

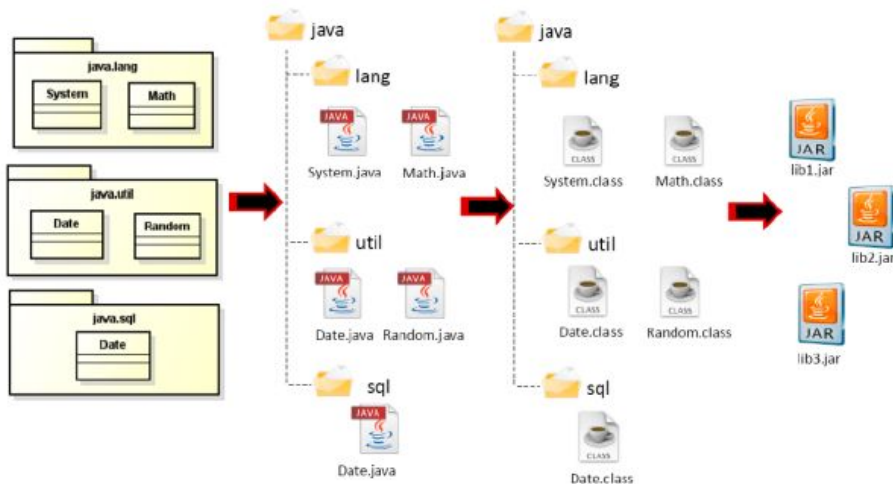
# M03. PROGRAMACIÓ

**UF5. OOP. Llibreries de classes fonamentals**

# 1. JAVA MODULES

# JAVA API

En les versions anteriors a JDK9, es treballava amb una plataforma Java monolítica. Així, una única llibreria nadiua de Java amb tots els seus components (llibreries) centralitzats. Aquesta API estava organitzada en packages (paquets), on podíem trobar totes les classes natives de Java. Aquest fet comportava que, per tal d'executar una aplicació senzilla, calia descarregar el JRE amb totes les llibreries de la plataforma.



# THE JIGSAW PROJECT

Es redissenya l'arquitectura de Java, afegint un component més d'abstracció: els modules. De manera que ara, tenim:

- **packages**: agrupen classes relacionades
- **modules**: agrupen packages relacionats

Els mòduls no només encapsulen els paquets i classes de les nostres aplicacions, sinó que determinen qui pot accedir o a qui deixen utilitzar aquestes classes i paquets.

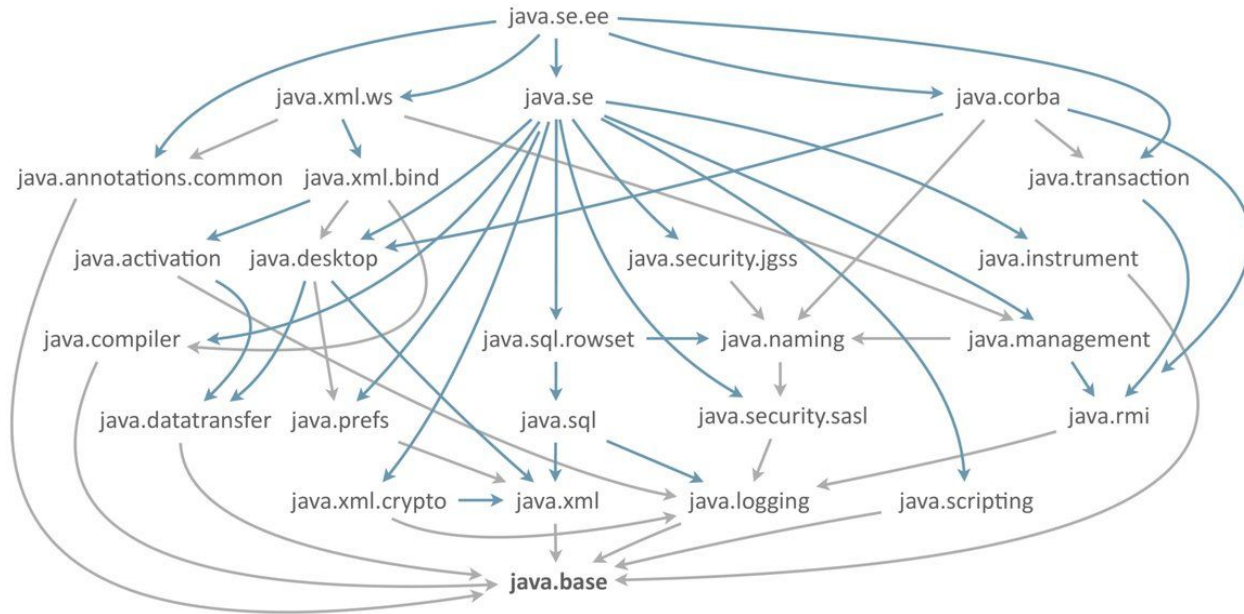
**Important!** Un package no pot tenir les seves classes distribuïdes per diferents modules.

# THE JIGSAW PROJECT



La nova estructura de l'API es redissenya, partint ara del mòdul `java.base` com el mòdul principal (conté els paquets bàsics (`lang`, `io`, `math`,...)).

# THE JIGSAW PROJECT



Gràfic de dependències de mòduls (API Java)

## 2. OBJECT CLASS

# OBJECT: THE HIGHEST SUPERCLASS

Tots els objectes extenen de la classe principal **Object**. Així, cada superclasse extén implícitament la classe Object. Es considera el component més alt i general de qualsevol jerarquia. És l'única classe que no té una superclasse i conté mètodes molt generals que cada classe hereta.

Els mètodes definits en la classe Object estan disponibles per a qualsevol classe, tot i que el seu funcionament no està adaptat a les diferents classes de l'aplicació. És per aquest motiu que cal substituir-los, sobreescrivint les seves funcionalitats a les nostres classes.

Els mètodes que habitualment se sobreescriuen són:

- toString
- equals
- finalize



# OBJECT.FINALIZE

És un mètode que el Garbage Collector invoca (de manera automàtica) quan aquesta determina que no hi ha més referències a l'objecte.

```
@Override
protected void finalize() throws Throwable {
    try {
        ... // cleanup subclass state
    } finally {
        super.finalize();
    }
}
```

Atenció!! És un mètode deprecated.

# OBJECT.EQUALS

Si apliquem l'operador de comparació "==" sobre dades de tipus primitius, compara si les dues dades contenen el mateix valor. Què passarà si ho apliquem a dos objectes?

```
System.out.println(cotxe2 == cotxe1);  
> false
```

L'operador compara si les dues referències (instàncies de l'objecte) fan referència a un mateix objecte, no si els continguts dels dos objectes són iguals. Si no se sobreesciu el mètode de la classe Object, resulta que `x.equals(y)` dóna el mateix resultat que "`x == y`".

```
System.out.println(cotxe2.equals(cotxe1));  
> false
```

# OBJECT.EQUALS

```
@Override
public boolean equals(Object obj) {
    if (obj == this) {
        return true;
    }

    if (!(obj instanceof Car)) {
        return false;
    }

    Car c = (Car) obj;
    return c.getCompanyName().
        equalsIgnoreCase(this.getCompanyName());
}
```

# OBJECT.HASHCODE

Retorna un codi hash de l'objecte (instància). No hi ha directrius respecte la generació d'aquest identificador d'instància (es pot fer de manera senzilla o més complexa).

```
@Override  
public int hashCode() {  
    return this.addPlateNumber(MAX);  
}
```

```
@Override  
public int hashCode() {  
    return Objects.hash(x, y, z);  
}
```

# OBJECT.TOSTRING

El mètode `System.out.println()` mostra una cadena per consola. Què passarà si intentem mostrar un objecte (la seva instància)?

```
System.out.println(cotxe);  
  
> cat.mvm.myapp.entities.Car@681a9515
```

El que ens mostra és el *fully-qualified-class-name@hashcode* de la instància.

En general, s'acostuma a sobreescrivre el mètode `toString()`, per tal d'obtenir la sortida adequada quan s'utilitza un objecte en invocar els mètodes `print()` o `println()`.

### 3. MANIPULACIÓ DE LA INFORMACIÓ

# GENERIC TYPES

Un **generic type** (tipus genèric) és una classe genèrica o interfície que està parametritzada sobre tipus.

Propietats:

- Tenen un paràmetre de tipus (l'operador de diamant que conté el tipus)
- Els paràmetres de tipus poden ser acotats

# GENERIC TYPES

Els noms de paràmetres de tipus més utilitzats són:

- E: Element (utilitzat àmpliament pel Java Collection Frameworks)
- K: Clau
- N: Número
- T: Tipus
- V: Valor
- S,U,V, etc.: 2n, 3r, 4t tipus



# GENERIC TYPES

```
public class Space<T> {  
    private T minX, maxX, minY, maxY;  
  
    public Space(T minX, T maxX, T minY, T maxY) {  
        this.setMinX(minX);  
        this.setMaxX(maxX);  
        this.setMinY(minY);  
        this.setMaxY(maxY);  
    }  
}
```

```
Space<Integer> spaceInt = new Space<Integer>(0,0,1920, 1280);  
Space<Double> spaceDoub = new Space<Double>(0.0,0.0,50.0, 75.5);  
System.out.println(spaceInt);  
System.out.println(spaceDoub);
```

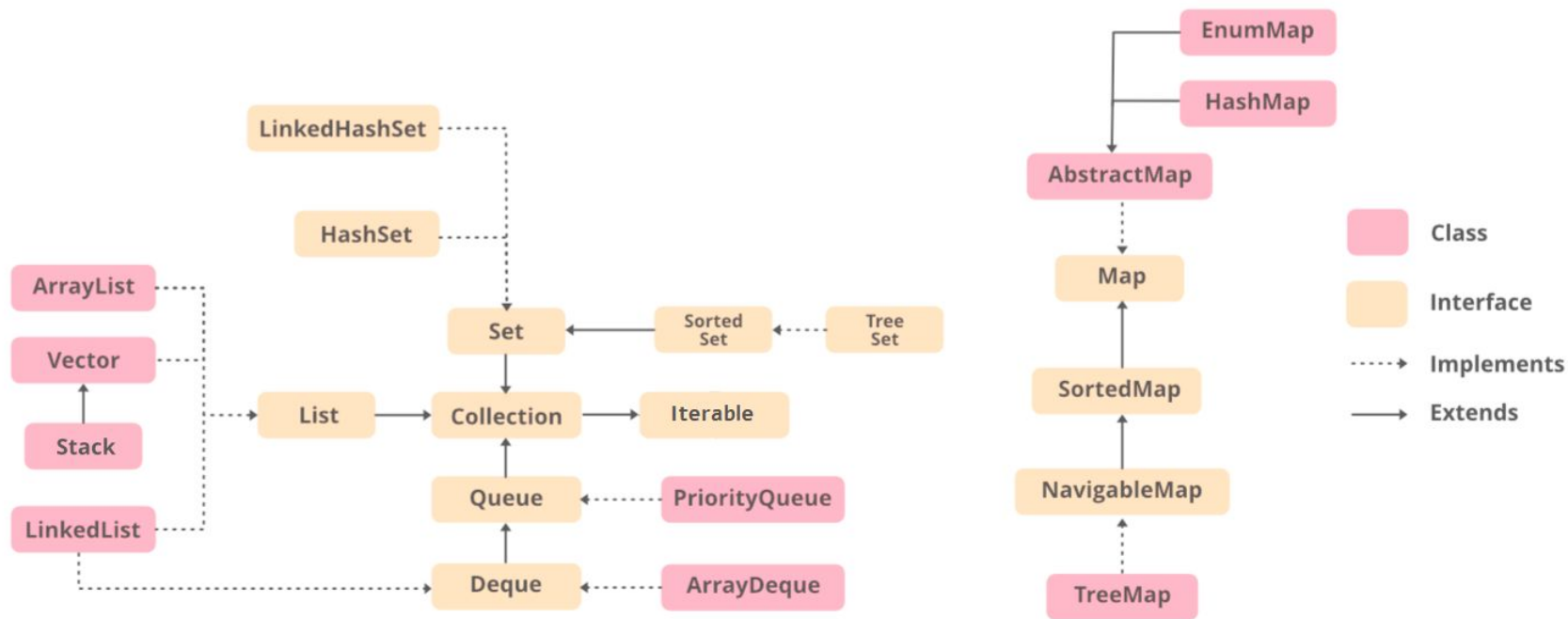
# COLLECTIONS

Una col·lecció (**Collection**) és un tipus de dades que agrupa objectes d'un mateix tipus.

- Algunes col·leccions admeten elements duplicats, unes altres no.
- Algunes mantenen els elements ordenats segons determinats criteris, unes altres no.

En Java, les col·leccions es representen mitjançant el framework Collections, dins del paquet [java.util](#).

# COLLECTIONS



# INTERFACES

Una *interface* centralitza un conjunt d'operacions que certs tipus d'objectes han d'implementar. És una manera d'assegurar que els nostres tipus de dades segueixin un comportament específic.

Els mètodes, per defecte i encara que no ho indiquem, són públics i abstractes i els atributs, `public static final`. Des de Java 8, disposem de noves interfaces (funcionals), dins del package [`java.util.function`](#).

# INTERFACES

Amb les noves versions de la plataforma, s'introdueixen diversos canvis a les *interfaces*:

Method type	Versió des de la qual és disponible
public abstract	Java 7
public default	Java 8
public static	Java 8
private	Java 9
private static	Java 9

# FUNCTIONAL INTERFACES

Una *functional interface* (interfície funcional) és aquella que només defineix un únic mètode abstracte. Se les coneixen com **SAM** (Single Abstract Method) interfaces i són aquelles que ens permetran treballar amb *expressions lambda*.

Dins la *functional interface*, podem crear diferents tipus de mètodes:

- **defender methods**: mètodes amb una implementació per defecte que no obliga a les classes implementadores a implementar-los. La seva visibilitat és pública i es declaren amb la paraula reservada *default*.
- **mètodes estàtics**: permetran agrupar de manera estàtica utilitats sense haver d'estar creant classes pròpies amb mètodes de tipus helper.

Indicant l'anotació *@FunctionalInterface*, obliguem a només declarar un mètode abstracte.

# CLASSES ANÒNIMES

Una *classe anònima* permet sobreescriure el mètode abstracte de la interface que es vol implementar. Aquesta pràctica era habitual fins a la versió 8 de java i es pot veure en diverses implementacions actualment (SWING).

```
//classes anònimes
var lc3 = new LoveCalculator(){
    @Override
    public int getCompatibilityBetween(String name1, String name2) {
        var sum1 = 0.0;
        for(char c : name1.toCharArray()){
            sum1 += (int)c;
        }


        var sum2 = 0.0;
        for(char c : name2.toCharArray()){
            sum2 += (int)c;
        }

        return (sum1 <= sum2) ? (int)((sum1/sum2) *
            LoveCalculator.MAX_VALUE) :
            (int)((sum2/sum1) * LoveCalculator.MAX_VALUE);
    }
};
System.out.println("lc3.
    getCompatibilityBetween(name1: "Josep", name2: "Maria"));
```

# LAMBDA EXPRESSIONS

Les expressions lambda ( $\lambda$ ) són una manera fàcil de fabricar-se un objecte d'un tipus anònim que implementa l'únic mètode abstracte de la interface funcional, per tal de poder executar aquest mètode. Se centra en els paràmetres de l'operació i en la implementació del mètode.

`(arguments) -> {function body}`



una funció lambda pot contenir diversos arguments, separats per coma

el cos de la funció pot contenir o no la sentència return



# LAMBDA EXPRESSIONS

```
//Expressió lambda
LoveCalculator lc4 = (String name1, String name2) -> {
    var sum1 = 0.0;
    for(char c : name1.toCharArray()){
        sum1 += (int)c;
    }

    var sum2 = 0.0;
    for(char c : name2.toCharArray()){
        sum2 += (int)c;
    }

    return (sum1 <= sum2) ?
        (int)((sum1/sum2) * LoveCalculator.MAX_VALUE) :
        (int)((sum2/sum1) * LoveCalculator.MAX_VALUE);
};

System.out.println("lc4.
    getCompatibilityBetween(name1: "Josep", name2: "Maria"));
```

# METHOD REFERENCES

Un method reference (mètode per referència) és una forma simplificada d'escriure una expressió lambda. Només d'aquelles que contenen una ÚNICA sentència.

```
Object::methodName
```

Per exemple:

```
Consumer<String> c = s -> System.out.println(s);
```

I el seu method reference:

```
Consumer<String> c = System.out::println;
```

# JAVA.UTIL.FUNCTION

- **Predicate<T>**: avaluarà si l'argument que rep compleix amb la condició indicada
- **Consumer<T>**: consumirà l'objecte que li passem com a argument
- **Function<T>**: retornarà un resultat a partir dels paràmetres que rep. S'utilitza en Streams
- **Supplier<T>**: proveirà d'un resultat
- **Operator<T>**: operació que retornarà un valor diferent al rebut però del mateix tipus

# JAVA STREAMS API

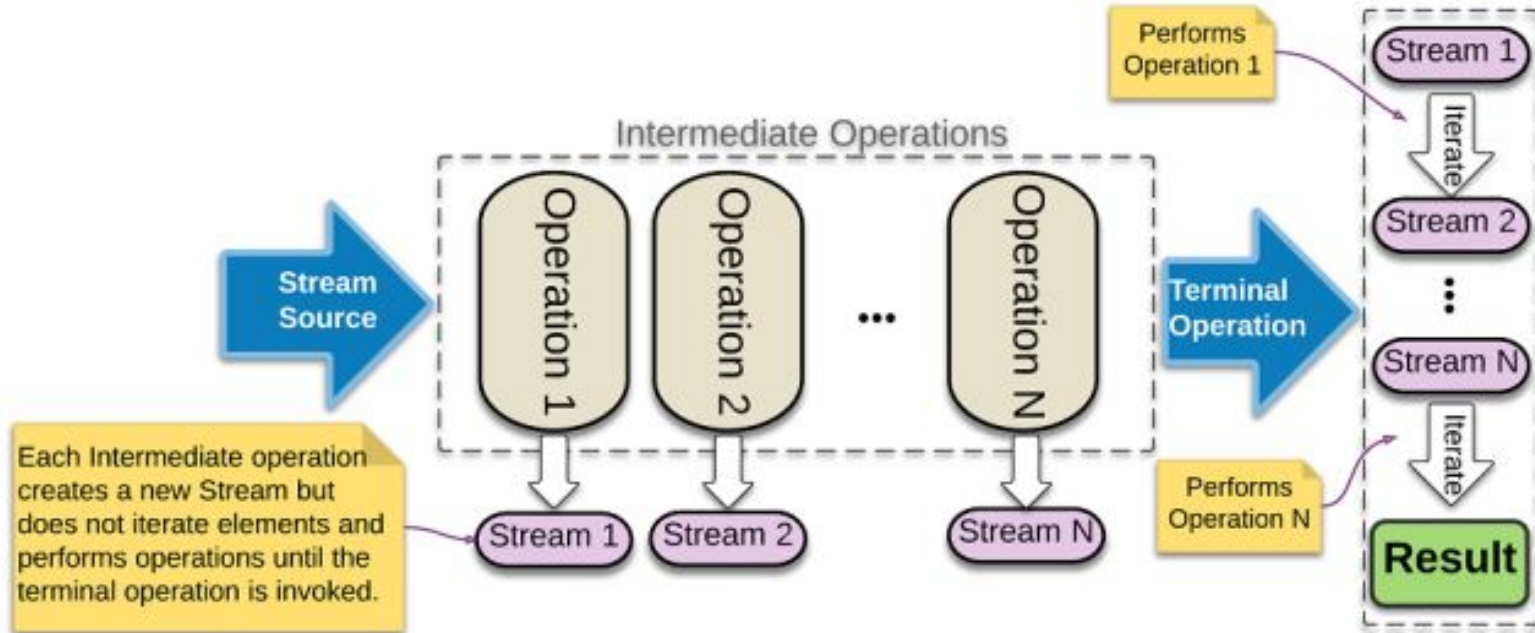
Un [Stream](#) és una seqüència d'objectes que volem manipular.

No és una col·lecció ni emmagatzema dades (no és una estructura de dades). No modifica la font de dades original, sinó que crea una seqüència d'objectes en memòria i realitza operacions amb ells (pipeline). Totes les operacions que executem en un *Stream* són en mode lazily (mandrós), ja que no s'executen fins que se li diu a l'*Stream* que s'ha de processar.

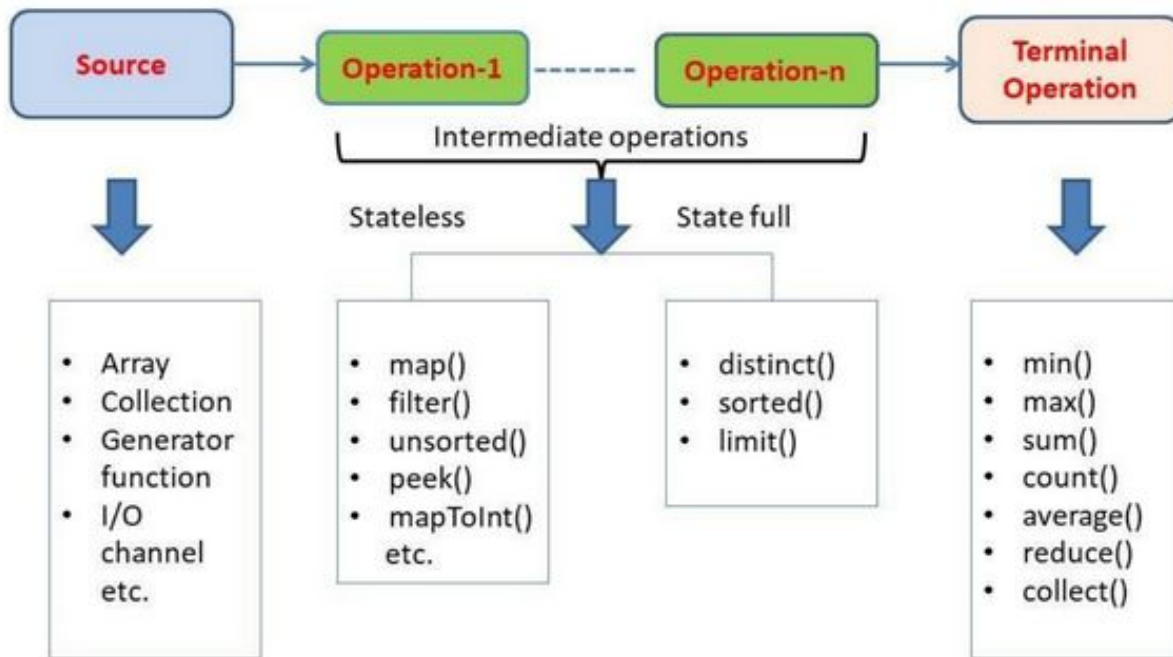
Per exemple, en el cas de Linux, podem encadenar instruccions mitjançant l'operador | (pipe):

```
cat file1 file2 | tr "[A-Z]" "[a-z]" | sort | tail -3
```

# JAVA STREAMS API



# JAVA STREAMS API



Cada operació intermèdia genera un nou Stream.

L'operació terminal "força" l'execució de totes les operacions intermèdies.