1. Java reinforcement

Service and Process Programming

Arturo Bernal
Nacho Iborra

IES San Vicente

Index of contents

1. Inheritance and polymorphism

In this unit we will review and reinforce some Java fundamentals learnt in the first course. We will also introduce some new concepts, specially some new stuff and improvements that come with Java 8. A basic knowledge of Java syntax and how to create and compile a project in your favourite IDE (Netbeans recommended) is assumed. We will first review inheritance and polymorphism.

1.1. Composition in Java

The composition/aggregation is usually represented in the code with a reference to the contained object or a collection of those objects. In the following example, we can see a class named **Classroom** that contains a collection (List) of objects of the type **Student**:

```
public class Classroom {
	private List<Student> studentList = new ArrayList<>();
}
```

1.2. Basic inheritance in Java

When we want a class to inherit the characteristics (public and protected) from another class we use the reserved word **extends** (ChildClass **extends** ParentClass). Like this:

```
public class ComputerClassroom extends Classroom {
}
```

All attributes and methods (public and protected) are inherited and can be used from the descendant class:

```
public class Store {
	public void welcome() {
		System.out.println("Welcome to our store!");
	}
}

public class LiquorStore extends Store {
}

// MAIN
LiquorStore lqStore = new LiquorStore();
lqStore.welcome(); → Prints "Welcome to our store!"
```

We also can **override** a method, meaning that we can redefine its functionality rather than keeping the same as the parent's. Still, we can call the original parent's method from the child class method, or call the parent's constructor, using the reserved word **super**.

```
public class LiquorStore extends Store {
	@Override
	public void welcome() {
		super.welcome();
		System.out.println("If you are younger than 18, go back home!");
	}
}

// MAIN
LiquorStore lqStore = new LiquorStore();
lqStore.welcome();
"Welcome to our store!" → super method call
"If you are younger than 18, go back home!"
```

Example calling super from the constructor:

```
public class Classroom {
	private int maxStudents;

	public Classroom(int maxStudents) {
		this.maxStudents = maxStudents;
	}

	public int getMaxStudents() {
		return maxStudents;
	}
}

public class ComputerClassroom extends Classroom {
	private int numComputers;

	public ComputerClassroom(int maxStudents, int numComputers) {
		super(maxStudents);
		this.numComputers = numComputers;
	}

	public int getNumComputers() {
		return numComputers;
	}
}

// MAIN
ComputerClassroom compClass = new ComputerClassroom(30, 20);
System.out.println("Students max.: " + compClass.getMaxStudents() +
                   ", computers: " + compClass.getNumComputers());

"Students max.: 30, computers: 20"
```

A class can only inherit from one parent class in Java. Other languages like C++ allow a class to inherit from multiple classes at the same time.

1.2.1. Abstract classes

Not all the methods have to be implemented in a class. We can define an abstract class that **cannot be instantiated** and has one or more abstract methods, which have no implementation inside them. Classes that inherit from an abstract class must implement its abstract methods. Otherwise, it has to be also an abstract class.

```
public abstract class Store {
	public abstract void welcome();
}


public class LiquorStore extends Store {
	@Override
	public void welcome() {
		System.out.println("Welcome to our liquor store. "
				+ "If you are younger than 18, go back home!");
	}
}


// MAIN
Store store = new Store(); → ERROR
LiquorStore liqStore = new LiquorStore(); → OK
```

Exercise 1

Create a project named "**Stores**" implementing the two classes above and a **Main** class with the **main** method. **Store** will have two new private fields (**cash** → double and **drinkPrice** → double) and a constructor that will receive the drinkPrice and initialize cash to 0.0. It will also implement a method called **payDrinks(int numOfDrinks)** that will add to the cash variable the payment (number of drinks * price).

LiquorStore will have a private field called **tax** → int. Its constructor will receive the drinkPrice and tax values, calling the parent's constructor when necessary. This class will also overwrite **payDrinks(int)** calling first to the **parent's method** (pay without tax), and then adding the tax value to the cash field.

In the main method, create a LiquorStore with drinkPrice = 8.95€ and tax = 20%. Pay for 10 drinks and print the cash to see if its value is 107.40€ (print 2 decimal numbers).

Implement getters and setters when needed.

1.3. Interfaces

While in Java we can't inherit from more than 1 class at the same time, we still can implement some form of multiple inheritance using **interfaces**. An interface is like an abstract class, but with some restrictions. It can only contain **public abstract** methods (doesn't use the word abstract, all methods are by definition), and static constants.

```
public interface ISayHello {
	public void sayHello();
}

public interface ISayGoodbye {
	public void sayGoodbye();
}
```

When you want to inherit from an interface, you use the word **implements** before the name of the interface and after the *extends* clause (if any). You can implement from more than one interface using the comma as a separator.

```
public class Student implements ISayHello, ISayGoodbye {
	@Override
	public void sayHello() {
		System.out.println("Hello, I'm a student!");
	}

	@Override
	public void sayGoodbye() {
		System.out.println("Goodbye people!");
	}
}
```

When implementing an interface, you must implement all its methods. It can be said that an interface defines a **behaviour** all classes that implement it must have. You can also inherit the methods of an interface from another interface (using the word **extends** in this case):


```
public interface ISayThings extends ISayHello, ISayGoodbye {
	// Includes methods from ISayHello and ISayGoodbye
}
```

1.3.1. Anonymous class

Being like an abstract class, we cannot instantiate an object from an interface. We must implement it in a non-abstract class in order to being able to instantiate an object that has its methods.

To keep things simple, when we only want to implement one or two methods from the interface and nothing else, we don't have to create a class (and its file) only to do this. Since Java 7, we can use what is called an **anonymous class**. Instead of declaring a new class, we directly instantiate an object based on an Interface or a class we want to extend and we implement all the obligatory methods and extras we want in the instantiation itself.

```
// MAIN
ISayThings sayThings = new ISayThings() {
	private String name = "Peter";

	@Override
	public void sayHello() {
		System.out.println("Hello, my name is " + name);
```

```
	}

	@Override
	public void sayGoodbye() {
		System.out.println("Goodbye friends!");
	}
};
```

We can also instantiate an anonymous class to implement missing methods in an abstract class (the other methods will remain as they are defined, unless we override them):

```
public abstract class Store {
	public abstract void welcome();
}

// MAIN
Store store = new Store() {

	@Override
	public void welcome() {
		System.out.println("Welcome to our anonymous store!");
	}
};
```

Exercise 2

Using the project **Store** from exercise 1. In the main method we are going to instantiate a Store using an anonymous class. The drink price will be 8.95€ and the welcome method will say “Welcome to anonymous store!, Our drink price is price€” (with 2 decimals). Call the welcome method and also pay for 10 drinks. Show the resulting cash (should be 89.50€).

1.3.2. New in Java 8

Until now, an interface could only have public abstract methods and static constants (final). In Java 8, interfaces are extended and can also have **static methods** and **default methods**. A default method is an implemented method (we couldn't do that in previous versions), and you must precede the declaration with the word **default**:

```
public interface ISayHello {
	public void sayHello();

	default void sayHelloPeople(List<String> people) {
		for(String name: people) {
			System.out.println("Hello " + name);
		}
	}
}
```

1.4. Inheritance and composition. Is-A vs Has-A

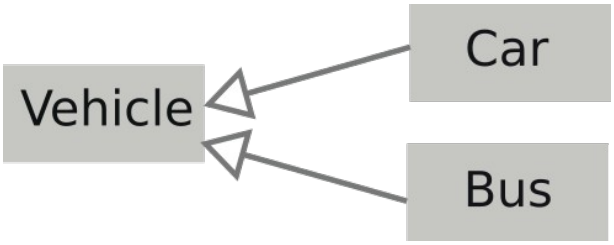
When we create a project, we know that we need to create some classes which represent the different entities that compose our program. Those classes or entities have to interact with each other (communicate) in different ways.

There's also more than one way to establish relationships between those classes. One common situation is when a class represents an entity that is part of another (**Has-A**):

- *Composition*: Is when an object is an indivisible part of another object. For example, a **Room** is part of a **House** (and only of that house), an **Square** is part of a **Chessboard**, and so on. The main characteristic of this type of relationship is that when you destroy the principal object, all objects that are related with it (composition) are also destroyed. We represent this relationship like this:

- *Aggregation*: Is when an object is part of another object (or maybe part of two or more objects) and it can exist with the object that contains it. An example of this would be a **Player**, who is part of a **Team** (or maybe more), or a **Student**, who belongs in a **Classrom** (or more). In these cases when the Team or the Class cease to exist, players and students continue to exist, can join other team/classroom, etc. We represent this relationship like this:

In contrast to this, other common relationship between classes is **inheritance** (**Is-A**). In this relationship, one class is a subtype of another class, in other words, shares the characteristics of the ancestor and introduces some new ones. One example of this is a Car, which is a subtype of Vehicle. Another could be a Mammal, which is a subtype of Animal. Or a ComputerClassroom, which is a subtype of Classroom that also has computers in it. This is the way to represent this kind of relationship:



1.5. Polymorphism

In Object Oriented Programming, **polymorphism** can mean different things, but in this case we will talk about **subtype polymorphism**. That means that an object can behave as if it was an instance from an extended class (even Object, from which every class in Java inherits) or an implemented interface.

```
public abstract class Store {
	public abstract void welcome();
}

public class LiquorStore extends Store {
	@Override
	public void welcome() {
		System.out.println("Welcome to our liquor store. "
				+ "If you are younger than 18, go back home!");
	}

	public void buyLiquor() {
		System.out.println("Do you want beer, wine, rum, whisky or vodka?");
	}
}

// MAIN
Store store = new LiquorStore(); // this LiquorStore behaves as a generic Store
store.buyLiquor(); // ERROR. We only can access Store methods
store.welcome(); // Ok, and executes LiquorStore implementation
```

We can cast a variable that references an extended class back to it's original class or any other extended class or implemented interface anytime. We can use the instruction **instanceof** to find out if it's really an instance of the subtype we believe is.

```
if(store instanceof LiquorStore) {
	LiquorStore liqStore = (LiquorStore) store;
	liqStore.buyLiquor(); // OK
}
```

If we just want to execute some method, we don't need to create a new variable:

```
if(store instanceof LiquorStore) {
	((LiquorStore) store).buyLiquor();
}
```

If a method's parameter is an interface or a class that other classes inherit from, we can pass any subtype class as the value. We must have in mind that we can only use the behaviour of the interface or the super-class (unless we do a casting like before):

```
public class Student implements ISayHello, ISayGoodbye {
	… // Implemented methods
	public void enterStore(Store store) {
		sayHello();
		store.welcome();
	}
}


// MAIN
LiquorStore liqStore = new LiquorStore();
Student student = new Student();
```

```
student.enterStore(liqStore); // Will be treated as a Store inside
```

Exercise 3

Create a project called **KillEnemies**. You must create the following inside:

- Interface **Character**: With a method called isEnemy() that returns a boolean.
- Class **Friend**: Implements **Character**. IsEnemy() returns false.
- Class **Enemy**: Implements **Character**. IsEnemy() returns true. Also implements a method called kill() that shows this message “Ahhhggg, you killed me, bastard!”.
- Class **Main** containing the main method. You must create an **ArrayList** of 10 Characters (5 friends and 5 enemies), then, using **Collections.shuffle(List)**, randomize the order of the items. You have to travel through all characters checking if they are enemies, and if they are, you kill them (call kill() method). An example of execution message would be:

```
Character 0 is a friend! :-)
Character 1 is a friend! :-)
Character 2 is an enemy! kill it!
Ahhhggg, you killed me, bastard!
...
```

1.6. Generics

We know that some classes like for example **ArrayList**, lets you create a dynamic array that can store objects of any type you want to. You specify the type of objects when you create a variable:

```
ArrayList<Store> stores; // Will hold objects which are a Store or inherit from
it
```

This is accomplished with **generics**. A generic is an special type which represents an unknown class or interface. The compiler will only know it when a variable that has a generic is instantiated, on that moment you must specify the type of the generic.

When you look at Java's ArrayList reference web page (https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html) , you'll see that this class is defined as **ArrayList<E>**. **<E>** is a notation used to define a generic class, a class that the compiler doesn't know which is until we instantiate an object. Another common notation is **<T>,<S>, <U>,** etc.. In general, a capital letter between **< >**.

If we create a generic, by default it can be any class, so the only properties and methods that the compiler will allow us to use are the ones inherited from **Object** (all classes in Java inherit from Object):

```
public class GenericExample<T> {
	private T generic;

	public GenericExample(T generic) {
		this.generic = generic;
	}

	public void showType() {
		System.out.println(generic.getClass().getName().toString());
		// We can't use for example .substring() because <T> can be
```

```
            anything.
      }

      public T getGeneric() {
            return generic;
      }
}
```

We define the type of <T> when we instantiate an object of GenericExample:

```
GenericExample<String> genEx = new GenericExample<>("Hello world!");
genEx.showType(); → java.lang.String
// Out of the class, in this context we can use a String method with the generic
object because the compiler knows that the generic is a string
System.out.println(genEx.getGeneric().length());
```

We can specify that the generic must be a subtype of class or implement some interface. In this case as it happens with polymorphism, we can use the superclass or the interface's methods:

```
public class GenericExample<T extends Store> {
      …
      public void welcome() {
            generic.welcome(); // generic can use Store methods
      }
      …
}
```

```
// MAIN
GenericExample<String> genEx = new GenericExample<>("Hello world!"); → Error
GenericExample<LiquorStore> genEx = new GenericExample<>(new LiquorStore()); →
OK
```

We can use more than one generic in a class:

```
public class GenericExample<T extends Store, E extends Classroom> {
}
```

Imagine we have a class called Inventory that can store items (any object that inherits from class Item). Let's see the difference in this case between using generics to define that class and using polymorphism:

```
public class Item {
      private float price;
      private int weight;

      public float getPrice() {
            return price;
      }
      public void setPrice(float price) {
```

```
            this.price = price;
        }
        public int getWeight() {
            return weight;
        }
        public void setWeight(int weight) {
            this.weight = weight;
        }
}


public class Potion extends Item {
        public void drink() {
            System.out.println("Gulp gulp gulp.");
        }
}


public class Weapon extends Item {
        private int damage;

        public int getDamage() {
            return damage;
        }

        public void setDamage(int damage) {
            this.damage = damage;
        }

}
```

Polymorphism

```
public class Inventory {
        private List<Item> items = new ArrayList<>();

        public void addItem(Item item) {
            items.add(item);
        }

        public Item getItem(int index) {
            return items.size() > index?items.get(index):null;
        }
}


// MAIN
Inventory inv = new Inventory();
inv.addItem(new Potion()); // Index 0
inv.addItem(new Weapon()); // Index 1

Item it = inv.getItem(0); // returns an Item, usually we don't know which type

if(it instanceof Potion) { // Cast to a potion
        ((Potion) it).drink();
} else if(it instanceof Weapon) { // Cast to a weapon
        System.out.println("Damage: " + ((Weapon) it).getDamage());
}
```

```
public class Inventory<T extends Item> {
	private List<T> items = new ArrayList<>();

	public void addItem(T item) {
		items.add(item);
	}

	public T getItem(int index) {
		return items.size() > index?items.get(index):null;
	}
}

// MAIN
Inventory<Item> inv  = new Inventory<>(); → Same behaviour as the example
before!

Inventory<Potion> potInv = new Inventory<>(); // Only allows potions
potInv.addItem(new Potion()); → OK
potInv.addItem(new Weapon()); → ERROR, <T> must be a Potion


Potion pot = potInv.getItem(0); // Ok, compiler knows is a Potion. No casting
needed
```

In summary, when we always want to be able to use more than one type of object inside a class instance, we can use polymorphism (an inventory with different types of items), althought we still can use generics for this → <Item>. When we want the possibility to use only one class at a time, defined when we instantiate an object, we only can do that with generics (we can create an inventory that only allows potions, other that only allows weapons and so on, using the same class for all).

We can also declare a generic (or more than one) only for a specific method, the same way we did with the class:

```
public <U extends T> void add(U item) {
	items.add(item);
}
```

Exercise 4

Create a project called **ShapeComparator** containing the following classes (including a Main class):

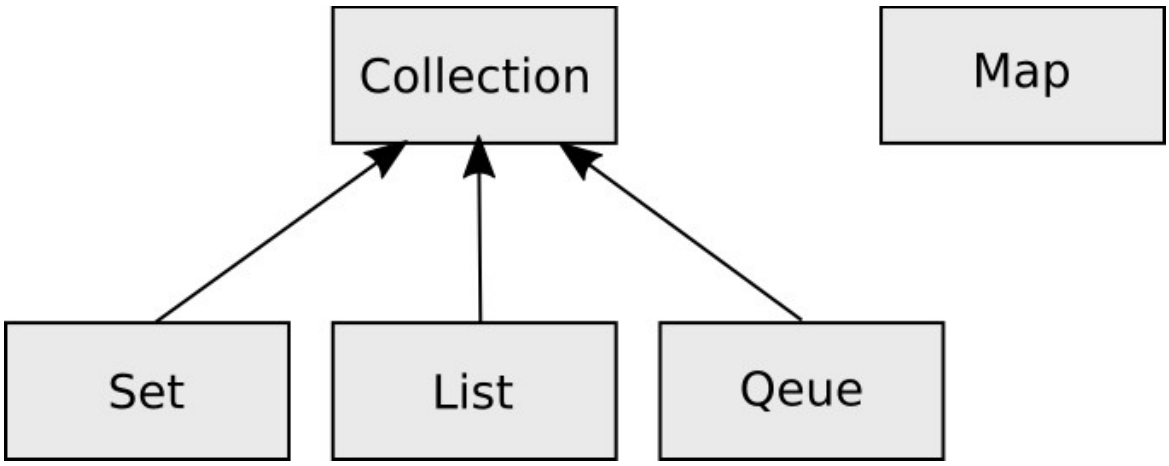
- Interface **Shape**: with the method **double getArea();**
- Classes **Triangle**, **Rectangle** and **Circle** implementing **Shape**, with the necessary properties to calculate the area (values are given in their respective constructors). Implement the method getArea() in each.
- Class **ShapeComparator**, that holds two objects (shape1 and shape2) derived from Shape (received in the constructor). **Use generics** to define the type of these two objects in this class (each object can be from a different type but always derived from Shape). This class has a method called compare(), that prints in

console which of the two shapes has a bigger area (or if they're equal), the first or the second.

- In the main method, instance two or more ShapeComparators, each to compare different kind of shapes (one can compare only Triangle vs Rectangle, another Rectangle vs Circle, and so on…). See that depending how you define the generics in every instance it only allows you to set or get (implement setters and getters) that kind of shape and not others.

2. Collections

A collection is a group of objects we want to store and operate with them in a particular way. Depending how we want to store them (ordered, unique,….) and operate with them (compare, iterate, …) there is one collection type more suited to do so than the others.



2.1. List

A list is an ordered (by an index → 0, 1, 2, …., N) and a sequential collection (iterable). It can have duplicates (the same object in different positions). In Java, List is an **interface**, so we must instantiate always a subtype of it (ArrayList, LinkedList, Stack, Vector,…) depending our needs. https://docs.oracle.com/javase/8/docs/api/java/util/List.html

2.1.1. ArrayList

This class is an implementation of a list that uses an internal array to store elements. It's main advantage is accessing an index to get an element O(1), but it's not the best implementation when we want to do many add and remove operations O(n).

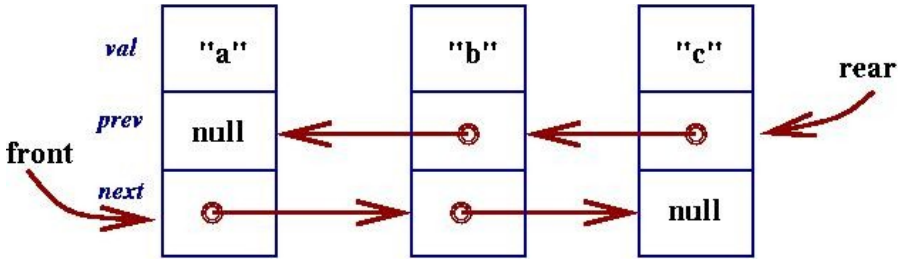
Complexities:

```
get(int index) is O(1), main benefit of ArrayList<E>
add(E element) when the interal array is big enough is O(1), but O(n) worst-case
since the array must be resized and copied
add(int index, E element) is O(n - index) if doesn't need to resize, but O(n)
worst-case (as above)
remove(int index) is O(n - index) (i.e. removing last is O(1))
Iterator.remove() is O(n - index)
ListIterator.add(E element) is O(n - index)
```

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

2.1.2. LinkedList

This type of list is implemented as a doubly-linked list. Elements are linked by a reference to the next and previous element in the list (The first element previous and last element next would be null).



There is no internal array, so nothing has to be resized, elements are inserted and removed only by modifying references in the previous and next object. This is the main advantage of this type of list. The main disadvantage is accessing a random position because it has to travel through all items to get to the specified index (there's no internal index to access a position directly).

Complexities:

```
get(int index) is O(n)
add(E element) is O(1)
add(int index, E element) is O(n)
remove(int index) is O(n)
Iterator.remove() is O(1), main benefit of LinkedList<E>
ListIterator.add(E element) is O(1), main benefit of LinkedList<E>
```

This implementation is more suited to traverse with an Iterator or ListIterator than with the typical for loop (will work but much less efficient). If you want to access random positions very frequently, it's better to use ArrayList.

```
List<String> strings = new LinkedList<>();
strings.add("Hello");
strings.add("world!");
strings.add("How");
strings.add("are");
strings.add("you?");
ListIterator<String> strIt = strings.listIterator();
while (strIt.hasNext()) {
      System.out.print(strIt.next() + " ");
}
```

https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html

2.1.3. Vector

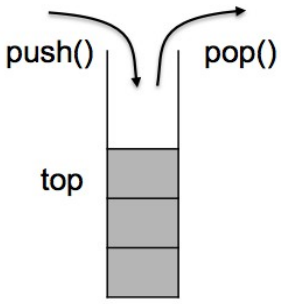
Vector is a list implementation very similar to ArrayList, but with the main difference that operations are all **synchronized**, so it's a good option when using more than 1 thread to operate with the list. Otherwise, is recommended to use ArrayList instead (more efficient).

We can also create a **synchronized** (thread safe) ArrayList instead of using Vector:

```
List list = Collections.synchronizedList(new ArrayList(...));
```

https://docs.oracle.com/javase/8/docs/api/java/util/Vector.html

2.1.4. Stack



An stack is an implementation of a LIFO (Last In First Out) list. It extends Vector class, so it has the same methods plus five methods that allow this list to behave like a LIFO list. However when we want to use a stack (LIFO), a queue (FIFO) or a mix, it's usually better to use an implementation of the interface Qeue or Deque as we'll see later.

https://docs.oracle.com/javase/8/docs/api/java/util/Stack.html

Exercise 5

Crete a project called **ListBenchmark**. We are going to test in which cases is better to use an ArrayList, or a LinkedList. To measure the time an operation takes, you can use this piece of code:

```
Instant start = Instant.now();
// Some operation with ArrayList
Instant end = Instant.now();

Duration dur = Duration.between(start, end);
System.out.printf("ArrayList: The operation ... takes: %dms\n", dur.toMillis());
```

You have to compare these situations (ArrayList<Double> vs LinkedList<Double>). Create one of each empty and reuse the same lists in every comparison:

1. Add 100.000 (double) random items always at position 0. Compare times.
2. Delete the first 50.000 items (always delete the first one).
3. Add 50.000 random items in random positions.
4. Delete 50.000 items from random positions.
5. Delete items that are in even positions (divisible by 2) using an Iterator.

You'll see that when using a lot of random accesses (index), ArrayList is much faster (LinkedList needs to count from the beginning). When adding or deleting items at the beginning, or using an iterator the situation is the opposite (ArrayList has to reorder internal indexes every time, while LinkedList doesn't need to).

2.2. Map

Map is a type of collection that maps a key (represented with an object) with a value (also an object). Keys must be unique, and the main difference with other types of collections such lists is that a key doesn't have to be an integer.

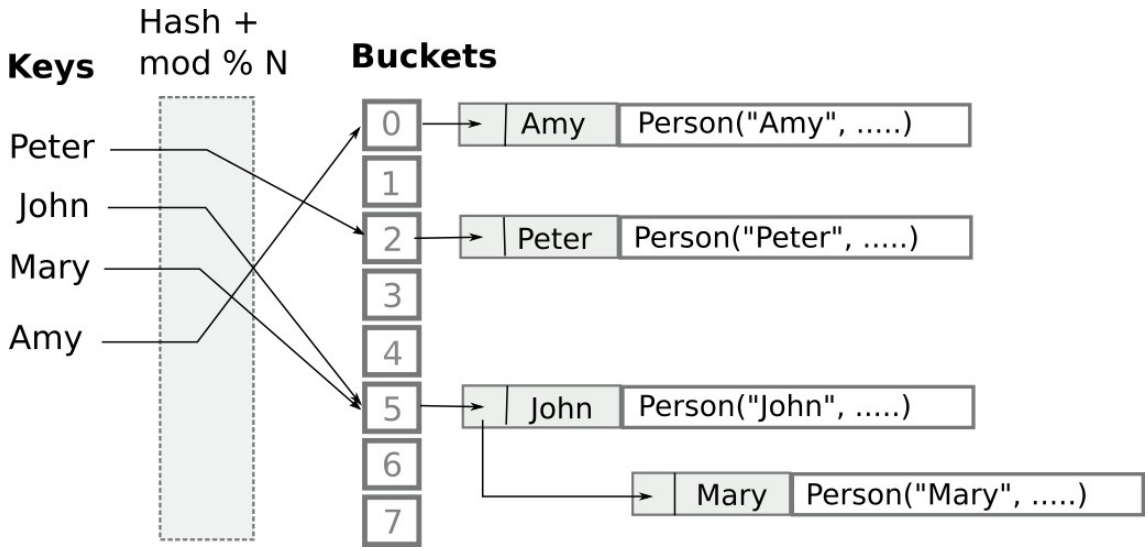
https://docs.oracle.com/javase/8/docs/api/java/util/Map.html

There are also many implementations of this interface, each suited for different needs:

2.2.1. HashMap

HashMap is a Map implementation using a hash function for distributing keys. The main difference with HashTable is that the latter is synchronized, so it's thread safe. Other difference is that with HashMap you can use ***null*** as a key.

This implementation is more complex than a list. In fact, it's like a combination of an ArrayList (numbered buckets represent the index), containing a LinkedList (when more than one value goes in the same bucket). The process is simple, a **hash function** is applied to the key, and then a **module** (number of buckets) to see which bucket corresponds to that key. Then, the key plus the value are stored in that bucket (assuming they're not already there, there's a search first). When you want to get a value the process is the same, with the difference that when you know the right bucket, you have to travel through all the values stored there (usually one or just a few) to find a match with the key.



The main advantage is that when you want to search for a value it's very fast to find (compared to a list for example), at the cost of calculating a hash function. When you want to insert a lot of entries and search a lot too, instead of iterating, a Map implementation is usually better suited than a List.

A HashMap (and HashTable) has two parameters that determine its performance, initial capacity (number of initial buckets, default 16), and load factor (default 0.75). A low “load factor” ensures that usually only one key or very few are stored in the same bucket (great search performance), but at the cost of more memory and frequent table growth. When the number of stored values reaches number of buckets * load factor, the HashMap size is doubled and all its keys hashes are recalculated (costly operation). It's important to estimate the approximate number of values that will be stored in order to initialize it with the right values.

https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

2.2.2. HashTable

It's the same as HashMap, with the difference that this one is synchronized and doesn't allow ***null*** as a key. When there are many threads accessing this collection, it's better to use **ConcurrentHashMap**. Otherwise, if only one thread is accessing this collection it's better to use **HashMap**.

https://docs.oracle.com/javase/8/docs/api/java/util/Hashtable.html

2.2.3. LinkedHashMap

The difference in this implementation compared to HashMap is that, in addition of having a Map structure, an internal linked list is maintained between values inserted. That linked list is ordered keeping the same order the values were added. The additional cost of this implementation is having two extra references for each value inserted (previous and next item). The advantage is that you can iterate through elements easily in the same order they were added, or generate a new copy of a LinkedHashMap, knowing that the order will be maintained (even if capacity and load factor are different).

https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html

2.2.4. TreeMap

This implementation does **not** use a hash function. Instead, it stores keys and values in a tree structure (red-black tree → https://en.wikipedia.org/wiki/Red%E2%80%93black_tree). In this structured, keys must maintain an order, so thy mus be comparable. By default, it uses natural order to compare keys (Integers → numerical, Strings → alphabetical, ….), but with more complex objects, you can provide a **Comparator** implementation to do so (for example Person objects that must be ordered by its DNI number,….).

This implementation is usually more costly than HashMap when you search and insert elements, but it's compensated by not having to calculate a hash function every time a key is used, so it has its advantages.

https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html

2.3. Set

Set defines a collection of objects with the main characteristic that it doesn't contain duplicates, and also doesn't keep those objects ordered with an index. It's just a repository of objects that lets you know if an object is in the set or not.

You can walk through the contained objects with an iterator, or generating an array with the method .toArray(). As always there are many implementations of this interface, each with its own pros and cons.

https://docs.oracle.com/javase/8/docs/api/java/util/Set.html

2.3.1. HashSet

The HashSet implementation uses an internal HashMap to store values (as keys that don't contain anything). It has its advantages: fast insertion, deletion and search. And its disadvantages: hash function, no established order, costly iteration.

https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html

2.3.2. TreeSet

It's based on TreeMap internally with its pros and cons. Values are stored as keys in the internal TreeMap.

https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html

2.3.3. LinkedHashSet

Equivalent to LinkedHashMap, it maintains the internal order in which elements where inserted.

https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html

Exercise 6

Create a project named **ShapeTree**, in which you must have the Shape interface defined in exercise 4 and its derived classes. Implement the method **toString()** in all Shape's derived classes returning the type of shape (triangle, rectangle, …) and the area.

In the main method, create a TreeSet that holds objects of type Shape. Insert at least two of each shape with the values you want. You must provide a **Comparator** instance to the TreeSet in the constructor using an anonymous class (section 1.3.1). This Comparator will compare the area of the shapes.

When you have inserted all shapes, iterate through all them, showing their .toString() message to see if they are ordered by area.

Exercise 7

Create a project named **Companies** with these classes (including Main):

- Company: Has name and money (double) attributes (value set in the constructor).

- Person: Has name and age attributes (value set in the constructor).

In the main method, create a **TreeMap** (ordered by companies money) in which the key is a Company and the value is a **TreeSet** of Persons (ordered by person's age). Populate the TreeMap with 3 companies (do not add them ordered by money) and each company will have a list of 3 people (not ordered when adding).

Iterate through the TreeMap (use the .entrySet() method to get a Set of keys ordered) and show in console the companies with their people (both should be ordered by money and age).

2.4. Queue

Queue is an special type of list that is designed to add elements always at the end of the list (add() and offer() methods), and do the get and remove operations at the beginning (FIFO) or at the end (LIFO or stack) of the list. It's not designed to access positions with an index (you should use a list for that). When you want to restrict operations to the beginning or the end of a list, it's more efficient to use Queue (or Deque) than List.

An special type of queue that allows to insert and remove from the beginning or the end of the queue is called **Deque** (Double Ended Queue), and it's also an interface. It also allows you to retrieve or remove an object that it's not on the edge of the queue.

https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html

https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html

When using multiple threads is better to use an implementation of **BlockingQueue** or **BlockingDeque** interfaces like **LinkedBlockingDeque**.

2.4.1. ArrayDeque

An implementation of a double ended queue (Deque) using an internal array. It's similar to ArrayList but designed for head and tail operations rather than using an index to access or modify items in the middle of the list.

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayDeque.html

2.4.2. LinkedList

This type of list we saw earlier, also implements the Deque interface with all its methods, so you can use it as a double ended queue.

https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html

2.5. Collections

The Java **Collections** class contains static methods designed to operate with, or generate new collections. Those methods include generating synchronized collections (for safe operation with multiple threads), order a List (with some kind of comparator), reverse a List, rotate a List, get the minimum element, maximum, …..

https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html

3. Enumerations

When you need to use a set of constants like months names, week days, colours, etc… It's better to use an enumeration than, for example, a list, or creating some string or integer constants. Enumerations can be iterated like a list, and are a limited set of read-only values.

In Java enumerations are classes. That means they can have a constructor (always private, so it works as a Singleton and you can't create more than one instance at a time → https://en.wikipedia.org/wiki/Singleton_pattern) and methods. They also can implement interfaces (and all their associated methods).

In a enumeration, all its values must be defined at the beginning of it, separated by comma, and usually in upper case (they're like constants).

```
public enum MyColours {
	RED, GREEN, BLUE, BLACK, WHITE;
}
```

As said before, you can implement a constructor and different methods. You must call the constructor when defining the values, in each one (each value will be an instance of the enum):

```
public enum MyColours {
	RED("FF0000"),
	GREEN("00FF00"),
	BLUE("0000FF"),
	BLACK("000000"),
	WHITE("FFFFFF");

	private final String rgbValue;

	private MyColours(String rgbValue) {
		this.rgbValue = rgbValue;
	}

	public String getRgbValue() {
		return rgbValue;
	}
}
```

You can access a value and its methods like this:

```
System.out.println(MyColours.RED.getRgbValue()); //Will print → FF0000.
```

You can use **.values()** method to get an array and iterate through all values:

```
for(MyColours col : MyColours.values()) {
	System.out.println(col.name() + " -> " + col.getRgbValue());
}
```

4. Exceptions

Exceptions are a commonly and usually recommended way to deal with errors in our applications, specially serious ones. When using a Java API (arrays, strings, files, dates, ….), any error, expected or not, will be thrown as an exception. Exception (and its subclasses) is an special class in Java (and other languages). What we are really getting when there is an error is an Exception (or a sobclass of Exception) object.

If we don't capture the exception object, or throw it so it can be captured above, a fatal error will be generated and crash our application inmediately. This policy can be defined as "*If there is an error that we are not expecting (by try...catch), it may cause a corruption in our valuable data, so it's better to stop the application that let it run and just inform (the computer can't know if an error is severe or not, that's why it stops the running operation so the programmer can decide what to do with that error in the future)*".

An exception is captured by a try...catch statement. You can capture as many exception types as you want inside a try block (multiple catch). It's **recommended** that you specify as many catch blocks as necessary for every type of exception instead of capturing a generic "Exception". You also can specify an finally block at the end to do operations that must be done independently if there's an error or not (like closing an stream).

```
String line;
BufferedReader bfRead = null;
try {
	bfRead = new BufferedReader(new FileReader("/home/pepe/file.txt"));

	line = bfRead.readLine();
	while(line != null) {
		line = bfRead.readLine();
	}
} catch (FileNotFoundException e) {
	System.err.println("File /home/pepe/file.txt doesn't exist!");
} catch (IOException e) {
	System.err.println("Error reading /home/pepe/file.txt.");
} finally {
	if(bfRead != null)
		try {
			bfRead.close();
		} catch (IOException e) {
			System.err.println("Couldn't close C:\file.txt stream.");
		}
}
```

Luckily, since Java 7, we can create variables (separated by semicolon ';') like arguments after the **try** word. Those variables classes must implement the **AutoCloseable** interface (like StreamReader). Those variables will exist locally only in the try block, and Java will close them automatically after that block ends. This practice will save a lot of ugly additional code:

```
String line;
```

```
try (BufferedReader bfRead = new BufferedReader(new FileReader("/home/pepe/file.txt"))){
	line = bfRead.readLine();
	while(line != null) {
		line = bfRead.readLine();
	}
} catch (FileNotFoundException e) {
	System.err.println("File /home/pepe/file.txt doesn't exist!");
} catch (IOException e) {
	System.err.println("Error reading /home/pepe/file.txt.");
}
```

If you think that an exception shouldn't be managed inside the method where it's being generated, but instead, you want to delegate that management to the "parent" method in the call stack, you can use the **throws** instruction at the beginning of the method's definition. After throws, you can write all the exceptions types you want (separated by comma) to be sent up. When a exception of that type is generated, the method will be interrupted returning the exception object generated that must be caught in the method that made the call. You can do that as many times as you want until the Main method. However, is usually not recommended to throw exceptions that are not created by ourselves.

You can always create a new exception (usually generic) and throw it with the special instruction **throw**:

```
public void welcome() throws Exception {
	throw new Exception("Error, nobody can pass!");
}
```

Custom exceptions

We can create our own exceptions by creating classes that inherit from **Exception** class. We can then throw an custom exception whenever we want and manage it in the method that throws it or send it up to the method it will return to.

```
public class CustomException extends Exception {
	public CustomException(String msg) {
		super(msg);
	}
}


public class Store {
	public void welcome() throws CustomException {
		throw new CustomException("Error, nobody can pass!");
	}
}

public static void main(String[] args) {
	Store store = new Store();
	try {
		store.welcome(); // This method can throw a CustomException
	} catch (CustomException e) {
		System.err.println(e.getMessage());
	}
```

```
}
```

Exercise 8

Create a project called **CustomException**. In this project you're going to implement:

- A class called **NegativeSubtractException**. This class will inherit from Exception and will be created when a subtraction result is negative. The constructor will receive 2 parameters (the two numbers that caused a negative subtraction result in order). The message generated will be: “NegativeSubstractException: 'N1 - N2' result is negative”.
- In the **Main** class create a **static** method that throws this type of custom exception. This method will be called **int positiveSubtract(int n1, int n2)**, and will generate and throw this kind of exception if the result is negative. Within the **main** method call this method with parameters that would give a negative result and catch the corresponding exception, showing its message on console.

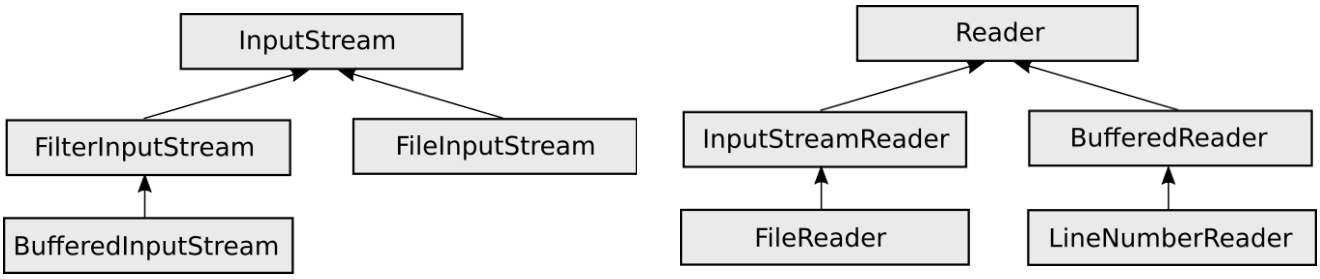
5. Java I/O

When operating with files or accessing remote services like web services, there are two many operations we can do, write (out), and read (in) into the stream of data that communicates our application with the file or service.

The most basic classes for managing these in/out operations are:

- **InputStream**: Represents an stream of data (bytes) that comes to our application for reading (input).
- **OutputStream**: Represents an stream of data (bytes) that goes from our application (write - output).
- **Reader**: Represents a **character** stream that comes to our application (input).
- **Writer**: Represents a character stream that goes from our application (output).

These classes provide functionality that is usually too basic for most common operations like file reading or writing. There are subclasses of these classes that implement more advanced functionality, which also usually have subclasses with more advanced or specific functionalities. This is called a **Decorator Pattern** in software design (every layer adds more specific functionalities).



You can use a combination of Readers for example:

```
try(BufferedReader buffer = new BufferedReader(new FileReader("/home/pepe/file.txt"))) {
	String line;
	while((line = buffer.readLine()) != null) {
		System.out.println(line);
	}
} catch (FileNotFoundException e) {
	System.err.println("File /home/pepe/file.txt doesn't exist!");
} catch (IOException e) {
	System.err.println("Error reading /home/pepe/file.txt.");
}
```

Or Writers:

```
try (PrintWriter print = new PrintWriter(new FileWriter("/home/pepe/file.txt")))
{
	print.println("Hello world!");
} catch (IOException e) {
	System.err.println("Error writing: " + e.getMessage());
}
```

It's possible to write your own implementation of one of these classes using inheritance and add some new methods or modifying some functionality.

```
public class UpperCaseReader extends BufferedReader {

	public UpperCaseReader(Reader reader) {
		super(reader);
	}

	@Override
	public String readLine() throws IOException {
		String line = super.readLine();
		return line != null?line.toUpperCase():null;
	}
}
```

// MAIN

```
try(UpperCaseReader buffer = new UpperCaseReader(new FileReader("/home/pepe/file.txt")))
{
	String line;
	while((line = buffer.readLine()) != null) {
		System.out.println(line);
	}
} catch (FileNotFoundException e) {
	…
} catch (IOException e1) {
	…
}
```

Exercise 9

Create a project named **RegexReader** containing the following things:

- A class called **RegexReader** that inherits from **BufferedReader**. This class will receive a Reader in the constructor and a String containing a regular expression to match when reading lines. You'll have to override the readLine() method. It will use the parents readLine() method and return the first line that matches the regular expression received in the constructor (use the **.matches(String)** method from the class String), ignoring all other lines (until it gets **null**, you'll have to return that too when that happens).
- In the **main** method, create a **RegexReader** object that will give you only lines containing a date (dd/mm/yy or dd/mm/yyyy format), mixed with text. A regular expression in Java always tries to match all the String to be valid, so you'll have to add **.*** at the beginning and at the end of the expression, also **\d** is invalid, you'll have to use **\\d**. Create a file that contains a date in some lines and try it.

5.1. Filesystem operations

Since Java 7, new classes have been added to facilitate operations with the OS filesystem. We are mainly talking about **Paths**, **Filesystems** and **Files** classes, which

provide static methods for this purpose. Most of the time, these classes return a **Path** or a **Filesystem** object.

We'll use a Java **Path** interface object to represent a file in the filesystem:

```
Path file = Paths.get("/home/pepe/file.txt");

System.out.println("Path: " + file.getParent());
System.out.println("Separator: " + file.getFileSystem().getSeparator());
System.out.println("File name: " + file.getFileName());
```

We can do all basic operations with files and directories (Path objects) using Files methods:

```
Path file = Paths.get("/home/pepe/file.txt");
Path parent = file.getParent();
Path destiny = Paths.get("/home/pepe/docs");

// Create
if(Files.isDirectory(parent) && Files.isWritable(parent) &&
      !Files.exists(file)) {
      try {
            Files.createFile(file);
      } catch (IOException e) {
            e.printStackTrace();
      }
}

// Move
if(Files.isDirectory(destiny) && Files.isWritable(destiny)) {
      try {
            Files.move(file, destiny);
      } catch (IOException e) {
            e.printStackTrace();
      }
}

// Delete
try {
      Files.deleteIfExists(file);
} catch (IOException e) {
      e.printStackTrace();
}
```

https://docs.oracle.com/javase/8/docs/api/java/nio/file/Path.html

https://docs.oracle.com/javase/8/docs/api/java/nio/file/Paths.html

https://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html

https://docs.oracle.com/javase/8/docs/api/java/nio/file/FileSystem.html

https://docs.oracle.com/javase/8/docs/api/java/nio/file/FileSystems.html

6. Annotations

Annotations are meta-instructions and won't compile like the rest of our code. They provide useful information to the compiler and different tools like: automatic code generation tools, documentation tools (JavaDoc), Testing Frameworks (Junit,…), ORM (Hibernate,….), etc.

They're placed before a class, a field or a method definition, affecting what comes next. They start with the character '@'. Some examples of annotations that the compiler interprets are:

- **@Deprecated** → Applies to a method. It can be used but will give you a warning when you try to use it somewhere. This is for giving advice that they should stop using this method and use some new functionality instead, because it won't be available soon, maybe in the next version of the program or library.
- **@Override** → Informs the compiler that the method is overriden. It's not necessary to use this but if you do and the method doesn't exist in the parent class (not overriden), the compiler will throw an error.
- **@SuppressWarnings(“type”)** →Disables a type of warning inside a method, some examples of warning types are “unused” (unused variables), “deprecated” (use of deprecated methods), …

7. New in Java 8

Java 8 was released in March 2014, and it brings to the table some new interesting functionalities. Some of them, like lambda expressions (also called anonymous functions), brings functional programming style to Java, like many other languages nowadays (C#, PHP, C++ (since C++11 standard), Ruby, Javascript, and many others).

7.1. StringJoiner

Before Java 7, String concatenation '+' was not very efficient. Internally, a new String object was created for every concatenation, and if there were more than 1 concatenation, intermediate objects were also being deleted at the same time, consuming a lot of resources. The recommended way to concatenate various Strings was to use **StringBuilder**:

```
StringBuilder strBuild = new StringBuilder("Hello");
strBuild.append(" ").append("world!").append(" Yes!");
String str = strBuild.toString();
```

This was more efficient than doing String str = “Hello” + “ “ + “world!” + “ Yes!”. Since Java 7, the compiler uses internally StringBuilder to process String concatenations, so there's no problem with that anymore (if you still use Java 6, better user StringBuilder manually).

Since Java 8, you can use **StringJoiner**, a more flexible version of StringBuilder. For example, you can specify a default separator for every string you add:

```
StringJoiner joiner = new StringJoiner(", ");
joiner.add("Peter").add("John").add("Mary");
System.out.println(joiner.toString()); // Will print -> Peter, John, Mary
```

You can also specify a prefix and a postfix:

```
StringJoiner joiner = new StringJoiner(", ", "[", "]");
joiner.add("Peter").add("John").add("Mary");
System.out.println(joiner.toString()); // Will print -> [Peter, John, Mary]
```

You can use it from the static method **.join()** of Array class. The first argument is the separator and the second an array or an iterable collection.

```
String[] names = {"Peter", "John", "Mary"};
System.out.println(String.join(", ", names)); // Will print -> Peter, John, Mary
```

7.2. Dates

Before Java7, most date operations were handled by the class Date. Java 7 deprecated most of its methods and you were forced to rely on the **Calendar** (or **GregorianCalendar**) class.

```
Date d = new Date(); // Represents NOW. All other constructors are deprecated
```

We can still use Date objects, but almost all its methods are deprecated, so most of the operations must be done with a Calendar object:

```
// If I want to create an specific date, I must use Calendar
Calendar cal = Calendar.getInstance(); // NOW -> default locale and timezone
cal.set(2014, 1, 10); // 10th Feb. 2014 (January -> 0)
cal.add(Calendar.DAY_OF_MONTH, 7); // Add 7 days
Date weekLater = cal.getTime(); // Returns Date object
```

Since Java 8, **java.time** package classes replace Date, Time, TimeStamp and Calendar functionality (still supported for compatibility with all code). Dates and time processing becomes much more flexible and ready for international timezones adaptability.

Instant

Represents a moment (date and time) with a precision of nanoseconds, and it's immutable.

Duration

It's the amount of time between two Instant. You can get its value in different units, add or subtract time, etc.

```
Instant start = Instant.now();
// Some code
Instant end = Instant.now();

Duration dur = Duration.between(start, end);
long milliseconds = dur.toMillis();
```

LocalDate

Replaces Date class:

```
LocalDate now = LocalDate.now();
LocalDate birth = LocalDate.of(1950, Month.JULY, 13);
```

Period

Is the amount of time between two LocalDate. Same concept as Duration.

```
LocalDate now = LocalDate.now();
LocalDate birth = LocalDate.of(1950, Month.JULY, 13);

Period age = birth.until(now);

System.out.println("Lived: " + age.getYears() + " years, " +
            age.getMonths() + " months, " + age.getDays() + " days.");

System.out.println("Total days lived: " + birth.until(now, ChronoUnit.DAYS));
```

TemporalAdjuster and TemporalAdjusters

Useful to add or subtract an amount of time to an Instant or LocalDate.

```
LocalDate now = LocalDate.now();
LocalDate nextMonday = now.with(TemporalAdjusters.next(DayOfWeek.MONDAY));
```

TemporalAdjusters provides 14 static methods for these kind of operations, like firstDayOfMonth(), lastDayOfMonth(), firstDayOfYear(),…

LocalTime

LocalTime represents the time of a day:

```
LocalTime now = LocalTime.now();
LocalTime bedTime = LocalTime.of(23, 0);
LocalTime wakeUp = bedTime.plusHours(8);
System.out.printf("I will wake up at %02d:%02d",
      wakeUp.getHour(), wakeUp.getMinute());
```

LocalDateTime

Has information about both date and time:

```
LocalDateTime dateTime = LocalDateTime.now();
```

Zoned Time

With Java 8, you can easily manage zoned DateTime, that means, that a ZonedDateTime (date + time) contains also information about the world time zone.

ZoneId class provides ids for every time zone supported (https://www.iana.org/time-zones)

```
Set<String> zonesIds = ZoneId.getAvailableZoneIds();
ZoneId spainZone = ZoneId.of("Europe/Madrid");
```

This is how we create a zoned datetime:

```
ZonedDateTime zonedDate = ZonedDateTime.of(
            1969, Month.AUGUST.getValue(), 14, // year, month, day
            14, 25, 0, 0,                      // hour, min, sec, nanos
            ZoneId.of("Europe/Madrid"));
```

We can add or substract units to this date, change timezone, etc...

```
ZonedDateTime date2 = zonedDate.plus(Period.ofMonths(5)); // Add 5 months
ZonedDateTime date3 = zonedDate.plus(15, ChronoUnit.DAYS);
ZonedDateTime dateUk = zonedDate.withZoneSameInstant(ZoneId.of("Europe/London"));
```

Format dates

Using DateTimeFormatter predefined formats or creating our own:

```
// Will print 14/25/1969
System.out.println(DateTimeFormatter.ofPattern("d/m/Y").format(zonedDate));
// Will print Thu, 14 Aug 1969 14:25:00 +0100
System.out.println(DateTimeFormatter.RFC_1123_DATE_TIME.format(zonedDate));
```

Bridges between APIs

How to interoperate with legacy Date API (before Java 8):

- Instant ↔ Date

```
Date date = Date.from(Instant.now()); // Instant → Date
Instant instant = date.toInstant();   // Date → Instant
```

- Instant ↔ TimeStamp

```
Timestamp time = Timestamp.from(Instant.now()); // Instant → TimeStamp
Instant instant = time.toInstant(); // TimeStamp → Instant
```

Exercise 10

Create a project named **OrderedComments** and a file called **comments.txt** that contains user comments (do not order them by date) separated by semicolon ';' in this format:

username;comment;dd/mm/yyyy;hour:min:sec;timezone.

Example: peter;Hello everybody;23/09/2013;10:04:54;Europe/London

Create a class named **Comment** that holds the username, the comment and the ZonedDateTime date.

In the main method, read the file and create a **Comment** object for every comment (you can use the String's **split** method to divide fields), change the timezone of each comment's date to "Europe/Madrid" (use **withZoneSameInstant** method), and add comments to a List. After that, order the list by date of comments using a Comparator. To compare a date to another (in seconds), you can use:

date1.until(date2, ChronoUnit.SECONDS) → "date2 - date1" in seconds

Finally, create a new file called **ordered_comments.txt** and save all comments in the same format as before but they will be ordered by date and in the same timezone (Madrid). To convert the ZonedDateTime to the date format in the file use the method **.format(DateTimeFormatter)** with this formatter:

```
DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("dd/MM/yyy;HH:mm:ss;VV");
```

https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html

7.3. Lambda expressions

Since Java 8, lambda expressions, also called anonymous functions in many languages, are an easy and fast way to implement an Interface method without having to create a new class for doing that.

Lets give an example. We have this **functional interface** (interface with only one implementable method) for accepting files (java.io.FileFilter):

```
public interface FileFilter {
	boolean accept(File file);
}
```

Before Java 7, we had to create a new class for every new implementation we wanted to create. For example:

```
public class JavaFileFilter implements FileFilter {

	@Override
	public boolean accept(File file) {
		return file.getName().endsWith(".java");
	}
}
```

And then use it like this:

```
File dir = new File("/home/arturo");
File[] javaFiles = dir.listFiles(new JavaFileFilter());
```

Since Java 7 we could create **anonymous classes**, interfaces or abstract classes whose methods are implemented when we create an object (we could make different implementations every time).

```
File dir = new File("/home/arturo");
File[] javaFiles = dir.listFiles(new FileFilter() {

	@Override
	public boolean accept(File file) {
		return file.getName().endsWith(".java");
	}

});
```

Since Java 8. When we are implementing an interface with a single method (functional interface), we can do it with less code using a **lambda expression**:

```
File dir = new File("/home/arturo");
File[] javaFiles = dir.listFiles((File file) -> file.getName().endsWith(".java"));
```

We don't have to specify that is a FileFilter Interface what we are implementing because the compiler knows that the **listFiles()** method needs a FileFilter derived object. We also don't need to use the **return** word because the compiler will assume it. We can even omit the parameter type because the compiler can look at it in the interface definition:

```
File[] javaFiles = dir.listFiles(file -> file.getName().endsWith(".java"));
```

Example with Comparator using String length:

```
// Java 7
Comparator<String> comp = new Comparator<String>() {

	@Override
	public int compare(String s1, String s2) {
		return Integer.compare(s1.length(), s2.length());
	}
};

// Java 8
Comparator<String> lambdaComp = (s1,s2) -> Integer.compare(s1.length(), s2.length());

List<String> list = Arrays.asList("Hello", "Hi", "Goodbye", "Farewell", "Bye");
Collections.sort(list, lambdaComp);
```

Example with Runnable, for implementing threads in Unit 3:

```
// Java 7
Runnable run = new Runnable() {

	@Override
	public void run() {
		for(int i = 0; i < 3; i++) {
			System.out.println("This is thread: " +
					Thread.currentThread().getName());
		}
	}
};
```

```
// Java 8
Runnable lambdaRun = () -> {
	for(int i = 0; i < 3; i++) {
		System.out.println("This is thread: " +
				Thread.currentThread().getName());
	}
};

Thread t = new Thread(lambdaRun);
t.start();
try {
	t.join();
} catch (InterruptedException e) {
	e.printStackTrace();
}
```

There is even a way to make a lambda expression shorter. When it contains a method call that takes the same parameters as the lambda expression and in the same order, we can just write a reference to that method, omitting even the parameters, using this especial syntax:

```
Comparator<Integer> comp  = (i1, i2) -> Integer.compare(i1, i2);
Comparator<Integer> comp2 = Integer::compare;

Consumer<String> con  = s -> System.out.println(s);
Consumer<String> con2 = System.out::println;
```

Lambda vs Anonymous class instance

When using lambda expressions, the compiler doesn't create full objects, but instead it creates an special and lighter type of object, so it's usually a better choice.

Exercise 11

Create a project called **ListFilter** that includes the following:

- A class called **Student** that has these properties (getters/setters when necessary):
  - Name
  - Age
  - List of subjects (as Strings)
- A **Main** class with a main method and also a **static** method called:

  List<Student> filterStudents(List<Student> srcList, Predicate<Student> predicate)

The method **filterStudents** receives a student list and returns another list with only the items which meet the condition defined in the Predicate. A Predicate is a functional interface (included in java 8) that needs to implement a method (**boolean test(T t)**). Implement it using lambda expressions.

In the main method you'll have to create a list of at least 8 students, and then, using the method filterStudents generate 3 other lists that only hold students who:

1. Are older than 20.

2. Are inscribed in the “Programming” subject.
3. His name contains “Peter”.

7.4. Interfaces

Until Java 8, an Interface could only have abstract methods and constants (final static). Now, an interface can also have static (an implemented) methods, and **default** methods (also implemented) that will be inherited in the classes that implement that interface.

For example, the **Iterable** Java interface (**java.lang**), apart from the **iterator()** method, which is abstract, has introduced two new default methods. One of them is **forEach()**, that takes a Consumer implemented interface (we'll see what a Consumer is soon) and applies it to all the elements of a collection for example.

```
public interface Iterable<E> {
	// Other methods

	default void forEach(Consumer<E> consumer) {
		Objects.requireNonNull(consumer);
		for(E e: this) {
			consumer.accept(e);
		}
	}
}
```

A default method will be inherited as if it was a regular class method. Static methods can also be defined if needed.

A **functional** interface is an interface that only has **one** abstract method, so you can implement them using **lambda expressions**. You can inform the compiler that an Interface is functional with the new **@FunctionalInterface** annotation. In Java 8, many new functional interfaces for generic operations have been defined inside package **java.util.function**.

These functional interfaces are divided in 4 categories:

1. **Supplier**: Does not take any argument. Provides an object.

```
@FunctionalInterface
public interface Supplier<T> {
	T get();
}
```

2. **Consumer**: Takes an object (or more) as argument and doesn't return anything (System::println).

```
@FunctionalInterface
public interface Consumer<T> {
	void accept(T t);
}
@FunctionalInterface
public interface BiConsumer<T, U> {
	void accept(T t, U u);
}
```

```
	// MAIN
	List<String> strings = Arrays.asList("one", "two", "three", "four",
"five");
	List<String> result = new ArrayList<>();

	Consumer<String> cPrint = System.out::println;
	Consumer<String> cAdd = result::add;

	// Will print and then add to the other list (chaining consumers)
	strings.forEach(cPrint.andThen(cAdd));
```

3. **Predicate**: Takes an object (or more) and returns a boolean.

```
@FunctionalInterface
public interface Predicate<T> {
	boolean test(T t);
}


@FunctionalInterface
public interface BiPredicate<T, U> {
	boolean test(T t, U u);
}

// MAIN
Predicate<String> p1 = s -> s.length() < 20;
Predicate<String> p2 = s -> s.length() > 10;
Predicate<String> p3 = p1.and(p2); // Applies both p1 and p2
// You can also create new predicates with .or(Predicate) and .negate()
```

4. **Function**: Takes an object (or more) and returns another object.

```
@FunctionalInterface
public interface Function<T ,R> {
	R apply(T t);
}

@FunctionalInterface
public interface BiFunction<T, U ,R> {
	R apply(T t, U u);
}

@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {
	// A Function that receives and returns the same type
}

@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T, T, T> {
	// A BiFunction that receives and returns the same type
}


BiFunction<String, Integer, String> biFunc = (s,i) -> s.substring(i);
Function<String, String> func = s -> s.toUpperCase();
// Combine both functions in a third one
BiFunction<String, Integer, String> biFunc2 = biFunc.andThen(func);
```

https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

Exercise 12

Using the same project " **ListFilter**" created in exercise 11, we're going to modify it:

Instantiate in the **main** method a **BiFunction** called **commonSubjects**, that receives two Students and returns a **List<String>**. Use a lamba expression. It will return the list of subjects that both students have in common.

Instantiate a **Function** called **join** that takes a **List<String>** and returns a String containing all strings in the list separated by commas (use lambda).

Create a third **BiFunction** called **commonJoin** that takes 2 students and returns a String. This Function will be the combination of **commonSubjects** and **join** together. Try it with some students printing the results in console.

7.5. Streams

In Java, a stream is a new concept designed to process large (or small) amounts of data using a new level of abstraction in combination with lambda expressions that can be easily parallelizable to take advantage of all available CPU cores without having to do it ourselves manually (split the computation and create new threads).

A stream:

- Is an special object on which we can define operations (lambda expressions)
- It doesn't hold any data (acts as an intermediary).
- It doesn't change the data it processes. The compiler allows you to do so but you shouldn't → unpredictable behaviour.
- It processes all data of an operation and passes it to the next operation. It doesn't do anything until you call a '**final**' or '**terminal**' operation (we'll see what's a final operation)
- It behaves, in some ways, similar to SQL.

Stream vs Collection

First of all, for compatibility issues, the Collection framework hasn't been transformed to work like streams do, so they are separate things. Most of the time, streams are generated from collections and several times they generate a new collections as a result, but they're not the same.

With a Collection, the programmer has to iterate and operate with all its values manually, and if he/she wants to parallelize the operation, has to create the necessary threads and divide the problem by him/herself. With a stream, you only have to define the operations that will be done with all data, and it will automatically iterate and apply the defined operations, and also divide the problem and generate threads when using parallel streams.

Intermediary (lazy) and final (terminal) operations

When using streams, nothing is executed until we do what is called a “final operation”. All operations before the final one are called intermediary operations, and return another stream so we can chain as many of them as we want.

Filter operation

A filter operation is an **intermediary** operation. It takes a Predicate as an argument, meaning that it will accept data that matches the condition and reject data that doesn't.

```
List<Person> persons = new ArrayList<>(10);
persons.add(new Person(16, "Peter"));
persons.add(new Person(22, "Mary"));
persons.add(new Person(43, "John"));
persons.add(new Person(70, "Amy"));

Stream<Person> stream  = persons.stream();
Stream<Person> stream2 = stream.filter(p -> p.getAge() > 18); // Intermediary
stream2.forEach(p -> System.out.println(p.toString())); // Final (processes all)

// A shorter way to do exactly the same
persons.stream().filter(p -> p.getAge() > 18).forEach(System.out::println);
```

Mapping operations

Mapping operations are also **intermediary** operations. It uses a Function, meaning that it takes an item (input) and returns a different output. For example, we can use it to extract persons ages in our previous example:

```
persons.stream()
      .filter(p -> p.getAge() > 18)
      .map(p -> p.getAge())  // .map(Person::getAge) works too
      .forEach(System.out::println); // Prints only ages
```

Reduction operations

Reductions are **final** or **terminal** operations. An important thing to note is that a Stream can only have **one final operation**. Once it has processed data, the stream can't be reused, so you need to create another stream.

Reductions return one element that can be of any type. Sometimes, they won't return anything (for example, the minimum element of an empty list), and that would be a problem if we assign that result to a variable. We can use the class **Optional** to manage that without problems.

Types of reductions:

- Returns Optional
  - max(), min()
  - findAny(), findFirst()
- Returns long
  - count()
- Returns boolean

    - allMatch(), noneMatch(), anyMatch()
- Other (the return value depends on the operation done)
    - reduce()

Let's see an example:

```
int sumAges = persons.stream()
      .filter(p -> p.getAge() > 18)
      .map(p -> p.getAge())
      .reduce(0, (a,b) -> a + b);
```

In this case, we are calculating the sum of all ages above 18. The reduce method's first argument is the **base case**, the value that will be taken if the list is empty. Other operations like **max**, doesn't make sense to take a base value, so they will return an **Optional** (in case the list is empty there's no result).

```
Optional<Integer> maxAge = persons.stream()
      .filter(p -> p.getAge() > 18)
      .map(p -> p.getAge())
      .reduce(Integer::max);

System.out.println(maxAge); // Will print Optional[70]
System.out.println(maxAge.isPresent()?maxAge.get():"No max age");// Better
```

With mapToInt(), mapToDouble(), etc… values are casted into primitive types and you can do more and simpler operations. Like calculate the average of all ages (older than 18):

```
OptionalDouble avgAge = persons.stream()
      .filter(p -> p.getAge() > 18)
      .mapToInt(p -> p.getAge()).average();

System.out.println(
      avgAge.isPresent()?
            "Avg: " + avgAge.getAsDouble()
            :"No ages");
```

Another example:

```
// Search if there is anyone younger than 18
if(persons.stream().anyMatch(p -> p.getAge() < 18)) {
      System.out.println("There are children in the collection!");
}
```

Collecting operations

These kind of operations are also **final**. Instead of returning an object, a primitive value, or nothing (that's why we use Optional), they usually return a Collection, or sometimes a String (joining operation). The method used for this operations is **.collect()**.

The most basic operation is to pass one parameter, a **Collector** object, which we can create from the **Collectors** class:

```
String names = persons.stream()
      .filter(p -> p.getAge() >= 18)
```

```
        .map(p -> p.getName())
        .collect(
                Collectors.joining(",", "Adults: ", "")
        );

System.out.println(names); // Adults: Mary,John,Amy
```

We can generate new Lists:

```
List<Person> older = persons.stream()
        .filter(p -> p.getAge() >= 18)
        .collect(
                Collectors.toList()
        );
```

We can return a Map using Collectors.groupingBy(key, calculated value):

```
Map<Integer, Long> ages = persons.stream()
        .filter(p -> p.getAge() >= 18)
        .collect(
                    Collectors.groupingBy(
                                p -> p.getAge(),
                                Collectors.counting()
                    )
        );

System.out.println(ages.toString());
// {70=1, 22=1, 43=1}
```

With only one parameter, groupingBy returns a List of items grouped by every key:

```
Map<Integer, List<Person>> ages = persons.stream()
        .filter(p -> p.getAge() >= 18)
        .collect(
                    Collectors.groupingBy(
                                p -> p.getAge()
                    )
        );

System.out.println(ages.toString());
// {70=[Name: Amy, age: 70], 22=[Name: Mary, age: 22], 43=[Name: John, age: 43]}
```

We can choose to generate a different list passing a function ike this:

```
Map<Integer, List<String>> ages = persons.stream()
        .filter(p -> p.getAge() >= 18)
        .collect(
                Collectors.groupingBy(
                        p -> p.getAge(),
                        Collectors.mapping(
                                p -> p.getName(), // Insert only the name
                                Collectors.toList()
                        )
                )
        );

System.out.println(ages.toString());
// {70=[Amy], 22=[Mary], 43=[John]}
```

Parallelism

With Streams, you can parallelize easily the operations you want to do. All you have to do is create a Parallel Stream calling .parallelStream(), instead of .stream() (or .stream().parallel()) in a Collection, for example. Java will divide the problem into pieces and execute each in a different thread (running in parallel in different processors if available). There are two main problems with this aproach:

- When the data set is small, adding the overhead of dividing the problem and create and manage threads is not worth it. Usually it will be slower than running a sequential stream.
- Java uses the Fork/Join Framework (We'll look at it in unit 3) to create and manage these threads. Then problem is that those threads are created in the same pool as the main program's thread instead of creating them in another pool with less priority. That means, if there are less available processors than threads and some of them are slow to compute, they will block sometimes the main thread (and other threads too). It's recommended to avoid use of methods like .forEach() and .peek(), as they can have unpredictable results.

When generating collections in parallel streams, you should use the concurrent versions of those collections:

```
ConcurrentMap<Integer, List<String>> ages = persons.parallelStream()
	.filter(p -> p.getAge() >= 18)
	.collect(
		Collectors.groupingByConcurrent(
			p -> p.getAge(),
			Collectors.mapping(
				p -> p.getName(),
				Collectors.toList()
			)
		)
	);
```

https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html

To sum up, we'll see an example of how we did the same kind of operations with Java 7, and the difference with using streams and lambda expressions in Java 8:

```
// Store the names of adult people sorted by age. JAVA 7
List<Person> older = new ArrayList<>();

for(Person p : persons){
	if(p.getAge() >= 18){
		older.add(p);
	}
}

Collections.sort(older, new Comparator<Person>(){
	public int compare(Person p1, Person p2){
		return Integer.compare(p1.getAge(), p2.getAge());
	}
});
```

```
List<String> namesOlder = new ArrayList<>();
for(Person p : older){
      namesOlder.add(p.getName());
}

// Store the names of adult people sorted by age. JAVA 8
List<String> namesOlder = persons.stream()
            .filter(p -> p.getAge() >= 18)
            .sorted(Comparator.comparing(Person::getAge))
            .map(Person::getName)
            .collect(Collectors.toList());

// Could use: .sorted((p1, p2) -> Integer.compare(p1.getAge(), p2.getAge()))
// And: .map(p -> p.getName())
```

http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html

http://www.oracle.com/technetwork/articles/java/architect-streams-pt2-2227132.html

Exercise 13

Using the project **ListFilter** of exercises 11 and 12, we're going to add some things.

In the Main class add another static method called **getOldestNames(List<Student> list)** that returns List<String>. Inside, you'll have to create a stream out of the list, and generate a list of the names of the three oldest people of the list (use **.limit(3)** after sorting). Try it and print the results to see if it works well.

Exercise 14

Using the project **ListFilter** of exercises 11, 12 and 13, we're going to add some things.

We'll use streams to get our results. Create another **static** method in the **Main** class called **Set<String> getAllSubjects(List<Student> list)**. In this method we're going to create a stream out of the student's list and generate a list (Set) that contains all subjects which appear at least in one student (or more) of the list, **ordered alphabetically**. Try it passing some list and printing the results.

7.6. Files and I/O

A **.lines()** method has been added to **BufferedReader** that returns a **Stream<String>**:

```
try(BufferedReader reader = new BufferedReader(
            new FileReader("/home/arturo/file.txt"))) {

      reader.lines()
            .filter(line -> line.contains("Login:"))
            .forEach(System.out::println);
} catch (FileNotFoundException e) {
      // Do something
} catch (IOException e) {
      // Do something
}
```

Files.lines() method even creates the Stream without having to create a **BufferedReader**, and it's **AutoCloseable**, so you can create it inside the try declaration:

```
try(Stream<String> stream = Files.lines(Paths.get("/home/arturo", "file.txt")))
{

	stream
		.filter(line -> line.contains("Login:"))
		.forEach(System.out::println);
} catch (IOException e) {
	// Do something
}
```

Files.list() returns a Stream<Path> containing a list of all files and directories present in the current directory (passed as a parameter):

```
try(Stream<Path> stream = Files.list(Paths.get("/home/arturo"))) {

	stream // Prints subdirectories
		.filter(path -> path.toFile().isDirectory())
		.forEach(System.out::println);
} catch (IOException e) {
	// Do something
}
```

Files.walk() is very similar but it also explores subdirectories. The second parameter is the maximum depth you want to go in the directory tree:

```
try(Stream<Path> stream = Files.walk(Paths.get("/home/arturo"), 2)) {

	stream // Prints subdirectories
		.filter(path -> path.toFile().isDirectory())
		.forEach(System.out::println);
} catch (IOException e) {
	// Do something
}
```

Exercise 15

Create a porject named **OrderedStreamComments** that does the same as in exercise 10, but using a stream to read the file and generate the ordered list of comments.

7.7. Collections

New method, **forEach()**, in **Iterable** collections:

```
List<Integer> list = Arrays.asList(4,6,7,9);

list.forEach(elem -> {
	if(elem % 2 == 0) {
		System.out.println(elem);
	}
```

```
});
```

Collection → removeIf(Predicate): Returns a boolean if operation has been done. It modifies the original collection, so it doesn't work with immutable collections:

```
// Arrays.asList returns a fixed size list (you can't remove or add), so
// you have to generate another type of list from it
List<Integer> list = new ArrayList<>(Arrays.asList(4,6,7,9,12,15));

if(list.removeIf(elem -> elem < 10)) {
	System.out.println(list.stream().map(i -> i.toString())
		.collect(Collectors.joining(", ")));
}
```

List → replaceAll(UnaryOperator): Applies a Function (UnaryOperator) to all its elements modifying them:

```
List<Integer> list = Arrays.asList(4,6,7,9,12,15);

list.replaceAll(Math::incrementExact); // Increments all by 1
System.out.println(
	list.stream().map(i -> i.toString()).collect(Collectors.joining(", ")));
```

List → sort(Comparator): Orders the elements in the list modifying the original list:

```
List<Integer> list = Arrays.asList(4,6,7,9,12,15);

list.sort(Comparator.reverseOrder());
System.out.println(
		list.stream().map(i -> i.toString()).collect(Collectors.joining(",
")));
```

Comparator. New methods. For example, what can we do with nulls?:

```
list.sort(Comparator.nullsFirst(Comparator.naturalOrder()));
list.sort(Comparator.nullsLast(Comparator.naturalOrder()));
```

A comparator to order people by their first name, and if it's equal, use the first name (Java 7 vs Java 8):

```
List<Person> list = new ArrayList<>();

// JAVA 7
list.sort(new Comparator<Person>() {
	@Override
	public int compare(Person p1, Person p2) {
		int last = p1.getLastName().compareTo(p2.getLastName());
		if(last == 0) { // Equal last name
			return p1.getFirstName().compareTo(p2.getFirstName());
		}
		return last;
	}
});

// JAVA 8
list.sort(Comparator.comparing(Person::getLastName)
```

```
            .thenComparing(Person::getFirstName));
```

Map → New Methods:

- forEach(BiConsumer). Same as List::forEach but with BiConsumer (key,value):

```
Map<String, Person> map = new HashMap<>();
...

map.forEach((key,person) ->
      System.out.println("Key: " + key +
            ". Person: " + person.getFullName()));
```

- getOrDefault(key, defaultValue). If the key doesn't have a value it returns the default value (received as a parameter).

```
Map<String, Person> map = new HashMap<>();
Person defaultPerson = ...;

Person p = map.getOrDefault(key, defaultPerson);
```

- putIfAbsent(key, value). With put(), if a key already holded a value, it modified it without asking.

```
Map<String, Person> map = new HashMap<>();

map.putIfAbsent(key, person); // Only if key doesn't already exist
```

- replaceAll(BiFunction). Applies the function passed as a parameter to all the elements transforming them.

```
Map<String, Person> map = new HashMap<>();

map.replaceAll((key, p) ->
      new Person(p.getFirstName().toUpperCase(),
                 p.getLastName().toUpperCase()));
```

- remove(key, value). The main difference with remove(key) is that if the received value isn't associated with the key, it doesn't remove anything.
- compute(), computeIfPresent(), computeIfAbsent(). Similar to replaceAll(), but only applies to one key:

```
Map<String, Person> map = new HashMap<>();

map.computeIfPresent(key,
            (key,person) -> new Person(p.getFirstName().toUpperCase(),
                                       p.getLastName().toUpperCase()));
```

- merge(). Receives a key and a value (new). If the key is empty, the new value is assigned. Otherwise, a BiFunction will be computed for merging the current and new values:

```
map.merge(key, newValue, (currentVal, newValue) -> ...);
```

```
Map<Integer, List<Person>> map1 = new HashMap<>();
// Add values...
Map<Integer, List<Person>> map2 = new HashMap<>();
// Add values...


// Add all map2 elements to map1
map2.entrySet().stream()
      .forEach(
            entry -> // Map2 (val,key)
            map1.merge(entry.getKey(),
                        entry.getValue(), // If key doesn't exists, assigns this
                        (map1Val, map2Val) -> {
                              map1Val.addAll(map2Val);
                              return map1Val; // Assigns map1 + map2 elements
                         })
      );
```

Exercise 16

Create a project named **AggregateMoney**. You'll have to implement the following:

Create a **Map<String, Double>** which contains several people's DNI as keys and their money as value. After that, add a 5% to all values using the method **.replaceAll(...)**.

Create another Map with the same characteristics, and store some other values (use some of the keys from the first Map too).

Merge both Maps into the first one. When a person's DNI exists in both Maps, add the money stored in both (use the **merge** method as in the example above). Print the final result.