Systems Programming

Recap

Julio Villena Román (LECTURER)

<jvillena@it.uc3m.es>

CONTENTS ARE MOSTLY BASED ON THE WORK BY:

M.Carmen Fernández Panadero and Natividad Martínez Madrid

Universidad
Carlos III de Madrid

Systems Programming

First steps in Java

Julio Villena Román (Lecturer)

<jvillena@it.uc3m.es>

Contents are mostly based on the work by:

M.Carmen Fernández Panadero and Natividad Martínez Madrid

Scenario I:
Install and configure the environment

- Today is your first day at work in the programming department of PROTEL. Your department has to update an old application with new functionality
- You are provided with a laptop to work and a URL from where to download the code developed up to date

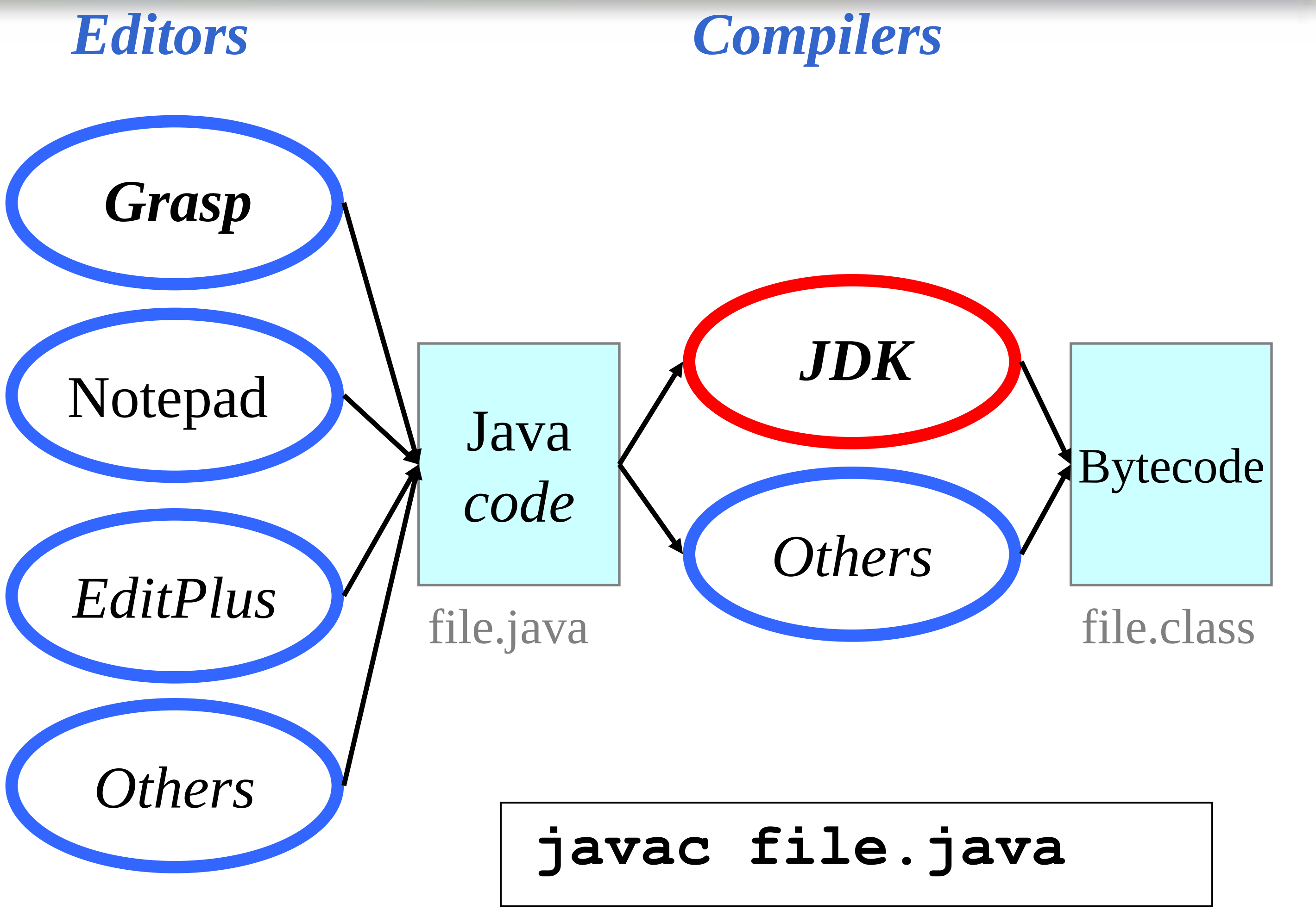
- Objective: Be able to ***edit, compile execute and debug*** an existing program.
- Workplan: Download, install and configure the software in order to test (edit, compile, execute and debug) the application

Development Architecture

Step I: Edit
Step II: Compile

IDEs

- Eclipse
- Netbeans
- J Builder
- Visual Café
- Java Workshop
- Visual Age
- J++



Editors

Grasp

Notepad

EditPlus

Others

Java *code*

file.java

Compilers

JDK

Others

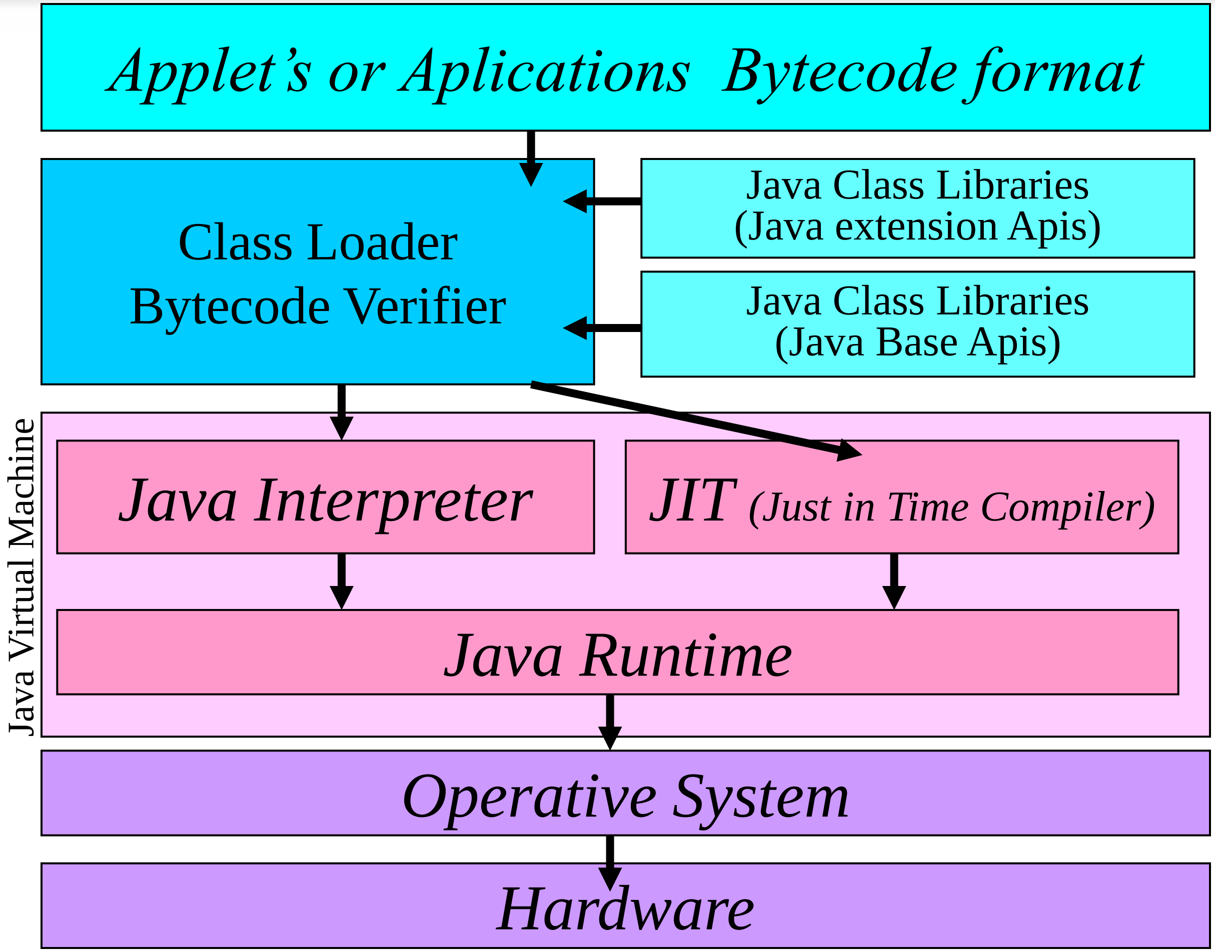
Bytecode

file.class

```
javac file.java
```

Execution Architecture

Step III: Load
Step IV: Verify
Step V: Execute



Applet's or Aplications Bytecode format

Class Loader
Bytecode Verifier

Java Class Libraries
(Java extension Apis)

Java Class Libraries
(Java Base Apis)

Java Virtual Machine

Java Interpreter

JIT (Just in Time Compiler)

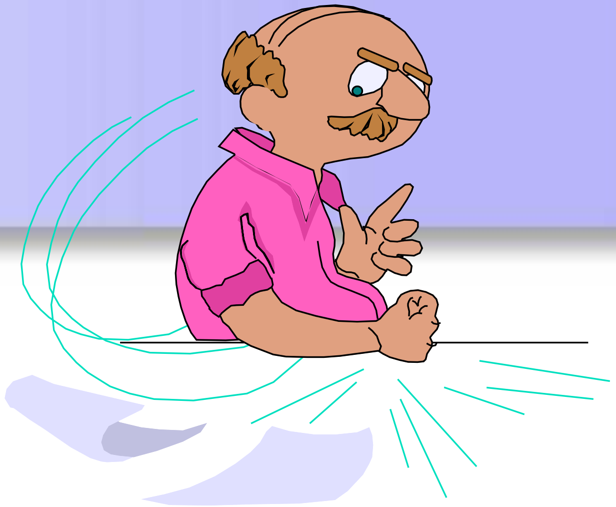
Java Runtime

Operative System

Hardware

`java file`

What can go wrong?



- Compile → “Syntax Error”
- Load → “Class not found Exception”
- Verify → “Security Exception”
- Execute → “Null Pointer Exception”

Where do I start?

- Development environment: JDK

  `http://www.oracle.com/technetwork/Java/index.html`

- Editor: Eclipse

  `http://www.eclipse.org`

- Documentation: Java API

  `http://docs.oracle.com/Javase/7/docs/api/`

Config (if needed)

- Configuration:
  - CLASSPATH: Set of directories containing the files.class you want to execute (not necessary since v1.2).

    It must contain, at least, $JAVA_HOME/lib/files.class
  - PATH: Directories to search for executable files

    It must contain, at least, $JAVA_HOME/bin

How to configure Environment Variables

Windows 95-98 (Type in MSDOS Window or modify c:\autoexec.bat):

```
set PATH=c:\jdk1.2\bin;C:\WINDOWS\COMMAND\
set CLASSPATH=c:\jdk1.2\lib\classes.zip;.
```

Preserving the old value of environment variables:

```
set PATH=c:\jdk1.2\bin;%PATH%
set CLASSPATH=c:\jdk1.2\lib\classes.zip;%CLASSPATH%;.
```

Linux (Type in a terminal window or modify in .bash file to conserve the value):

```
PATH=$JAVA_HOME/bin:/usr/bin
CLASSPATH=$JAVA_HOME/lib/classes.zip:.
```

Preserving the old value of environment variables :

```
PATH=$JAVA_HOME/Java/bin:$PATH
CLASSPATH=$JAVA_HOME/lib/classes.zip:$CLASSPATH
```

How to configure Environment Variables

Windows NT

- Start – Control panel – System
- Select: Environment – [look for user and system variables]

Windows 2000

- Start – Control panel – System
- Select: Advanced – [look for user and system variables]

Windows XP

- Start – Control panel – System
- Select: Advanced – click on environment variables

Windows ME

- Start – Program files - Accesories – System tools – System info
- Select: Tools-System configuration
- Select: Environment – [select variable]- click edit

Windows 7, 8, 8.1

- Start – Control panel – System and Security – system
- System advanced configuration – Advanced options – Environment variables

Systems Programming

Java Language Code Structure

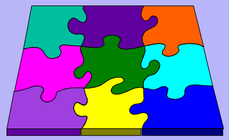
Julio Villena Román (LECTURER)

<jvillena@it.uc3m.es>

CONTENTS ARE MOSTLY BASED ON THE WORK BY:

M.Carmen Fernández Panadero and Natividad Martínez Madrid

Scenario II:
Understanding Java code

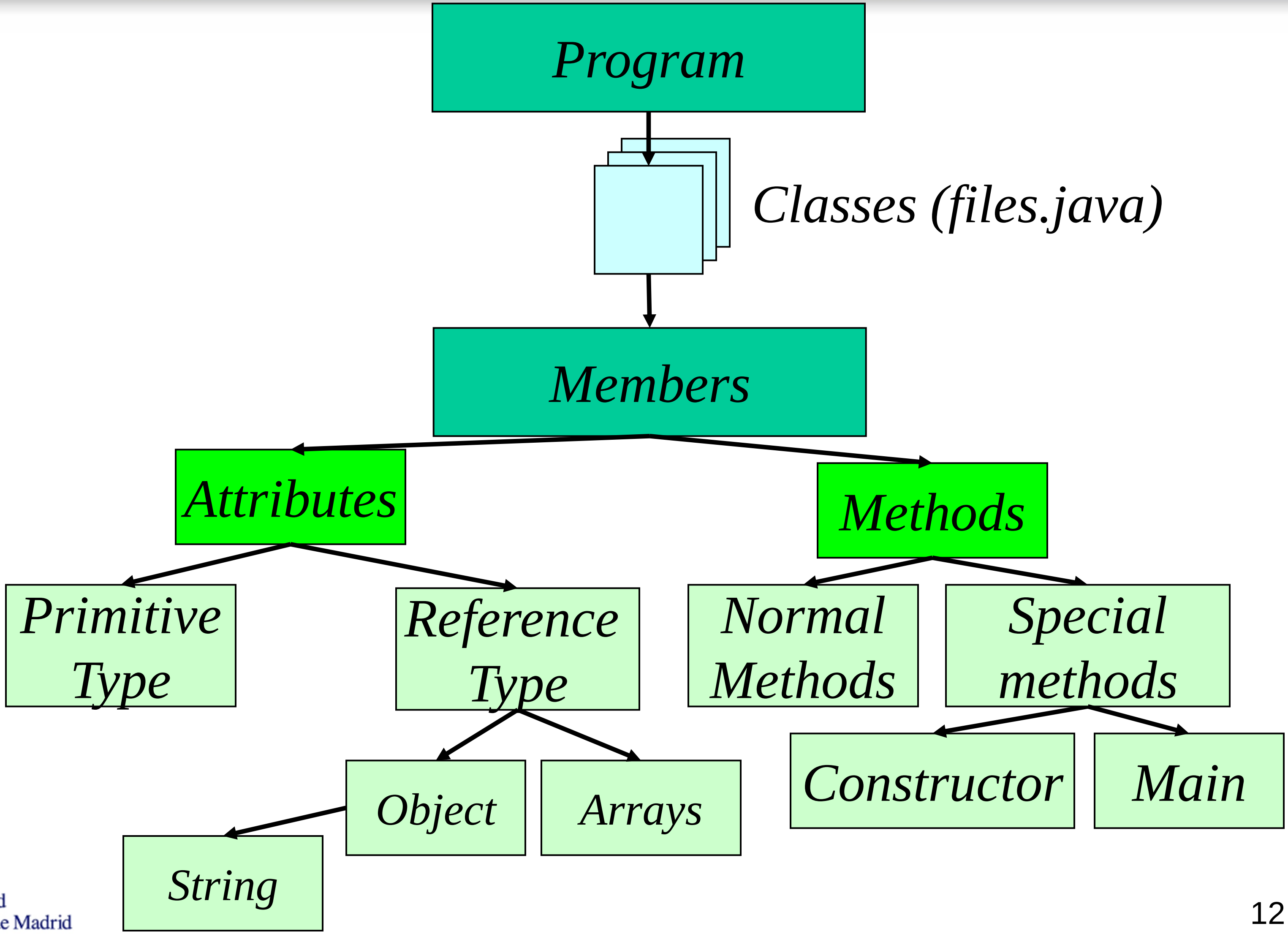


- Your first meeting as a programmer will be in an hour. By this time you must have reviewed the code and you must have understood how the application works

- Objective: Be fluent in reading Java structures related with classes, attributes and methods. Understand, at a glance, a complex Java program with several files
- Workplan:
  - **Review** Java **syntax** (identifiers, reserved words, etc.) in order to distinguish between words from Java-language and naming for a specific application
  - **Identify** language structures related with **class declaration, attribute declaration** (basic and reference types) **and method declaration**
  - **Draw UML diagrams** to represent a set of Java files in order to identify object types, their characteristics (attributes) and behaviour (methods)
  - **Understand and explain the main method** (when exists) to see in which order the objects are created, the methods invoked and the sentences executed

Universidad
Carlos III de Madrid

Code Structure



Program

Classes (files.java)

Members

Attributes

Methods

Primitive Type

Reference Type

Normal Methods

Special methods

Object

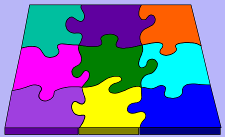
Arrays

Constructor

Main

String

How to represent classes and objects in Java



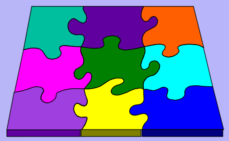
OO

- Class declaration
- Attribute declaration (constants or variables)
- Method declaration
- Object creation

Java

- Identifiers
- Reserved words
- Primitive and reference types in Java

Identifiers

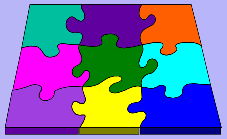


- Identifiers are used to give a name to variables, methods, classes, objects, and everything that the programmer needs to identify
- Starting with a letter, an underscore or a $ sign
- Case-sensitive, no maximum length

- **By convention:**
  - The names of variables, methods and objects begin with lowercase.
  - The class names begin with uppercase
  - If there are several words, use camel-case `likeInThisExample` (avoid spaces, underscores and hyphen)

Identifiers can not be reserved words

Reserved words



Reserved:

abstract	double	int	static
boolean	else	interface	super
break	extends	long	switch
byte	final	native	synchronized
case	finally	new	this
catch	float	null	throw
char	for	package	throws
class	goto	private	transient*
const *	if	protected	try
continue	implements	public	void
default	import	return	volatile
do	instanceOf	short	while

Reserved (not used):

cast	future	generic	inner
operator	outer	rest	var

Comments

Three types:

```
// Implementation comment (1 line)
```

```
/* Implementation block comment.
  continue
  finish */
```

```
/**Documentation comment to generate Javadoc
  @see ref to other class or method
  @version information about version number
  @author author name
  @since Date since code is available
  @param Params recived by the method
  @return Information and data type returned by the method
  @throws Exceptions that throws this method
  @deprecated The method is old
 */
```

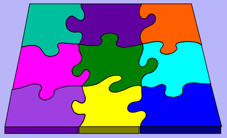
optional

For classes and methods

For classes

For methods

Class declaration



```
public class Car {
   // Attribute declaration
   //   (color, speed, etc.)
   // Method declaration
   //   (start, stop, etc.)
}
```

Car.java

Syntax

```
(modifiers) class className {
   // class implementation
}
```

Breaking this rule is considered in many compilers as a syntax error

Style

- File name = class name
- 1st letter capitalized
- No blanks
- Camel case: MyFirstClass
- Indentation

Variable Declaration

```
public class Car{
  //Atribute declaration
   String color;
   int speed;
  //Method declaration
  // (start, stop, etc.)
}
```

Car.java

Syntax

type name;
type name1, name2, name3;
type name = value;

Initialize the variable

Style

- Intuitive names
- 1st letter capitalized
- No blanks
- Camel case: myVariable
- Indentation

Variables

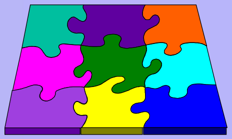
- Variables are fields in which programs store information
- ***“To declare*** a variable” means to specify its name and type
- We can find variables:
  - As ***members:*** Instance and class variables (within a class)
  - As ***local variables*** (within a method)
  - As ***parameters*** (within a method declaration)

Variables

- Three types:
  - ***Instance*** variables
  - ***Class*** variables
  - ***Local*** variables
- Variables
  - can be initialized in the declaration
  - may be declared uninitialized
    - when have been not initialized they have a ***default value*** (except local variables)
- ***Constants*** (variables that can not be modified):
  - Use reserved word: ***final***
  - Mandatory to be initialized in declaration

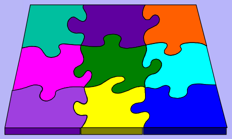
***Default values*:**
numbers = 0
booleans = false
references = null

Scope



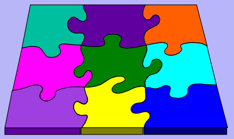
- The ***scope*** of a variable is the part of the program over which the variable name can be referenced
- ***Instance*** or ***class variables*** can be referenced inside the body of the class or from other classes depending on the permissions set:
  - **`private`**
  - **`protected`**
  - **`public`**
  - **`friendly`**
- ***Local*** variables can be referenced inside a statement block in brackets, such as inside a method or inside a while or for loops
- ***Parameters*** can be referenced only inside the body of the method

Data Types in Java



- All variables belong to a ***data type***
- The data type determines:
  - The **values** that the variable can take
  - The **operators** that can be used
- We will study:
  - ***Primitive types***
  - ***Reference types (objects and arrays)***

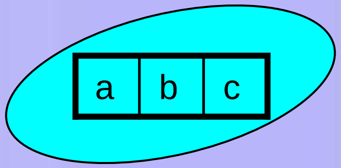
Primitive types



4 basic primitive types

type	literal	num of bits	double	float	long	int	short	byte	char
Real	`double`	64-bits	X						
	`float`	32-bits	X	X					
Integer	`long`	64-bits	X	X	X				
	`int`	32 bits	X	X	X	X			
	`short`	16 bits	X	X	X	X	X		
	`byte`	8 bits	X	X	X	X	X	X	
Character	`char`	Unicode (16 bits)	X	X	X	X			X
Boolean	`boolean`	1 bit							

Strings
Declaration, concatenation



- Sequence of characters implemented in a class named `String` (in `java.lang` package)
- Strings creation

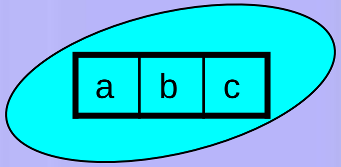
```
String emptyS = new String();
String emptyS = “”;
String message = “hello”
String messageCopy = message;
```

- Strings concatenation
  - String concatenation uses the overloaded `+` operator

```
“this” + “that”       // result: “thisthat”
“abc” + 5             // result: “abc5”
“a” + “b” + “c”       // result: “abc”
“a” + 1 + 2           // result: “a12”
1 + 2 + “a”           // result: “3a”
1 + (2 + “a”)         // result: “12a”
```

Strings

Comparation



- You must **not use** relational (<, >, <=, <=) and equality (==, !=) operators with Strings
  - **This operators compare the object not the content**
- There are specific **methods to compare** in the **`String`** class
  - Method: **`equals`**

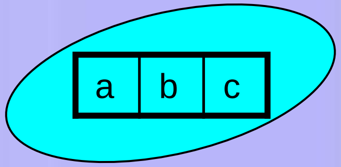
```
leftSide.equals(rightSide)
     • true, if leftSide and rightSide are identical
```

  - Method **`compareTo`**

```
leftSide.compareTo(rightSide)
     • negative int value, if leftSide is less than rightSide
     • 0, if leftSide is equal to rightSide
     • positive int value, if leftSide es mayor que rightSide
```

Strings

Useful methods of `String class`



- Length of an **`String`**
  - Method: **`length()`**
  - Don't forget parenthesis because it is a method
- Accessing individual characters inside the **`String`**
  - Method: **`charAt(`**position**`)`** ,
    - The first position is the String is `0`
- SubStrings
  - Method **`substring(`**1stPosIncluded, 1stPosExcluded**`)`**
    - Returns: a String reference
    - Parameters: the 1st position included and the 1st position excluded

```
String greeting = "hello";
int len = greeting.length();          // len is 5
char ch = greeting.charAt(1);         // ch is 'e'
String sub = greeting.substring(2,4); // sub is "ll"
```

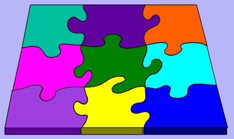
Strings

Conversion between String and primitive types

- Use calls to the wrapper class that is in `java.lang`
  - They are called wrappers because they wrap the primitive types: `Integer, Double, Float, Double, Character, …`
  - `String` conversion
    - Methods: `toString(…)`, `doubleValue()`
  - `String` conversion to a primitive type
    - Methods: `parseInt(…)`, `parseFloat(…)`
  - `String` conversion to an object of the wrapper class
    - `valueOf(…)`
  - Conversion from an object of the wrapper class to a primitive value
    - `doubleValue()`, `intValue()`

```
System.out.println(Integer.toString(55, 2));
int x = Integer.parseInt(“75”);
Double y = Double.valueOf(“3.14”).doubleValue();
```

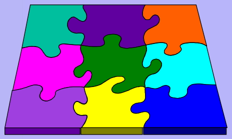
Constants defined by user



- Invariant values of basic types (primitives + **String**)
- Constants use the **final** modifier (and sometimes the **static** too)
  - **static**: Indicates global or class variable. This mean that it is stored only once. Objects can access this variable using the dot notation, ClassName.variableName
  - **final**: This modifier Indicates that the value never changes.
  - Constants can be **public**, **private** or **protected**
    - Depending on accesibility that user prefers
  - *Style:* All the characters in **UPPERCASE**

```
class Circle {
  private static final float PI = 3.14159;
  private float radio;
  private float area;
  public Circle (float radio) {
        area = 2 * PI * radio;
  } //constructor
} //class Circle
```

Reference types



- Its value is a ***reference*** (pointer) to the value represented by this variable.
- Some examples of reference types:
  - Arrays
  - Classes
  - Interfaces

An object as an attribute

Object declaration

```
public class Car{
  //Attribute declaration
   String color;
   int speed;
   Equipment standardEquipment;
  //method declaration
  // (start, stop, etc.)
}
```

Car.java

Style

- Remember that the class (type) uses 1st char capitalized and identifier (objectName) uses lower-case

Syntax

ClassName name;

ClassName name1, name2;

ClassName name = new Equipment();

Object declaration

similar to variable declaration, where we put the type, now we put the name of the class

Object creation

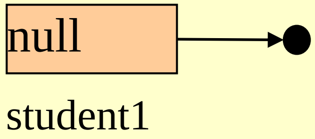
Variables are initialized, but Objects are created

Objects

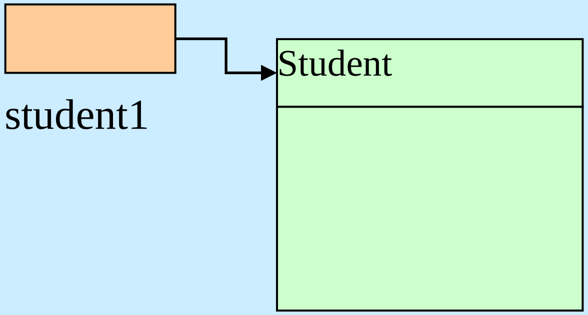
Declaration, creation, initialization

- Objects are created with the reserved word **`new`** and a call to the constructor
- Once the object is created, the reference to the object is reassigned to the memory location where the object is located

```
Student student1;
```



```
student1 = new Student();
```



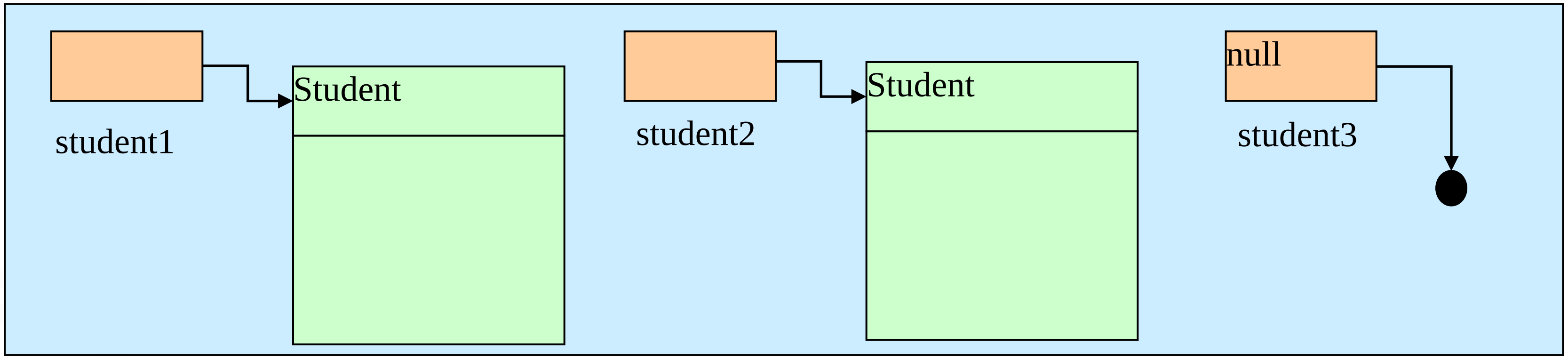
Objects

Null reference

- It may happen that a reference to an object has no instance assigned
  - Then the special value **`null`** is used
- Example:

```
Student student1;             // null by default
Student student2;
Student student3;

student1 = new Student();     // value /= null
student2 = new Student();     // value /= null
student3 = null;              // value null by assignment
```

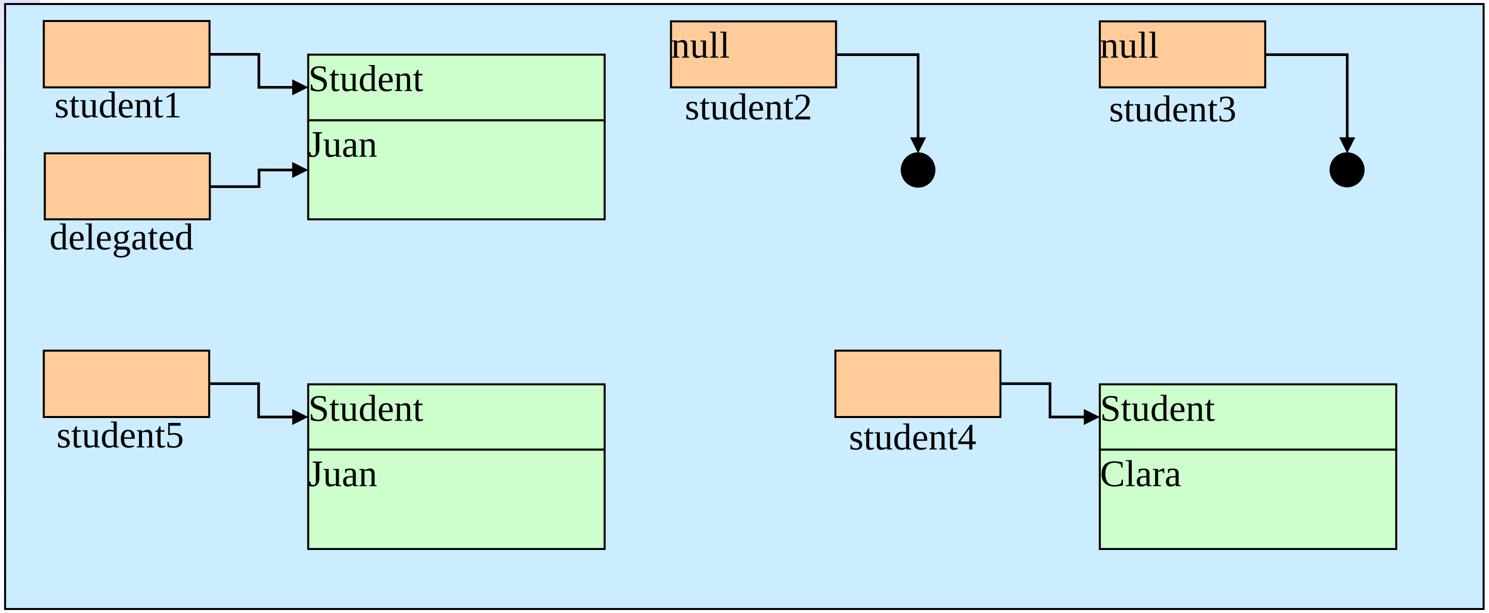


Objects
Alias

- An object can have several references, known as alias

```
Student delegate;
delegate = student1;
```

- What would be the result of comparing the different references in the figure?



Universidad Carlos III de Madrid

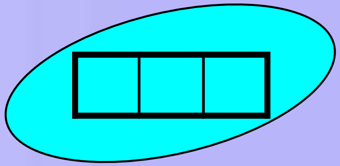
Arrays

What is an array?

- It is a set of elements belonging to the **same data type** and stored in one place
- The **index [ ]** operator is used to retrieve individual elements from the array
- The **length** (attribute) returns the number of array elements. (do not be confused with the method `length()` of the String class)
- Range of index
  - From `0` to `length - 1`
  - **Be careful!** Don’t exceed the maximum length
    - **Exception:** `IndexOutOfBoundsException`

An Array as an Attribute

Array declaration



```
public class Car{
   //Array declaration
   String equipment[] = new String [10];
   // ...
}
```

Ways to declare an array

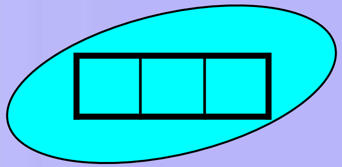
type ArrayName[];
type [] ArrayName;
type ArrayName[] = new type [arraySize];

Array creation
When you create an array you must specify its capacity

Array creation
Variables are initialized, but Arrays (like objects) are created

Arrays

Declaration, Creation, Initialization



- ***Declaration***: To assign an **identifier** to the array and specify the **data type** of the elements that will be stored
  - It can be done in two ways:

```
Type ArrayName[];
Type[] ArrayName;
```

  - No memory to store the array is allocated in the declaration, thus you can not access its contents yet

- ***Creation***: it consists on allocating memory for the array
  - You must use reserved word **new** and specify the array **size**

```
arrayName[] = new type[arraySize];
```

  - Once the array has been created, its elements have default values until the array is initialized

Default values:
int, short, long = 0
float, double = 0.0
booleans = false
String = null
Object = null

Arrays

Declaration, Creation, **Initialization**

- ***Initialization***: is to **assign value** to each element of the array. It can be done in several ways:
  - Element by element

```
arrayName[0] = element0;
arrayName[1] = element1;
...
```

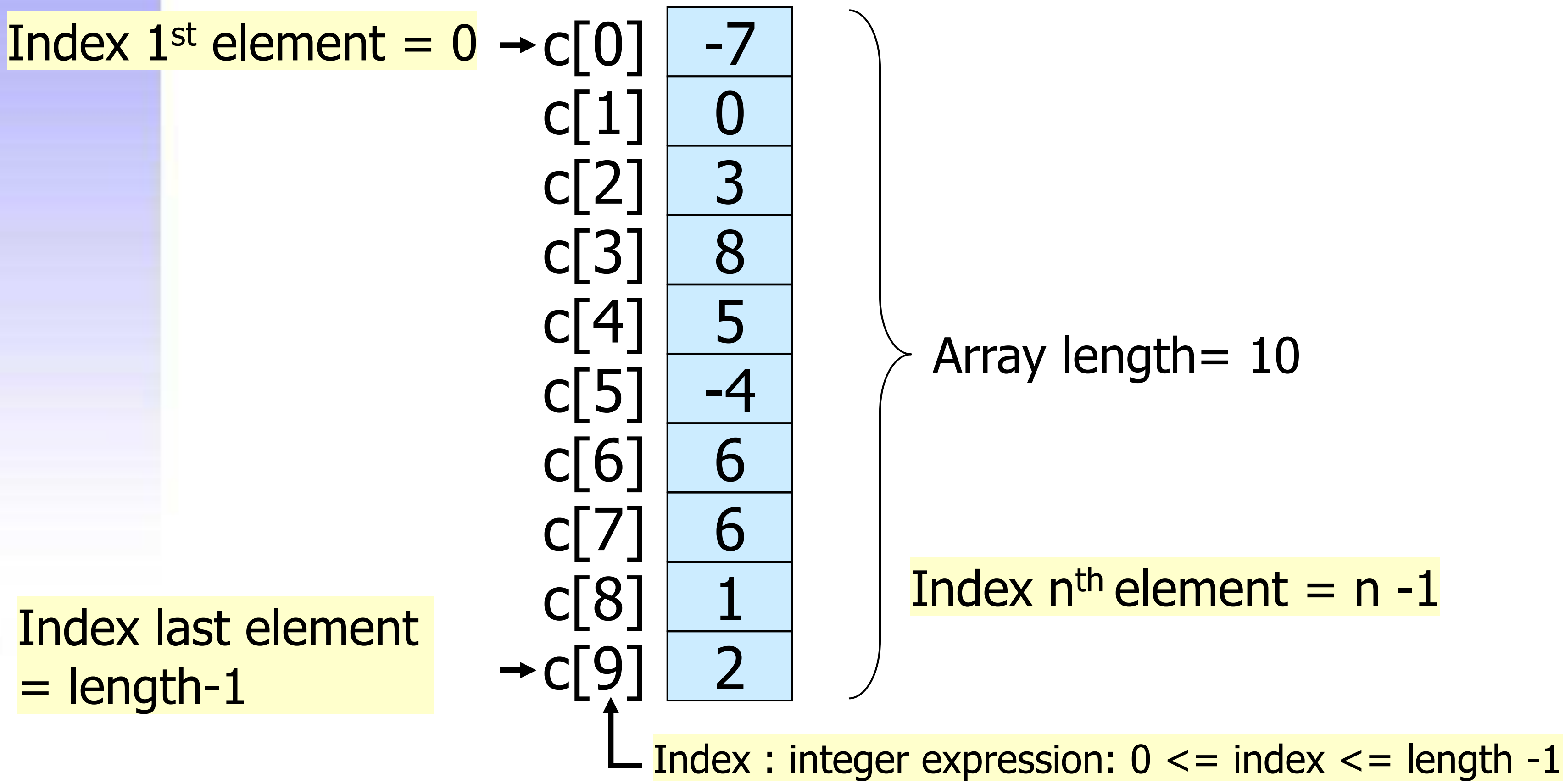
  - Using a Loop

```
for(int i = 0; i < arrayName.length; i++){
   arrayName[i] = element-i;
}
```

  - Direct assignment

```
arrayName = {elem1, elem2, elem3, ...};
```

Arrays



Index 1st element = 0

c[0]	-7
c[1]	0
c[2]	3
c[3]	8
c[4]	5
c[5]	-4
c[6]	6
c[7]	6
c[8]	1
c[9]	2

Array length= 10

Index nth element = n -1

Index last element = length-1

Index : integer expression: 0 <= index <= length -1

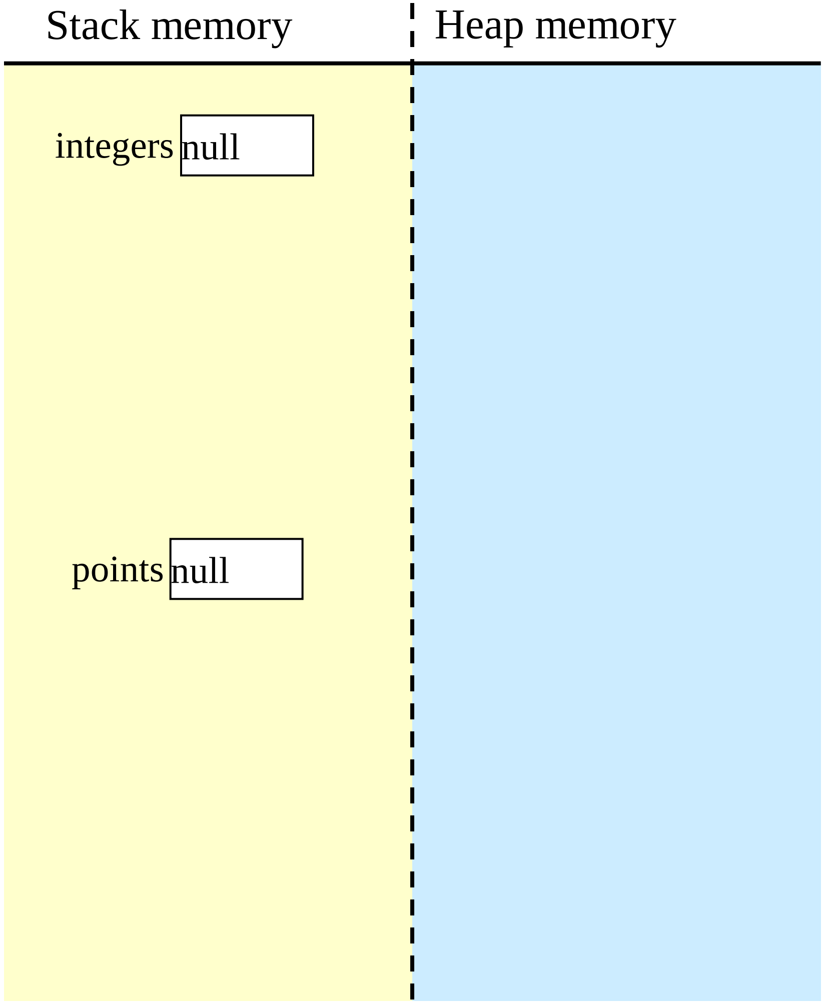
Universidad Carlos III de Madrid

Arrays
Memory usage in array declaration

```
int[] integers;
```

```
Point[] points;
```

```
class Point {
  int x;
  int y;
  Point (int x, int y){
    this.x = x;
    this.y = y;
  }
}
```



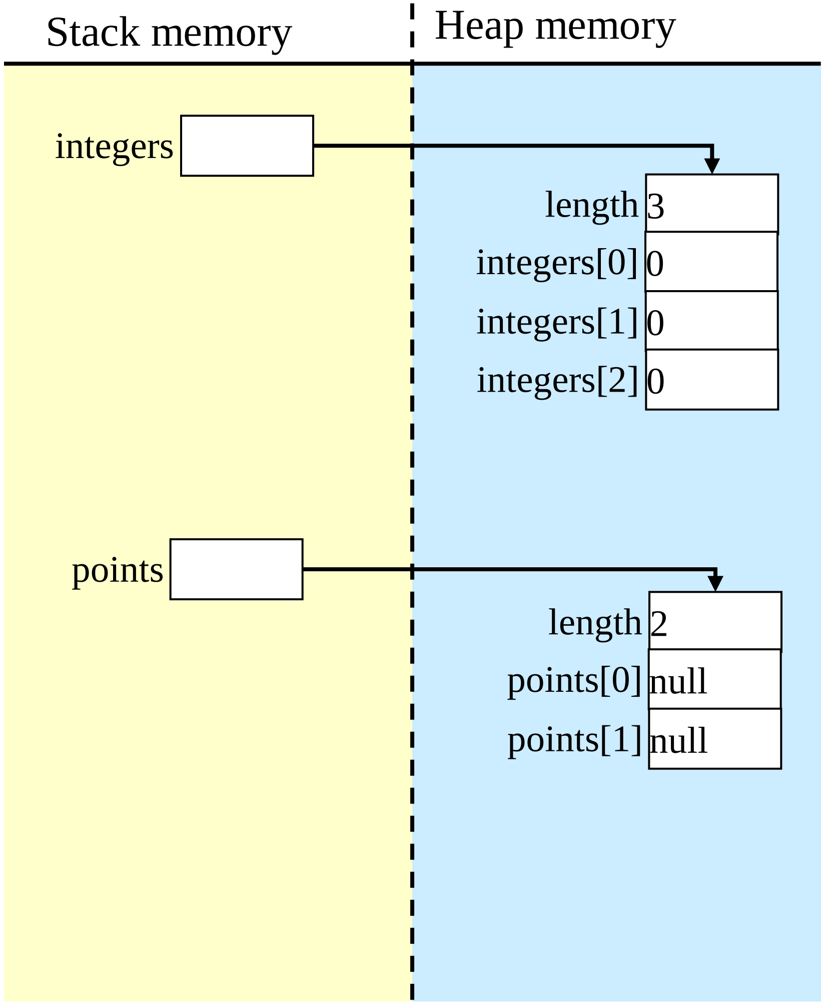
Arrays

Memory usage in array creation

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

```
points = new Point[2];
```

Watch out! This is not a constructor call

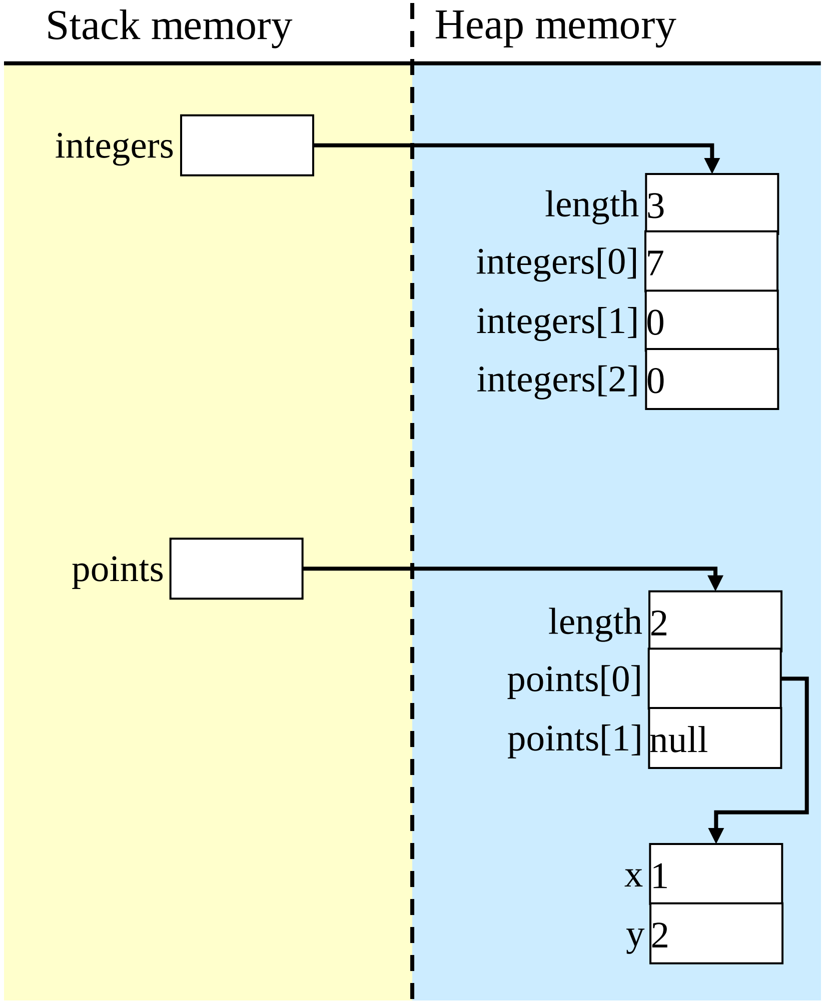


Arrays

Memory usage in array initialization

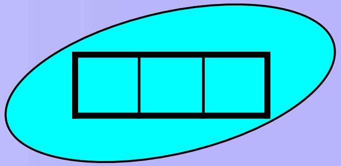
```
integers[0] = 7;
```

```
points[0] = new Point(1,2);
```



Arrays (examples)

Declaration, Creation, Inicialization



Arrays with primitive types

```
int a[];           //declaration
a = new int[3]  //creation
a[0]=1;          //initialization
a[1]=2;
a[2]=3;
```

```
int a[] = new int[3]  //declaration, creation
a[0]=1;                    //initialization
a[1]=2;
a[2]=3;
```

```
int a[] = new int[3]  // declaration, creation
for(int i=0; i<a.length;i++){ //initialization
   a[i]=i+1;
}
```

```
int a[] = {1, 2, 3};  / /Declaration, creation, initialization
```

Arrays with objects (reference types)

```
MyClass a[];      //declaration
a = new MyClass[3] //creation
a[0]=new MyClass(param1);
a[1]=new MyClass(param2);
a[2]=new MyClass(param3);
```

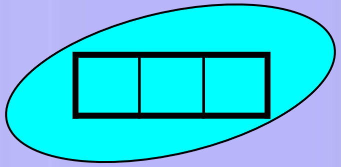
```
MyClass a[] = new MyClass[3]
//initialization
a[0]=new MyClass(param1);
a[1]=new MyClass(param2);
a[2]=new MyClass(param3);
```

```
MyClass a[] = new MyClass[3]
//initialization
for(int i=0; i<a.length;i++){
   a[i]=new MyClass(param-i);
}
```

```
MyClass[] a = {new MyClass(param1), new MyClass(param2), new myClass(param3)};
```

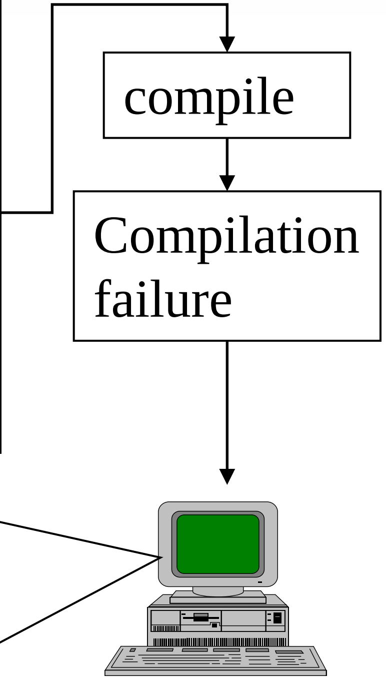
Arrays (common errors):

Declaration, Creation, Initialization



```
public class ArrayExamples{
    public static void main(String args[]){
      double myArray[];
     System.out.println(myArray[0]);
    }
}
```

WRONG



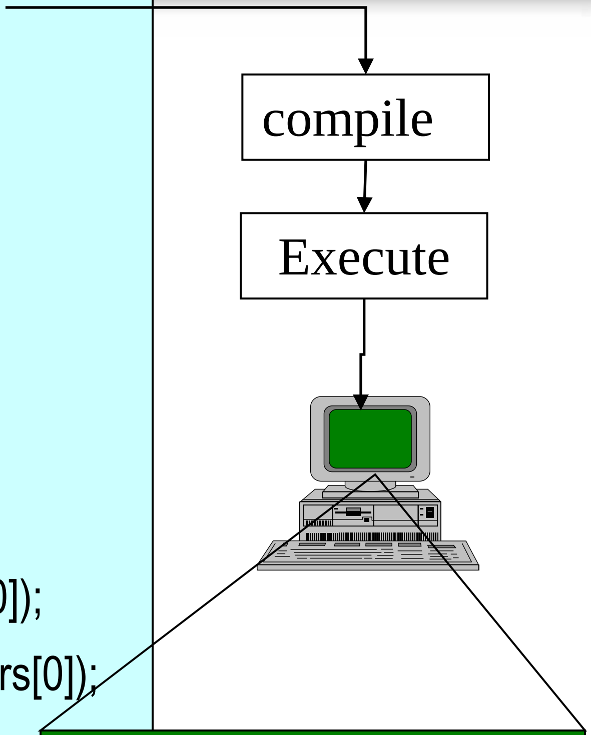
```
variable myArray may not have been initialized
```

When an array has been ***declared*** but not created or initialized, you have no access to its elements. The program does not compile and prints an ***error*** message

Arrays (Common errors):

Declaration, creation, inicialization

```
public class ArrayExamples2{
    public static void main(String args[]){
      int myArrayOfIntegers[] = new int[10];
      float myArrayOfReals[]= new float[10];
      boolean myArrayOfBooleans[] = new boolean[10];
      char myArrayOfCharacters[] = new char[10];
      String myArrayOfStrings[] = new String[10];
      Object myArrayOfObjects[] = new Object[10];
      System.out.println(“Integer by default: " + myArrayOfIntegers[0]);
      System.out.println("Real by default : " + myArrayOfReals[0]);
      System.out.println("Boolean by default : " + myArrayOfBooleans[0]);
      System.out.println(“Character by default : " + myArrayOfCharacters[0]);
      System.out.println("String by default : " + myArrayOfStrings[0]);
      System.out.println("Object by default : " + myArrayOfObjects[0]);
    }
```



```
Integer by default: 0
Real by default : 0.0
Boolean by default : false
Character by default :
String by default : null
Object by default : null
```

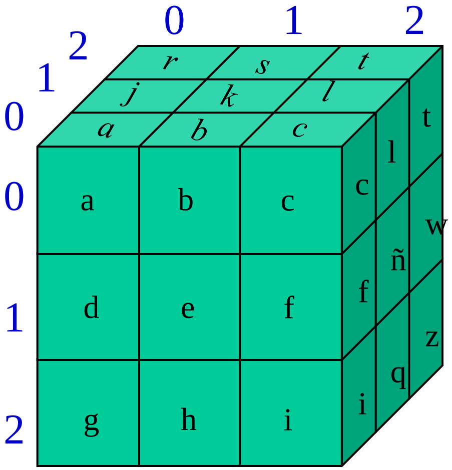
When the array has been ***declared and created*** but ***not initialized*** we can retrieve its elements but they have their ***default value***

N-dimensional Arrays

- When we need more than one index to retrieve its elements

	0	1	2
0	A	B	C
1	D	E	F
2	G	H	I

a[0][2]=‘C’



a[0][2][1]=‘l’

```
char a[][];             //declaration
a = new char[3][3]      //creation
a[0][0]=‘A’;            //initialization
...
```

```
char a[][][];               //declaration
a = new char[3][3][3]       //creation
a[0][0][0]=‘a’              // initialization
...
```

N-dimensional Arrays Examples

Direct declaration and creation

```
//Declaration and creation
String [][]myArray = new String[3][4]
```

null	null	null	null
null	null	null	null
null	null	null	null

Declaration and creation step by step

```
int [][] myArray ;                              // Array declaration
myArray  = new int[numRows][];                  // Creating the reference array for rows
for(int i=0; i<numRows; i++)                    // Allocate memory for rows
    myArray[i]= new int[numColumns];
```

Other examples

```
// Array 3x3 inicialized to 0
int [][] a= new int[3][3];
```

0	0	0
0	0	0
0	0	0

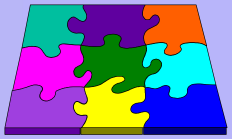
```
int [][] b= {{1, 2, 3},
             {4, 5, 6}};
```

1	2	3
4	5	6

```
int [][] c = new[3][];
c[0] = new int[5];
c[1] = new int[4];
c[2] = new int[3];
```

0	0	0	0	0
0	0	0	0	
0	0	0		

Method declaration



Car.java

```
public class Car{
  //Attribute declaration
   private String color;
   private int speed;
  //Method declaration
   public String getColor(){
       //implementation
     return color;
   }
   public void start(){
       //implementation
   }
   public void goForward(int speed){
       //implementation
   }
}
```

Style

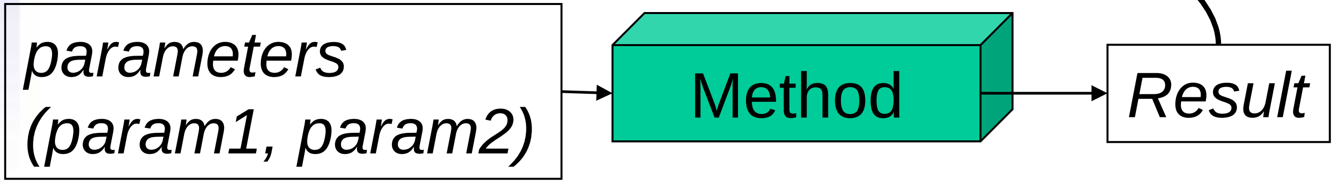
- Intuitive names
- 1st letter lower-case
- No blanks
- Camel-case `myMethod()`
- Indentation

Universidad
Carlos III de Madrid

Method declaration

```
public class Car{
  //...
   public String getColor(){
      //implementation
     return color;
   }
  //...
}
```

Car.java

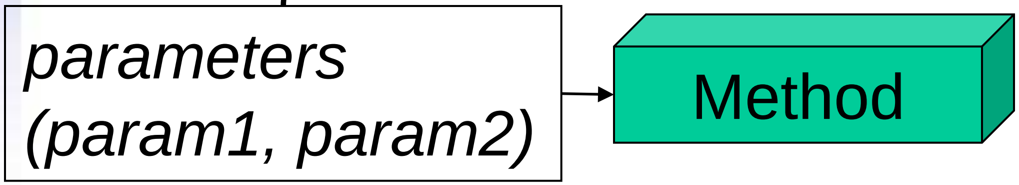


(modifiers) returnType methodName(type1 param1, type2 param2){
//implementation
return expression;
}

Method declaration

```
public class Car{
  //...
   public void goForward(int speed){
      //implementation
   }
  //...
}
```

Car.java

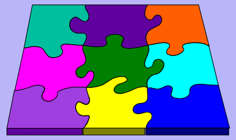


parameters (param1, param2)

Method

```
(modifiers) void methodName(type1 param1, type2 param2){
        // implementation
}
```

Method declaration

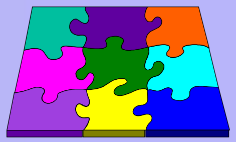


- Methods
  - Have 0, 1 or more ***parameters***
  - Define the ***data type*** of the result in their declaration (except constructors)
  - Can have ***local variables***. These variables **are not initialized** by default
- A method cannot contain other methods inside its body
- If one method produces a result, the last sentence of its execution must be a ***return sentence***

Constructor methods

- When an object is created, their members are ***initialized*** with the constructor method
- Constructor methods:
  - Have ***the same name*** as their container class
  - ***Do not have*** a ***returned data type*** in their declaration
- It is desirable that there be at least one
- There may be several that will be distinguished by the parameters acepted (***overload***)
- If there are no declared constructors, a default one is created and this ***default constructor*** initializes all variables to their own default value
- If the class has a constructor, the default constructor does not exists, but the programmer can declare a constructor without parameters with the same function than the default one.

The main method



- It is the ***first*** method than the runtime system calls to execute an application.
- The parameters of the main ***(String args[ ])*** represent an array of Strings that stores the arguments that are written in the command line to run the application

```
java HelloWorld arg1 arg2...
```

- ***void*** indicates that there are no return values
- ***static*** indicates that it is a global method. This method is the same for every instance of the class

Systems Programming

Imperative Java

Julio Villena Román (LECTURER)

<jvillena@it.uc3m.es>

CONTENTS ARE MOSTLY BASED ON THE WORK BY:

M.Carmen Fernández Panadero and Natividad Martínez Madrid

Scenario III: Method implementation

- Once the programmers' meeting has finished, you have to show your expertise before integrating into the team. Your boss asks you to implement several methods. As your first task, the methods are simple and work independently (do not invoke other attributes or methods)
- Objective:
  - Be able to decompose a problem in order to identify the basic steps for solving it (**algorithms design and representation)**
  - Use the basic structures of a programming language, variables, operators and flow control statements (loops, conditionals) to **implement an algorithm**
- Work plan:
  - Train in the design of algorithms and their representation. Break problems in small steps in order to resolve them without using code.
  - Memorize the syntax of Java in terms of (operators, loops and conditionals)
  - Train in use Java to implement previously designed algorithms
  - Take implementing ease and speed. Resolve typical problems (eg: in arrays, print all its elements, retrieve an specific element, swap elements between two positions, sorting)

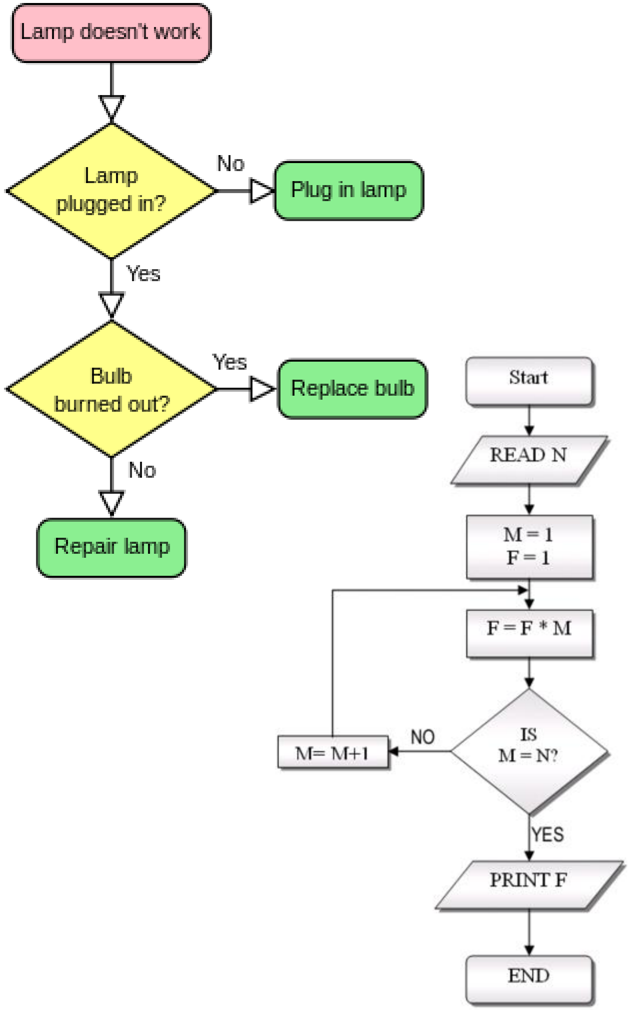
Step I: Thinking

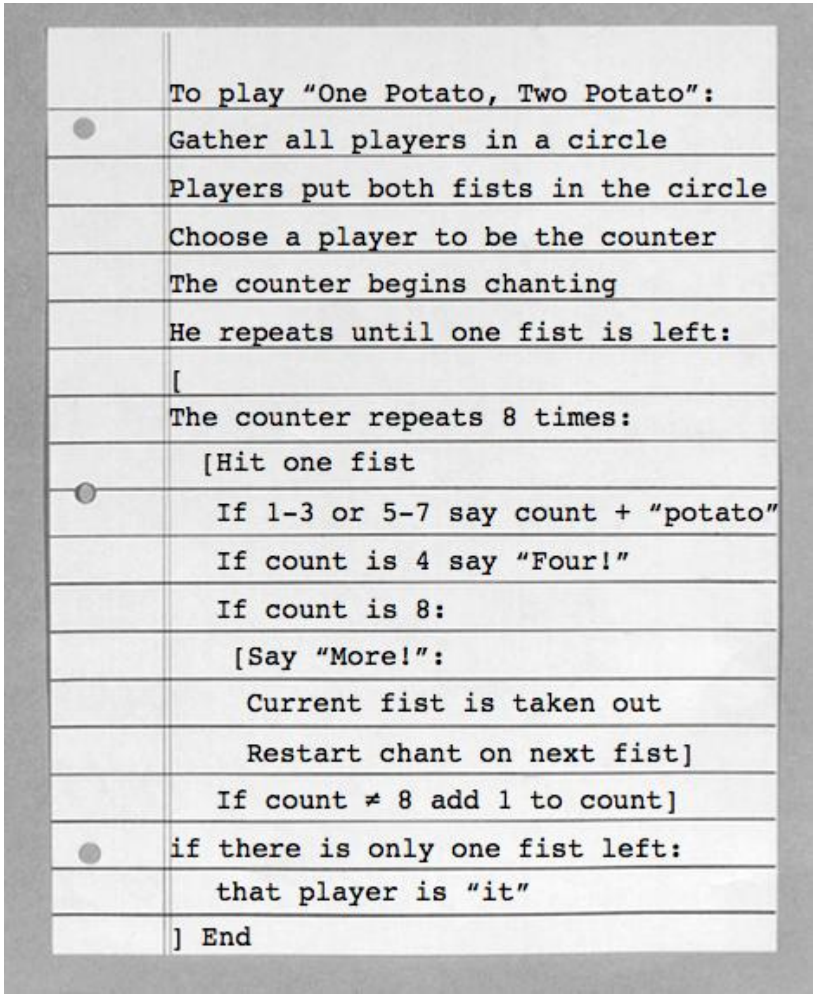
What tools do we have to represent algorithms?

- Once we have thought about the algorithm structure, we need to represent the steps to solve it:
  - Pseudocode
  - Flowcharts, organigrams
    - **Figures**: represent sentences
    - **Flow lines**: represent order in which they are executed

Step I: Thinking

Flowcharts vs Pseudocode





```
To play "One Potato, Two Potato":
Gather all players in a circle
Players put both fists in the circle
Choose a player to be the counter
The counter begins chanting
He repeats until one fist is left:
[
The counter repeats 8 times:
  [Hit one fist
   If 1-3 or 5-7 say count + "potato"
   If count is 4 say "Four!"
   If count is 8:
    [Say "More!":
     Current fist is taken out
     Restart chant on next fist]
   If count ≠ 8 add 1 to count]
if there is only one fist left:
   that player is "it"
] End
```

Step II: Algorithm implementation

What kind of expressions can we use in the method body?

- Variables
- Operators
  - By type
    - Aritmetical
    - Relational
    - Logical
  - By number of operands
    - Unary
    - Binary
- Operations with objects (not for this scenary)
  - Object creation
  - Attribute and method invocation

- Flow control structures (can be stacked and nested)
  - Sequence
  - Iteration (loops)
    - For
    - While
    - Do-while
  - Selection (conditionals)
    - If
    - If-else
    - Switch
- Breaking up the flow of execution
    - Break
    - Continue
    - Exception (not in this scenary)

Operators

- By **number** of operands
  - Unary (one operand ej: ++, --)
  - Binary (two operands ej: &&, %)
- By **type** of operator
  - Assignment (=)
  - Aritmetical (+, -, *, /, %)
  - Relational (>, >=, <, <=, ==, !=)
  - Logical (&&, II, !)
  - Conditional operator (`condition?sentence1:sentence2`)

```
System.out.println( studentGrade >= 5 ? "pass" : "not pass" );
```

Universidad
Carlos III de Madrid

Operators

Notes to remember

- Unary
  - i++ (first evaluates then increments)
  - ++i (first increments then evaluate)
  - Eg if i=3
    - i++ result = 3
    - ++i result = 4
- Binary (can be abbreviated)
  - x+=3 equals to x= x+3
- Assignment vs. comparison
  - The "=" operator assigns a value
    - Eg. `var = 5`, assigns `5` to `var`
  - The " == " operator compares
    - Eg. `var == 5`, returns `true` (after the previous assignment)
- The conditional operator is harder to understand than a simple if-else try not to use

Universidad
Carlos III de Madrid

Selection sentences
(Conditionals)

- If

```
if( condition) {
    sentences1;
}
```

- If-else

```
if( condition) {
    sentences1;
}else{
    sentences2;
}
```

```
if( condition) {
    sentences1;
}else if(condition2){
    sentences2;
}else{
    sentences3;
}
```

- switch

```
switch ( expression ) {
        case value1:
           sentences1;
          break;

        case value2:
           sentences2;
           break;

        default:
           sentences3;
        }
```

Universidad Carlos III de Madrid

Selection sentences

Notes to remember for if and if-else

- **Indent** the code contributes to its readability
- **Braces** { } fix the **scope** of every element declared between them
- **No braces** ~~{ }~~ is like to put them only in the first sentence

```
if (studentGrade >= 5)
    System.out.println ( “Pass” );
else
    System.out.println (“Not pass”);
```

Selection sentences

Notes to remember for switch

- Valid expression types: byte, short, int, long, char, ~~String~~
- Examples:
  - `int num=5; switch(num){}`
  - `char character='z' switch(character){}`
  - ~~`String string="myString"`~~ `switch(myString){}`
- If you do not use “**`break`**”, all the following code-blocks will be executed until a “**`break`**” or end of the **`switch`** is found
- It not necessary to place the block-code associated with each **`case`** between braces { }

Iteration sentences
(Loops)

- For:

```
for( initialization;condition;update) {
    sentences;
}
```

- While:

```
while( condition) {
    sentences;
}
```

- Do-while:

```
do {
    sentences;
}while(condition);
```

Universidad
Carlos III de Madrid

Iteration sentences

(Examples: for)

- Examples

```
int i=0;
for (i =0;i<10;)
{ i=i+2;}
```

```
int i=0;
for (i=13;i<10; i++)
{ i=i+2;}
```

```
int i=4;
for (;i<10;)
{ i=i+2;}
```

```
int i=0;
for ( ; ; )
{ i=i+2;}
```

```
int i sum;
for (i =0, sum=5;i<10;sum+=i)
{ i=i+8;}
```

How many times these loops are executed?

What is the value of “i” in each example at the end of the loop?

Iteration sentences

(Examples: for)

- Examples

```
int i=0;
f
{
```

```
4;
10;)
}
```

```
int
for (
{ i=i+
```

The one that will be most often used **(learn it by heart!)**

```
for (int i=0; i<5; i++) {
    //sentences
}
```

How man mes these loops are executed?

What is the value of “i” in each example at the end of the loop?

Universidad
Carlos III de Madrid

Iteration sentences

Notes to remember

- When the loop has several sentences (in initialization, comparation or update), they will be separated by commas

```
for(i=0, sum=0 ;  i<=n;  i++, sum+=n)  {
    sentences;
}
```

- Nested loops:
  - Program slows down
  - They are used to cover n-dimensional arrays (one loop per dimension)
- The sentences in a **`while`** might not run ever; in a **`do-while`** are executed at least once
- Avoid infinite loops (always check termination condition)
- A "**`for`**" loop always can be converted into a "**`while`**" one, and viceversa

Iteration sentences

Comparative

- for vs. while vs do while

	Init	Upd	Condition	Min Exe	Usage
for	Yes	Yes	Continue	0	High
while	Not	Not	Continue	0	High
do while	Not	Not	Continue	1	Low

- Init: Initialize variables
- Upd: Update variables
- Condition: Continue or exit
- Min exe: minimum number of times the block of code executes
- Usage: frequency of use of the control structure

Iteration sentences

Usage patterns

- When to use while or for

	for	while
The number of iterations is known (eg array)	X	
The number of iterations is unknown		X
Increase of variables in each cycle	X	
Variable initialization	X	X

E.g.: reading a file with while

E.g.: cover an array with for

Breaking up the flow of execution:
Break sentence

break: causes to break the execution and exit the structure in a **`while`**, **`for`**, **`do-while`** or **`switch`**

```
int j=0;
while(j<10){
  j++;
  break;
  System.out.println("This message is never printed");
}
System.out.println("j = "+j);
```

The loop runs only once and prints the message "j = 1"

Universidad
Carlos III de Madrid

Breaking up the flow of execution:

Continue sentence

continue: when **`continue`** appears in a **`while`**, **`for`** or **`do-while`** block of code, it skips the rest of the sentences of the loop and continues with the next iteration

```
int j=0
while(j<10){
  j++;
  continue;
  System.out.println("This message is never printed");
}
```

The message is never printed

Implementing a method:

Step 1.1: Think about the algorithm

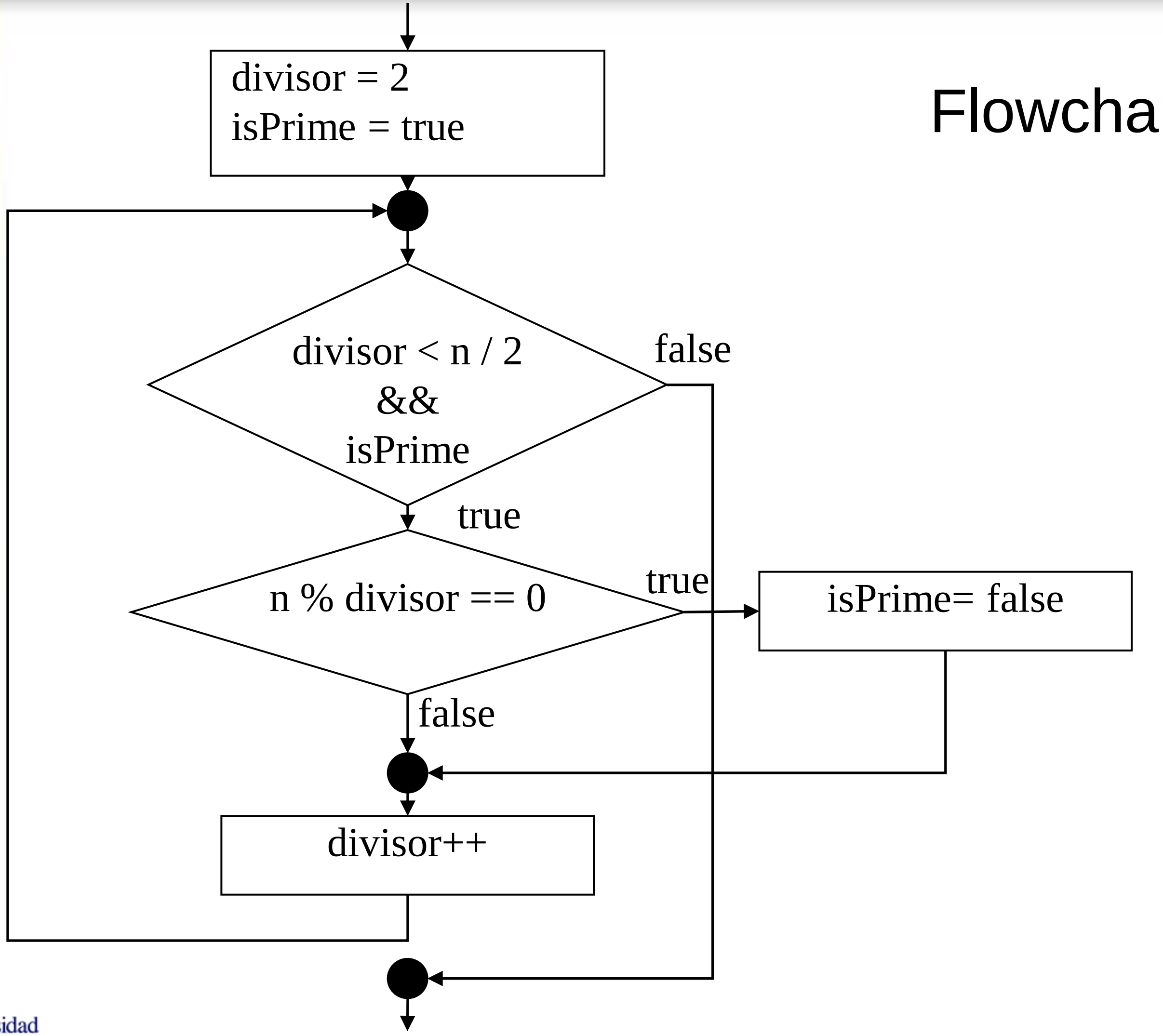
- **Problem**: Write a program that calculates whether a number n is prime

```
1   2   3   4   .   .   .   n/2    .   .   .   n
```

- **Step 1: Think about the algorithm** (split the problem into simpler steps)
  - Starting by 2, we check for each number if it is an integer divisor of n
  - Only needs repeating until n/2
  - Or until we find an integer divisor
  - *We will use a **sentinel***
    - Boolean variable that will help us control the loop

Implementing a method:

Step 1.2: Represent the algorithm



Flowchart

Implementing a method:

Step 2: Write the code

```
public boolean isAPrimeNumber (int number) {

   int divisor =2;
   boolean isPrime = true;


   while ((divisor < number/2) && isPrime) {
     if (number % divisor == 0)
       isPrime = false;
     divisor++;
   }

   System.out.println(“The number “ +number);
   if (isPrime)
     System.out.println(“ is prime.”);
   else
     System.out.println(“ is not prime.”);

   return isPrime;
 }
```

Universidad
Carlos III de Madrid

Implementing a method:
Examples: working with arrays

- Let’s practice
- Imagine that you have to implements methods to:
  - Print an array (practice loops)
  - Retrieve a specific element in an array
    - Practice: conditionals and nested loops
    - Practice comparation using different data types
      - Basic types (numbers, characters booleans)
      - String comparation
      - Object comparation
  - Swap two elements in an array (practice auxiliar variables)
  - Sort an array (copy elements between two arrays)

Review

Learning outcomes

- After this session you must be able to:
  - **Install and configure** an environment to work with Java
  - **Understand a program** with several files, be able to draw a class diagram, and know what is the first method that the runtime system calls to execute the application
  - **Identify basic structures associated with classes and objects** such as ***declarations*** of:
    - Classes
    - Members
      - Attributes
        - » **Basic types** (primitives, String)
        - » **Reference types** (objects and arrays)
      - Methods
        - » **main**
        - » **constructors**
        - » Normal methods
  - **Design and implements simple algorithms** inside the body of a method using **operators** and **basic control structures** (loops and conditionals)