

Object-Oriented Programming

Introduction in Java

Most Common Paradigm in Programming - Affects Analysis, Design (Design) and Programming

v Object-oriented analysis

- Determines what the system should do: Which actors involved? What activities are to be carried out?

- Decomposes the system into objects: What are they? What tasks will each object have to do?

v Object-oriented design

- Defines how the system will be implemented

- Model the relationships between objects and actors (you can use a specific language like UML)

- Uses and reuses abstractions such as classes, objects, functions, frameworks, APIs, design patterns

→ Java is object-oriented language, but also supports other paradigms (structured, imperative)

General features

v Open source software, available under the terms of the GNU General Public License

v Ease of internationalization (natively supports UNICODE characters)

v Wide range of libraries

v Facilities for creating distributed programs and multitasking

v Automatic freeing of memory by garbage collector process

v Dynamic code loading v Portability

Writing and running programs

v All source code is written in plain text files that end with the extension .java.

- They are compiled with the javac compiler into .class files.

v A .class file contains bytecode code that is executed by a virtual machine.

- does not contain native processor code

- runs on an instance of the Java Virtual Machine - JVM.

v **Advantages** => great portability

- The JVM is a program that loads and runs Java applications, converting bytecodes into native code.
- Thus, these programs are independent of the platform on which they operate.
- The same .class file can be run on different machines (running Windows, Linux, Mac OS, etc.).

v **Disadvantage** => lower performance

- The code is slower compared to the execution of native code (e.g. written in C or C ++).
-

Basic structure of a Java program

v What in other languages is called the main program is in Java a class declared as a public class in which we define a function called main ()

- Declared as public static void
 - With an args parameter, of type String []
- v This is the standard format, absolutely fixed
-

Primitive types and variables

v If we want to store data, we need to define variables, with a given type

Type	Size	Range	Default
boolean	1 bit	true or false	false
byte	8 bits	[-128, 127]	0
short	16 bits	[-32,768, 32,767]	0
char	16 bits	['\u0000', '\uffff'] or [0, 65535]	'\u0000'
int	32 bits	[-2,147,483,648 to 2,147,483,647]	0
long	64 bits	$[-2^{63}, 2^{63}-1]$	0
float	32 bits	32-bit IEEE 754 floating-point	0.0
double	64 bits	64-bit IEEE 754 floating-point	0.0

Declaring and initializing variables

v Local variables can be initialized in the following ways:

- at the time of definition: double weight = 50.3;
- ```
int day = 18;
```
- using an assignment instruction (symbol '='): double weight;
- ```
weight = 50.3;
```
- reading a value from the keyboard or other device: double km;
- ```
km = sc.nextDouble ();
```
-

## Operators

v Operators take one, two or three arguments and produce a new value.

v Java includes the following operators: - assignment: =

- arithmetic: \*, /, +, -, %, ++, -

- relational: <, <=, >, >=, ==, !=

- logical: !, ||, &&

- bit manipulation: &, ~, |, ^, >>, << - ternary decision operator?

---

### Unary arithmetic operators

v Unary increment (++) and decrement (-) operators can be used with numeric variables.

v When placed before the operand they are pre-increment (++ x) or pre-decrement (--x).

- the variable is first changed before being used.

v When placed after the operand they are post-increment (x++) and post-decrement (x--)

- the variable is first used in the expression and then changed.

---

### Constants / Literals

v Literals are invariant values in program 23432, 21.76, false, 'a', "Text", ...

v Usually the compiler knows how to determine its type and interpret it.

int x = 1234; char ch = 'Z';

v In ambiguous situations, we can add special characters:

- l / L = long, f / F = float, d / D = double

- 0x / 0Xvalue = hexadecimal value

- 0value = octal value.

long a = 23L;

double d = 0.12d;

float f = 0.12f; // required

---

### Variable type conversion

v We can store a value with less storage capacity in a variable with more storage

v The respective conversion will be done automatically: - byte -> short (or char) -> int -> long -> float -> double

v The reverse conversion generates a compilation error. - However, we can always perform a conversion

explicitly through a conversion operator:

int a = 3;

double b = 3.3;

double c = a; // automatic conversion from int to double

a = (int) b; // b is forcibly converted / truncated to int

---

## Print variables and literals

v `System.out.println (/ * ... * /);`  
- write whatever is between (..) and change the line  
v `System.out.print (/ * ... * /);`  
- write whatever is between (..) and do not change lines

---

## Read data

v We can use the Scanner class to read data from the keyboard.

```
import java.util.Scanner;
```

```
...
```

```
Scanner sc = new Scanner (System.in);
```

v Useful methods of the Scanner class:

- `nextLine ()` - read an entire line (String)
  - `next ()` - read a word (String)
  - `nextInt ()` - read an integer (int)
  - `nextDouble ()` - read a real number (double)
- 

## Precedence of operators

v The order of execution of operators follows rules of precedence.

```
int a = 5;
```

```
int b = -15;
```

```
double c = ++ a-b / 30;
```

v To change the order and / or clarify complex expressions, it is suggested that they use parentheses.

```
c = (++ a) - (b / 30);
```

---

## Referenced types

v Variables of these types do not contain the values but the addresses for accessing the effective values

v Include:

- Arrays - Objects
-

## Array

v We can declare arrays of variables of the same type

```
int [] vet1;
```

```
int vet2 []; // alternative syntax and equivalent to the previous one
```

v In addition to the declaration, we still need to define its size.

- initialization with default values:

```
int [] v1 = new int [3]; // vector with 3 elements: 0, 0, 0
```

- declaration and initialization with specific values 

```
int [] v2 = {1, 2, 3}; // vector with 3 elements: 1, 2, 3
```

// or

```
int [] v3 = new int [] {1, 2, 3};
```

---

v Vectors in Java have a fixed dimension and cannot be scaled at run time

v The new statement creates a vector with the indicated dimension and initializes all positions

- For primitive types with the default value - For referenced types, with the value null

---

## Access to vector elements

v Elements are accessed through indexes. - The index of the first element is 0 (zero).

```
int [] table = new int [3]; // indexes between 0 and 2 table [0] = 10;
```

```
table [1] = 20;
```

```
table [2] = 30;
```

```
table [3] = 11; // mistake!!
```

v The size of a vector v is given by v.length. 

```
System.out.println (table.length); // 3
```

---

## Multidimensional Vectors

v It is possible to create multidimensional vectors, i.e. vector vectors:

```
int [] [] a = {{1, 2, 3}, {4, 5, 6}}; System.out.println (a.length); // 2
```

```
System.out.println (a [0] .length); // 3 to [0] [1] = 9;
```

v Are vectors of arrays (arrays of arrays) - They are implemented using nesting

```
int table [] [] = new int [30] [20];
```

- Define table as being of type 

```
int [] []
```

- Allocates dynamically a vector of 30 elements, each one of them type 

```
int [20]
```

- Allocate 30 vectors of 20 integers and store the reference (address) for each of these in the vector of 30 positions

## Flow control in a program

- v The order of execution of a program's instructions is usually linear
  - one statement after another, in sequence
- v Some instructions allow you to change this order, deciding:
  - whether or not to execute a particular declaration
  - run a statement repeatedly, repeatedly
- v These decisions are based on Boolean expressions (or conditions)
  - which are assessed as true or false

---

## Boolean expressions

- v Boolean expressions return true or false.
- v Boolean expressions use operators
  - relational, equality, and logical (AND, OR, NOT)
- == equal to // Attention !!  $x == y$  is different from  $x = y$  != not equal to
- < less than
- > greater than
- <= less than or equal to
- >= greater than or equal to! NOT
- && AND
- || OR
- v Examples
  - $x >= 10$
  - $(y < z) \&\& (z > t)$

---

## Tables of truth

- v Boolean algebra is based on truth tables.
- v Considering A and B, eg:  $((y < z) \&\& (z > t))$ 
  - Both must be true for the expression  $A \&\& B$  to be true.
  - A true being is enough for the expression  $A || B$  to be true.

| <i>a</i> | <i>!a</i> | <i>a</i> | <i>b</i> | <i>a &amp;&amp; b</i> | <i>a    b</i> |
|----------|-----------|----------|----------|-----------------------|---------------|
| true     | false     | false    | false    | false                 | false         |
| false    | true      | false    | true     | false                 | true          |
|          |           | true     | false    | false                 | true          |
|          |           | true     | true     | true                  | true          |

## **Ternary operator**

v The ternary operator (? :) is also known as the conditional operator.

result = testCondition? valueIfTrue: valueIfFalse

- Evaluates an expression (1st operand) and, if true, the result is equal to the 2nd operand, otherwise the result is equal to the 3rd operand.

```
char code = 'F';
```

```
boolean capitalLetter = (code >= 'A') && (code <= 'Z'); System.out.println (capitalLetter?
"Yes": "no");
```

```
minVal = (a < b)? a: b;
```

---

## **Conditional statements**

v In Java there are two types of decision / selection statements:

- if

- switch

v The if statement has the following format:

```
if (Boolean expression)
```

```
// do_this;
```

```
else // optional
```

```
// do_this;
```

---

## **If decision statement**

v We can chain several if statements:

```
if (condition1)
```

```
 block1;
```

```
else if (condition2)
```

```
 block2;
```

```
else block3;
```

If a block includes more than one instruction, the block must be delimited by {..}.

## Switch selection statement

v The switch statement executes one of several paths (case), depending on the result of an expression

```
switch (expression) {
 case valor1:
block1;
 break;
 case valor2:
block2;
 break;
 // ...
 default:
Final block;
}
```

The result of the expression is searched in the list of alternatives existing in each case, in the order in which they are specified.

If the search is successful, the corresponding code block is executed. If there is a break statement, the execution of the switch ends. Otherwise, all the following options will be executed until break appears or the end of the switch is reached.

If the search is not successful and the default exists, the corresponding code block (Final block) is executed.

---

## Cycles

v Sometimes there is a need to execute instructions repeatedly.

- A set of instructions that are executed repeatedly is called a cycle.

v A cycle can be of the conditional type (while and do ... while) or of the counter type (for).

- We normally use conditional cycles when the number of iterations is unknown and counter-type cycles when we know the number of iterations in advance.

---

## While cycle

v The while cycle runs as long as the cycle condition is true.

- The condition is tested before each iteration of the cycle.

```
while (condition) execute_block;
- Example:
Scanner sc = new Scanner (System.in); int grade = -1;
while ((grade > 20) || (grade < 0)) {
 System.out.println ("Insert the student's grade.");
 nota = sc.nextInt ();
}
sc.close ();
```

---

## Do-While cycle



- v The do ... while cycle runs a first time and then check if it is necessary to repeat.
- The condition is tested at the end of each iteration of the cycle.

```
of
execute_block;
while (condition);
- Example:
Scanner sc = new Scanner (System.in); int note;
of {
System.out.println ("Insert the student's grade.");
nota = sc.nextInt ();
} while ((grade > 20) || (grade < 0));
sc.close ();
```

---

### **for cycle**

- v The for cycle is more general because it supports all situations of repeated execution.

```
for (initialization; condition; update)
bloco_a_executar;
```

1. Before the 1st test, do the initialization (only once)
  2. Then perform the condition test.  
If true, execute the block, if false ends
  3. At the end of each term, execute the update part and resume in point 2 above.
- 

### **For loop (foreach syntax)**

```
v The for cycle, when used with vectors, can have a more succinct form (foreach).
Scanner sc = new Scanner (System.in); double [] a = new double [5];
for (int i = 0; i < a.length; i ++)
a [i] = sc.nextDouble ();
for (double el: a)
System.out.println (el);
sc.close ();
```

---

### **Break and continue instructions**

- v We can end the execution of a block of instructions with two special instructions:
    - break and continue.
  - v The break statement allows for the immediate exit of the code block being executed.
    - It is normally used in switch but can also be used in cycles.
  - v The continue statement allows you to end the execution of the current iteration, forcing the passage to the next iteration (i.e. it does not end the cycle).
-

## Modular programming

- v Organization of programs as independent modules.
  - v Why? → Easier to share and reuse code to create bigger programs.
  - v In Java we can consider each .java file as a module.
  - v Each .java file contains a (public) class.
- 

## Basic class concept

- v Definition of a class (Example.java file):

```
public class Example {
 // Dice
 // methods
}
```

- v The Example.java file must contain a public class called Example.
    - We should use a nomenclature of type Person, SomeClass, SomeLongNameForClass, ...
    - Java is a case-sensitive language (i.e. Example! = Example)
  - v This class must be declared as public
- 

## Functions

- v A role
    - Performs a task.
    - It has zero or more input arguments. - Returns zero or an output value.
  - v Applications
    - Scientists use mathematical functions to calculate formulas.
    - Programmers use functions to build modular programs.
    - We will use them for both purposes.
  - v Examples  
Math.random (), Math.abs (), Integer.parseInt () System.out.println (), main ()
- 

## Static methods

- v To implement a function (static method), we need
    - Create a name
    - Declaring the type and name of the argument (s) - Specifying the type for the return value
    - Implement the method body
    - End with return declaration
- ```
public static void myFunction () {System.out.println ("My Function called");  
}  
public static double doisXSquare (double x) {return 2 * x * x;  
}
```
-

java.lang.Math

- ✓ The Math class contains static methods for performing basic numeric operations
- exponential, logarithmic, square root and trigonometric functions.

Modifier and Type	Method and Description
static double	abs(double a) Returns the absolute value of a double value.
static float	abs(float a) Returns the absolute value of a float value.
static int	abs(int a) Returns the absolute value of an int value.
static long	abs(long a) Returns the absolute value of a long value.
static double	acos(double a) Returns the arc cosine of a value; the returned angle is in the range 0.0 through <i>pi</i> .

- ✓ General functions Math.abs ()

Math.ceil () Math.floor () Math.floorDiv () Math.min () Math.max () Math.round ()
Math.random ()

- ✓ Exponential, logarithmic functions Math.exp ()

Math.log () Math.log10 () Math.pow () Math.sqrt ()

- ✓ trigonometric functions

Math.PI Math.sin () Math.cos () Math.tan () Math.asin () Math.acos () Math.atan ()
Math.atan2 () Math.sinh () Math.cosh () Math.tanh () Math.toDegrees () Math.toRadians ()

The String class

- ✓ The java.lang.String class makes it easy to manipulate character strings.

- ✓ Example:

```
String s1 = "java"; // creating string by java string literal
char ch [] = {'s', 't', 'r', 'i', 'n', 'g', 's'};
String s2 = new String (ch); // converting char array to string
System.out.println (s1);
System.out.println (s2);
```

java strings

String concatenation

- ✓ String concatenation

```
String data = "feve" + "reiro";
date = 10 + date;
date += "de" + 2019;
System.out.println (data);
```

- ✓ Objects of type String are immutable (constants).

- All methods whose objective is to modify a String in the actually build and return a new String
- The original String remains unchanged.

v Alternative use of type **StringBuilder**

```
StringBuilder sb = new StringBuilder (); sb.append (10);  
sb.append ("feve");  
sb.append ("reiro");  
sb.append ("de"); sb.append (2019);  
String data = sb.toString (); System.out.println (data);  
10 February 2019
```

String class methods

v This class has a set of methods that allow you to perform many operations on text.

char	charAt(int index) Returns the char value at the specified index.
int	codePointAt(int index) Returns the character (Unicode code point) at the specified index.
int	codePointBefore(int index) Returns the character (Unicode code point) before the specified index.
int	codePointCount(int beginIndex, int endIndex) Returns the number of Unicode code points in the specified text range of this String.
int	compareTo(String anotherString) Compares two strings lexicographically.
int	compareToIgnoreCase(String str) Compares two strings lexicographically, ignoring case differences.
String	concat(String str) Concatenates the specified string to the end of this string.
boolean	contains(CharSequence s) Returns true if and only if this string contains the specified sequence of char values.

Character length and access

v The length (number of characters) of a String can be determined using the length method.

v Accessing a character is done with the charAt (int index) method.

v Example:

```
String s1 = "University of Aveiro"; System.out.println (s1.length ());  
for (int i = 0; i < s1.length (); i ++)  
System.out.print (s1.charAt (i) + ",");
```

22

Aveiro University,

String Comparison

v Some methods

- equals, equalsIgnoreCase, compareTo

v Examples:

```
String s1 = "Aveiro";
```

```
String s2 = "aveiro";
```

```
System.out.println (s1.equals (s2)? "Equals": "Different"); System.out.println
```

```
(s1.equalsIgnoreCase (s2)? "Equal": "Different"); System.out.println (s1.compareTo (s2));
```

```
// <0 (minor s1), 0 (equal), > 0 (major s1)
```

Comparison of subStrings

v We can analyze parts of a String - contains, substring, startsWith, endsWith, ...

v Examples:

```
String s1 = "Aveiro";
```

```
String s2 = "aveiro";
```

```
System.out.println (s1.contains ("ve")); // true System.out.println (s1.substring (1, 3)); // ve
```

```
System.out.println (s1.startsWith ("ave")); // false System.out.println (s1.endsWith ("ro")); //
```

```
true
```

Formatting Strings

v The format method returns a new String formatted according to format specifiers.

```
long seconds = 347876;
```

```
String s1 =
```

```
String.format ("% 02d hours,% 02d minutes and% 02d seconds \n",
```

```
seconds / 3600,
```

```
(3600% seconds) / 60,
```

```
seconds% 60);
```

```
System.out.println (s1);
```

```
96 hours, 37 minutes and 56 seconds
```

v System.out.printf is a method, alternative to System.out.print, that uses formatting.

v Example:

```
long seconds = 347876;
```

```
System.out.printf ("% 02d hours,% 02d minutes and% 02d seconds \n",
```

```
seconds / 3600,
```

```
(3600% seconds) / 60,
```

```
seconds% 60);
```

Regular expressions (regex)

v Allows you to define patterns that can be searched for in Strings.

- The complete list of supported constructs is described in the documentation for the `java.util.regex.Pattern` class.

v The `matches` method of the `String` class checks whether a `String` includes standard data.

v Examples:

```
String s1 = "123"; System.out.println (s1.matches ("\\ d {2,4}"));
```

```
// 2-4 digits in a row
```

```
s1 = "abcdefg"; System.out.println (s1.matches ("\\ w {3,}"));
```

```
// at least 3 alphanumeric characters
```

```
true true
```

- | | |
|----------|---|
| - . | qualquer caracter |
| - \\d | dígito de 0 a 9 |
| - \\D | não dígito [^0-9] |
| - \\s | "espaço": [\\t\\n\\x0B\\f\\r] |
| - \\S | não "espaço": [^\\s] |
| - \\w | carater alfanumérico: [a-zA-Z_0-9] |
| - \\W | carater não alfanumérico: [^\\w] |
| - [abc] | qualquer dos carateres a, b ou c |
| - [^abc] | qualquer carater exceto a, b e c |
| - [a-z] | qualquer carater das gamas (inclusivas) a-z |
| - X? | um ou nenhum X |
| - X* | nenhum ou vários X |
| - X+ | um ou vários X |

Split method

v The `split` method separates a `String` into parts based on a regular expression and returns the resulting `String` vector.

What is a class?

v Classes are specifications for creating objects

v A class represents a complex data type

v Classes describe

- Types of data that make up the object (which can store)

- Methods that the object can perform (what they can do)

v Example:

```
public class Book {  
    String title;  
    int pubYear;  
}
```

Objects

v Objects are instances of classes

```
Book oneBook = new Book ();
```

```
Book otherBook = new Book ();
```

```
Book book3 = new Book ();
```

v All objects are manipulated through references

```
Person name1, name2;
```

```
name1 = new Person ("Manuel");
```

```
name2 = name1;
```

v All objects must be explicitly created.

```
Circle c1 = new Circle (p1, 5);
```

```
String s = "Book"; // Strings are an exception!
```

v In Java, objects are stored in heap memory and manipulated through a reference (variable), stored in the stack.

- Have been (attributes)
- Behave (methods)
- Have identity (the reference)

Methods

v Methods, messages, functions, procedures

v Invocation is always done through the dot notation.

```
oneBook.setTitle ("Tourism in Aveiro"); otherBook.setPubYear (2000);
```

v The recipient of the message is always on the left.

v The receiver is always a class or reference for an object.

```
Math.sqrt (34); otherBook.setPubYear (2000);
```

Namespace – Package

v In Java, the management of the namespace is done through the concept of package.

v Why namespace management?

V Avoid class name conflicts

- We generally have no problem distinguishing the names of the classes we build.
 - But how do we ensure that our Book class does not collide with another that may possibly already exist?
-

Package import

v Use

- Classes are referenced using their absolute names or using the import primitive.

`import java.util.ArrayList import java.util. *`

- The import clause must always appear on the first line of a program.

v When we write, `import java.util. *;`

- we are indicating a path for a package of classes allowing us to use them through simple names:

`ArrayList <String> al = new ArrayList <> ();`

v Otherwise we would have to write: `java.util.ArrayList <String> al = new java.util.ArrayList <> ();`

Create a package

v In the first line of code: `package poo;`

- ensures that the public class of this compilation unit will be part of the poo package.

v The namespace is based on a structure of sub-directories

- This package will correspond to a directory entry: `$ CLASSPATH / poo`

- Good practice to use inverted DNS: `en.ua.deti.poo`

v Its use will be in the form:

`poo.Book sr = new poo.Book ();`

- or

`import poo. *`

`Book sr = new Book ();`

toString

v All objects in Java understand the `toString ()` message

`Book oneBook = new Book (); oneBook.setTitle ("Tourism in Aveiro"); System.out.println (oneBook); // oneBook.toString ()`

`Book @ 33909752`

v It is usually necessary to redefine this method to provide a more suitable result.

`@Override`

`public String toString () {`

`return "Book: title =" + title + "; pubYear =" + pubYear;`

`}`

`Book: title = Tourism in Aveiro; pubYear = 0`

Insecure Programming

v Many of the programming errors result from:

- uninitialized data - some programs / libraries need to initialize components and make the task dependent on the programmer.
- incorrect management of dynamic memory - "forgetting" to free memory, insufficient reserve, ...

v To solve these two problems, the Java language uses the concepts of:

- constructor
 - garbage collector
-

Member initialization

v Within a class, initializing variables can be made in your statement.

```
class Measurement { int i = 25;  
char c = 'K'; }
```

v However, this initialization is the same for all objects of the class

- The most common solution is to use a constructor. class Measurement {
int i;
char c;
Measurement (int im, char ch) {
i = im; c = ch; }

Constructor

v The initialization of an object can imply the simultaneous initialization of several types of data.

v A special member function, constructor, is invoked whenever an object is created.

v Instantiation is done using the new operator. Car c1 = new Car ();

v The constructor is identified by the same name as the class.

v This method can be overloaded from to allow different ways of initialization.

```
Car c2 = new Car ("Ferrari", "430");
```

v Does not return any value

v Always assume the class name

v Can have input parameters

v It is called only once: when creating the object

```
public class Book {  
    String title;  
    int pubYear;  
    public Book (String t, int py) {title = t;  
    pubYear = py; }  
    // ...  
}
```

Default constructor

- v A constructor without parameters is called the default constructor or the default constructor.
- It is automatically created by the compiler if no constructor is specified.

```
class Book {  
    String title;  
    int pubYear;  
}
```

```
Book m = new Book (); // OK
```

- If there is at least one constructor associated with a given class, the compiler no longer creates the default one.

```
class Book {  
    String title;  
    int pubYear;  
    Book (int py) {pubYear = py; }  
}
```

```
Book m = new Book (); // mistake!
```

Default values for primitive types

- v If a variable is used as a member of a class the compiler takes care of initializing it by default

- This is not guaranteed in the case of local variables so we must always initialize all variables
-

Overloading

- v We can use the same name in several functions - as long as they have different arguments and conceptually perform the same action

```
void sort (int [] a);  
void sort (Book [] b);
```

- v The static link checks the signature of the function (name + arguments)

- v You cannot distinguish functions by the return value

- because it is allowed to invoke, e.g., void f () or int f () in the form f () where the return value is not used
-

Overlapping constructors

- v Allow different ways to start an object of a given class.

```
public class Book {  
    public Book (String title, int pubYear) {...} public Book (String title) {...}  
    public Book () {...}  
}
```

```
Book c1 = new Book ("The stone raft", 1986); Book c2 = new Book ("Galveias");
```

```
Book c3 = new Book ();
```

This reference

v The this reference can be used within each object to reference that same object

```
public class Book {
    String title;
    int pubYear;
    public Book (String title, int pubYear) {
        this.title = title;
        this.pubYear = pubYear; }
    }
    class Faucet {
        void closes () {/ * ... * /}
        void locks () {closes (); / * or this.close () * /}
```

v Another use of the this reference is to return, in a given method, the reference to that object.
- can be used in a chain (lvalue).

```
public class Contador { int i = 0;
    Contador increment() {

        i++; return this;
    }

    void print() { System.out.println("i = " + i);

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

        Contador x = new Contador();

        x.increment().increment().increment().print(); }

    }
```

Invoke one constructor within another

v When we write several constructors, we can call one within the other.
- the reference this allows to invoke another constructor on the same object.

```
public Book (String title, int pubYear) {this.title = title;
    this.pubYear = pubYear;
    }
    public Book (String title) {
        this (title, 2000);
    }
```

v This form can only be used within constructors;
- in this case, this must be the first instruction to appear;
- it is not possible to invoke more than one this constructor.

The static concept

- v Static methods have no associated this reference.
 - v Therefore, it is not possible to invoke non-static methods from static methods.
 - v You can invoke a static method without objects of that class.
 - v Static methods have the semantics of global functions (they are not associated with objects).
-

Static elements

- v Static variables, or class variables, are common to all objects in that class.
- v Your declaration is preceded by static.
- v The invocation is made on the class identifier

```
class Test {  
    public static int a = 23;  
    public static void someFunction () {...} // ...  
}  
Test.someFunction (); // invoked on the Test class  
s1 = new Test ();  
Test s2 = new Test (); System.out.println (Test.a);  
Test.a ++; // s1.a and s2.a will be 24
```

Initialization of static members

- v If static members are initialized, they take priority over all others.
- v A static member is only initialized when the class is loaded (and only then)
 - when the first object of that class is created or when it is used for the first time.

- v We can use a special block - initializer

static - to group the initializations of
static members

```
class Circle {  
    static private double list [] = new double [100]; static { // static initializer  
        // list initialization []  
    }  
}
```

Object Vectors

- v A vector in Java represents a set of references
 - previous rules apply to default values
`int [] a = new int [10]; // 10 int`
`Book [] xC = new Book [10]; // 10 refs! Not 10 Books !!`
-

Range / Scope

v An automatic variable can be used as long as it is defined until the end of that context

v Each block can have its own objects

```
{intk;  
{inti;  
    } // 'i' is not visible here  
    } // 'k' is not visible here
```

v Illegal example

```
{intx = 12;  
{intx = 96; / * error! * /  
}
```

Range of references and objects

v Example with references and objects {

```
Book b1 = new Book ("Elephant Memory"); }
```

```
    // 'b1' is no longer visible here
```

v In this case, reference b1 is released (removed) and the object can no longer be used

- Will be removed by the Garbage collector

Encapsulation

v Key POO ideas

- Information Hiding (Encapsulation)

- Inheritance

- Polymorphism

v Encapsulation

- Separation between what cannot be changed (interface)

and what can change (implementation)

- Interface visibility control (public, default, protected, private)

v It allows you to create different levels of access to the data and methods of a class.

v The levels of access control that we can use are, from the highest to the lowest access:

- public - can be used in any class

- "default" - visible within the same package

- protected - visible within the same package and derived classes

- private - only visible within the class

v A method of a class has access to all the information and methods of that class

Modifiers / Selectors

v Encapsulation allows you to hide an object's internal data

- But, sometimes it is necessary to access this data directly (reading and / or writing).

v Important rules!

- All attributes must be private.

- Access to the internal information of an object (private part) must always be done, through functions of the public interface.

v Selector

- Returns the current value of an attribute

```
public float getRadius () { // or public float radius () return radius;
}
```

v Modifier

- Modify the state of the object

```
public void setRadius (float newRadius) {
// or public void radius (float newRadius)
this.radius = newRadius; }
```

Private methods

v Internally a class can have several private methods that are only used internally by other methods of the class.

```
// example of auxiliary functions in a class class Screen {
    private int row ();
    private int col ();
private int remainingSpace ();
// ...};
```

What can a class contain

v The definition of a class can include:

- zero or more data attribute declarations
- zero or more method definitions
- zero or more constructors
- zero or more static initialization blocks
- zero or more class definitions or internal interfaces

v These elements can only occur within the 'class ClassName {...}' block

- "everything belongs" to some class
 - only 'import' and 'package' can occur outside of a 'class' (or 'interface') statement
-

Good practice

- v The construction semantics of an object must make sense
Personalap = new Pessoa (); L
Person p = new Person ("António Nunes"); K
Pessoa p = new Pessoa ("António Nunes", 12244, dataNasc); J
 - v We must give a minimum of public visibility when accessing an object
 - Only what is strictly necessary
 - v Sometimes it makes more sense to create a new object than that changing existing attributes
Point p1 = new Point (2,3); p1.set (4,5); L
 - v Join members of the same type
 - Do not mix static methods with instance methods
 - v Declaring variables before or after methods - Do not mix methods, constructors and variables
 - v Keep constructors together, preferably at the beginning
 - v If you need to define static blocks, define only one at the beginning or end of the class.
 - v The order of the members is not important, but following conventions improves the readability of the code
-

Class Relations

- v Part of the class modeling process consists of:
 - Identify entities applying for classes
 - Identify relationships between these entities
 - v Relationships between classes are easily identified using some real models.
 - For example, a Digital Clock and an Analog Clock are both types of Clock (specialization or inheritance).
 - A Digital Clock, on the other hand, contains a Battery (composition).
 - v Relations: - IS-A
 - HAS-A
-

Inheritance (IS-A)

- v IS-A indicates specialization (inheritance), that is, when a class is a subtype of another class.

- v For example:
 - Pinheiro is an (IS-A) tree.
 - A Digital Clock is an (IS-A) Clock.
- ```
class Watch { /* ... */
}
class Digital Watch extends Watch { /* ... */
}
```
-

## Composition (HAS-A)

v HAS-A indicates that a class is composed of objects from another class.

v For example:

- Forest contains (HAS-A) Trees.
- A Digital Watch contains (HAS-A) Battery.

```
class Stack { /* ... */
}
class Digital Watch extends Watch {
 Stack p;
 /* ... */
}
```

---

## Class reuse

v Whenever we need a class, we can:

- Use an existing class that meets the requirements
  - Write a new class from "from scratch"
  - Reuse an existing class using composition
  - Reuse an existing class through inheritance
- 

## Inheritance Identification

v Typical signs that two classes have an inheritance relationship

- Have common aspects (data, behavior)
- Have different aspects
- One is a specialization of the other

v Examples:

- Cat is a Mammal - Circle is a Figure - Water is a Drink
- 

## Inheritance – Concepts

v Inheritance is one of the main characteristics of OOP

- The CDeriv class inherits, or is derived from, CBase when CDeriv represents a subset of CBase

v Inheritance is represented in the form: class CDeriv extends CBase { /\* ... \*/ }

v Cderiv has access to CBase data and methods - which are not private in CBase

v A base class can have multiple derived classes but a derived class cannot have multiple base classes

- In Java multiple inheritance is not possible
-



## Constructors with parameters

v In parameterized constructors the base class constructor is the first statement to appear in a derived class constructor.

```
class Game {
 int num;
Game (int code) {...}
// ...}
class BoardGame extends Game { // ...
BoardGame (int code, int numPlayers) {super (code);
// ...}
}
```

---

## Method Inheritance

v When inheriting methods, we can:

- keep them unchanged

```
class Person {
private String name;
public Person(String n) { name = n; } public String name() { return name; } public String toString() { return "PERSON";}
}
class Student extends Person {

private int nmec;
public Student(String s, int n) { super(s); nmec=n; } public int num() { return nmec; }

}
public class Test {

public static void main(String[] args) { Student stu = new Student("Andreia", 55678); System.out.println(stu + " : " +
stu.name() + ", " + stu.num()); }

}
```

- add new features to it

```
class Person {
private String name;
public Person(String n) { name = n; } public String name() { return name; } public String toString() { return "PERSON";}
}

class Student extends Person {
private int nmec;
public Student(String s, int n) { super(s); nmec=n; } public int num() { return nmec; }
public String toString()

{ return super.toString() + " STUDENT"; } }
```

- redefine them

```
class Person {
private String name;
public Person(String n) { name = n; } public String name() { return name; } public String toString() { return "PERSON";}

}

class Student extends Person {
private int nmec;
public Student(String s, int n) { super(s); nmec=n; } public int num() { return nmec; }
public String toString() { return "STUDENT"; }

}
```

---

## Inheritance and access control

- v We cannot reduce the visibility of inherited methods in a derived class
- Methods declared as public in the base class must be public in the subclasses
- Methods declared as protected in the base class must be protected or public in the subclasses. They cannot be private
- Methods declared without access control (default) cannot be private in subclasses
- Methods declared as private are not inherited

---

## Final

- v The final classifier indicates "cannot be changed"
- v Can be used on:
  - Data - final constants `int i1 = 9;`
  - Methods - non-resettable `final int swap (int a, int b) {/:`  
`}`
  - Classes - not inherited `final class Rat {/ ...`  
`}`
- v "final" sets constant attributes of primitive types but does not set objects or vectors
- in these cases what is constant is simply the reference to the object

---

## Inheritance - Good Practices

- v Program for the interface and not for the implementation
  - v Look for aspects common to various classes and promote them to a base class
  - v Minimize relationships between objects and organize related classes within the same package
  - v Use inheritance judiciously - whenever possible, favor composition
-

## Methods common to all objects

v In Java, all classes are derived from the super class java.lang.Object

v Methods of this class: - toString ()

- equals ()

- hashCode () - finalize ()

- clone ()

- getClass () - wait ()

- notify ()

- notifyAll ()

---

### toString ()

Circle c1 = new Circle (1.5, 0, 0); System.out.println (c1);

c1.toString () is automatically invoked

Circle @ 1afa3

- the toString () method must always be redefined to behave according to the object

```
public class Circulo { //
```

```
@Override
```

```
public String toString () {
```

```
return "Centro: (" + centro.x () + "," + centro.y () + ")" + "Radius:" + radius;
```

```
}
```

---

### equals ()

v The expression c1 == c2 checks whether references c1 and c2 point to the same object

- If c1 and c2 are automatic variables, the previous expression compares values

v The method tests whether two objects are equal

```
Circle p1 = new Circle (0, 0, 1);
```

```
Circle p2 = new Circle (0, 0, 1); System.out.println (p1 == p2); // false System.out.println
```

```
(p1.equals (p2)); // false (why?)
```

v equals () must be redefined whenever objects of this class can be compared

- Circle, Point, Complex ...

v Equality properties

- reflective: - symmetrical: - transitive:

```
x.equals (x) à true x.equals (y) à y.equals (x)
```

```
x.equals (y) AND y.equals (z) à x.equals (z)
```

v We must respect the Object.equals (Object o) signature

```
public class Circulo { // ...
```

```
@Override
```

```
public boolean equals (Object obj) { // ... }
```

```
}
```

v Problems

- What if 'obj' is null?

- What if you reference an object other than a circle?

---

## hashCode ()

v Whenever the equal () method is rewritten, hashCode () must also be

- Equal objects must return equal hash codes

v The purpose of the hash is to help identify any object through an integer

```
// Circulo.hashCode () - Very simple example !!! public int hashCode () {
return radius * centro.x () * centro.y (); }
```

```
// ..
```

```
Circle c1 = new Circle (10,15,27); Circle c2 = new Circle (10,15,27); Circle c3 = new Circle
(10,15,28);
```

- Building a good hash function is not trivial. Other sources are recommended for its construction

---

## Summary - Why inheritance?

v Many real objects have this characteristic

v Allows you to create simpler classes with more watertight and better defined features

- We should avoid classes with very "extensive" interfaces

v It allows to reuse and extend interfaces and code

v Lets you take advantage of the polymorphism

---

## Upcasting and downcasting

Handgeschrieben Seite 24

---

## Polymorphism

v Base idea:

- the type declared in the reference need not be exactly the same as the type of the object it points to - can be of any derived type

```
Circle c1 = new Target (...); Object obj = new Circle (...);
```

v Polymorphic reference

- T ref1 = new S (); // OK as long as the whole S is a T

v Polymorphism is, together with Inheritance and Encapsulation, one of the fundamental characteristics of OOP.

- Different shapes with similar interfaces.

v Other designations:

- Dynamic binding, late binding or run-time binding

v This feature allows us to take more advantage of the heritage.

- We can, for example, develop an X () method with CBase parameter with the guarantee that it accepts any argument derived from CBase.

- The X () method is only resolved in execution.

v All methods (with the exception of final ones) are late binding.

- The final attribute associated with a function, prevents it from being redefined and simultaneously gives an indication to the compiler for static linking (early binding) - which is the only way of linking in languages with C.

---

## Generalization

v Generalization consists of improving the classes of a problem in order to make them more general.

v Forms of generalization:

v Make the class as comprehensive as possible to cover the widest range of entities.

class ZooAnimal;

v Abstract different implementations for similar operations in abstract classes in a upper level.

ZooAnimal.draw ();

v Gather behaviors and characteristics and make them rise as far as possible in the class hierarchy.

ZooAnimal.weight;

---

## Abstract classes

v A class is abstract if it contains at least one abstract method.

- An abstract method is a method whose body is not defined.

```
public abstract class Forma {
```

```
// can define constants
```

```
public static final double DOUBLE_PI = 2 * Math.PI;
```

```
// can declare abstract methods public double area (); public double perimeter ();
```

```
// can include non-abstract methods
```

```
public String aka () {return "euclidean"; }}
```

v An abstract class is not instantiable.

Form f; // OK. We can create a reference to Form f = new Form (); // Mistake! We cannot create Forms

v In an inheritance process, the class is no longer abstract only when it implements all abstract methods.

```
public class Circulo extends Forma {
```

```
protected double r;
```

```
public double area () {return Math.PI * r * r;
```

```
}
```

```
public double perimeter () {return DOUBLE_PI * r;
```

```
}}
```

Form f;

f = new Circle (); // OK! We can create Circles

---

## Interfaces

v An interface works like a class that only contains signatures

- From Java 8 onwards it started to include default and static methods.

```
public interface Drawable { // ...
}
```

v Acts as a protocol to the classes that implement them.

```
public class Grafico implements Drawable { // ...
}
```

v A class can inherit from a single base class and implement one or more interfaces.

---

## Main features

v All of its methods are implicitly abstract. - The only allowed modifiers are public and abstract.

v An interface can inherit (extends) more than one interface.

v Constructors are not allowed.

v Variables are implicitly static and constant

- static final ..

v A (non-abstract) class that implements a

interface must implement all of its methods. v An interface can be empty

- Cloneable, Serializable

v You cannot create an instance of the interface

v You can create a reference to an interface

---

## Java 8 interfaces

v Default methods

- We can define the body of the methods in the interface

v Static methods

- We can define the body of static methods in the interface. Must be invoked on the interface (Interface Methods)

v Functional interfaces

- (we'll talk about this later ...)

v why (complicate with) these new features?

v Default Methods

- Offer a default implementation

- Can be rewritten in classes that implement the interface

v Static Methods

- Similar to default methods

- Cannot be rewritten in classes that implement the interface

---

## Abstract Classes versus Interfaces

### Abstract Classes

- v Objective: to describe entities and properties
- v Can implement interfaces
- v Allow simple inheritance
- v Relationship in the simple class hierarchy

### Interfaces

- v Objective: describe functional behaviors
  - v Cannot implement classes
  - v Allow multiple inheritance
  - v Horizontal implementation in the hierarchy
- 

### Instanceof

- v Statement indicating whether a reference is a member of a class or interface
  - v Example, considering  
class Dog extends Animal implements Pet {...}  
Animal fido = new Dog ();  
v the following statements are true:  
if (fido instanceof Dog) .. if (fido instanceof Dog) .. if (fido instanceof Pet) ..
- 

## Enumerated Types - Motivation

- v How can we restrict the possible values of a variable?

- Example 1

```
public class SomeClass {
 void someFunction (int weekday) {
 // weekday must be between 1 ()
```

Sunday

) e7 (

Saturday

}}

- Example 2

```
public class SomeOtherData {
 void someFunction (int estacaoDoAno) { // estacaoDoAno must be
 }}
```

---

## Solution

v Enumerated types (enum)

- We created a new data type with predefined values and constants
- Example 1

```
public
enum Day {
, Monday Tuesday Wednesday Thursday Friday Saturday
Sunday

}
public class SomeClass {
void someFunction (Day of the Week) {
// diaDaSemana varies between Dia.domingo and Dia.sabado}
}
```

---

- Example 2

Convention: we should preferably use capital letters

```
public
enum
,, FALL, WINTER

}
SPRING
Season {
SUMMER
public class SomeOtherData {
void someFunction (estacaoDoAno) {
// and }
}
```

---

## Enumerated Types – characteristics

v Important Value: “compile-time type safety” i.e. "void someFunction (Season estacaoDoAno)"

v Declaration in general form - We created a java file

- i.e. "Color.java"

```
public enum Color {
WHITE, BLACK, RED, YELLOW, BLUE
}
```

- How to use?

```
Color cor = Color.WHITE;
```

...

```
if (color == Color.WHITE) ...
```

```
color = Color.BLUE;
```



## Enumerated Types – characteristics

v Enum declaration within an existing Class.

```
public class Painter {...
```

```
enum
```

```
}
```

```
... // Painter body
```

```
}
```

- How to use?

```
Painter.Color cor = Painter.Color.WHITE; ...
```

```
if (color == Painter.Color.WHITE)
```

```
...
```

```
color = Painter.Color.BLUE;
```

---

## Enumerated types - another example

v We define an enum for the Months of the Year

```
public enum Mes {
```

```
JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST,
SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER;
```

```
}
```

v We can use it in the Data class

```
// Date Builder
```

```
public Data (int iDia, Mes iMes, int iAno) {
```

```
// ...}
```

```
// Main
```

```
Data d1 = new Data (11, Mes.MAIO, 1900);
```

v Important limitation

- We cannot read an enum through a Scanner

---

## Available methods

```
public enum Season {
```

```
SPRING, SUMMER, FALL, WINTER}
```

v Provide some useful methods:

- valueOf (String val): converts to String (element of the set) to a value

- ordinal (): position (int) of the value in the list of elements

- values (): returns the list of elements

```
Season s1 = Season.valueOf ("WINTER"); System.out.println (s1); System.out.println
```

```
(s1.ordinal ()); for (Season s2: Season.values ())
```

```
System.out.println (s2);
```

v The switch statement works with enumerates

```
Color myColor = Color.BLACK;
```

```
switch (myColor) {case WHITE: ...; case BLACK: ...; ...
```

```
case BLUE: ...;
```

```
default: ...; }
```

---

## Enums in Java

v enum is a "class", not a primitive type.

- Can implement interfaces
  - Values are objects
  - Support comparison (== or equals ()).
  - v The enumerated types are not integers.
  - v They only have private builders.
  - v The enumerated values are constant
  - They are automatically public, static, final.
- 

### Enum - a Class

v We can have Enumerated types with associated data and operations

```
public enum Color {
 WHITE (21), BLACK (22), RED (23), YELLOW (24), BLUE (25);
}
// Dice
private int code;
// Constructor
private Color (int c) {
// Method
public int getCode () {
 code = c; } return code; }
```

---

### Enum - implements Interface

v enum types can implement public enum interfaces

```
Color implements Runnable {
 WHITE, BLACK, RED, YELLOW, BLUE;
 public void run () {System.out.println ("name () =" + name () +
 ", toString () =" + toString ());
 }}
```

- Use:

```
for (Color c: Color.values ()) {c.run ();}
```

- Or

```
for (Runnable r: Color.values ()) {r.run ();}
```

---

## Error Handling Exceptions – Problem

v Not all errors are detected when compiling.

v These are generally the ones that are most overlooked by programmers:

- Compiles without errors - Works !!!

v Classic Treatment:

- If we know from the outset that an error situation can arise in a certain passage, we can deal with it in that context (if ...)

```
if ((i = doTheJob ()) != -1) { /* error handling */
}
```

---

## Exception

v An exception is generated by something unforeseen that cannot be controlled.

v Use of Exceptions:

- Error handling in the local context

```
try {
```

```
/* What you want to do */
```

```
catch (ErrorType a) { }
```

- Error delegation - generate an exception object (throw) in which this treatment is delegated.

```
if (t == null)
```

```
throw new NullPointerException ();
```

```
// or
```

```
// throw new NullPointerException ("reference t can't be null");
```

---

## Exception Control

v Exception handling is done through a special block: try .. catch.

```
try {
```

```
// Code that might generate exceptions Type1, // Type2 or Type3
```

```
} catch (Type1 id1) {
```

```
// Handle exceptions of Type1
```

```
} catch (Type2 id2) {
```

```
// Handle exceptions of Type2
```

```
} catch (Type3 id3) {
```

```
// Handle exceptions of Type3
```

```
} finally {
```

```
// Executed regardless of whether or not there is an exception
```

```
}
```

v Exception handling can also be done using a try-with-resources block.

- Ensures that resources are closed

- In this case there is no finally block

```
try (Code that might generate exceptions Type1, Type2 or Type3) {
```

```
// other code that does not cause exceptions
```

```
....
```

---

## Advantages of Exceptions

- v Clear separation between regular code and error handling code
  - v Propagation of errors in successive calls
  - v Grouping errors by types
- 

### code separation - example

```
readFile {
 try {
 open the file;
 determine its size; allocate that much memory; read the file into memory; close the file;
 } catch (fileOpenFailed) {doSomething;
 } catch (sizeDeterminationFailed) {doSomething;
 } catch (memoryAllocationFailed) {doSomething;
 } catch (readFailed) {doSomething;
 } catch (fileCloseFailed) {doSomething;
 }
}
```

---

## Types of Exceptions

- v checked
    - If we invoke a method that generates a checked exception, we have to tell the compiler how we are going to solve it:  
1) Solve try .. catch, or 2) Propagate throw
  - v unchecked
    - These are programming or system errors (we can use Assertions in these cases)
    - They are subclasses of java.lang.RuntimeException or java.lang.Error
- 

## Statement of Exceptions

- v When designing methods that can generate exceptions, we must explicitly flag them
  - v Declaration throws public void istoPodeDarNaseira ()  
throws TooBigException, TooSmallException, DivByZeroException { // ...  
}
- 

## Create New Exceptions

- v We can use the inheritance mechanism to customize some exceptions
- ```
class MyException extends Exception { // base interface  
  public MyException () {}  
  public MyException (String msg) {  
    super (msg); }  
  // we can add constructors and data}
```
-

Good Practices

- ✓ Use exceptions only for exceptional conditions
 - A well-designed API should not force the client to use exceptions for flow control
 - An exception should not be used for a simple test
 - ✓ Preferably use standard exceptions
 - `IllegalArgumentException`
inappropriate parameter value
 - `IllegalStateException` Incorrect object state
 - `NullPointerException`
 - `IndexOutOfBoundsException`
 - ✓ Always handle exceptions (or delegate them)
-

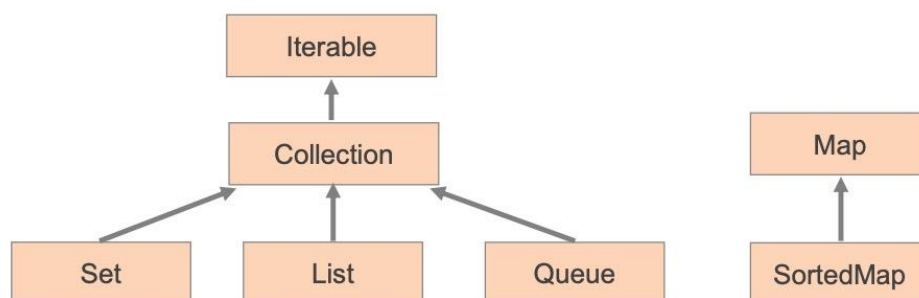
Java Collections

Java Collections Framework (JCF)

- ✓ Set of classes, interfaces and algorithms that represent various types of data storage structures
 - Lists, Vectors, Stacks, Trees, Maps, ...
 - They allow the aggregation of objects of a parametric type
 - > data types are also a Parameter
 - Example:
`ArrayList<String> cities = new ArrayList<>();`
`Cidades.add ("Aveiro");`
`cities.add ("Paris");`
 - Do not support primitive types (int, float, double, ..). In this case, we need to use adapter classes (Integer, Float, Double, ...)
-

Main Interfaces

- ✓ Set of 4 main interfaces:
 - Sets: without position notion (without order), without repetition
 - Lists (List): sequences with a sense of order, with repetition
 - Queues (Queue): are the First in First Out queues
 - Maps (Map): associative structures where objects are represented by a key-value pair.



Advantages of Collections

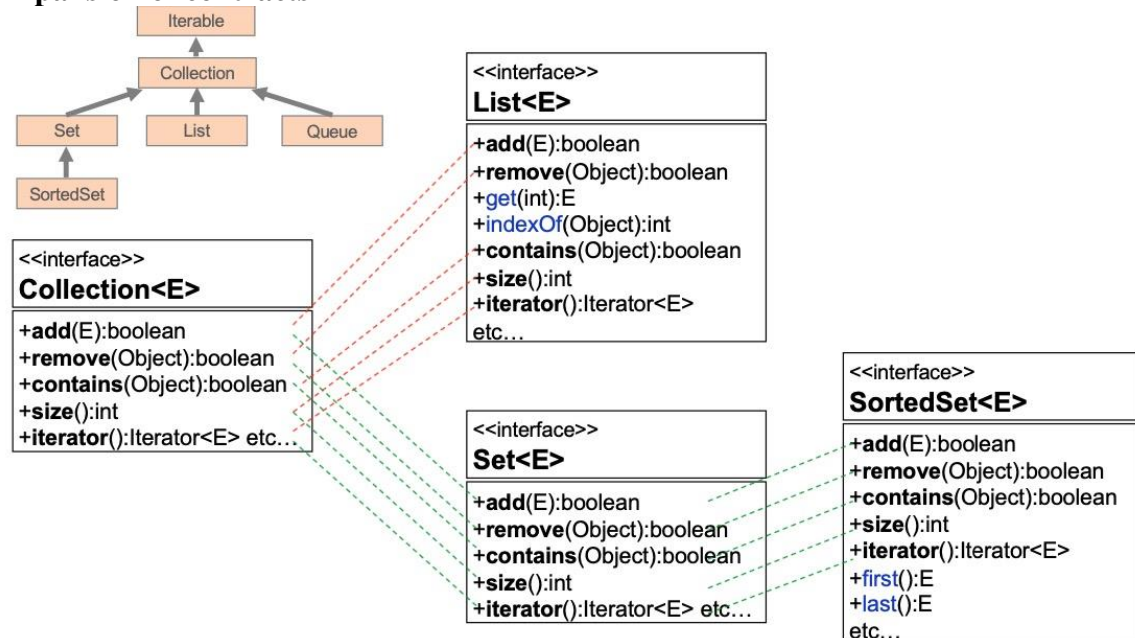
✓ Advantage of creating interfaces:

- Separate the specification from the implementation
- One implementation can be replaced by another more efficient one without major impacts on the existing structure.

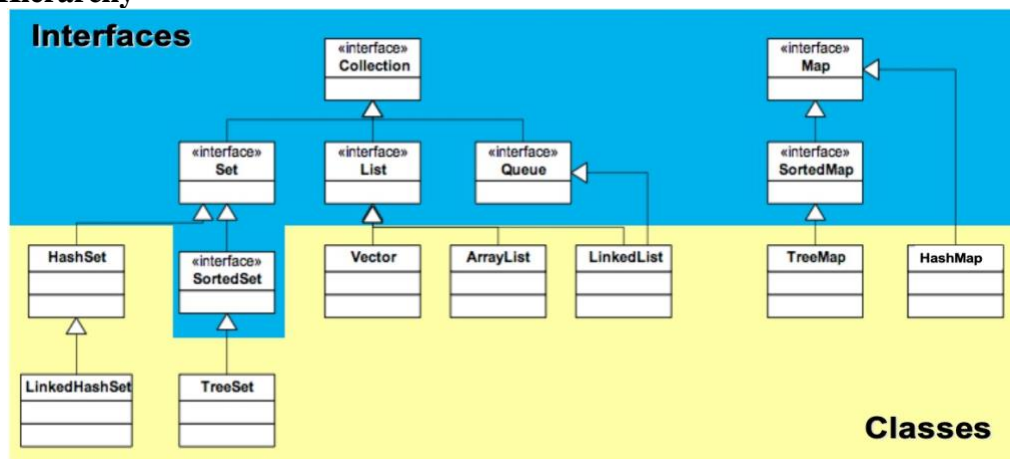
✓ Example:

```
Collection <String> c = new LinkedList <> (); c.add ("Aveiro");
c.add ("Paris");
Iterator <String> i = c.iterator ();
while (i.hasNext ()) {
    System.out.println (i.next ());
}
```

Expansion of contracts



Class Hierarchy



Interfaces and Implementations

Collections					
	Implementações				
Interfaces	Hash table	Resizable array	Balanced Tree (sorted)	Linked list	Hash table + Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Set

- ✓ A collection that cannot contain duplicate elements.
- ✓ Contains only the methods defined in the Collection interface
 - New contracts in add, equals and hashCode methods
- ✓ Implementations:
 - HashSet
 - TreeSet
 - ..

HashSet

- ✓ Uses a hash table (Hash Map) to store the elements.
 - ✓ The insertion of a new element will not be performed if the equals function of the element to be inserted with any element of the Set returns true.
 - It is essential to implement the equals function in all classes that can be used as elements of scatter tables (HashSet, HashMap, ...)
 - ✓ Constant performance,
 - The (~ 1) for add, remove, contains and size
 - ✓ no sense of position (no order) when inserting elements
-

TreeSet

- v It allows the ordering of elements according to their “natural order”.
 - Objects inserted in TreeSet must implement the Comparable interface.
 - or using an object of type Comparator in the TreeSet constructor. (we'll see this later)
 - v Implementation based on a balanced tree structure.
 - v Performance $\log(n)$, for add, remove and contains
-

Lists

- v Implement List
 - v Can contain duplicates.
 - v In addition to operations inherited from Collection, the List interface also includes:
 - Positional Access: manipulation of elements based on their position (index) in the list
 - Search: for a particular element in the list. Returns your position.
 - ListIterator: extends the semantics of the Iterator taking advantage of the sequential nature of the list.
 - Range-View: execution of operations on a range of list elements.
- `list.subList (fromIndex, toIndex) .clear ();`
-

Lists – Classes

More common:

- v ArrayList - Dynamic array
- v LinkedList - Linked lists

Others:

- v Vector - Dynamic array
 - (!) Vector is synchronized. If a thread-safe implementation is not needed, it is recommended to use ArrayList in place of Vector.
 - v Stack
 - extends Vector
-

Maps

- v Interface Map does not descend from Collections
 - Interface Map $\langle K, V \rangle$
 - v A map is a set that associates a key (K) with a value (V)
 - Does not contain duplicate keys
 - v It is also referred to as a dictionary or associative memory
 - v Available methods:
 - add: put (K key, V value)
 - remove: remove (Object key)
 - get an object: get (Object key)
-

Views

- v Maps are not Collections.
 - v However, we can get views of the maps.
 - v Views are of the Collections type
 - v There are three views available:
 - set of keys
 - collection of values
 - set of key / value pair entries
-

Map – Implementations

- v HashMap
 - Uses a scatter table (Hash Table)
 - There is no sorting in pairs
 - v LinkedHashMap
 - Similar to HashMap, but preserves the order of insertion
 - v TreeMap
 - Based on a balanced tree
 - Pairs are ordered based on the key
 - Performance for insertion and removal is $O(\log N)$
-

TreeMap

- v Same characteristics as described for the TreeSet but adapted to key / value pairs.
 - v TreeMap offers the possibility to order objects
 - using the “Natural Order” (compareTo) or an object of the Comparator type
 - similar use to HashSet examples
-

Iterate over Collections

```
v Iterator
public interface Iterator <E> {boolean hasNext ();
And next ();
void remove (); // optional
}
v "for each" cycle
List <String> names = new LinkedList <> ();
// ... add some names to the collection
for (String name: names)
System.out.println (name);
```

Reading and writing data Files

Introduction

- v Without the ability to interact with the "rest of the world", our program becomes useless
 - This interaction is called "input / output" (I / O)
 - v Problem → Complexity
 - Different and complex I / O devices (files, consoles, communication channels, ...)
 - Different access formats (sequential, random, binary, characters, line, words, ...)
 - v Need for Abstraction
 - Free the programmer from the need to deal with the specificity and complexity of each I / O
-

Java IO and NIO

- v The java language provides two packages to allow input / output operations
 - v Java IO
 - Stream oriented
 - Blocking IO
 - v Java NIO (new IO)
 - Buffer oriented
 - Non blocking IO
 - Channels
 - Selectors
-

java.io - Stream oriented

- v The program can read from an input stream and write to an output stream.
 - v Each stream has a unique meaning
-

java.io - Types of Streams

- v Byte Streams
 - machine-formatted binaries
 - data transferred without being altered in any way
 - are not interpreted
 - no judgments are made about its value
 - v Character Streams
 - Data is in the form of characters (human-readable data)
 - interpreted and transformed according to text representation formats
-

java.io - Streams

- v We have 4 abstract classes to deal with I / O:
 - InputStream: byte-input
 - OutputStream: byte-output
 - Reader: text-input
 - Writer: text-output
 - v All I / O classes are derived from these
 - Data entry
 - InputStream (byte)
 - Reader (char)
 - Data output
 - OutputStream (byte)
 - Writer (char)
 - v These classes are included in the java.io package
-

InputStream / Reader

- v Reader and InputStream have similar interfaces but different data types
 - v Reader
 - int read ()
 - int read (char cbuf [])
 - int read (char cbuf [], int offset, int length)
 - v InputStream
 - int read ()
 - int read (byte cbuf [])
 - int read (byte cbuf [], int offset, int length)
-

OutputStream / Writer

- v Writer and OutputStream have similar interfaces but different data types
 - v Writer
 - int write ()
 - int write (char cbuf [])
 - int write (char cbuf [], int offset, int length)
 - v OutputStream
 - int write ()
 - int write (byte cbuf [])
 - int write (byte cbuf [], int offset, int length)
-

Standard I / O

- v System.in is of type InputStream
 - byte [] b = new byte [10];
 - InputStream stdin = System.in;
 - stdin.read (b);
 - v System.out is of type PrintStream (subtype of OutputStream)
 - OutputStream stdout = System.out;
 - stdout.write (104); // ASCII 'h'
 - stdout.flush ();
-

Files - Main Classes

- v Java IO
- File
- Scanner
- FileReader
- FileWriter
- RandomAccessFile
- v Java NIO
- Path
- Paths
- Files
- SeekableByteChannel

The previous classes allow "low level" operations over data streams. We will discuss only a few interfaces / classes that allow "high level" operations

java.io.File

- v The File class represents either a file name or the set of files in a directory
- v Provides useful information and operations on files and directories
- canRead, canWrite, exists, getName, isDirectory, isFile, listFiles, mkdir, ...

v Examples:

```
File file1 = new File ("io.txt");
File file2 = new File ("C: / tmp /", "io.txt"); File file3 = new File ("POO / Slides");
if (! file1.exists ()) {/ * do something * /}
if (! file3.isDirectory ()) {/ * do something * /}
```

java.util.Scanner

- v Class that makes it easy to read primitive types and strings from an input stream.

- Read from the keyboard

```
Scanner sc1 = new Scanner (System.in);
```

```
int i = sc1.nextInt ();
```

- Read from a string

```
Scanner sc2 = new Scanner ("really long \ nString \ n \ t \ tthat I want to pick
apart \ n ");
```

```
while (sc2.hasNextLine ())
```

```
System.out.println (sc2.nextLine ());
```

- Read from a file

```
Scanner input = new Scanner (new File ("words.txt"));
```

```
while (input.hasNextLine ())
```

```
System.out.println (input.nextLine ());
```

Reading text files

v Example 1: without exception handling

```
public class TestReadFile
{
    public static void main (String [] args) throws FileNotFoundException {
        Scanner input = new Scanner (new File ("words.txt"));
        while (input.hasNextLine ())
            System.out.println (input.nextLine ());
    }
}
```

v The "words.txt" file must be:

- In the local folder, if the program is run via commando line
 - In the project folder, if executed in Eclipse
-

Reading text files

v Example 3: try-with-resources

- The code that declares and creates resources is placed in the try () entry.
- Resources are objects that implement AutoCloseable and have to be closed after being used.

```
public static void main (String [] args) {
    try (Scanner input = new Scanner (new File ("words.txt"))) {
        while (input.hasNextLine ())
            System.out.println (input.nextLine ());
    } catch (FileNotFoundException e) {
        System.out.println ("File does not exist!");
    }
}
```

java.nio - Reading text files

v We can use static methods from the Files and Paths classes of the java.nio.file package.

v Example 4:

```
public class ReadFileIntoList {
    public static void main (String [] args) throws IOException {
        List <String> lines = Files.readAllLines (Paths.get ("words.txt"));
        for (String ln: lines)
            System.out.println (ln);
    }
}
```

Writing text files

v class java.io.PrintWriter

- Allows us to use the println and printf methods to write in text files.
- Formats the values of primitive types into text, such as when printed on the screen.

```
public class FileWritingDemo {
    public static void main (String [] args) throws IOException {
        PrintWriter out = new PrintWriter (new File ("file1.txt"));
        out.println ("Weekend at the beach");
        out.printf ("Travel:% d \ nHotel:% d \ n", 345, 1000);
        out.close ();
    }
}
```

Writing text files - append

v We can append more information to an existing file

```
public class FileWritingDemo {  
    public static void main (String [] args) throws IOException {  
        FileWriter fileWriter = new FileWriter ("file1.txt", true);  
        PrintWriter printWriter = new PrintWriter (fileWriter);  
        printWriter.append ("adding more notes ... \n");  
        printWriter.close ();  
    }  
}
```

Binary files

v java.io.RandomAccessFile

- See a file as a sequence of bytes
 - It has a pointer (seek) to read or write anywhere in the file.
 - Generically, includes seek, read, write operations
- v We can only read or write primitive types
- writeByte (), writeInt (), writeBoolean ()
 - writeChars (String s), writeUTF (String str),
 - String readLine ()
-

Binary files - example

v Assuming the following file organization

// In the file “./mydata”, copy bytes 10-19 to 0-9.

```
RandomAccessFile file = new RandomAccessFile (“mydata”, “rw”);  
byte [] buf = new byte [10]; // Reserve a 10-byte buffer (buf: 10 bytes in memory)  
file.seek (10); // Changing the pointer  
file.read (buf); // Read to memory  
file.seek (0); // Change the pointer  
file.write (buf); // Write to file  
file.close ();
```

v Append to a file that already exists.

```
File f = new File ("um_ficheiro_any");  
RandomAccessFile raf = new RandomAccessFile (f, "rw");  
raf.seek (f.length ()); // Seek to end of file  
raf.writeChars ("now is the end"); // Append to the end  
raf.close ();
```

→ FÜR DAS BEISPIEL IN FOLIEN SCHAUEN

Java NIO (Java 7+)

v Significant changes in the main classes

v Class `java.nio.file.Files`

- Only static methods for manipulating files, directories, ..

v Class `java.nio.file.Paths`

- Only static methods to return a `Path` by converting a string or Uniform Resource Identifier (URI)

v Interface `java.nio.file.Path`

- Used to represent the location of a file or file system, typically system dependent.

v Common use:

- Use `Paths` to obtain a `Path`.

- Use `Files` to perform operations.

`java.nio.file.Path`

v Notation dependent on the OS

- / home / sally / statusReport

- C: \ home \ sally \ statusReport

v Relative or absolute

v Symbolic links

v `java.nio.file.Path`

- Interface

- Path might not exist

`java.nio.file.Paths`

v Auxiliary class with 2 static methods

v Allows you to convert strings or a URI to a `Path`

static `Path get (String first, String ... more)`

- Converts a path string, or a sequence of strings that when joined form a path string, to a `Path`.

static `Path get (URI uri)`

- Converts the given URI to a `Path` object.

`java.nio.file.Path`

v Create

`Path p1 = Paths.get ("/ tmp / foo");`

`Path p11 = FileSystems.getDefault (). GetPath ("/ tmp / foo"); // <=> p1`

`Path p2 = Paths.get (args [0]);`

`Path p3 = Paths.get (URI.create ("file: ///Users/joe/FileTest.java"));`

v Create in the home directory `logs / foo.log` (or `logs \ foo.log`)

`Path p5 = Paths.get (System.getProperty ("user.home"), "logs", "foo.log");`

java.nio.file.Files

v Only static methods

- copy, create, delete, ..
- isDirectory, isReadable, isWritable, ..

v Example of copying files

```
Path src = Paths.get ("/ home / fred / readme.txt");
Path dst = Paths.get ("/ home / fred / copy_readme.txt");
Files.copy (src, dst, StandardCopyOption.COPY_ATTRIBUTES,
StandardCopyOption.REPLACE_EXISTING);
```

v Move

- Supports atomic move

```
Path src = Paths.get ("/ home / fred / readme.txt");
Path dst = Paths.get ("/ home / fred / readme.lst");
Files.move (src, dst, StandardCopyOption.ATOMIC_MOVE);
```

v delete (Path)

```
try {
Files.delete (path);
} catch (NoSuchFileException x) {
System.err.format ("% s: no such" + "file or directory% n", path);
} catch (DirectoryNotEmptyException x) {
System.err.format ("% s not empty% n", path);
} catch (IOException x) {
// File permission problems are caught here.
System.err.println (x);
}
```

v deleteIfExists (Path)

- No exceptions

v Check if two Paths indicate the same file

- In a file system with symbolic links we can have two distinct paths representing the same file

- Use isSameFile (Path, Path) to compare

```
Path p1 = ...; Path p2 = ...;
if (Files.isSameFile (p1, p2)) {
// Logic when the paths locate the same file
}
```

Lambda expressions Functional interfaces

Lambda calculation

v Functional programming languages are based on the lambda calculation (λ -calculation).

- Lisp, Haskell, Scheme

v Lambda calculation can be seen as an abstract programming language in which functions can be combined to form other functions.

v General idea: mathematical formalism - $x \rightarrow f(x)$ i.e. x is transformed into $f(x)$

v Lambda calculation treats functions as first-class elements

- can be used as arguments and returned as values for other functions.

Syntax

v A lambda expression describes an anonymous function. It is represented in the form:

- (argument) \rightarrow (body) (inta, intb) \rightarrow {return + b;}

v It can have zero, one, or more arguments - () \rightarrow {body}

() \rightarrow System.out.println ("Hello World");

- (arg1, arg2 ...) \rightarrow {body}

v The type of arguments can be explicitly stated or inferred

- (type1 arg1, type2 arg2 ...) \rightarrow {body} (inta, intb) \rightarrow {return + b;}

a \rightarrow return a*a // an argument - we can omit (a)

v The body can have one or more instructions

lambda expression	equivalent method
() \rightarrow { System.gc(); }	void nn() { System.gc(); }
(int x) \rightarrow { return x+1; }	int nn(int x) return x+1; }
(int x, int y) \rightarrow { return x+y; }	int nn(int x, int y) { return x+y; }
(String... args) \rightarrow {return args.length;}	int nn(String... args) { return args.length; }
(String[] args) \rightarrow { if (args != null) return args.length; else return 0; }	int nn(String[] args) { if (args != null) return args.length; else return 0; }

How to use?

v A lambda expression cannot be isolated

(n) \rightarrow (n% 2) == 0 // Compilation error

v We need another additional mechanism

- Functional interfaces

- where lambda expressions become implementations of abstract methods.

- The Java compiler converts a lambda expression to a private method of the class (this is an internal process).

Functional interface

v Contains only one abstract method

v Example

- Given the interface:

@FunctionalInterface

```
interface MyNum {  
    double getNum (double n);  
}
```

- We can use:


```
public class Lambda1 {  
    public static void main (String [] args) {  
        MyNum n1 = (x) -> x + 1;  
        // any expression that turns double into double  
        System.out.println (n1.getNum (10));  
        n1 = (x) -> x * x;  
        System.out.println (n1.getNum (10));  
    }  
}
```

Lambda expressions as an argument

v We can define generic interfaces (with parameters).

v For example:

```
interface MyFunc<T> {  
    T func(T n);  
}  
...  
// Funções que aceita uma expressão lambda e o seu argumento (T n)  
static String stringOp(MyFunc<String> sf, String s) {  
    return sf.func(s);  
}  
...  
// Outro exemplo  
static Person PersonOp(MyFunc<Person> sf, Person s) {  
    return sf.func(s);  
}
```



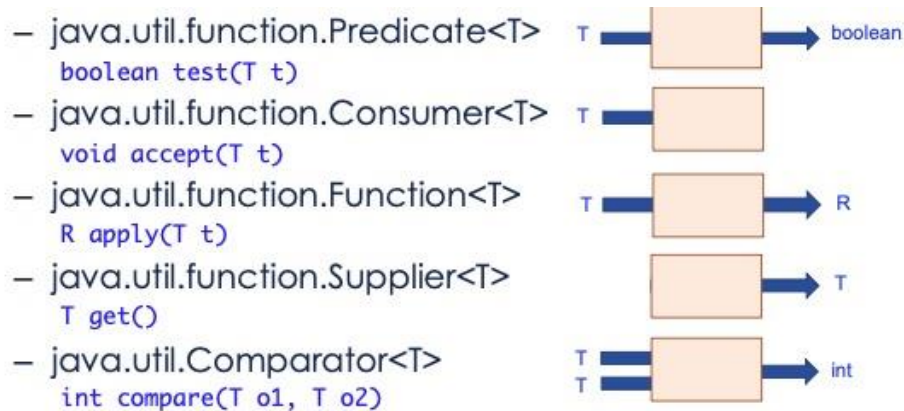
v Use

```
String inStr = "Lambdas add power to Java";  
String outStr = stringOp((str) -> str.toUpperCase(), inStr);  
System.out.println("The string in uppercase: " + outStr);  
// This passes a block lambda that removes spaces.  
outStr = stringOp((str) -> {  
    StringBuilder result = new StringBuilder();  
    for(int i = 0; i < str.length(); i++)  
        if(str.charAt(i) != ' ')  
            result.append( str.charAt(i) );  
    return result.toString();  
}, inStr);  
System.out.println("The string with spaces removed: " + outStr);
```

```
The string in uppercase: LAMBDA$ ADD POWER TO JAVA  
The string with spaces removed: LambdasaddpowertoJava
```

Pre-defined functional interfaces

- v We generally do not need to create new functional interfaces
- We use those already defined in Java. v Examples



Reference methods

- v They are a special type of lambda expressions.
 - Allows you to create lambda expressions using methods existing.
 - v Examples
 - We can replace:
`str -> System.out.println (str)`
`(s1, s2) -> {return s1.compareToIgnoreCase (s2); }`
 - with:
`System.out :: println String :: compareToIgnoreCase`
-

Use of lambda expressions

- v Java Collections

```
// solução 1
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
for (Integer n: list) {
    System.out.println(n);
}

// solução 2
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
list.forEach(n -> System.out.println(n));

// solução 3, method reference (:: double colon operator)
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
list.forEach(System.out::println);
```

Algorithms

- v Java libraries provide a set of algorithms that can be used in collections and vectors
 - v Two abstract classes provide static methods for global use
 - java.util.Collections - Note that it is different from java.util.Collection (interface) !!
 - java.util.Arrays - Class that contains several methods for manipulating vectors (sorting, searching, ..). It also allows you to convert vectors to lists.
 - v Examples of methods:
 - sort, binarySearch, copy, shuffle, reverse, max, min, etc.
-

Stream API

UA.DETI.POO

Iterar sobre coleções

❖ Iterator

```
List<String> names = Arrays.asList("Ana", "Ze", "Rui");  
Iterator<String> it = names.iterator();  
while (it.hasNext())  
    System.out.println(it.next());
```

❖ ciclo "for each"

```
List<String> names = Arrays.asList("Ana", "Ze", "Rui");  
for (String name : names)  
    System.out.println(name);
```

❖ Método forEach

```
List<String> names = Arrays.asList("Ana", "Ze", "Rui");  
names.forEach(s -> System.out.println(s)); // forEach com lambda  
names.forEach(System.out::println); // forEach com método referência
```

❖ Stream operations

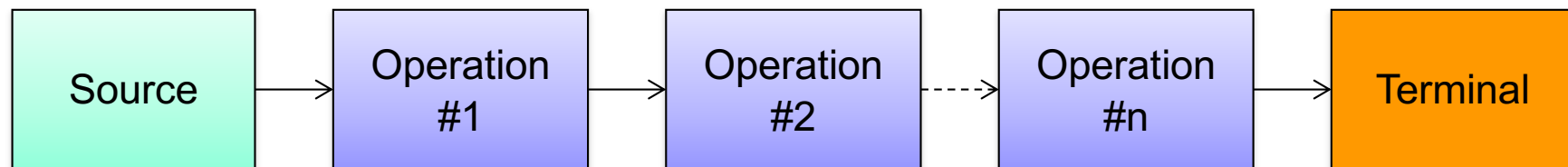
– *Agregate operations*

Aggregate Operations – Streams API

- ❖ The preferred method of iterating over a collection is to obtain a stream and perform aggregate operations on it.
- ❖ Aggregate operations are often used in conjunction with lambda expressions
 - to make programming more expressive, using less lines of code.
- ❖ Package `java.util.stream`
 - The key abstraction introduced in this package is stream.

Stream Pipeline

- ❖ (1) Obtain a stream from a source
- ❖ (2) Perform one or more intermediate operations
- ❖ (3) Perform one terminal operation



- ❖ Usage: `Source.Op1.Op2 .. .Terminal`

java.util.stream – Sources

- ❖ Streams sources include:
 - From a `Collection` via the `stream()` and `parallelStream()` methods;
 - From an `Array` via `Arrays.stream(Object[])`;
 - *and many more (files, random, ..)*

java.util.stream – Intermediate operations

- **filter** - excludes all elements that don't match a Predicate
- **map** - perform transformation of elements using a Function
- **flatMap** - transform each element into zero or more elements by way of another Stream
- **peek** - performs some action on each element
- **distinct** - excludes all duplicate elements (equals())
- **sorted** - ordered elements (Comparator)
- **limit** - maximum number of elements
- **substream** - range (by index) of elements
- *(and many more -> see java.util.stream.Stream<T>)*

```
List<Person> people = ...;  
Stream<Person> tenPersonsOver18 = people.stream()  
    .filter(p -> p.getAge() > 18)  
    .limit(10);
```

java.util.stream – Terminating operations

❖ Reducers

- reduce(), count(), findAny(), findFirst()

❖ Collectors

- collect()

❖ forEach

❖ iterators

```
// Accumulate names into a List
List<Person> people = ...;
List<String> names = people.stream()
    .map(Person::getName)
    .collect(Collectors.toList());
```

Stream.Filter

- ❖ Filtering a stream of data is the first natural operation that we would need.
- ❖ Stream interface exposes a filter method that takes in a Predicate that allows us to use lambda expression to define the filtering criteria:

```
List<String> l = Arrays.asList("Ana Maria", "Mariana", "Rui");  
  
l.stream().filter(n -> n.length()>3)  
    .forEach(System.out::println);
```

Stream.Map

- ❖ The map operations allows us to apply a function that takes in a parameter of one type and returns something else.

```
Stream<Student> map = persons.stream()  
    .filter(p -> p.getAge() > 18)  
    .map(person -> new Student(person));
```

```
// other example with Map && Consumer
```

```
List<String> l = Arrays.asList("Ana", "Ze", "Rui");  
l.stream().map(n -> "Nome = " + n)  
    .forEach(System.out::println);
```

Stream.Reduce

- ❖ A reduction operation takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation
- ❖ For instance, finding the sum or maximum of a set of numbers, or accumulating elements into a list.

```
// example with Map & Reduce
```

```
List<Integer> costBeforeTax = Arrays.asList(100, 200, 300, 400, 500);
```

```
double bill = costBeforeTax.stream()  
    .map(cost -> (cost*1.23))  
    .reduce(0.0,(sum, cost) -> sum + cost));
```

```
System.out.println("Total : " + bill);
```

Stream.Collect

- ❖ The Stream API provides several “terminal” operations.
- ❖ The collect() method is one of those, which allows us to collect the results of the operations:

```
List<Student> students = persons.stream()  
    .filter(p -> p.getAge() > 18)  
    .map(Student::new)  
    .collect(Collectors.toList());
```

```
// other example with Map && Collect  
List<String> l = Arrays.asList("Ana", "Ze", "Rui");  
List<String> res = l.stream()  
    .map(n -> "Nome: " + n)  
    .collect(Collectors.toList());  
res.forEach(System.out::println);
```

Some examples using a list of strings

```
public static void listExample() {
    List<String> words = new ArrayList<String>();
    words.add("Prego");
    words.add("no");
    words.add("Prato");
    // old fashioned way to print the words
    for (int i = 0; i < words.size(); i++)
        System.out.print(words.get(i) + " ");
    System.out.println();

    // Java 5 introduced the foreach loop and Iterable<T> interface
    for (String s : words)
        System.out.print(s + " ");
    System.out.println();

    // Java 8 has a forEach method as part of the Iterable<T> interface
    // The expression is known as a "lambda" (an anonymous function)
    words.stream().forEach(n -> System.out.print(n + " "));
    System.out.println();

    // but in Java 8, why use a lambda when you can refer directly to the
    // appropriate function?
    words.stream().forEach(System.out::print);
    System.out.println();

    // Let's introduce a call on map to transform the data before it is printed
    words.stream().map(n -> n + " ").forEach(System.out::print);
    System.out.println();

    // obviously these chains of calls can get long, so the convention is
    // to split them across lines after the call on "stream":
    words.stream()
        .map(n -> n + " ")
        .forEach(System.out::print);
    System.out.println();
}
```

```
Prego no Prato
Prego no Prato
Prego no Prato
PregonoPrato
Prego no Prato
Prego no Prato
```


Some examples with an array of int

```
public static void arraysExample() {
    int[] numbers = {3, -4, 8, 73, 507, 8, 14, 9, 3, 15, -7, 9, 3, -7, 15};

    // want to know the sum of the numbers? It's now built in
    int sum = Arrays.stream(numbers)
        .sum();
    System.out.println("sum = " + sum);

    // how about the sum of the evens?
    int sum2 = Arrays.stream(numbers)
        .filter(i -> i % 2 == 0)
        .sum();
    System.out.println("sum of evens = " + sum2);

    // how about the sum of the absolute value of the evens?
    int sum3 = Arrays.stream(numbers)
        .map(Math::abs)
        .filter(i -> i % 2 == 0)
        .sum();
    System.out.println("sum of absolute value of evens = " + sum3);

    // how about the same thing with no duplicates?
    int sum4 = Arrays.stream(numbers)
        .distinct()
        .map(Math::abs)
        .filter(i -> i % 2 == 0)
        .sum();
    System.out.println("sum of absolute value of distinct evens = " + sum4);
}
```

```
sum = 649
sum of evens = 26
sum of absolute value of evens = 34
sum of absolute value of distinct evens = 26
```

Sumário

❖ JAVA Stream API

❖ java.util.stream

- Interfaces
 - BaseStream
 - Collector
 - DoubleStream
 - DoubleStream.Builder
 - IntStream
 - IntStream.Builder
 - LongStream
 - LongStream.Builder
 - Stream
 - Stream.Builder
- Classes
 - Collectors
 - StreamSupport