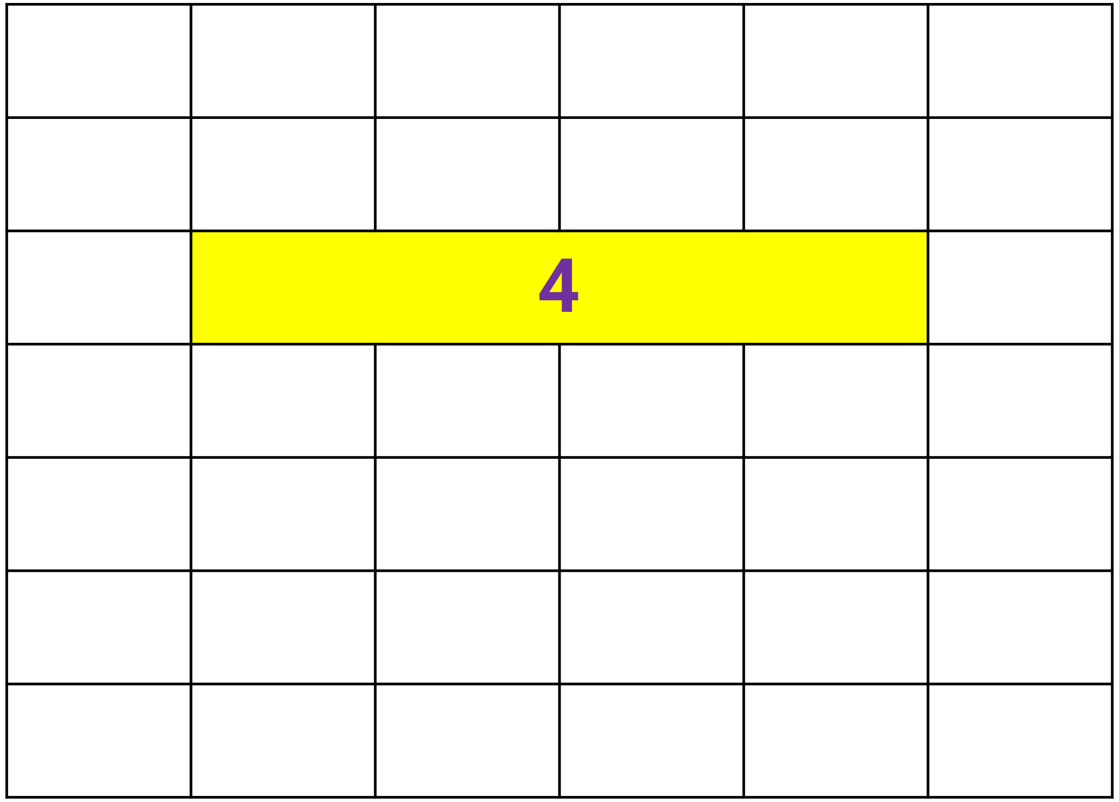
When we assign the value of 4 into `i`, this block of memory will literally store the value of 4.



It actually stores the bit pattern of 4, i.e. 00000000000000000000000000000100

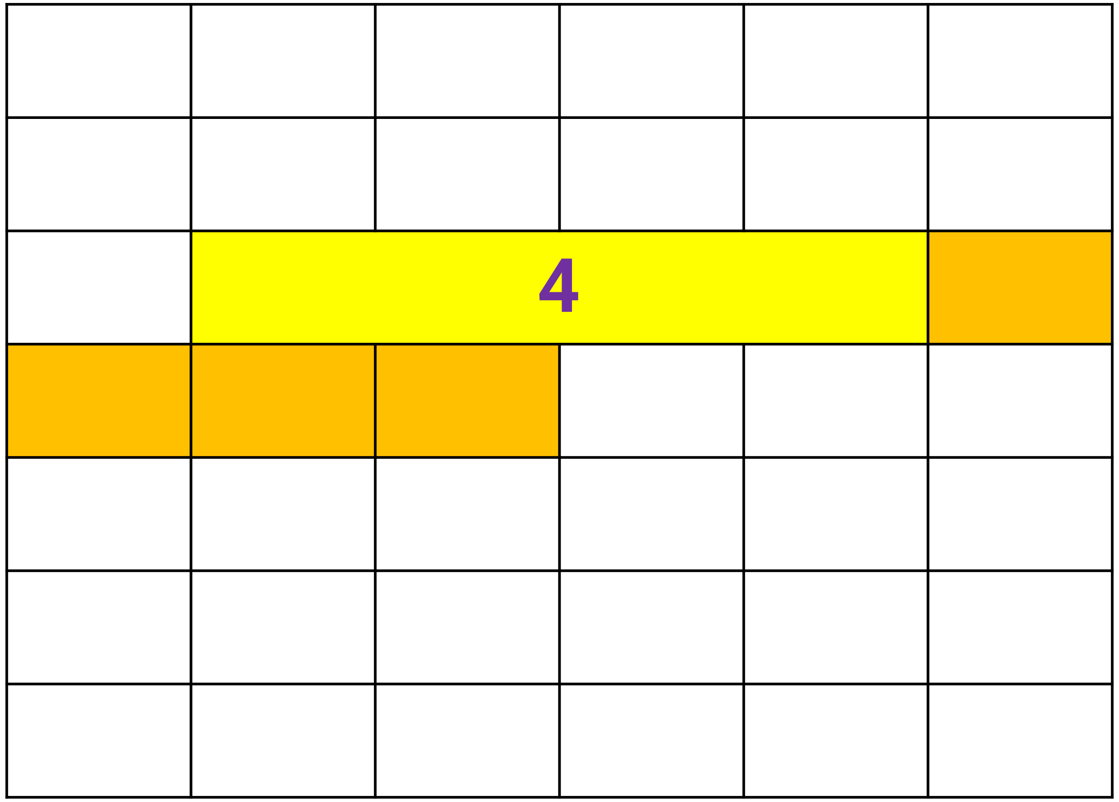
UNDERSTANDING OBJECTS

Let's say we now have `int j = i;`



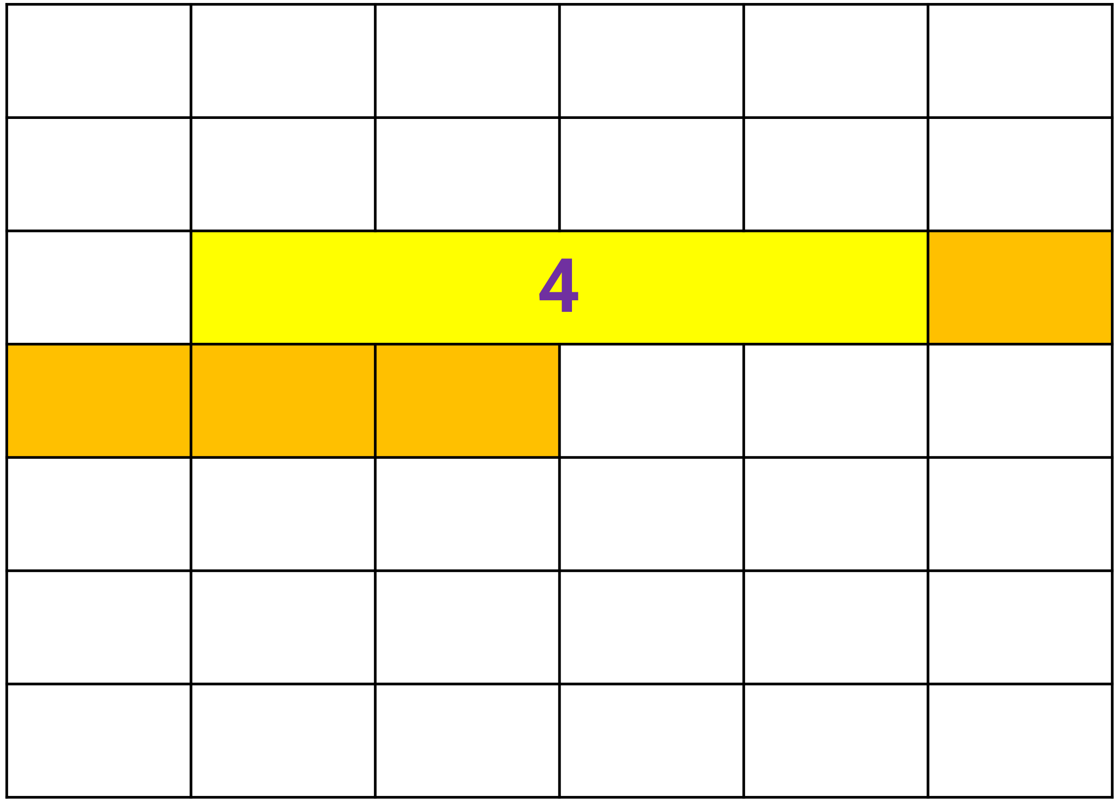
UNDERSTANDING OBJECTS

Similarly, we reserve some memory for `j`



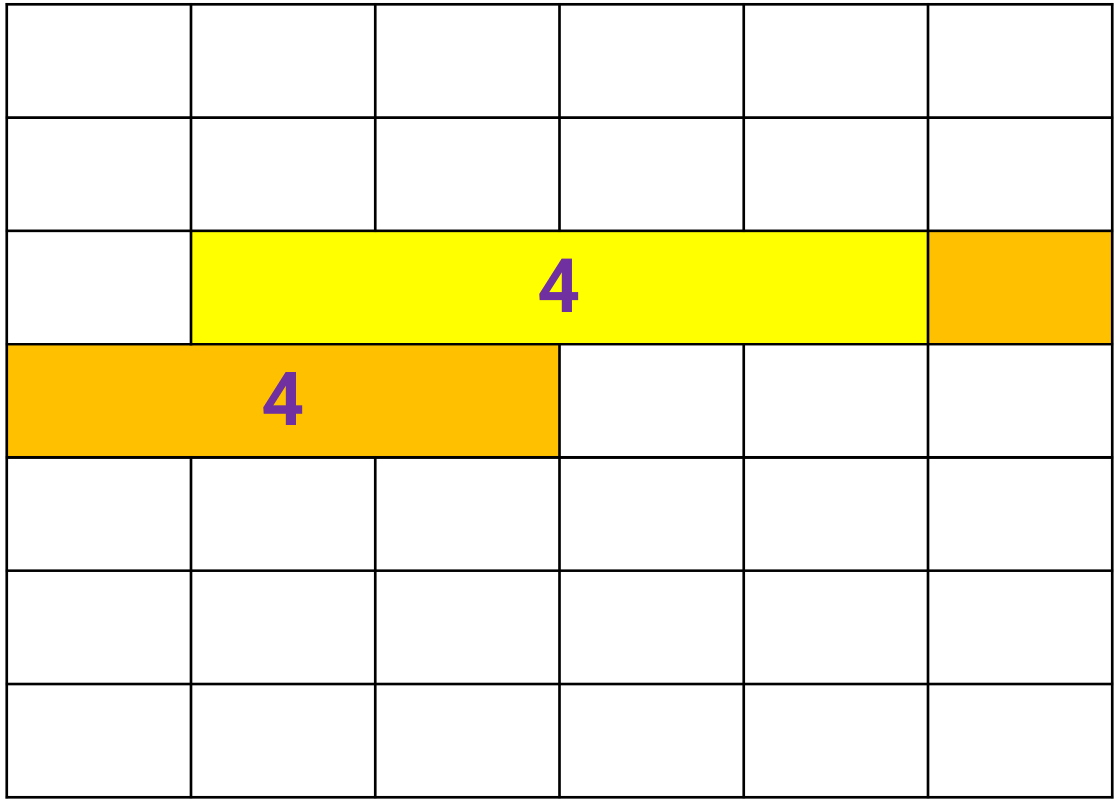
UNDERSTANDING OBJECTS

And store the value of `i` into `j`.



UNDERSTANDING OBJECTS

Which stores 4 into `j`.



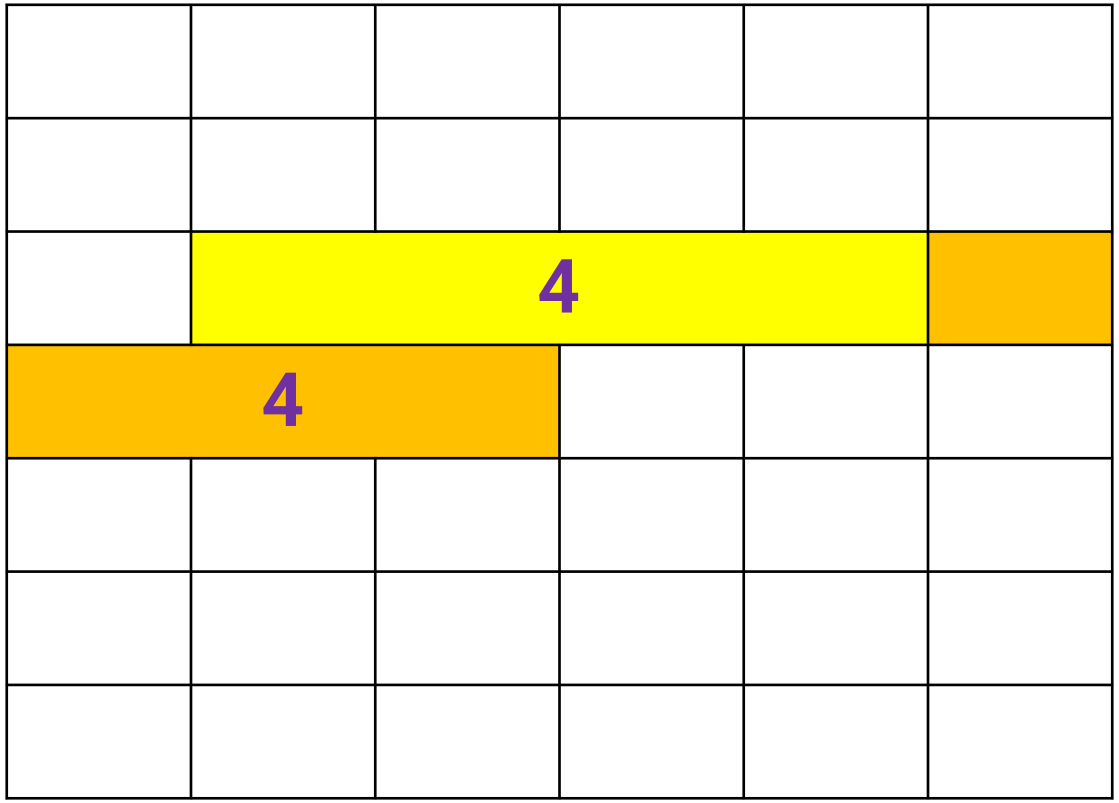
UNDERSTANDING OBJECTS

What is the value of `j` at the end of these statements?

```
int i = 4;
int j = i;
i++;
```

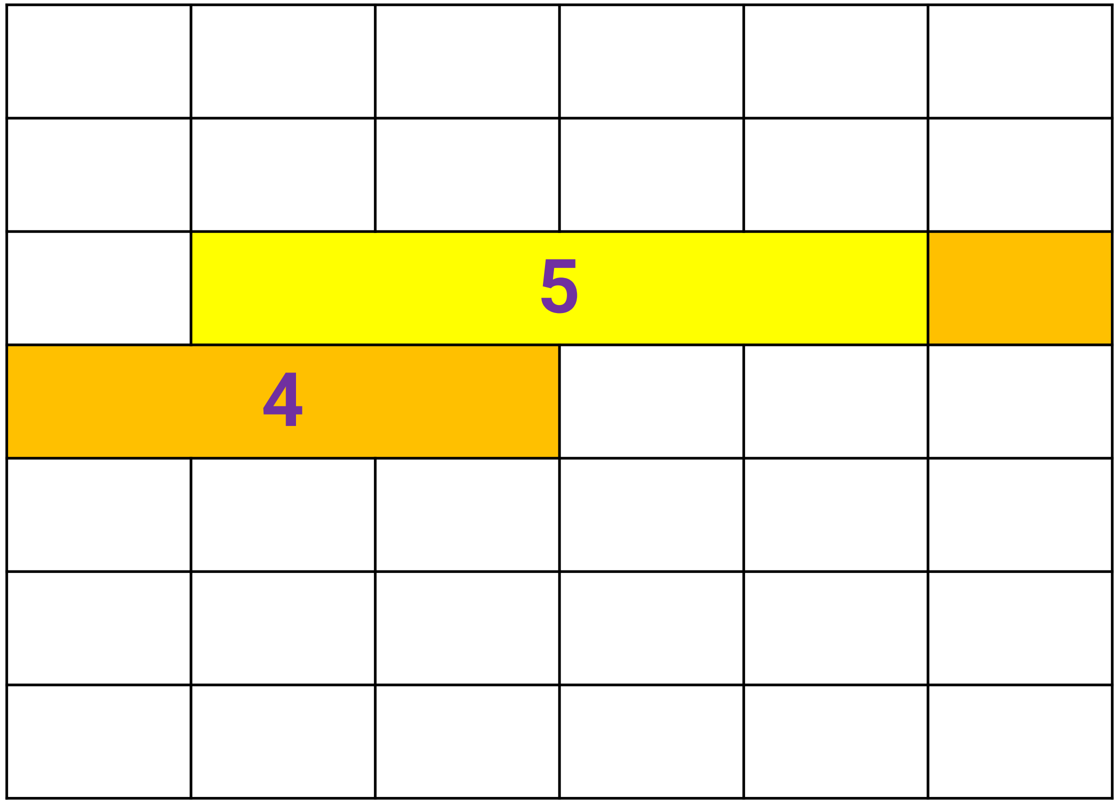
UNDERSTANDING OBJECTS

After the first two statements, we have this, just as before.



UNDERSTANDING OBJECTS

Now we increment the value of `i` by 1.

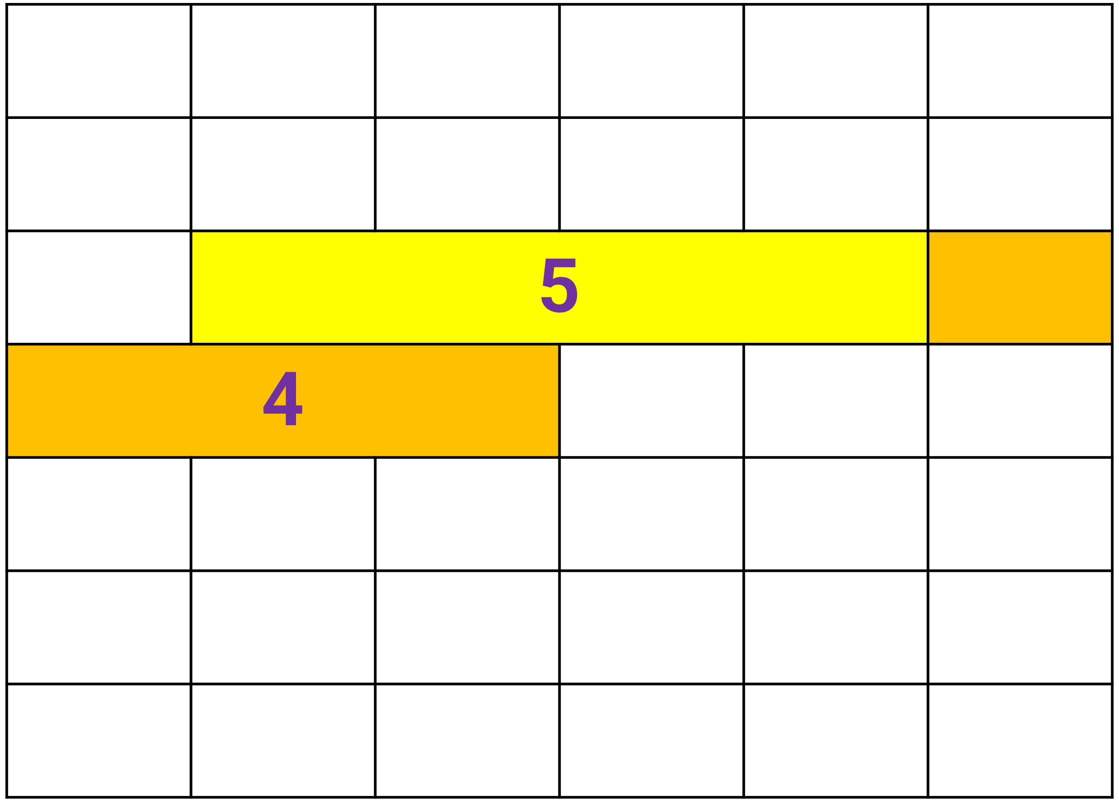


UNDERSTANDING OBJECTS

What is the value of `j`?

UNDERSTANDING OBJECTS

What is the value of `j`?



UNDERSTANDING OBJECTS

What is the value of `j`?

The value of `j` still remains as 4.

UNDERSTANDING OBJECTS

This is (roughly) what happens for primitive variable types, i.e. `short`, `int`, `long`, `float`, `double`, `boolean`, `char` and `byte`.

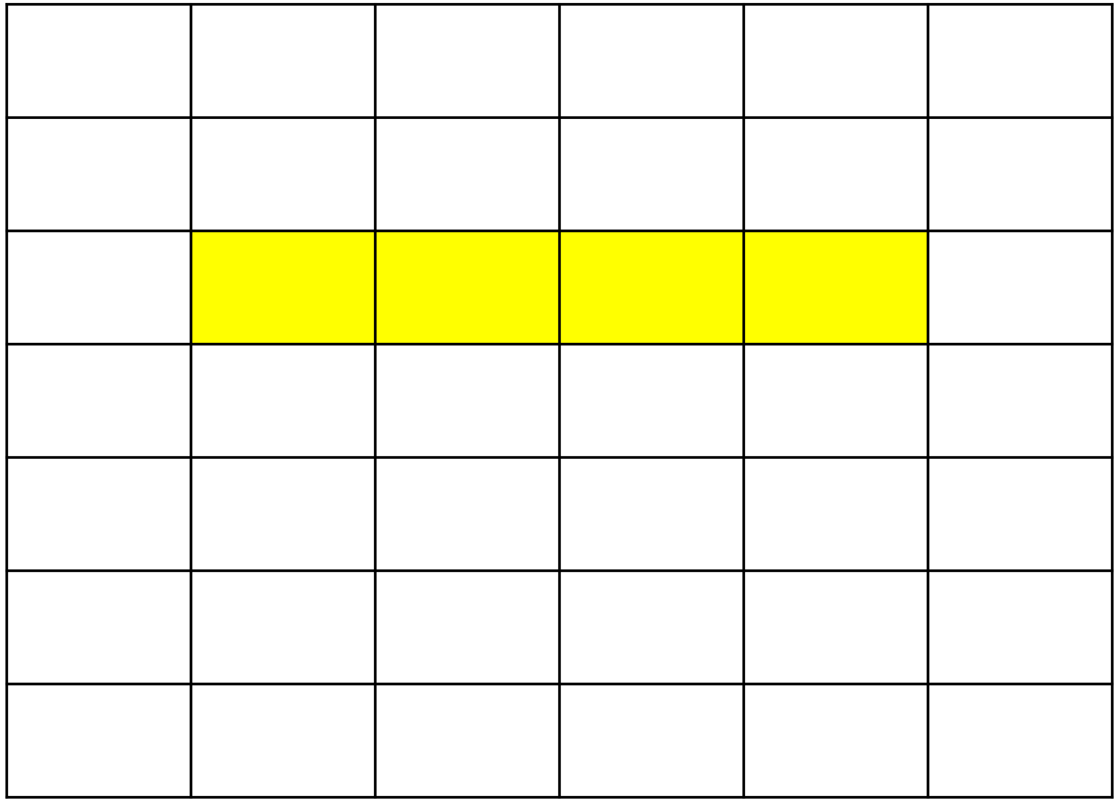
What about objects?

What happens when we have

```
Student s;
```

UNDERSTANDING OBJECTS

When we declare `Student s`, we are reserving some memory to store a **reference** to an instance of `Student`.



UNDERSTANDING OBJECTS

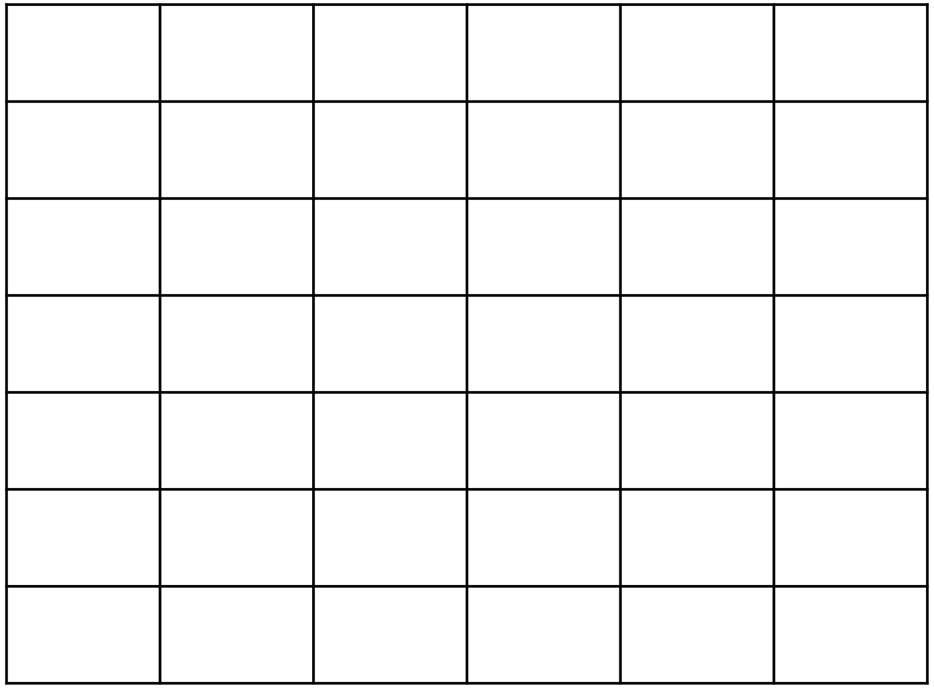
What happens when we have

```
Student s = new Student(1, “Bob”);
```

UNDERSTANDING OBJECTS

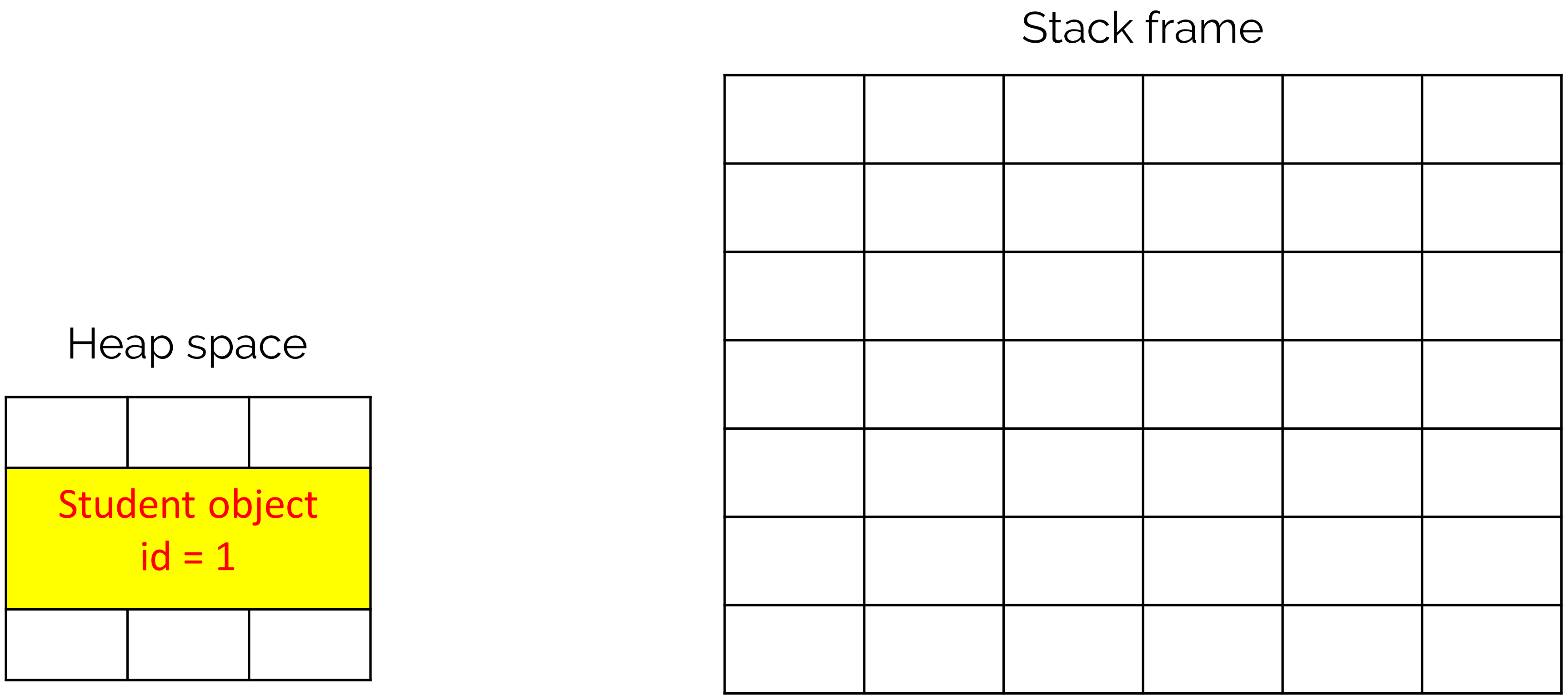
Back to same computer.

Stack frame



UNDERSTANDING OBJECTS

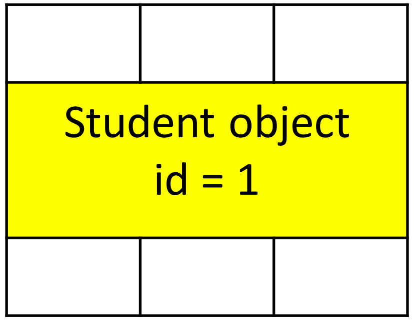
When we say `new Student(1, "Bob");`, we create an instance of the `Student` class in the **heap space**.

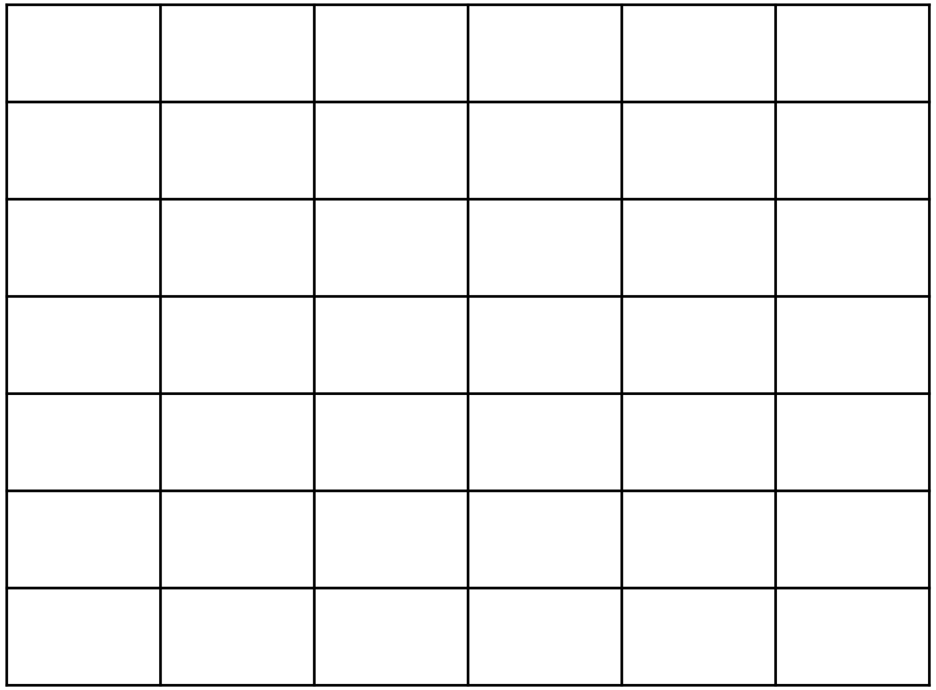


UNDERSTANDING OBJECTS

This object has a reference, which is its location in the heap space.

Heap space

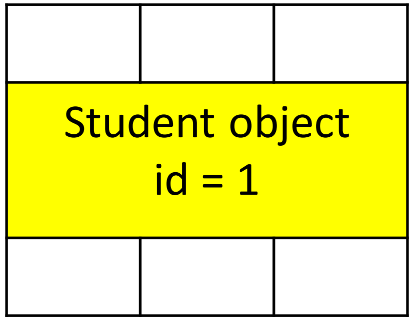


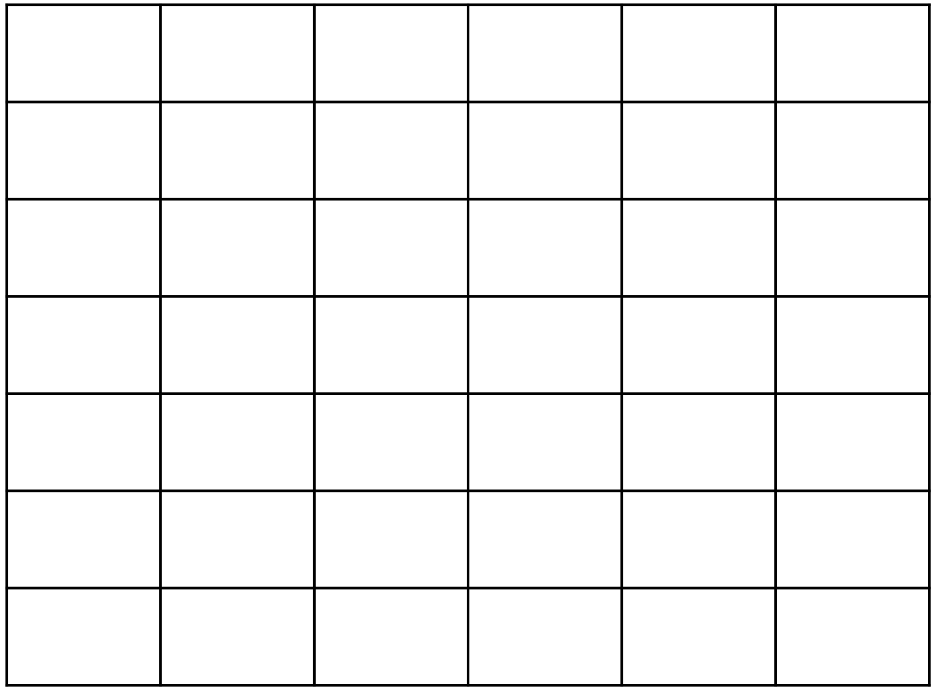


UNDERSTANDING OBJECTS

The evaluated value of the constructor is the reference to that object, in this case, let's say its 12345678???.

Heap space





UNDERSTANDING OBJECTS

Now, we are assigning the reference to that object into `s`.

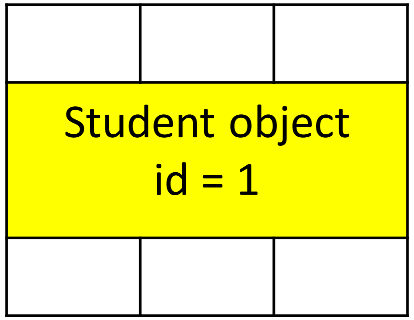
```
Student s = new Student(1, “Bob”);
```

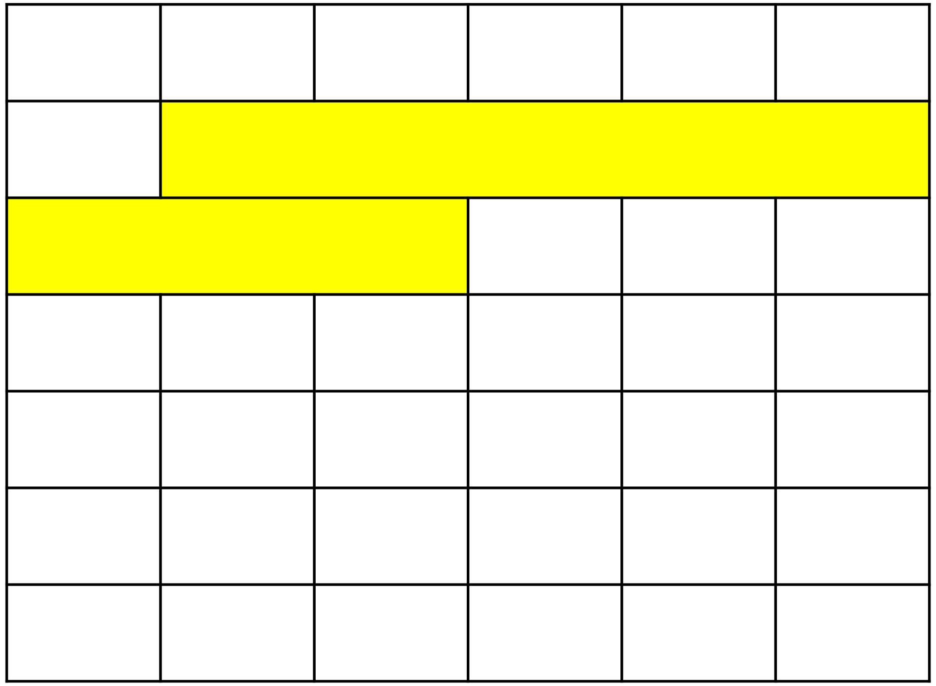
```
Student s = Student@12345678???;
```

UNDERSTANDING OBJECTS

We leave some memory for the variable **s**, that stores a **reference** to a Student instance.

Heap space

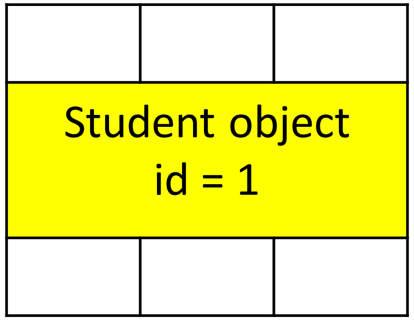


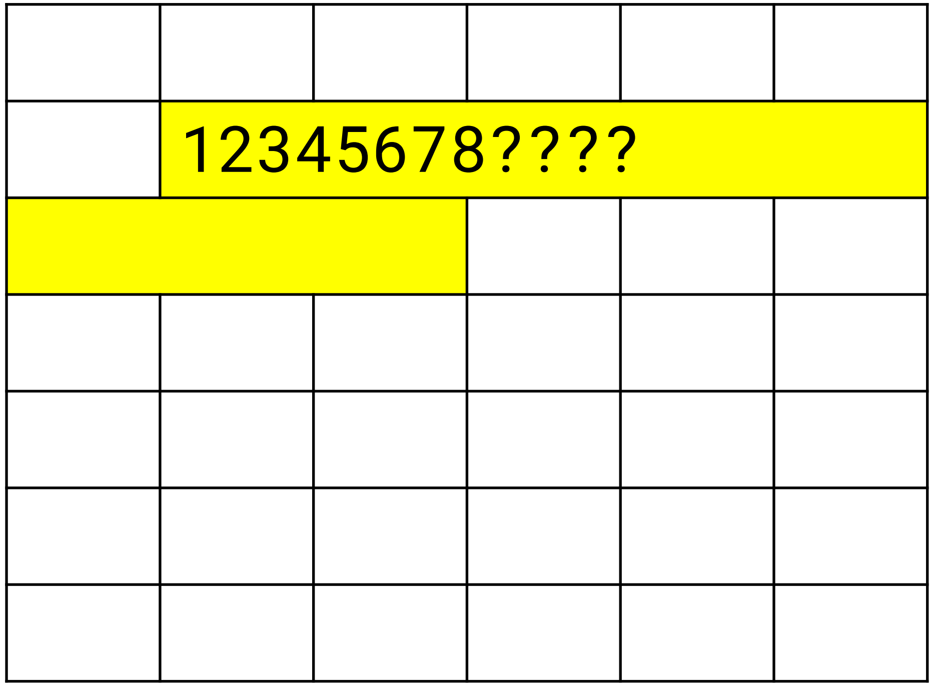


UNDERSTANDING OBJECTS

We then store the reference of this object into variable s.

Heap space





UNDERSTANDING OBJECTS

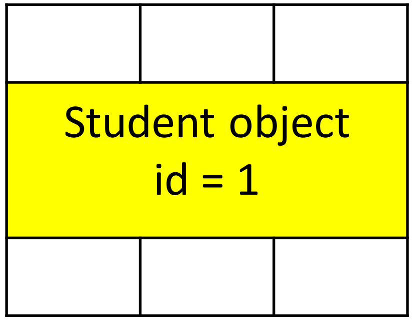
Now, say we have

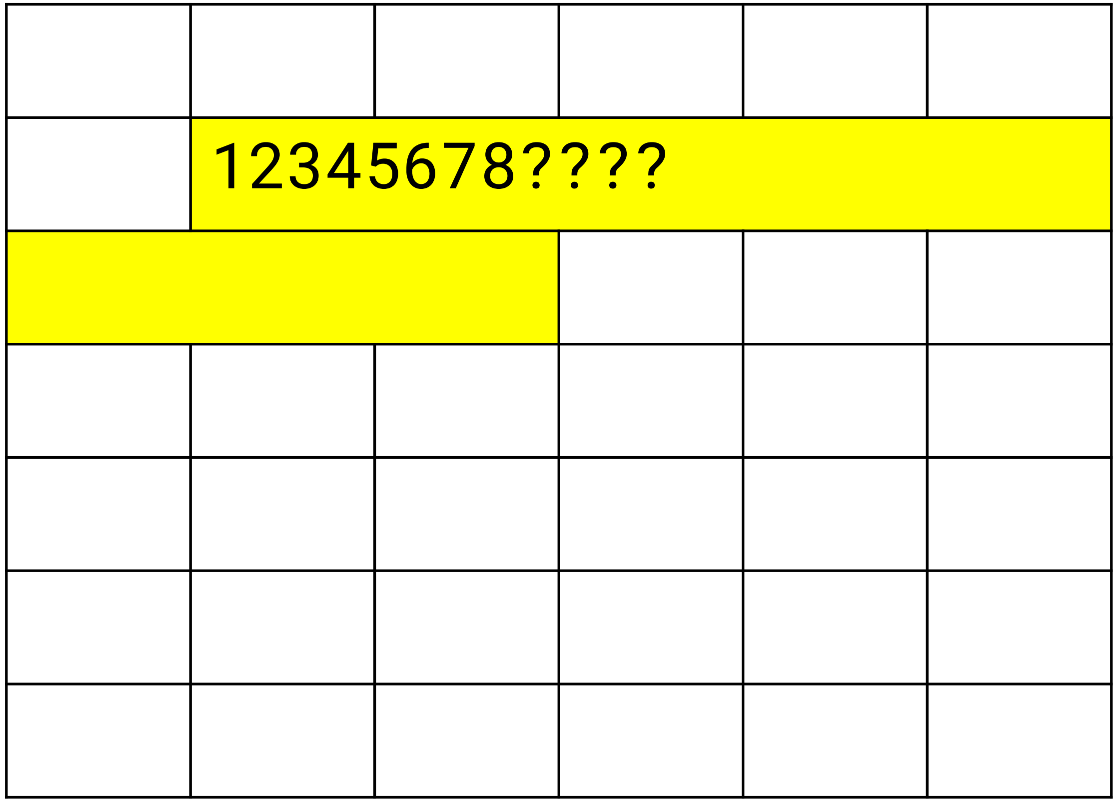
```
Student s = new Student(1, “Bob”);
Student t = s;
```

UNDERSTANDING OBJECTS

In the second line, we don't have the new keyword: no new objects are created.

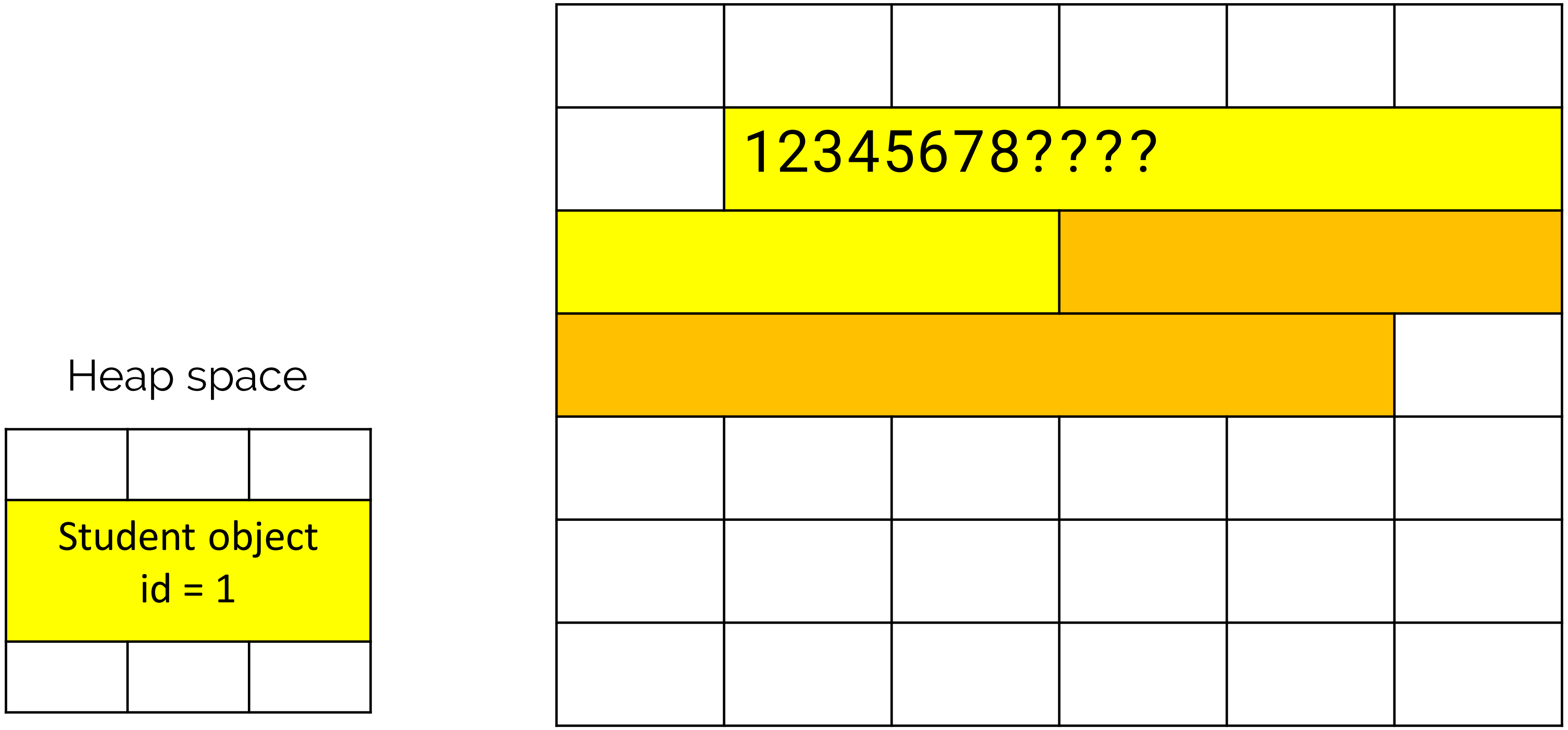
Heap space





UNDERSTANDING OBJECTS

We create `t`, which stores a reference to an instance of `Student`



UNDERSTANDING OBJECTS

`s` evaluates to the reference of the object we just created.

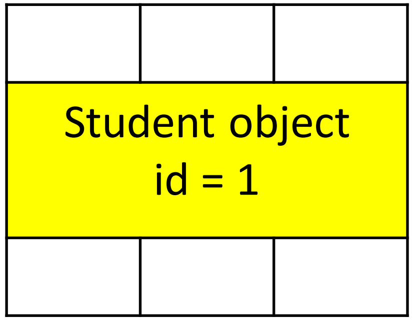
```
Student t = s;
```

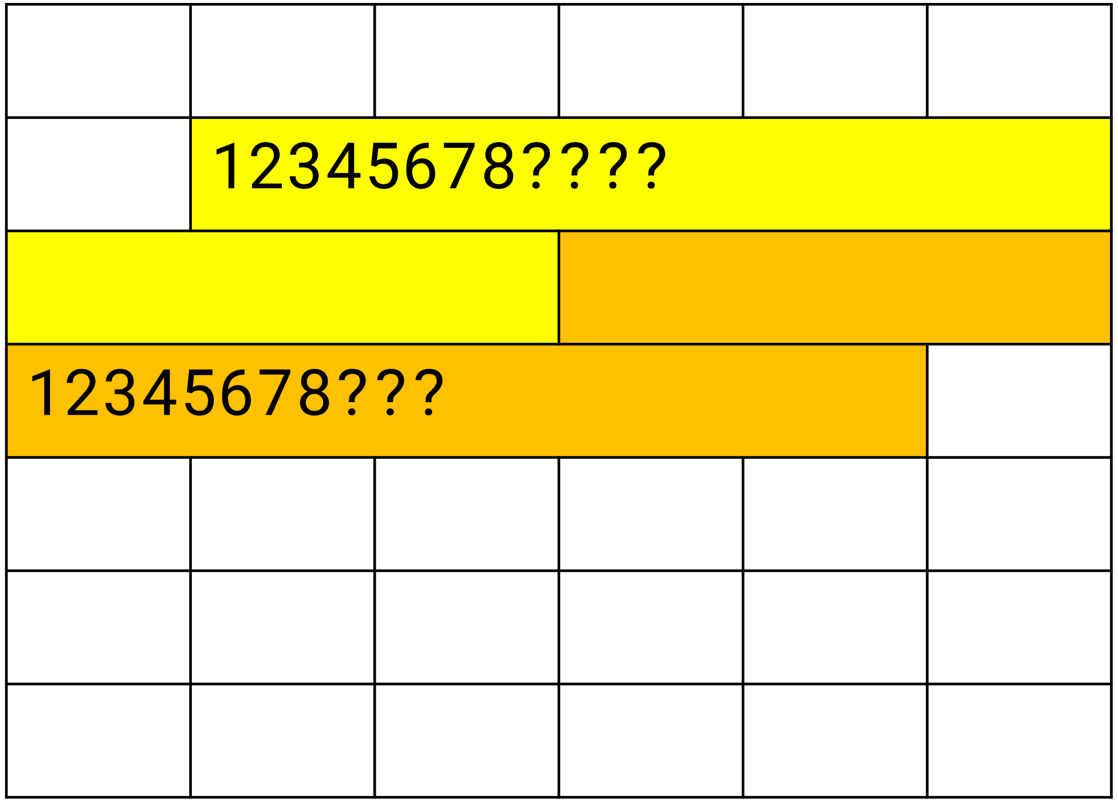
```
Student t = Student@12345678???;
```

UNDERSTANDING OBJECTS

We assign the reference stored in `s` into `t`.

Heap space





UNDERSTANDING OBJECTS

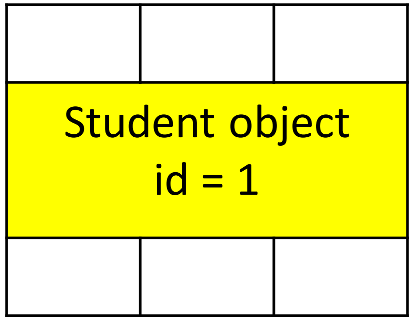
Now, say we have

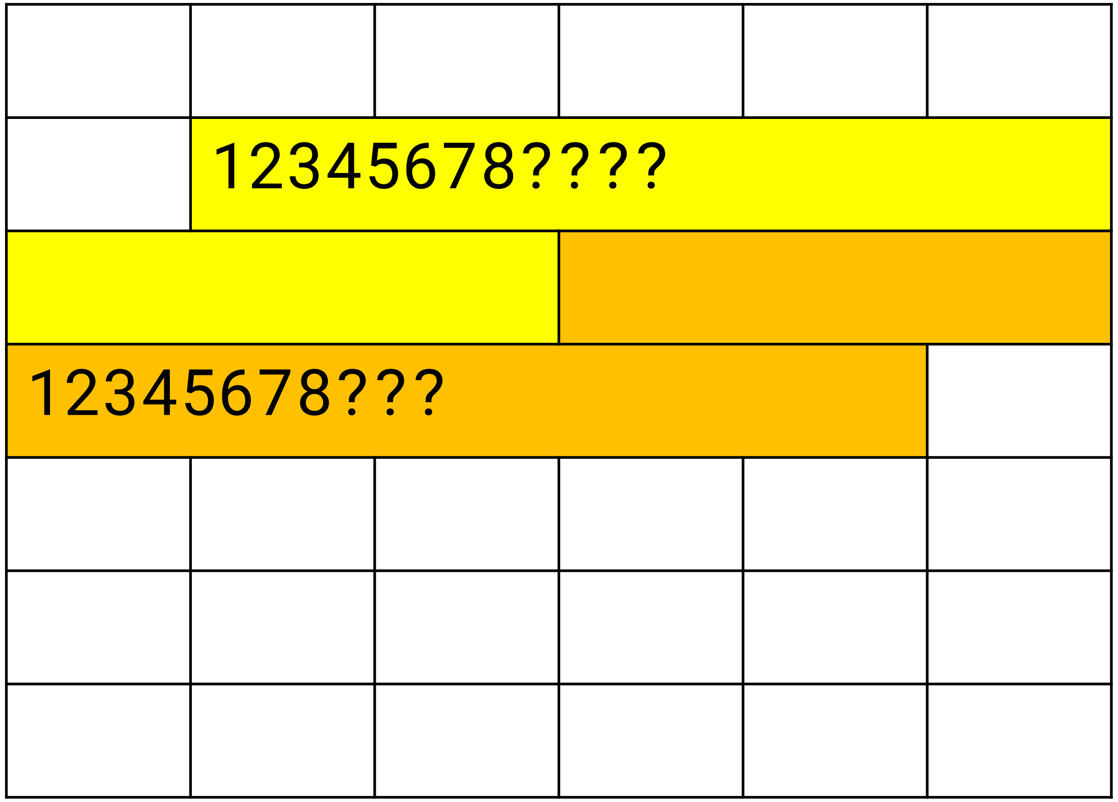
```
Student s = new Student(1, “Bob”);
Student t = s;
s.id = 2;
System.out.println(t.id);
```

UNDERSTANDING OBJECTS

We find the object stored in the reference of s.

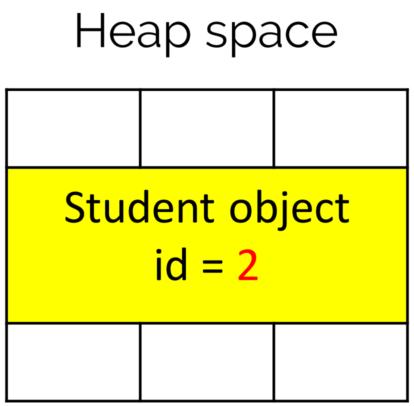
Heap space

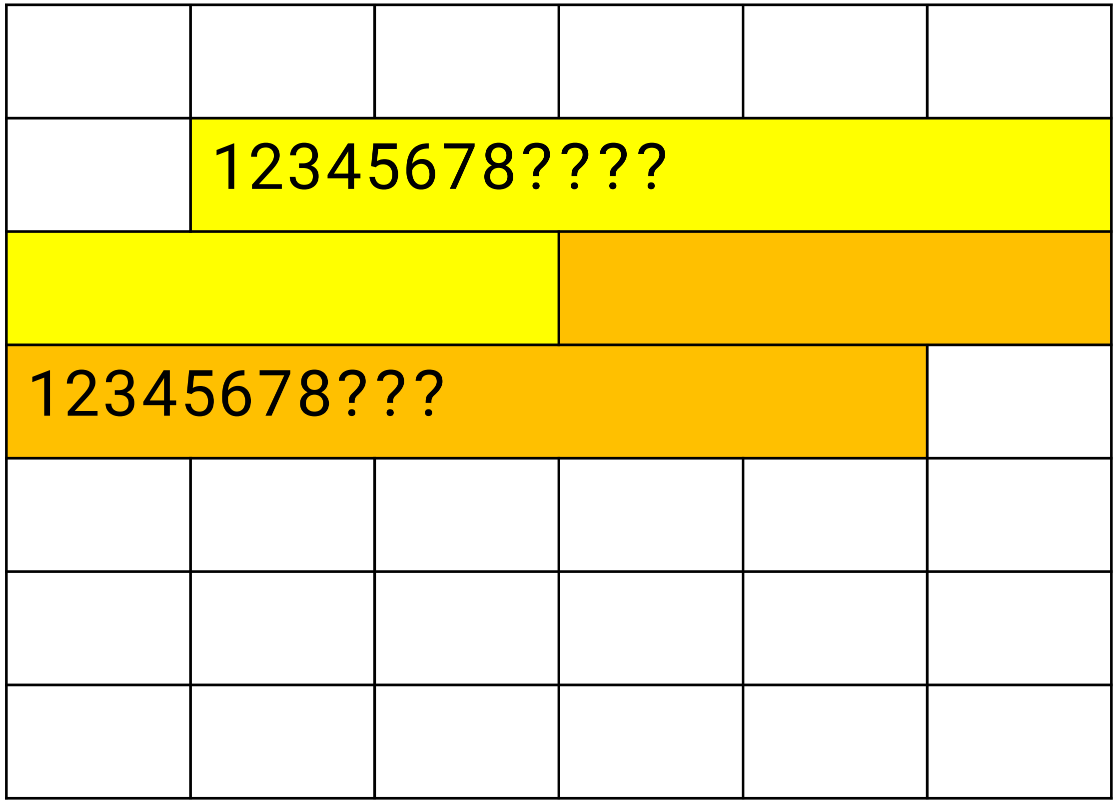




UNDERSTANDING OBJECTS

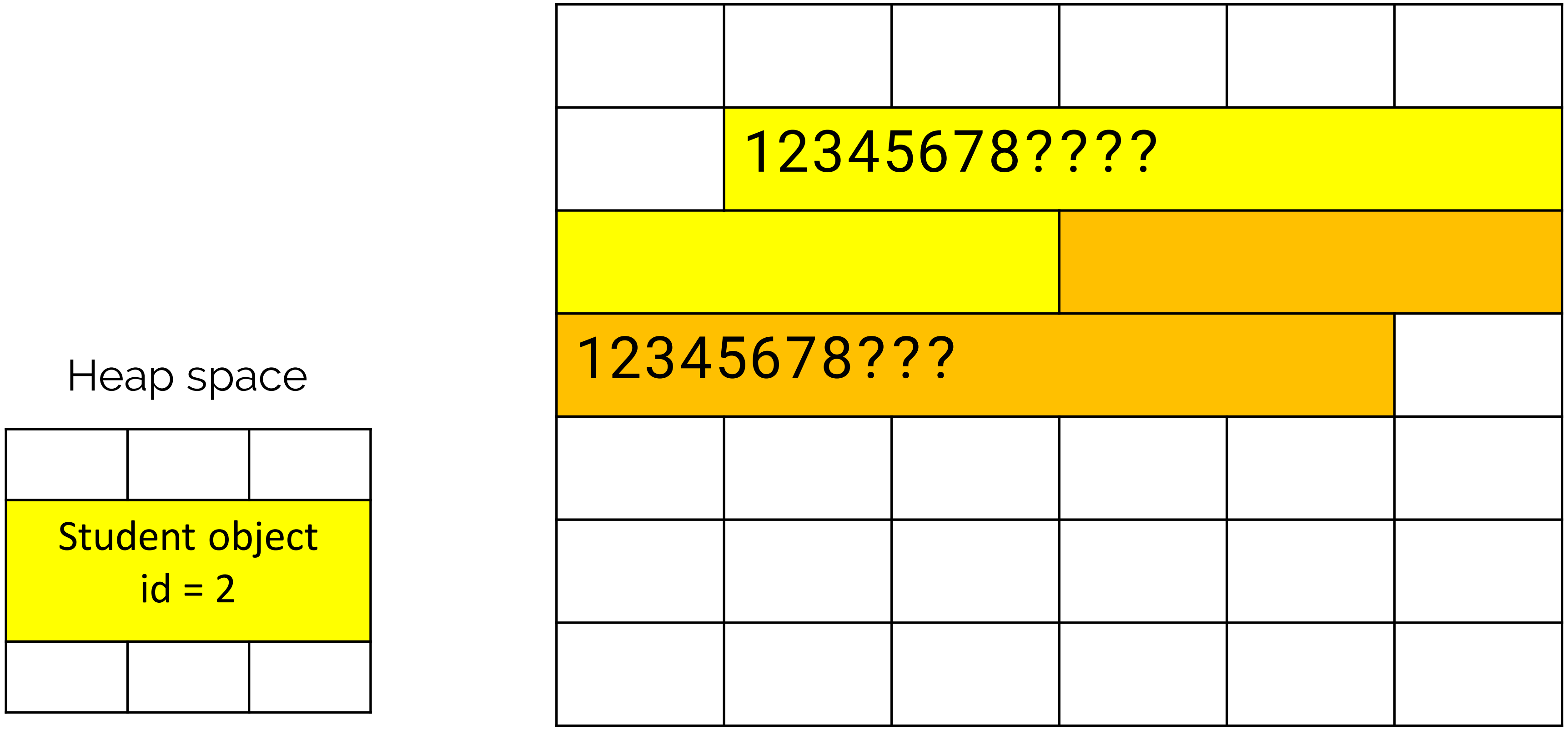
We set it's `id` to be 2





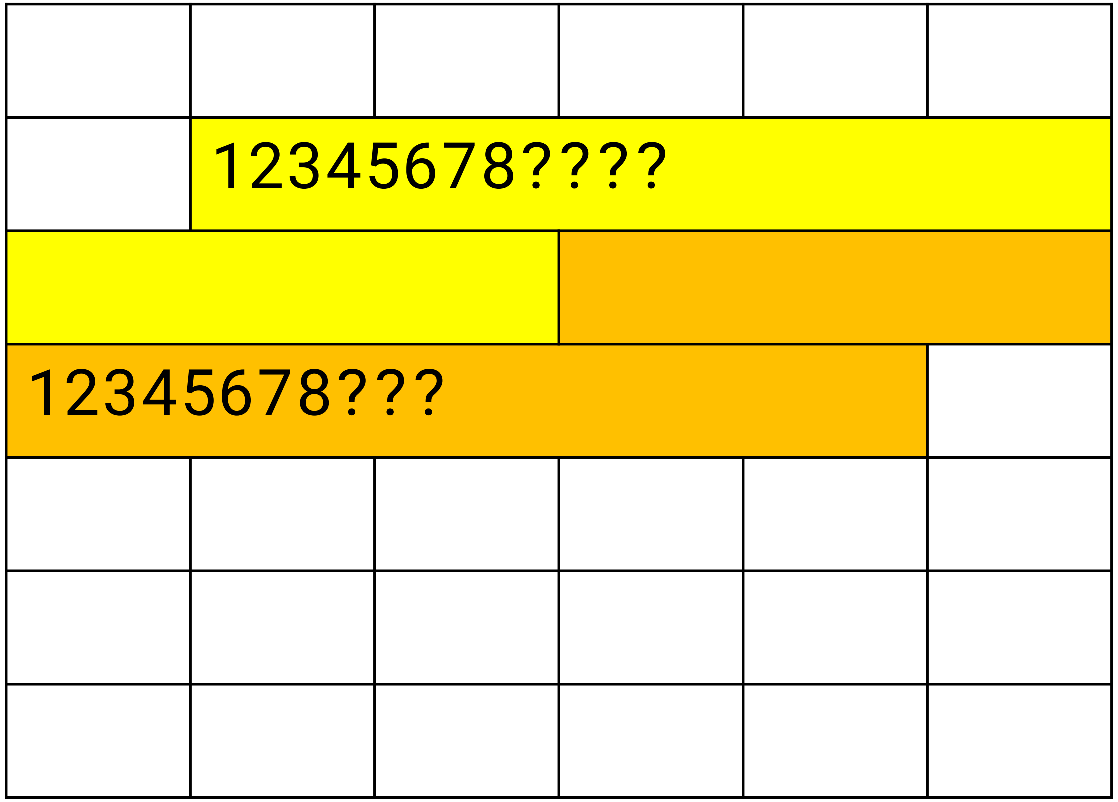
UNDERSTANDING OBJECTS

Now, we find the object stored in the reference of `t` (same object as `s`!).

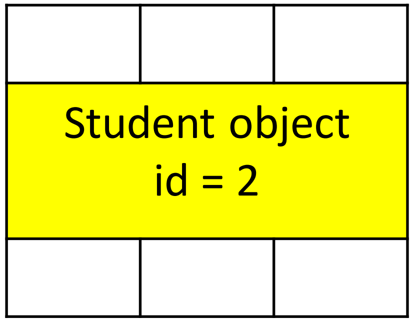


UNDERSTANDING OBJECTS

And we print it’s `id` out.



Heap space



UNDERSTANDING OBJECTS

```
Student s = new Student(1, “Bob”);
Student t = s;
s.id = 2;
System.out.println(t.id);

Output:
2
```

UNDERSTANDING OBJECTS

What about arrays?

Say we have `int[] arr = new int[3];`

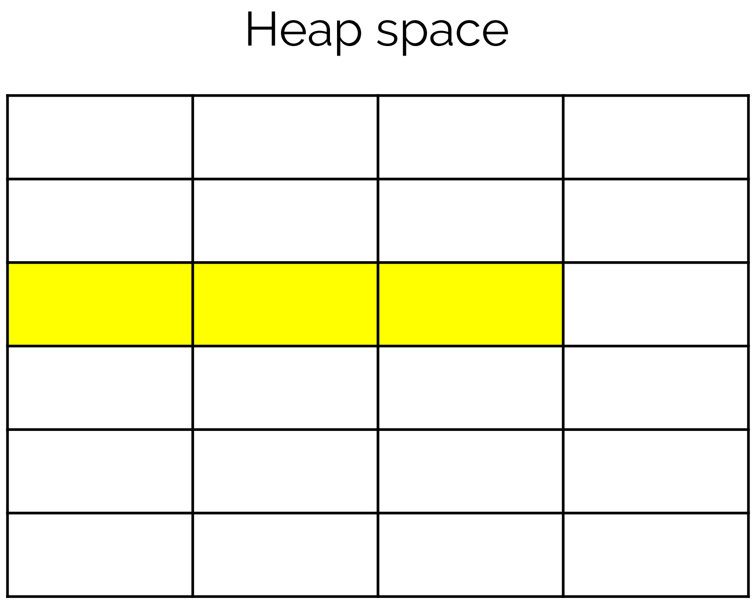
What happens?

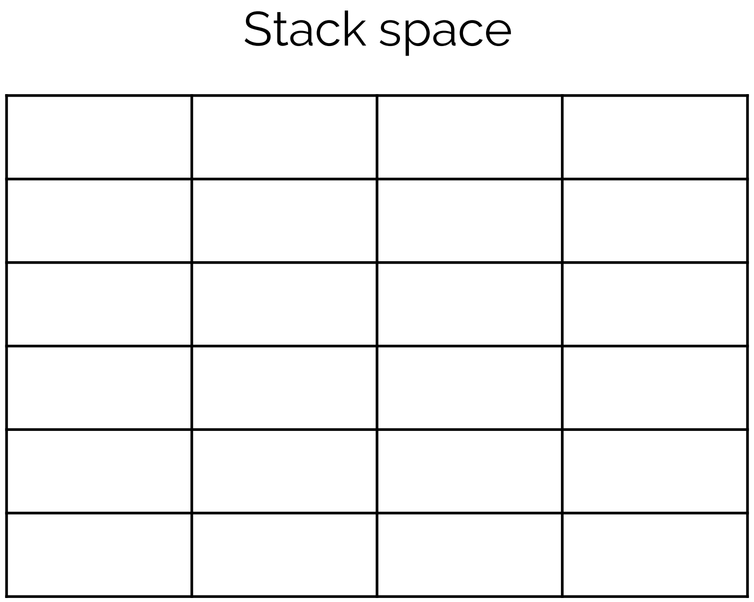
UNDERSTANDING OBJECTS

Arrays are objects as well. When we create a one-dimensional array, it is created in the heap space as well. Let’s have a look once again.

UNDERSTANDING OBJECTS

`new int[3]` reserves three blocks of memory of integers in the heap space for the array object.

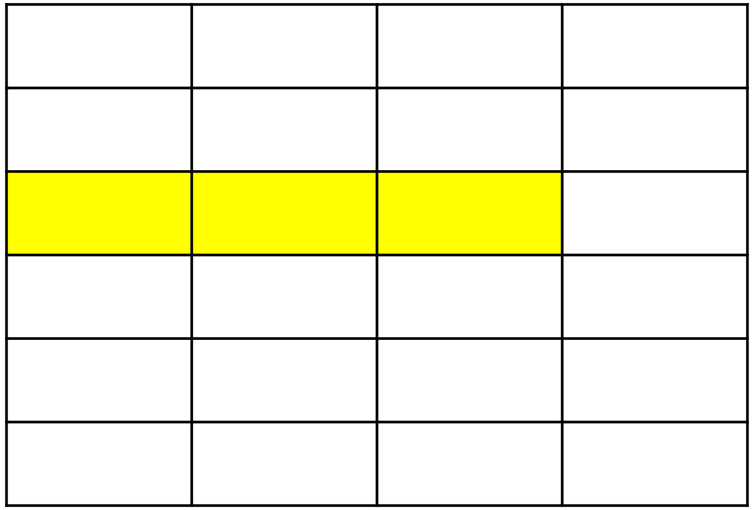




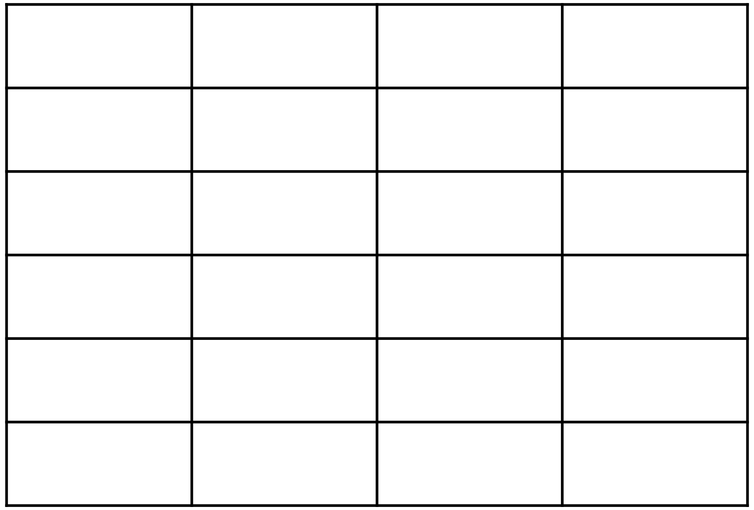
UNDERSTANDING OBJECTS

This array has a reference. The constructor of the array evaluates to its reference, say 87654321???.

Heap space



Stack space



UNDERSTANDING OBJECTS

```
int[] arr = new int[3];
```

```
int[] arr = int[]@87654321???;
```

UNDERSTANDING OBJECTS

Some memory is allocated in the stack space for the `arr` variable, which holds a reference to an `int[]`.

UNDERSTANDING OBJECTS

The array's reference is stored in that block of memory.

UNDERSTANDING OBJECTS

What about multi-dimensional arrays?

```
int[][] c = new int[][] {{1, 2}, {3, 4}};
```

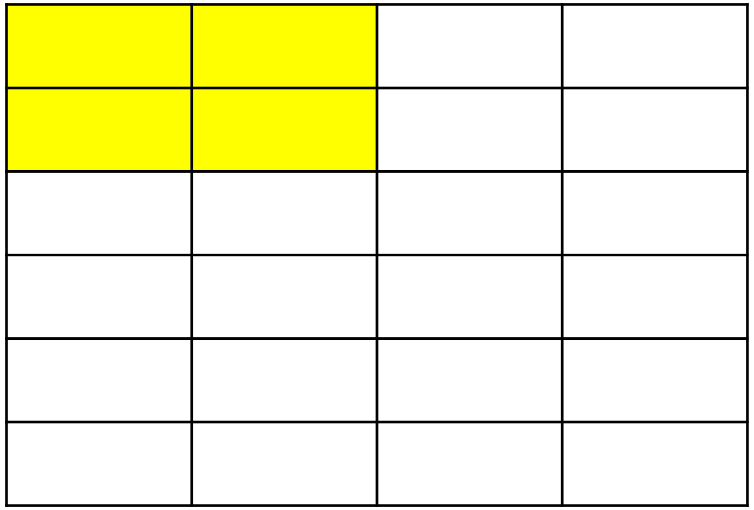
The important thing to note is that there are no two-dimensional arrays in Java.

Two dimensional arrays are simply one-dimensional arrays of multiple one-dimensional arrays.

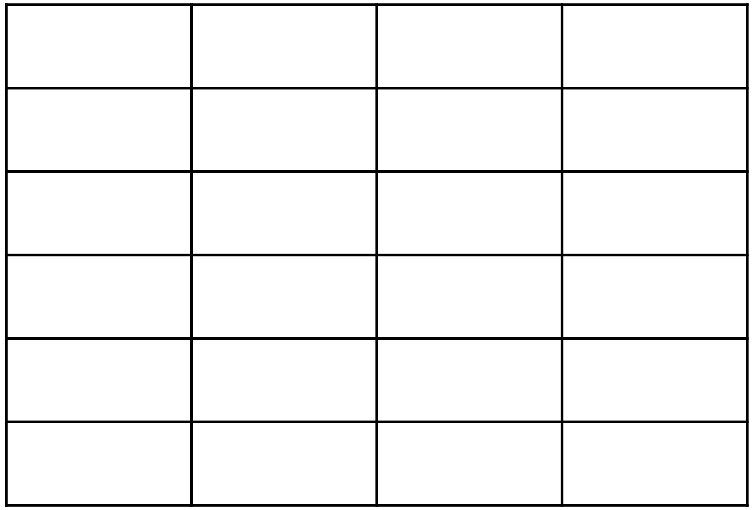
UNDERSTANDING OBJECTS

`new int[][] {{1, 2}, {3, 4}};` first needs to create two `int` arrays.

Heap space



Stack space



UNDERSTANDING OBJECTS

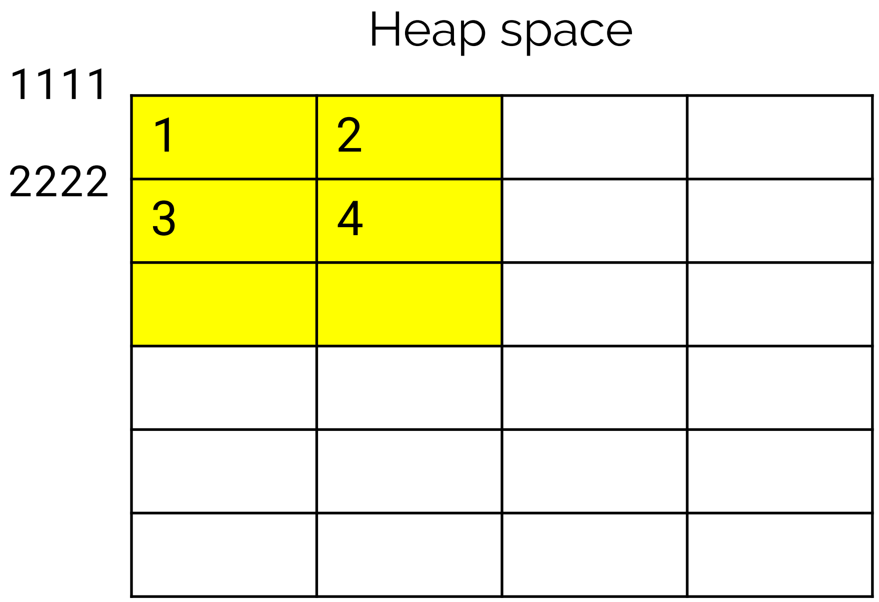
As the elements in the array are primitive types, they are stored as-is in the arrays.

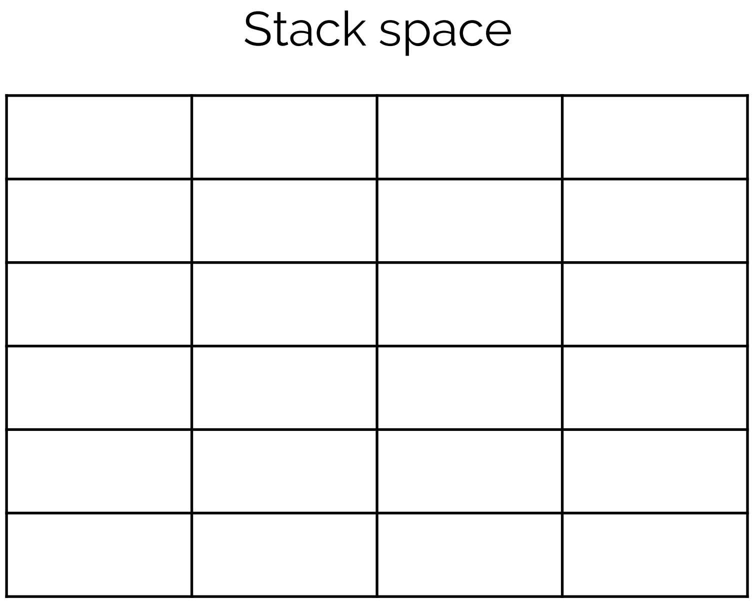
UNDERSTANDING OBJECTS

Each of the individual arrays have references, let's say they are 1111 and 2222.

UNDERSTANDING OBJECTS

A new `int[][]` object is created, which stores references to `int[]`.





UNDERSTANDING OBJECTS

The two array references are stored as elements in the new `int[][]`.

UNDERSTANDING OBJECTS

The `int[][]` has a reference as well. The call of its constructor evaluates to its reference, say 3333.

Heap space

1111	1	2		
2222	3	4		
3333	1111	2222		

Stack space

UNDERSTANDING OBJECTS

```
int[][] c = new int[][] {{1, 2}, {3, 4}};
```

```
int[][] c = int[][]@3333;
```

UNDERSTANDING OBJECTS

Memory is reserved in the stack space to store a reference to an `int[][]` for our variable `c`. The reference of our new `int[][]` object is stored there.

Heap space

1111	1	2		
2222	3	4		
3333	1111	2222		

Stack space

3333			

UNDERSTANDING OBJECTS

What happens here?

```
int[][] c = new int[][] {{1, 2}, {3, 4}};

int[] b = c[0];
b[0] = 5;
```

UNDERSTANDING OBJECTS

Memory is reserved in the stack space to store a reference to an `int[]` for our variable b. The reference stored in the first element of c is stored in b.

Heap space

1111	1	2		
2222	3	4		
3333	1111	2222		

Stack space

3333			
1111			

UNDERSTANDING OBJECTS

We then assign 5 to the first element in the array referred by b.

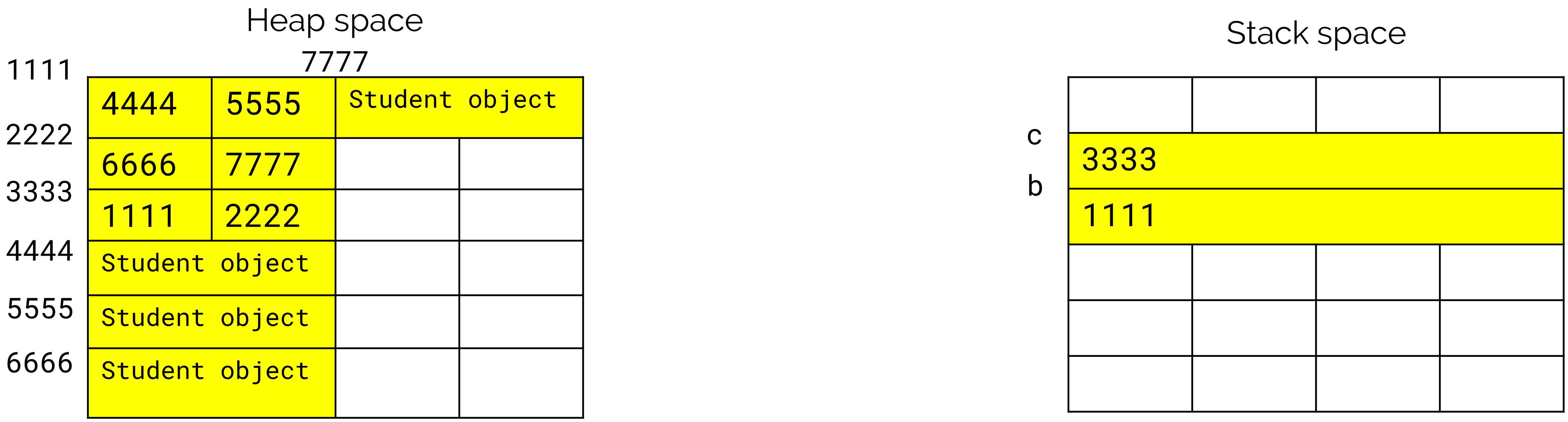
UNDERSTANDING OBJECTS

This means, that `c` is now `{{5, 2}, {3, 4}}`!

```
int[][] c = new int[][] {{1, 2}, {3, 4}};

int[] b = c[0];
b[0] = 5;
```

UNDERSTANDING OBJECTS

Object arrays work in the same way. Except, instead of storing values as-is, the one-dimensional arrays store references to the objects. A `Student[][]` might look like:



UNDERSTANDING OBJECTS

Let's take our understanding of primitive types vs reference types (objects) into lexical scoping.

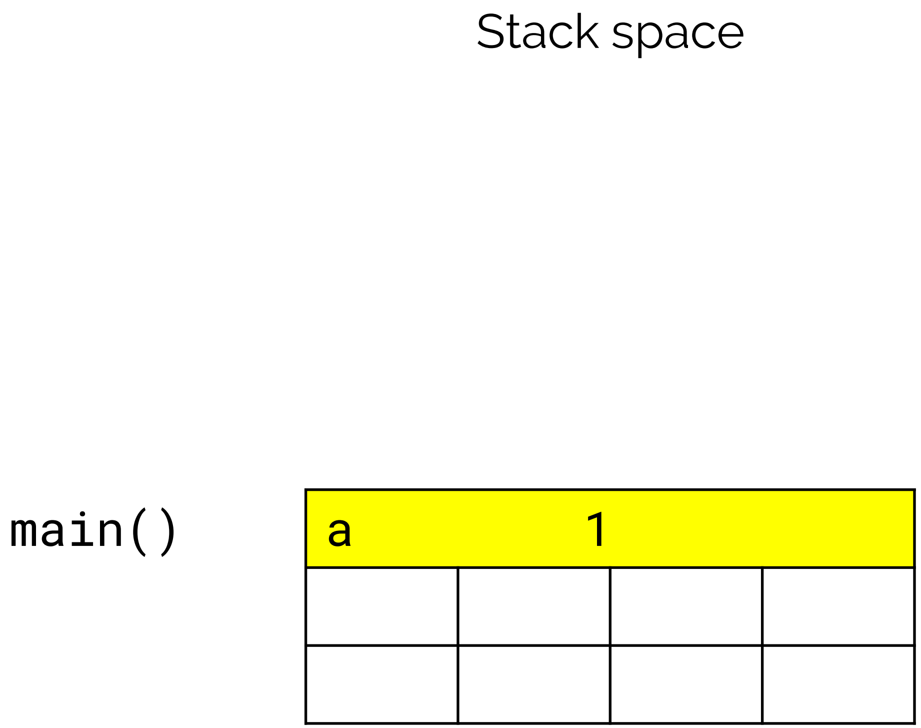
In `main()`:

```
int a = 1;
coolMethod(a);
System.out.println(a);
```

```
void coolMethod(int a) {
    a = 2;
}
```

UNDERSTANDING OBJECTS

In the main stack frame, the variable a is declared and initialised. Because a is a primitive type, the value is stored locally as-is.



UNDERSTANDING OBJECTS

In `main`, `a` evaluates to the value 1. It is passed in as an argument to `coolMethod`.

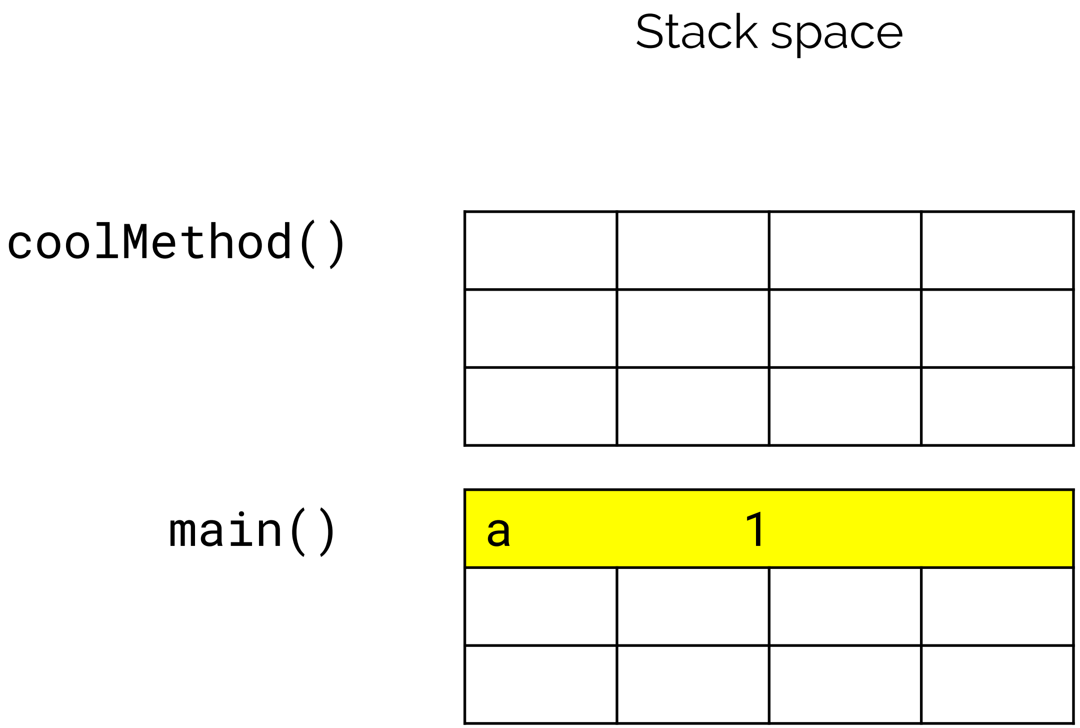
In `main()`:

```
int a = 1;
coolMethod(1);
System.out.println(a);
```

```
void coolMethod(int a) {
    a = 2;
}
```

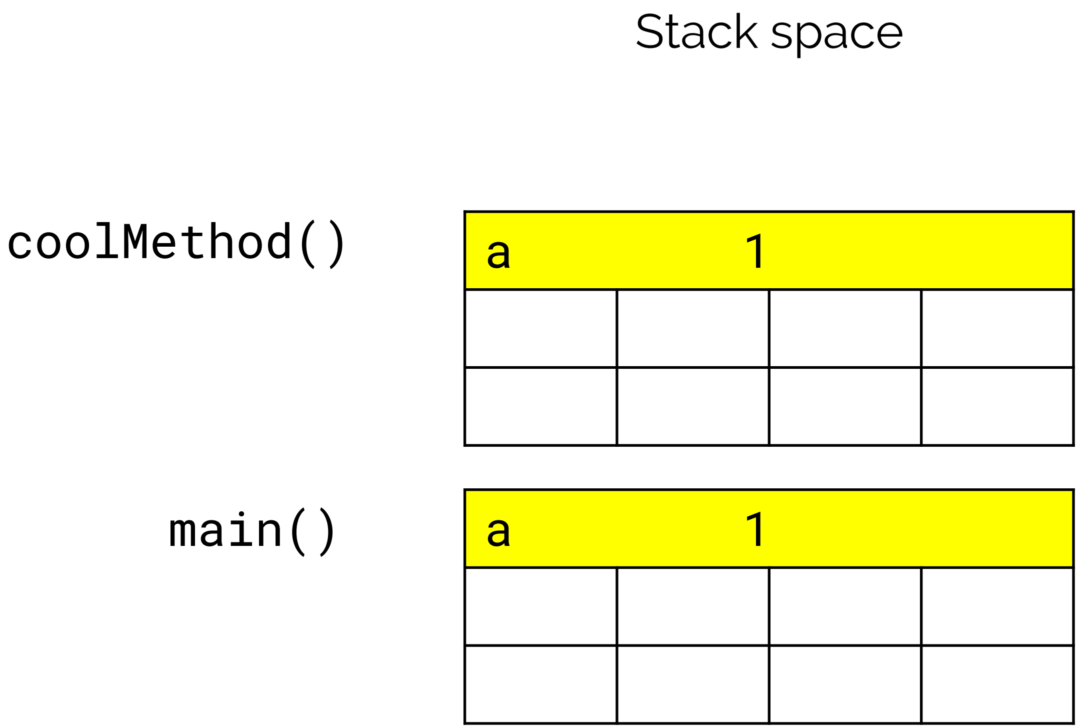
UNDERSTANDING OBJECTS

`main` makes a method call to `coolMethod`, which pushes it onto the stack.



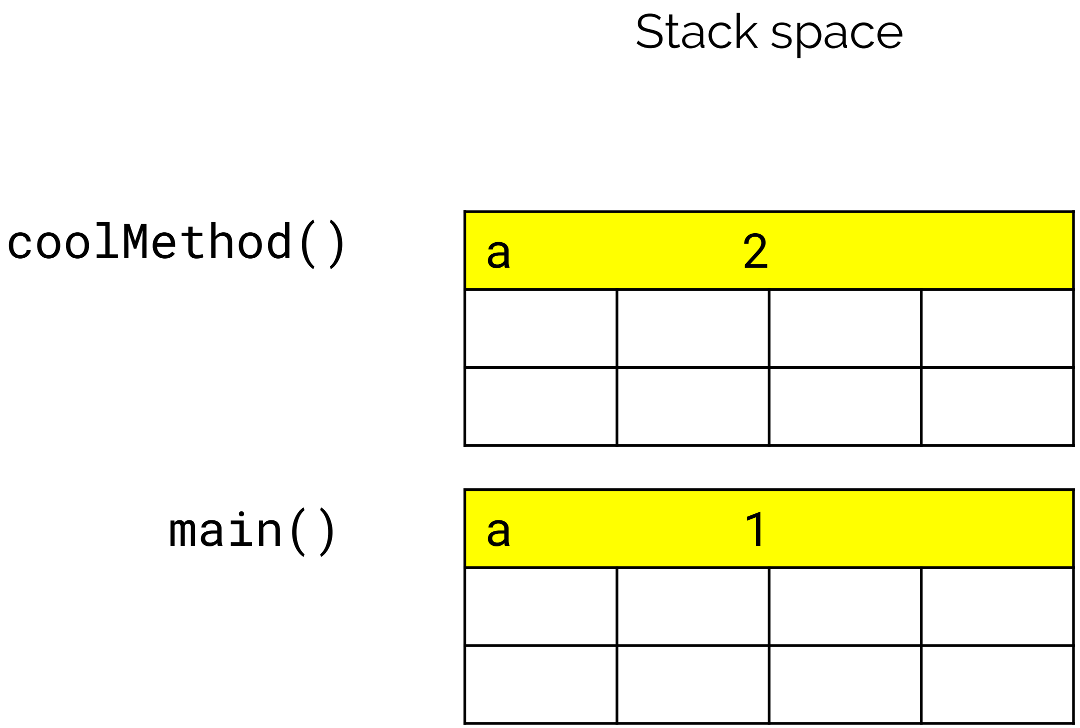
UNDERSTANDING OBJECTS

The value 1 is passed into `coolMethod`, and stored as a variable a.



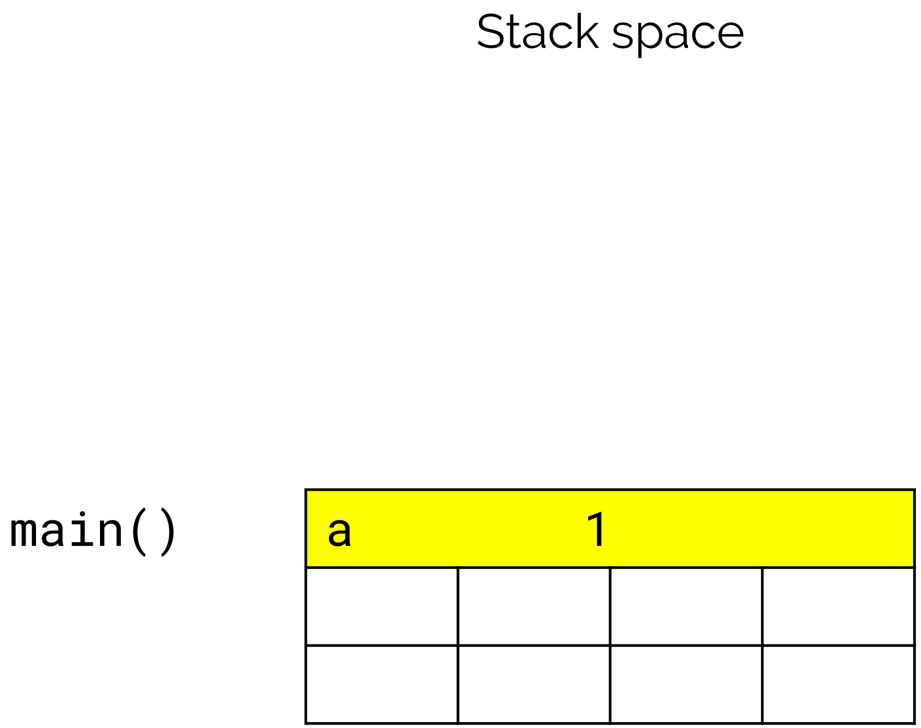
UNDERSTANDING OBJECTS

In `coolMethod`, we change **a** to 2.



UNDERSTANDING OBJECTS

`coolMethod` has completed, it is popped off the stack.



UNDERSTANDING OBJECTS

Hence, `main` will print 1 in the console.

In `main()`:

```
int a = 1;
coolMethod(a);
System.out.println(a);
```

```
void coolMethod(int a) {
    a = 2;
}
```

UNDERSTANDING OBJECTS

Earlier, we passed in primitive types to method calls. Let's try doing the same for reference types.

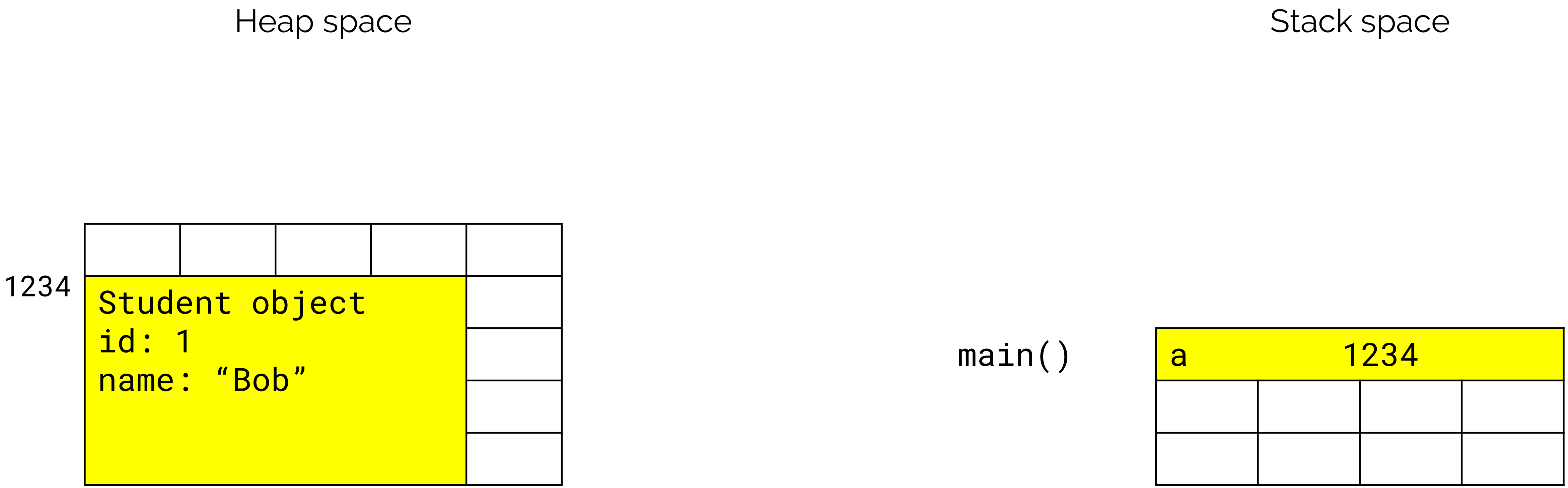
In `main()`:

```
Student a = new Student(1, “Bob”);
coolMethod(a);
System.out.println(a.id);
```

```
void coolMethod(Student a) {
    a.id = 2;
}
```

UNDERSTANDING OBJECTS

In the `main` stack frame, a new `Student` object is created, and its reference is stored in variable `a`.



UNDERSTANDING OBJECTS

In `main`, `a` evaluates to the reference `Student@1234`. It is passed in as an argument to `coolMethod`.

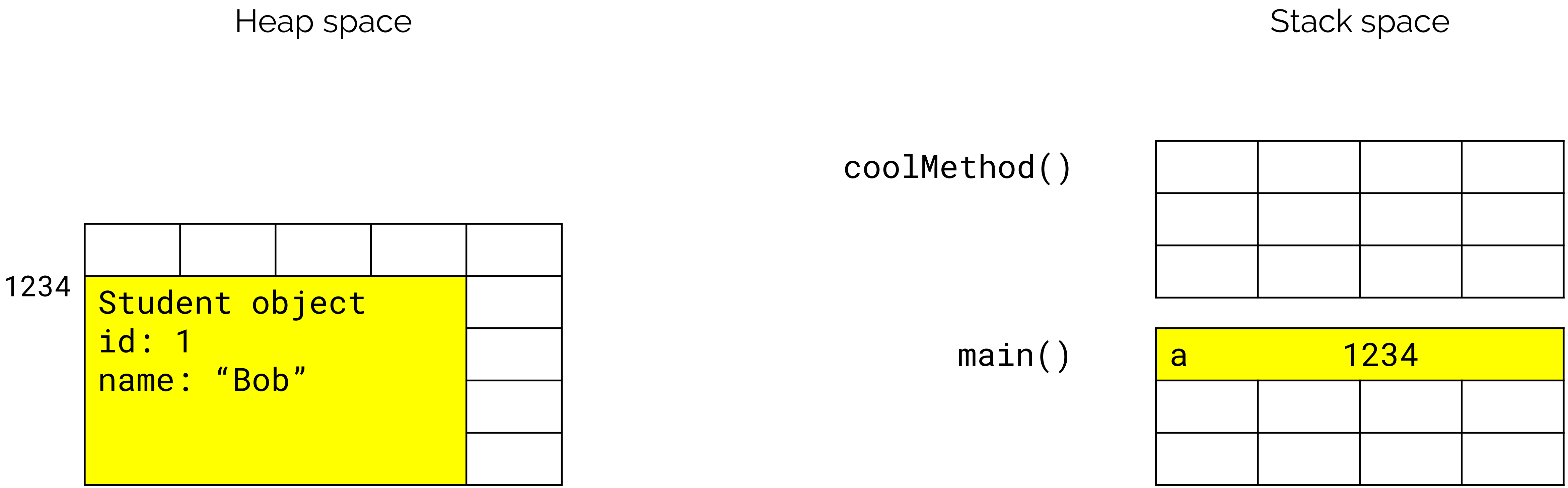
In `main()`:

```
Student a = new Student(1, “Bob”);
coolMethod(Student@1234);
System.out.println(a.id);
```

```
void coolMethod(Student a) {
    a.id = 2;
}
```

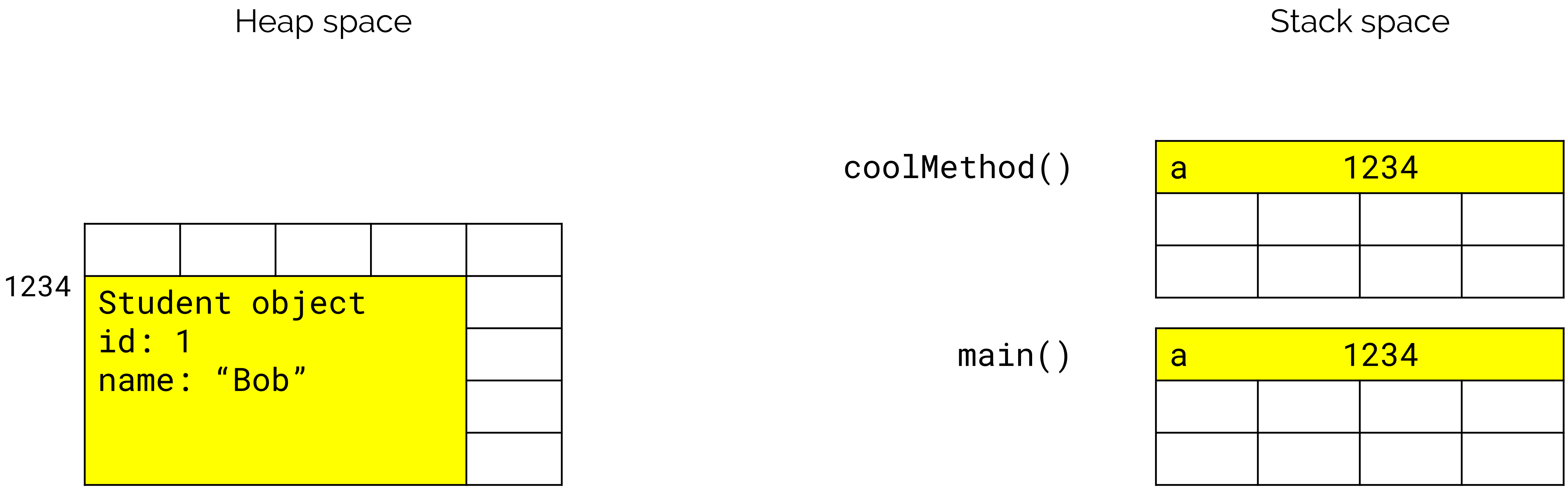
UNDERSTANDING OBJECTS

`main` makes a method call to `coolMethod`, which pushes it onto the stack.



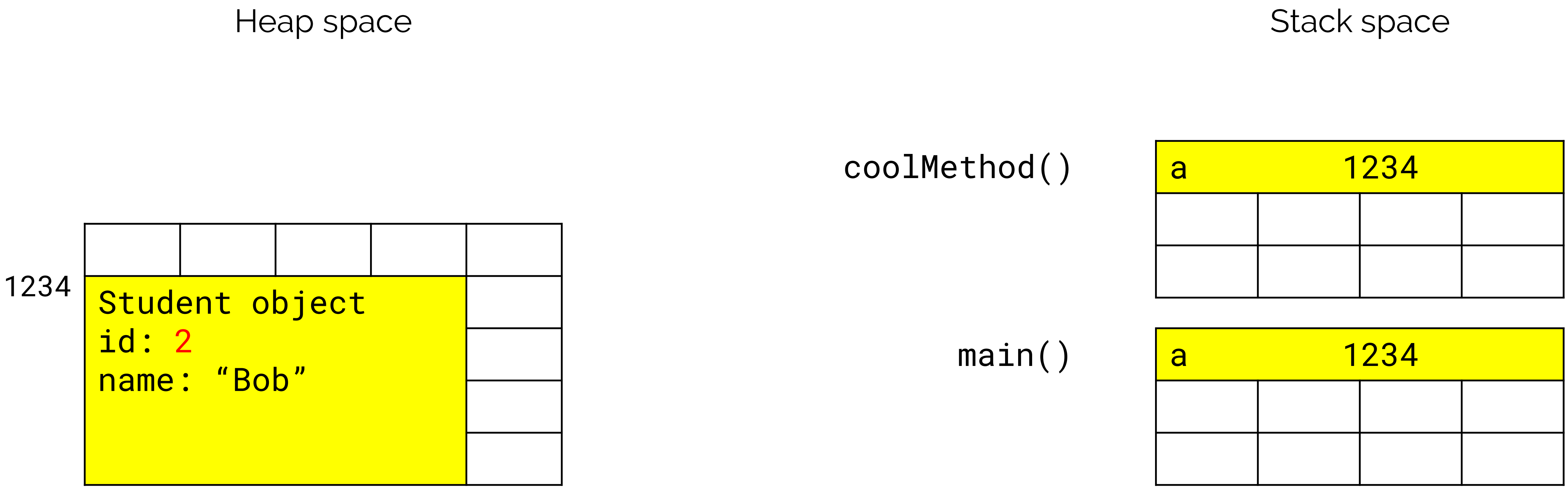
UNDERSTANDING OBJECTS

The value `Student@1234` is passed into `coolMethod`, and stored as a variable `a`.



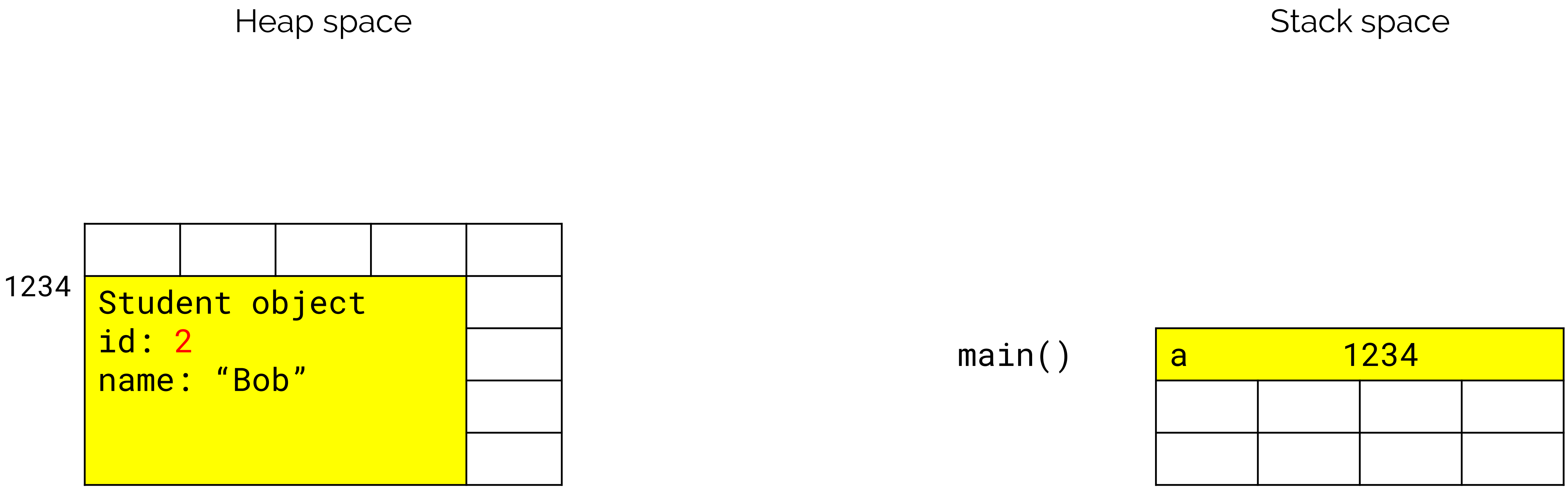
UNDERSTANDING OBJECTS

In `coolMethod`, we change the value of the `id` attribute of the object referred by `a` to 2.



UNDERSTANDING OBJECTS

`coolMethod` has completed, it is popped off the stack.



UNDERSTANDING OBJECTS

Hence, `main` will actually print 2 to the console!

In `main()`:

```
Student a = new Student(1, “Bob”);
coolMethod(a);
System.out.println(a.id);
```

```
void coolMethod(Student a) {
    a.id = 2;
}
```

This scenario will occur for all reference types, so be careful!

STATIC

The attributes and methods defined in our Student class are instance-level attributes.

We know this because different instances can have different values for their attributes, and the methods are called from the context of individual instances.

STATIC

For instance-level attributes/methods, each instance has their own copy.

Student class definition

```
id: int
name: String
crush: Student
myCoolMethod() {
    this.id++;
}
```

Object 1

```
id: 1
name: “Bob”
crush: Student object
       at 02384
myCoolMethod() {
    this.id++;
}
```

Object 2

```
id: 2
name: “Alice”
crush: Student object
       at 01321
myCoolMethod() {
    this.id++;
}
```

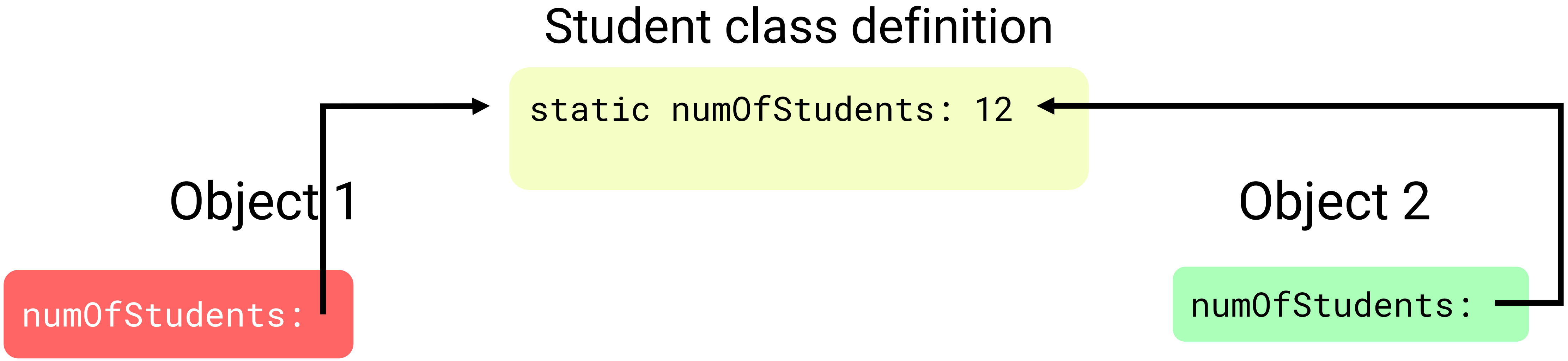
STATIC

We can also define **class-level attributes** and methods with the `static` keyword.

Class-level attributes/methods belong to the **class**; there is only **one copy**.

STATIC

If `numOfStudents` is a `static` attribute, each instance's `numOfStudents` will refer to the same attribute in the class.



STATIC

For example, for the following program, the output will be 2.

```
Student s = new Student(1, “Bob”);
Student.numOfStudents = 1;
s.numOfStudents++;
System.out.println(Student.numOfStudents);
```

STATIC

We place this definition back in the Student class.

```
class Student {
    // ...
    static int numberOfStudents = 0;
    // ...
}
```

STATIC

To keep track of the number of students, we can increment that attribute in the constructor.

```
class Student {
    // ...
    Student(int id, String name) {
        // ...
        Student.numberOfStudents++;
    }
}
```

STATIC

We can also create static methods which are bound to the class.

Let's create a factory method of the Student class that simply creates a student with a generic name and an id of 1.

```
static Student createStudent() {
    return new Student(1, “Generic Person”);
}
```

STATIC

Similarly, we can add it into the definition of the Student class.

```
class Student {
    // ...
    static Student createStudent() {
        return new Student(1, “Generic Person”);
    }
}
```

STATIC

We can access the attribute and method from the context of the class without any instances.

```
Student s = new Student(1, “Bob”);

s.numberOfStudents;                  // works
Student.numberOfStudents;            // works

s.createStudent();                   // works
Student.createStudent();             // works
```

ACCESS MODIFIERS

To adhere to Encapsulation (an OOP concept), we will need to hide internal details and restrict access of certain methods and attributes.

To do that, we use access modifiers.

ACCESS MODIFIERS

These are the access modifiers available to us. Attaching these to classes / methods / attributes will change their access.

MODIFIER	ACCESS
`public`	All
`protected`	Only available to classes in the same package, and its subclasses (regardless of package)
blank (known as package private)	Only available to classes in the same package
`private`	Only available to itself.

ACCESS MODIFIERS

Let's amend the access of the attributes and methods in our Student class.

For starters, the Student class itself should be `public`.

```
public class Student {
    // ...
}
```

ACCESS MODIFIERS

The constructor should also be `public`.

```
public class Student {
    public Student(int id, String name) {
        // ...
    }
}
```

ACCESS MODIFIERS

Other classes should not have the ability to directly amend the attributes in the class. As such, we should make them `private`.

```
public class Student {
    private int id;
    private final String name;
    private Student crush;
    private static int numOfStudents;
}
```

ACCESS MODIFIERS

The `sayHello` and `createStudent` methods should however, be `public`.

```
public class Student {
    public void sayHello() {
        // ...
    }
    public static Student createStudent() {
        // ...
    }
}
```

ACCESS MODIFIERS

Another class that tries to run the following codes will face these errors:

```
Student s = new Student(1, “Alice”);   // Ok
s.id = 1;                               // Error
s.name = “Bob”;                         // Error
s.crush = new Student(2, “Charlie”);   // Error
Student.numOfStudents = 1;              // Error
Student t = Student.createStudent();   // Ok
s.sayHello();                           // Ok
```

MAIN

We're almost done. We've created the blueprint for Student instances, but we have not stated what we want to do with them.

Introducing the `main` method!

```
public static void main(String[] args) {
    // ...
}
```

MAIN

The main method can live in any class. It serves as the entry point for our program to run.

This is where we can define what we want our program to do.

MAIN

As good practice, we should define a main “driver” class that contains the main methods and other necessary helper methods.

Let’s create a `Main` class in a file named `Main.java`.

You may enter `:tabedit Main.java` in vim’s command mode.

MAIN

This is where the logic of our program lies.

```
public class Main {
    public static void main(String[] args) {
        // ...
    }
}
```

After compiling your program, we will run this Main class on the JVM to execute our program.

JAVA API

Java provides a very robust **Application Programming Interface** (API) Reference which will help you use certain tools in the Java Development Kit (JDK).

We will be going through some useful API classes.

SCANNER

https://docs.oracle.com/javase/10/docs/api/java/util/Scanner.html

STRING

https://docs.oracle.com/javase/10/docs/api/java/lang/String.html

OBJECT

https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html

LIVE CODING EXAMPLE

PROBLEM STATEMENT

You are given a list of students and a prediction of who each student has a crush on.

However, each student can only have a crush on one other student aside from him/herself.

Your program must be able to discern if the prediction is possible.

EXERCISE

EXERCISE

Now it is your turn to code!

Imagine you live in a world where there are only three taxis, each with their own flag-down rates and per km rates.

The following table describes the rates for each taxi.

Name of Taxi	**GRAB**	**GOJEK**	**COMFORT**
Flag-down rate	$5.00	$4.30	$3.50
Per km rate	$0.22	$0.25	$0.30

EXERCISE

Because there are only three taxis, at the end of a trip, the taxi that drove you will wait for you. If in the next trip you choose to ride with that taxi again, there is no flag-down fare.

Name of Taxi	**GRAB**	**GOJEK**	**COMFORT**
Flag-down rate	$5.00	$4.30	$3.50
Per km rate	$0.22	$0.25	$0.30

EXERCISE

Write a program that reads in all your trips (in order) for the day. The program should output the total fare for each taxi for the day.

Example run (user input is underlined):

```
4
Grab 1
Gojek 1
Comfort 0.5
Comfort 0.5
Grab: $5.22
Gojek: $4.55
Comfort: $3.80
```