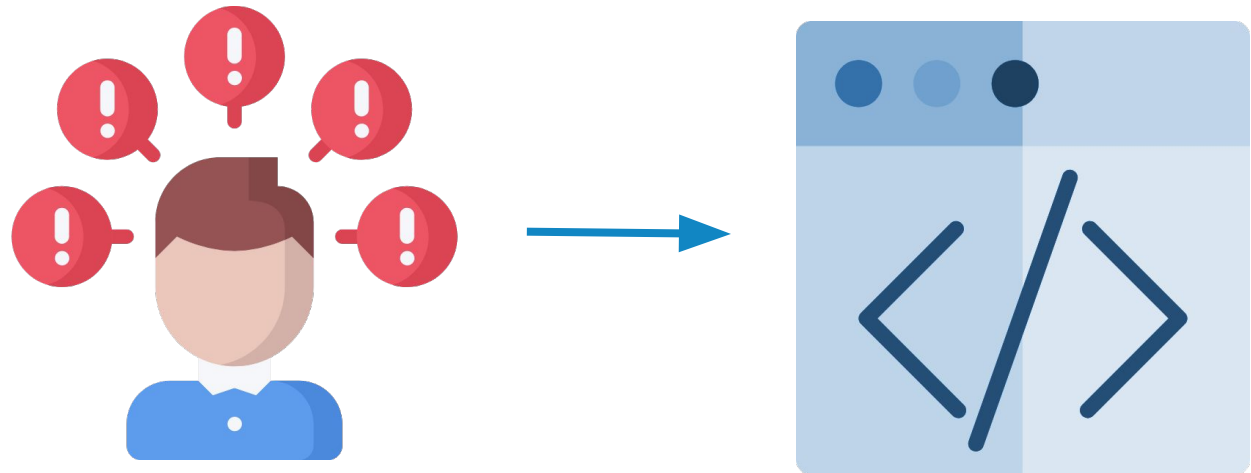


Orientación a Objetos

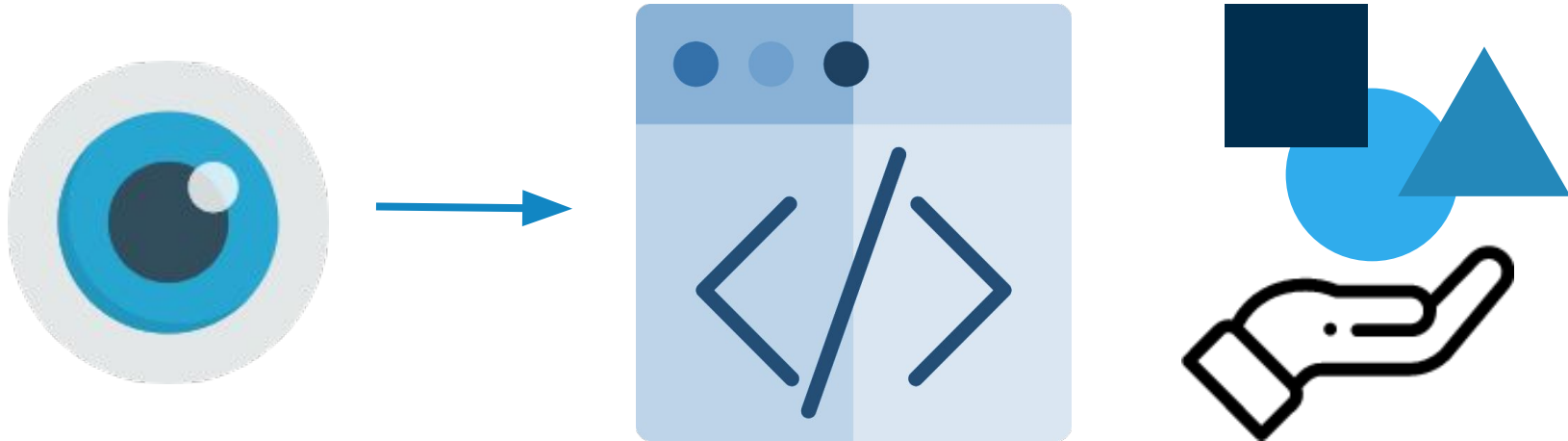
Orientación a Objetos

Surge a partir de los problemas que tenemos y necesitamos plasmar en código



Orientación a Objetos

Observar los problemas en
forma de objetos





Curso de Java SE Orientado a Objetos

Anahí Salgado
@anncode

1. Curso Básico de Java SE

**2. Curso de Java SE Orientado
a Objetos**

3. Curso de Java SE:
Programación Funcional

-
1. Curso de Programación Orientada a Objetos
 2. Curso de Java SE Orientado a Objetos

Paradigma

Programación
Orientada a Objetos

Paradigma

- + Teoría que suministra la base y modelo para resolver problemas

Se compone de
estos 4 elementos:

Clases

Propiedades

Métodos

Objetos

Tiene estos
pilares:

Encapsulamiento

Abstracción

Herencia

Polimorfismo

UML

Unified Modeling Language

UML

Unified Modeling

Language

Lenguaje de Modelado Unificado

UML

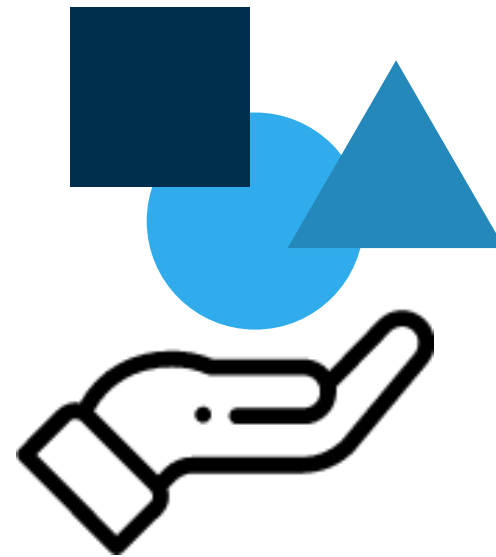
Unified Modeling Language

- + Clases
- + Casos de Uso
- + Objetos
- + Actividades
- + Iteración
- + Estados
- + Implementación

Objetos

Objetos

Cuando tengamos un problema lo primero que debemos hacer es **identificar Objetos**

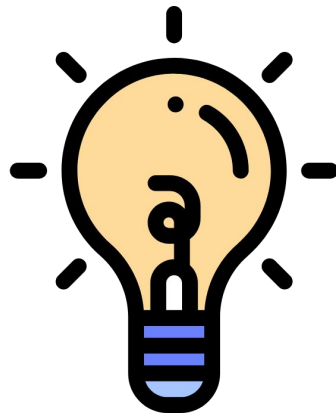
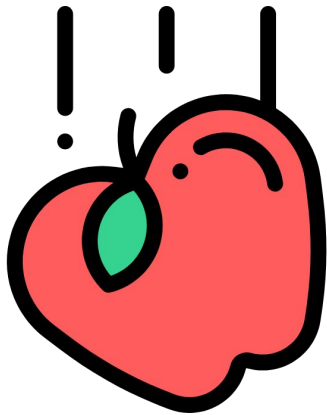


Objetos

Son aquellos que tienen
propiedades y
comportamientos

Objetos

Pueden ser Físicos o Conceptuales





User



Session

Propiedades

También pueden llamarse
atributos, serán
sustantivos

Propiedades

También pueden
llamarse **atributos**,
serán sustantivos

nombre, tamaño, forma, estado

Comportamientos

Serán todas las
operaciones del objeto,
suelen ser verbos o
sustantivo y verbo

Comportamientos

serán todas las
operaciones del objeto,
suelen ser verbos o
sustantivo y verbo

`login(), logout(), makeReport()`

“

Ejemplo ilustrativo

”

Clase

Clase

Es el modelo sobre el cual se construirá nuestro objeto

Clase

Las clases me permitirán generar más objetos

Analizar Objetos para crear Clases



Abstracción



Clases

Son los modelos sobre
los cuales construiremos
Objetos

UML

Nombre Clase

Atributo 1

Atributo 2

Atributo 3

Atributo n

Operación 1

Operación 2

Operación 3

Operación n

Identidad

Estado

Comportamiento

Person

name

walk ()

Modularidad

Diseño Modular





2 Seats + 4 Sides



4 seats + 5 Sides



6 seats + 8 sides



8 seats + 10 sides



Modular Orientado a Objetos

- + Reutilizar
- + Evitar colapsos
- + Mantenable
- + Legibilidad
- + Resolución rápida de problemas

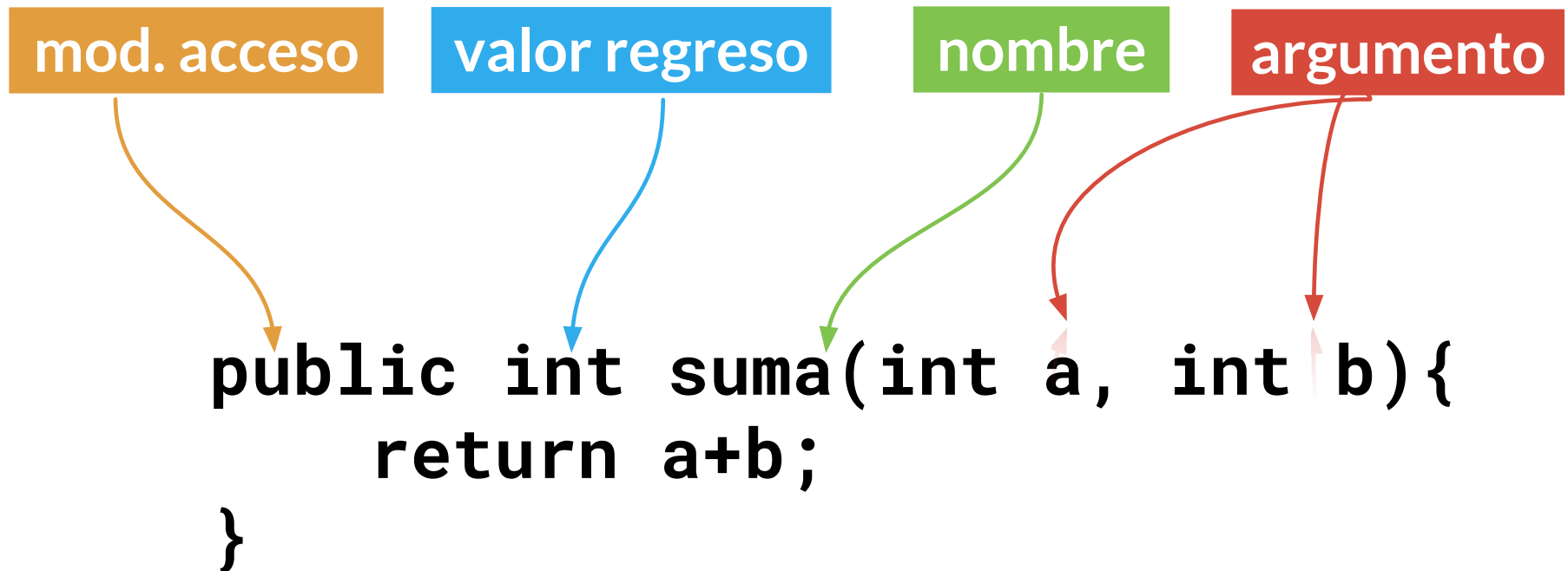
Clase

Clase

- Modularidad
- Divide el programa en diferentes partes o módulos / clases
- Separar las clases en archivos

Declarar Clases

Métodos



Declarando un objeto

Doctor



Tipo de Objeto

myDoctor;



Nombre del Objeto

Instanciando un objeto

myDoctor = new Doctor();

Nombre del Objeto

Creando el objeto

Declarando e instanciando un objeto

```
Doctor myDoctor = new Doctor();
```

Declarando
el objeto

Instanciando/
Creando el objeto

Utilizando el objeto

```
Doctor myDoctor = new Doctor();  
myDoctor.name = "Alejandro López";  
myDoctor.showName();
```

Método Constructor

```
myDoctor = new Doctor();
```

Método Constructor

```
myDoctor = new Doctor();
```



Método Constructor

Método Constructor

- Crea nuevas instancias de una clase.
- Tiene el mismo nombre que la clase que inicializa.
- Usa la palabra reservada **new** para invocarlo.

Método Constructor

Usa **cero o más argumentos** contenidos dentro de los paréntesis que siguen al nombre.

No regresa un valor.

Accesando a Métodos

```
Doctor myDoctor = new Doctor();  
myDoctor.showName();
```

Accesando a Métodos

```
Doctor myDoctor = new Doctor();  
myDoctor.showName();
```

```
Math.random();
```

```
Math.sqrt(25);
```

```
Math.PI;
```

Métodos **static**

Se puede usar en toda la clase.

Está definido por la palabra reservada **static**.

Métodos static

Puede ser **accesado** indicando el **nombre de la clase**, la notación punto y el nombre del método.

Métodos static

Se invoca en una clase que no tiene instancias de la clase.

Métodos static

```
public class Calculadora {
```

```
    public static int suma(int a, int b){  
        return a+b;  
    }
```

```
}
```

```
Calculadora.suma(5,2);
```

Métodos static

Puede ser invocado en una clase que **no tiene instancias de la clase.**

Miembros static

```
public class Calculadora{
```

```
    public static final double PI = 3.1415926
```

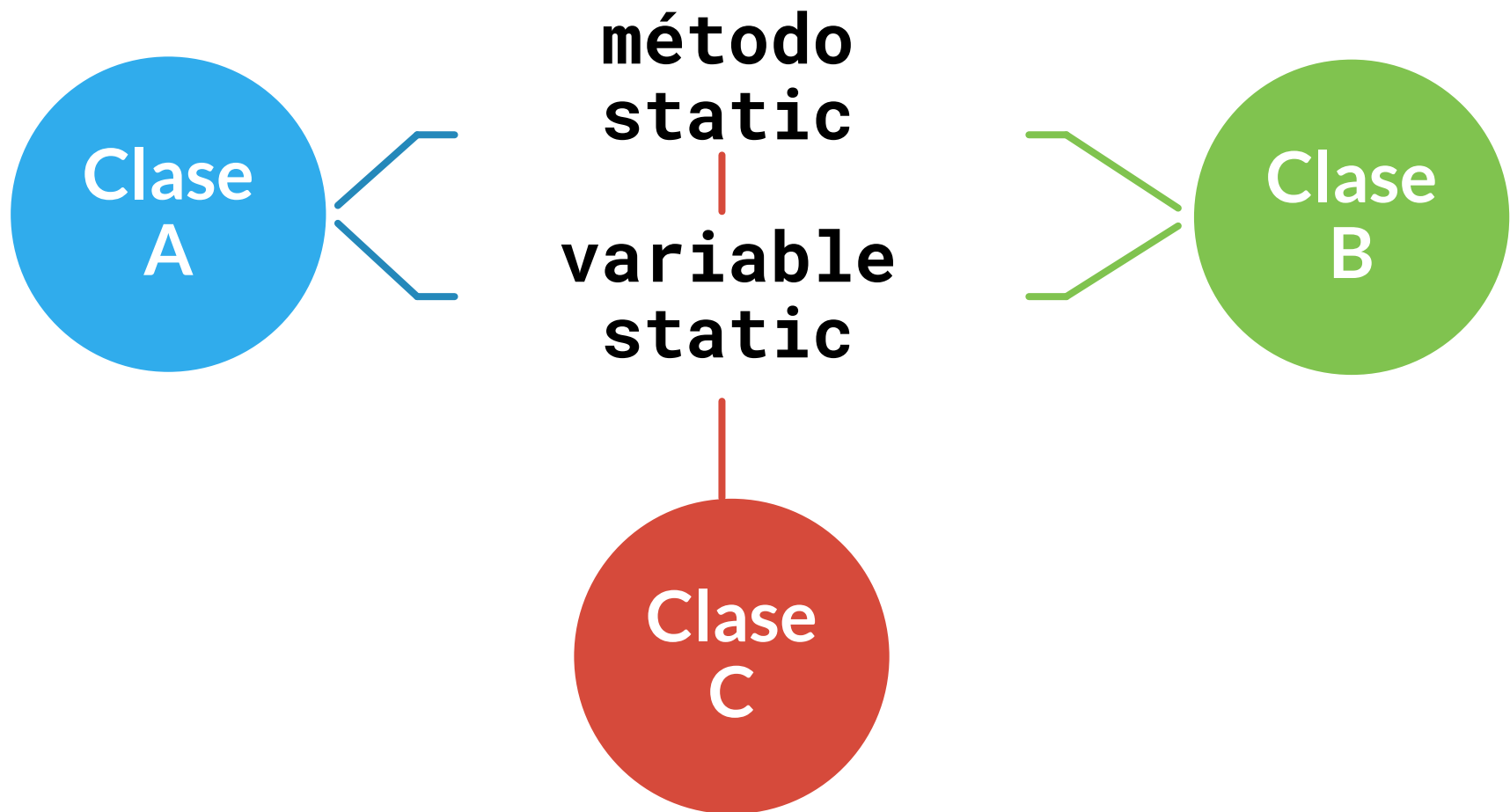
```
    public static int valor = 0;
```

```
}
```

Calculadora.PI;

Calculadora.valor;

Miembros static



Miembros static

```
import static com.anncode.operaciones.Calculadora.*  
import static java.lang.Math.*;  
  
public class Principal{  
  
    public static void main(String[] args){  
        System.out.println(suma(3, 5));  
        System.out.println(PI);  
    }  
  
}
```


Final

Miembros final

```
public class Calculadora{
```

```
    public static final double  
    PI = 3.1415926
```

```
}
```

```
    Calculadora.PI;
```


Sobrecarga

Sobrecarga

A veces necesitamos que dos o más métodos tengan el mismo nombre pero con diferentes argumentos

Sobrecarga

```
public class Calculadora{
```

```
    public int suma(int a, int b){  
        return a+b;  
    }
```

```
    public float suma(float a, float b){  
        return a+b;  
    }
```

```
    public float suma(int a, float b){  
        return a+b;  
    }
```

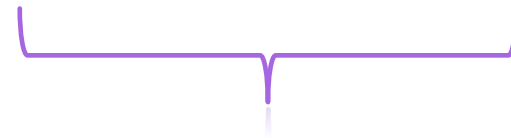
```
}
```

Sobrecarga de Constructores

La sobrecarga de constructores se usa para **inicializar objetos**

Sobrecarga de Constructores

```
Doctor myDoctor = new Doctor();
```



Método Constructor

Sobrecarga de Constructores

```
public class Doctor {  
    static int id = 0; //Autoincrement  
    String name;  
    String speciality;  
  
    public Doctor(){  
  
    }  
  
    public Doctor(String name, String speciality){  
        this.name = name;  
        this.speciality = speciality;  
    }  
}
```


Modificadores de Acceso

Sobrecarga de Constructores

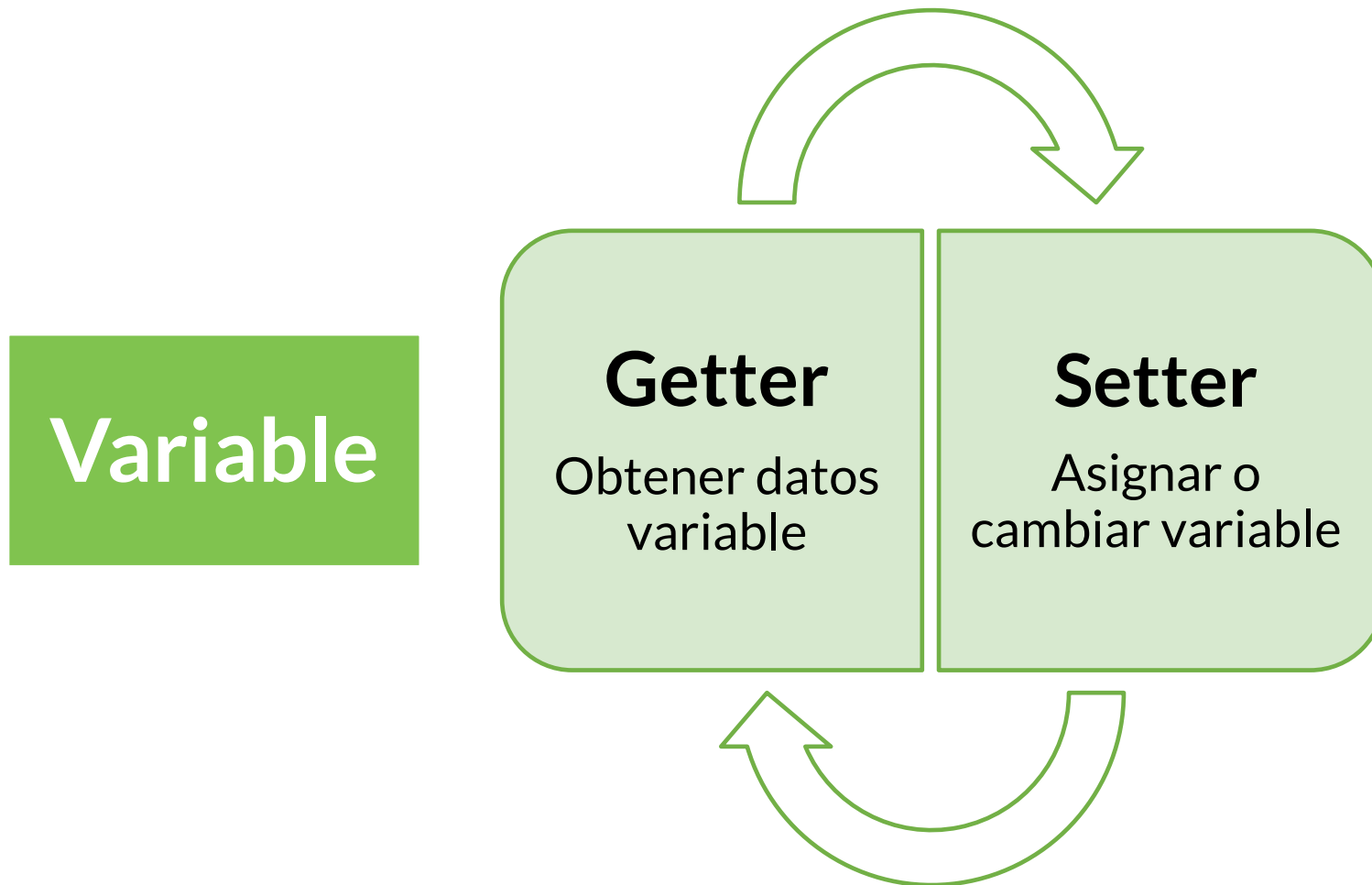
Modificador	Clase	Package	Subclase	Otros
public	✓	✓	✓	✓
protected	✓	✓	✓	●
default	✓	✓	●	●
private	✓	●	●	●

Getters y Setters

Getters y Setters

Leer/Escribir específicamente los valores de las variables miembro.

Getters y Setters



Variables \neq Objetos

Variables \neq Objetos

Variables son entidades elementales (muy sencillas)

- Un número
- Un carácter
- Un valor verdadero falso

Objetos son entidades **complejas** que pueden estar formadas por la **agrupación de muchas variables y métodos.**

Classes Wrapper / Objeto primitivo

Byte

Short

Integer

Long

Float

Double

Character

Boolean

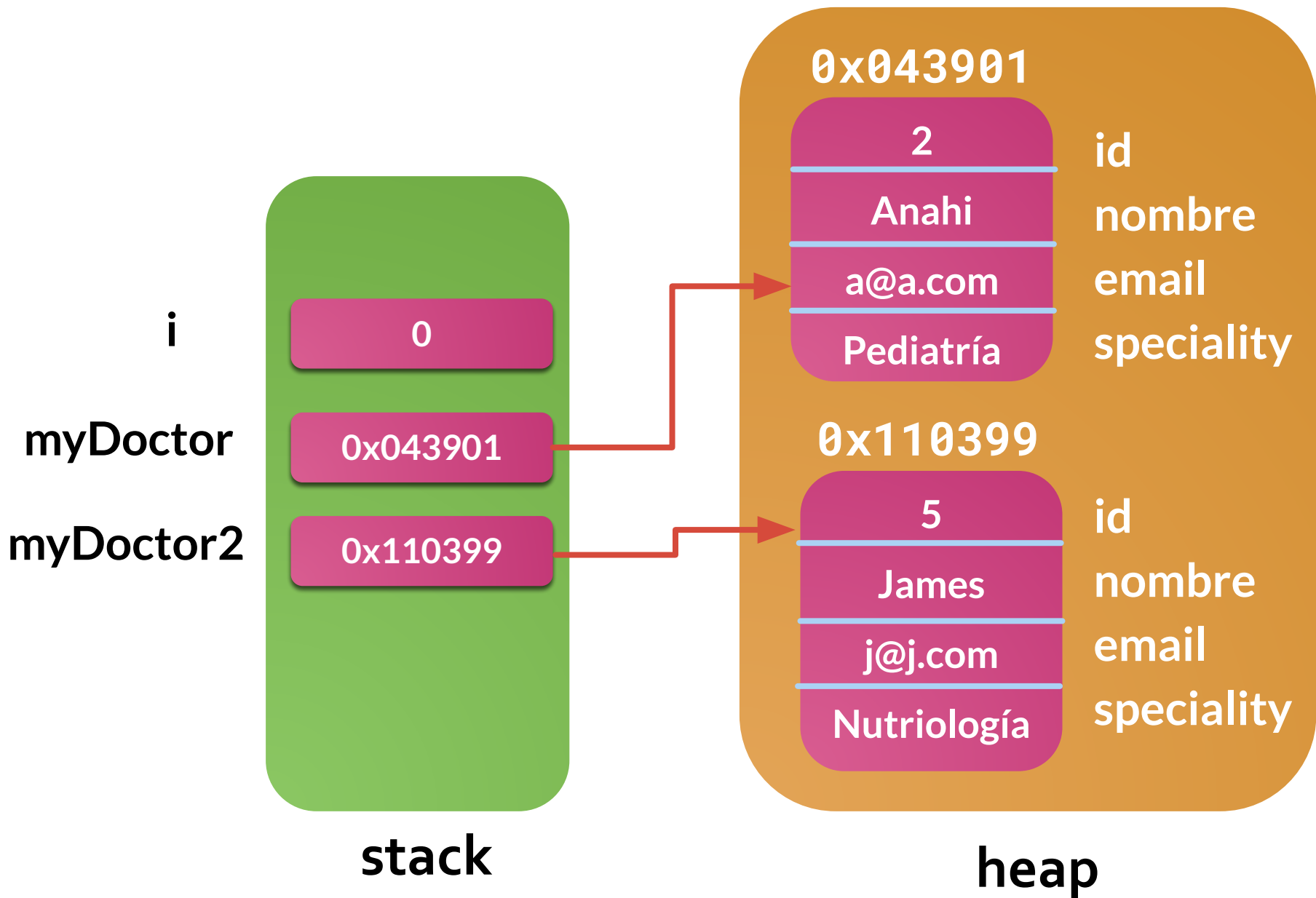
String

Variables \neq Objetos

```
int i = 0;
```

```
Doctor myDoctor = new Doctor();
```

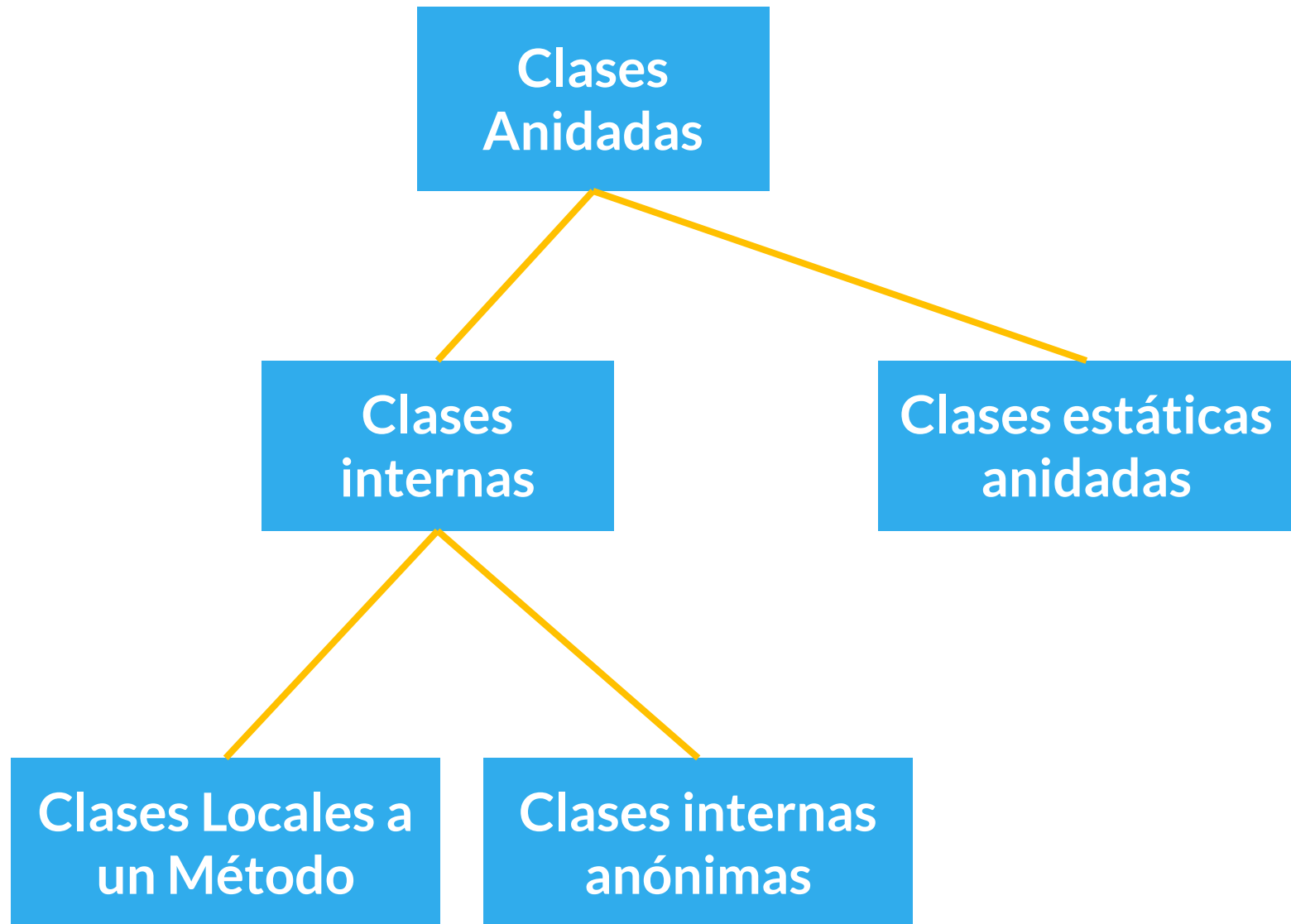
```
Doctor myDoctor2 = new Doctor();
```



Clases Anidadas


```
class ClaseExterior {  
    ...  
    class ClaseAnidada {  
        ...  
    }  
}
```

CLASES ANIDADAS



```
class ClaseExterior {  
    static class ClaseEstaticaAnidada {  
    }  
  
    class ClaseInterna {  
    }  
}
```

CLASES ANIDADAS

Clases anidadas

`static`



Estáticas



No Estáticas

Clases estáticas

No se necesitan crear instancias para llamarlas.

Solo se pueden llamar a los métodos estáticos.

Clases anidadas

Anidadas pueden llamar a cualquier tipo de elemento o método

Clases anidadas

Clases Helper

Agrupadas por lógica

Encapsulación

```
public class Enclosing {  
    private static int x = 1;  
  
    public static class StaticNested {  
        private void run() {  
            //implementación  
        }  
    }  
}  
  
public class Main {  
    public static void main(String[] args){  
        Enclosing.StaticNested nested = new Enclosing.StaticNested();  
        nested.run();  
    }  
}
```

CLASES ESTÁTICAS


```
public class Enclosing {  
    private static int x = 1;  
  
    public static class StaticNested {  
        private void run() {  
            //implementación  
        }  
    }  
}  
  
public class Main {  
    public static void main(String[] args){  
        Enclosing.StaticNested nested = new Enclosing.StaticNested();  
        nested.run();  
    }  
}
```

CLASES ESTÁTICAS

Clases Internas y Locales a un método

**Clases
Anidadas**

```
graph TD; A[Clases Anidadas] --> B[Clases internas]; A --> C[Clases estáticas anidadas]; B --> D[Clases Locales a un Método]; B --> E[Clases internas anónimas]; subgraph RedBox [ ]; D; E; end
```

**Clases
internas**

**Clases estáticas
anidadas**

**Clases Locales a
un Método**

**Clases internas
anónimas**

```
public class Outer {  
    public class Inner {  
  
    }  
  
}
```

```
public class Main {  
    public static void main(String[] args){  
        Outer outer = new Outer();  
        Outer.Inner inner = outer.new Inner();  
    }  
}
```

Classes Internas

```
public class Outer {  
    public class Inner {  
  
    }  
  
}
```

```
public class Main {  
    public static void main(String[] args){  
        Outer outer = new Outer();  
        Outer.Inner inner = outer.new Inner();  
    }  
}
```

Clases Internas

```
public class Enclosing {  
    void run() {  
        class Local {  
            void run() {  
  
            }  
        }  
  
        Local local = new Local();  
        local.run();  
    }  
}
```

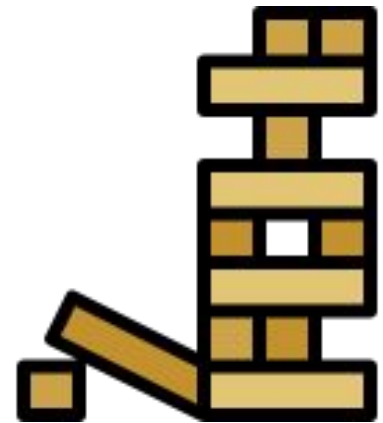
```
public class Main {  
    public static void main(String[] args){  
        Enclosing enclosing = new Enclosing();  
        enclosing.run();  
    }  
}
```

Clases Locales a un Método

Don't repeat yourself

DRY: *Don't repeat yourself*

- Promueve la reducción de duplicación en programación
- Las piezas de información **nunca deben duplicarse.**
- **Incrementa la dificultad** en los cambios y evolución



Reutilización



Herencia

Herencia

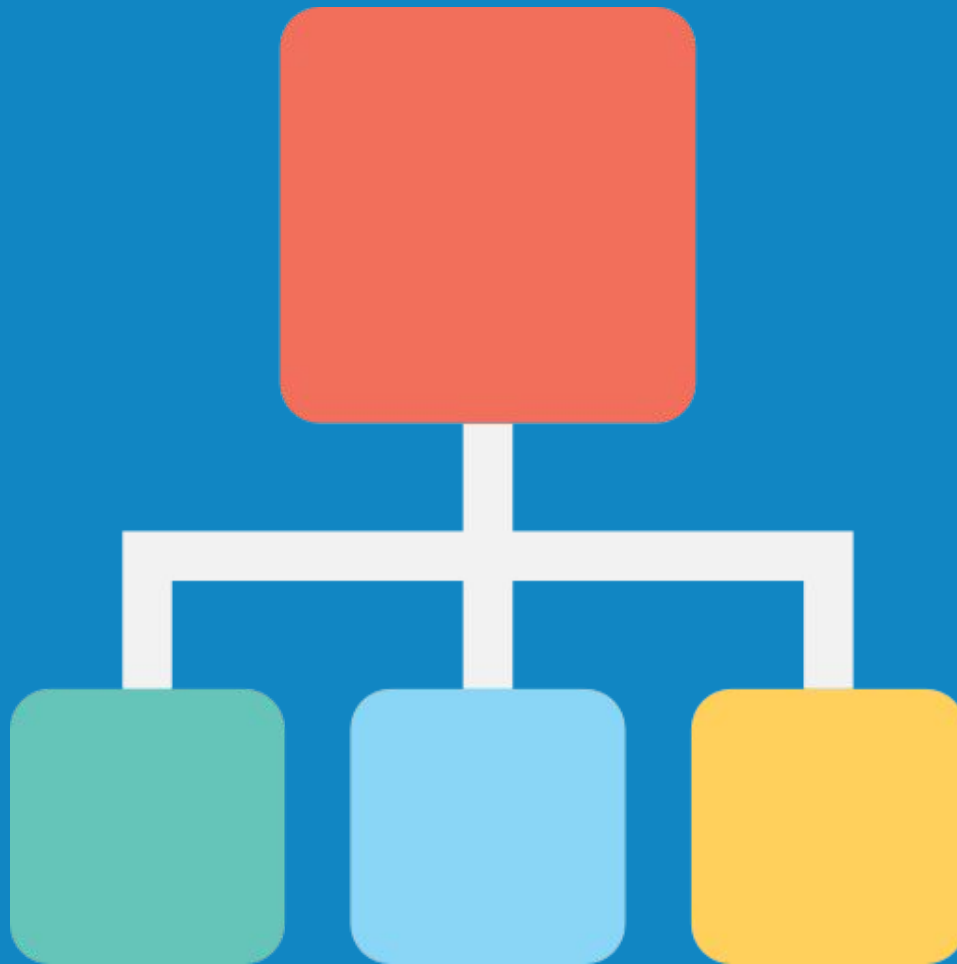
crearemos nuevas
clases a partir de otras



Herencia

Se establece una
relación **padre e hijo**





Superclass

Subclass

Herencia

```
public class User {
```

Súper Clase

```
}
```

```
public class Patient extends User {
```

```
}
```

Subclase

super y this

super

Indica que una variable o un método es de la clase Padre (superclase)

this

Permite especificar que la variable que está señalando (this.nombreVariable) es de la misma clase en la que se usa.

Sobreescritura

Sobreescritura

Cuando una clase hereda de otra y en esta **clase hija se redefine un método** con una implementación distinta a la de la clase padre

Sobreescritura

Los métodos marcados como `final` o `static` no se pueden sobrescribir.

Sobreescritura de Constructores

Un constructor en una subclase usando los miembros heredados de la superclase **con argumentos diferentes.**

Polimorfismo

Polimorfismo

Posibilidad de sobrescribir un método con **comportamientos diferentes**.

Interfaces

Interfaces

Es un tipo de referencia similar a una clase que podría contener solo **constantes y definiciones de métodos.**

Interfaces

Se establece la forma de una clase (nombres de métodos, listas de argumentos y tipos de retorno, pero **no bloques de código**).

Interfaces

```
public interface ISchedulable {  
    schedule(Date date, String Time);  
}
```

```
public class AppointmentDoctor  
implements ISchedulable {  
  
}
```


Clases Abstractas

Polimorfismo

Herencia Clases

Métodos sobreescritos

Muchas formas

Polimorfismo

Implementación Interfaces

Métodos sobreescritos

Muchas formas

Interfaces

A veces no necesitamos
implementar todos los métodos

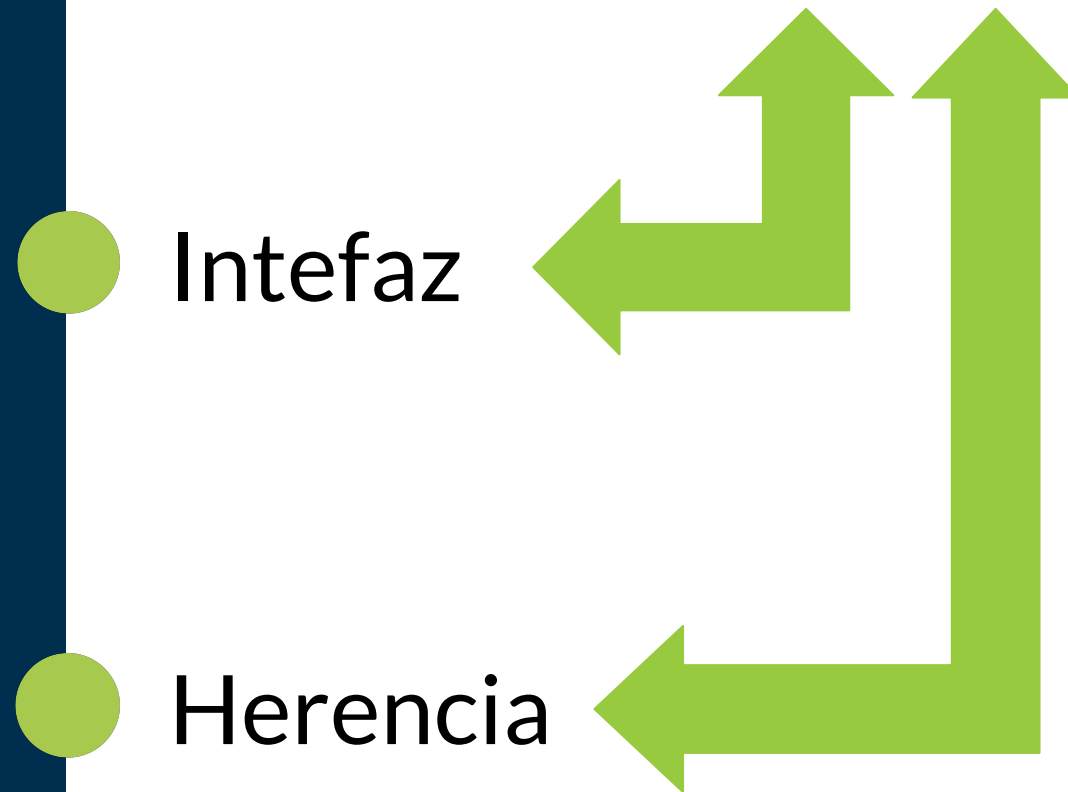
Herencia

Las clases podrían no necesitar heredar la implementación de un método

Herencia

A veces no necesitamos crear instancias de una clase padre, ya que es muy genérica

Clase Abstracta



Clase Abstracta

No implementaremos todos los métodos

No crearemos instancias


```
public abstract class Figura {
```

CLASES ABSTRACTAS

```
class Triangulo extends Figura {  
  
  
  
  
  
  
  
  
  
}
```

CLASES ABSTRACTAS

Métodos Abstractos

```
public abstract class Figura {  
    ····abstract void dibujate();  
}
```

CLASES ABSTRACTAS

```
class Triangulo extends Figura {  
  
    void dibujate(){  
        //Dibujar triangulo  
    }  
  
}
```

CLASES ABSTRACTAS

```
class Cuadrado extends Figura {  
  
    void dibujate(){  
        //Dibujar Cuadrado  
    }  
  
}
```

CLASES ABSTRACTAS

```
abstract class Triangulo extends  
Figura {  
  
    abstract void dibujate();  
  
}
```

CLASES ABSTRACTAS


```
class TrianguloIsosceles extends  
Triangulo {  
  
    void dibujate(){  
        //Dibujar triangulo isocceles  
    }  
  
}
```

CLASES ABSTRACTAS


```
1 | abstract class SimpleAbstractClass {  
2 |     abstract void run();  
3 | }
```

```
public class AnonymousInnerTest {  
  
    @Test  
    public void whenRunAnonymousClass_thenCorrect() {  
        SimpleAbstractClass simpleAbstractClass = new SimpleAbstractClass() {  
            void run() {  
                // method implementation  
            }  
        };  
        simpleAbstractClass.run();  
    }  
}
```

Clases Anónimas

Interfaces en Java 8 y 9

Interfaces

Métodos Abstractos
Campos constantes

Interfaces

Tipo de referencia

Polimorfismo similar Clases

Abstractas

Java 8 y 9

Java 8

default

Java 9

private

Interfaces

Ahora podemos tener
implementación en métodos

Modificador	Clase	Package	Subclase	Otros
public	✓	✓	✓	✓
protected	✓	✓	✓	<input type="radio"/>
default	✓	✓	<input type="radio"/>	<input type="radio"/>
private	✓	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

```
1 public interface MyInterface {  
2     default void defaultMethod() {  
3         privateMethod("Hello from the default method!");  
4     }  
5     private void privateMethod(final String string) {  
6         System.out.println(string);  
7     }  
8     void normalMethod();  
9 }
```

default y private Methods

DAO

Data Access Object

DAO - Data Access Object

Patrón de diseño

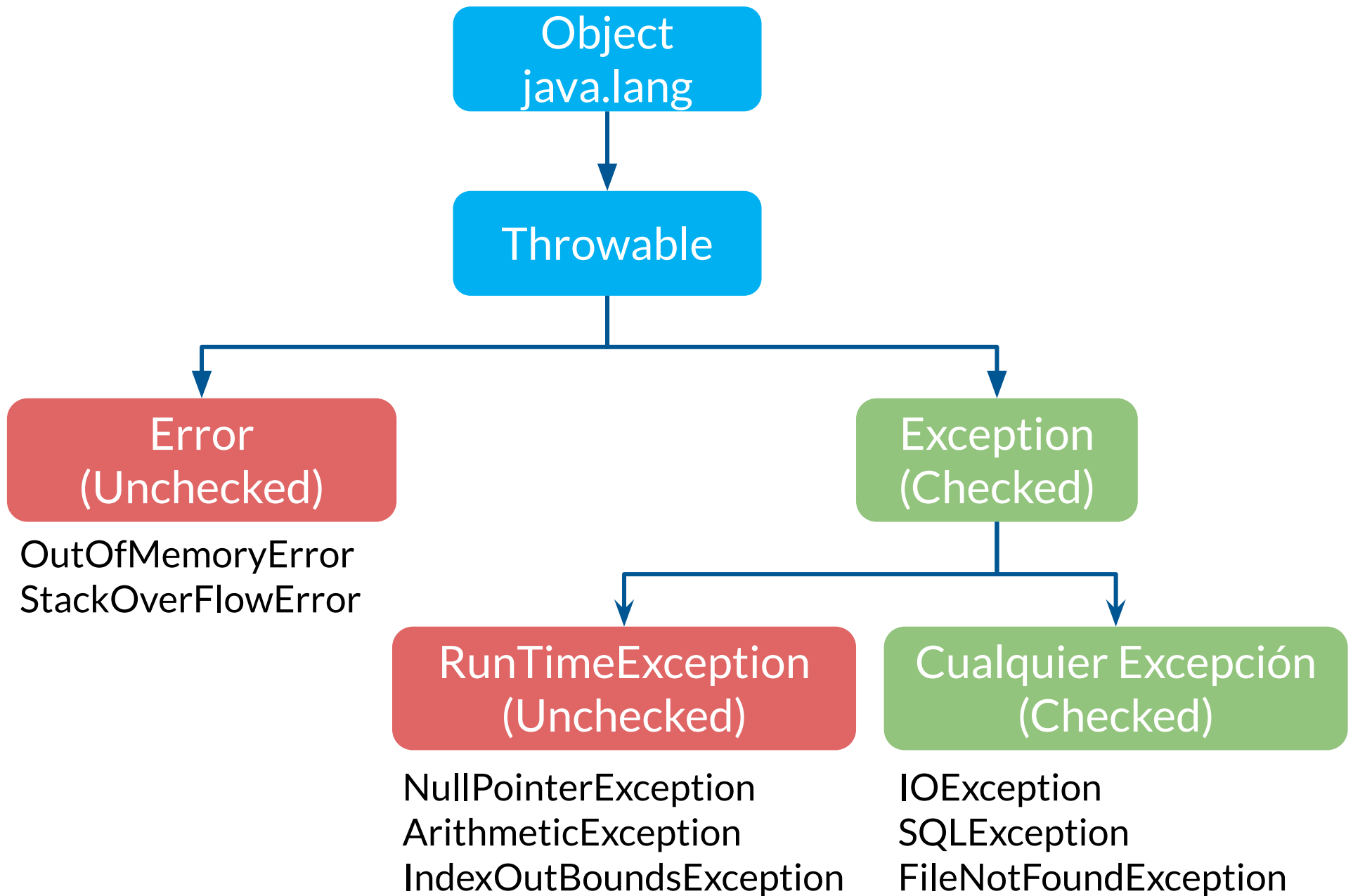
Métodos CRUD

(Create, Read, Update y Delete).

Excepciones

Excepciones

Manejar Excepciones significa que añadirás un bloque de código para manejar un error.



Try-catch-finally

```
try {
```

```
    ...
```

```
} catch (ExceptionType e) {
```

```
    ...
```

```
} catch (ExceptionType e) {
```

```
    ...
```

```
}
```

```
finally {
```

```
    ...
```

```
}
```


Try-with-resources

```
BufferedReader reader =  
new BufferedReader(new InputStreamReader(System.in));  
  
try (BufferedReader r1 = reader) {  
    //sentencias  
  
} catch (Exception e){  
    //sentencias  
  
}
```

```
BufferedReader reader =  
new BufferedReader(new InputStreamReader(System.in));  
  
try (BufferedReader r1 = reader) {  
    //sentencias  
  
} catch (Exception e){  
    //sentencias  
  
}
```

```
BufferedReader reader =
```

```
new BufferedReader(new InputStreamReader(System.in));
```

```
try (reader) {  
    //sentencias
```

```
} catch (Exception e){  
    //sentencias
```

```
}
```

Try-with-resources

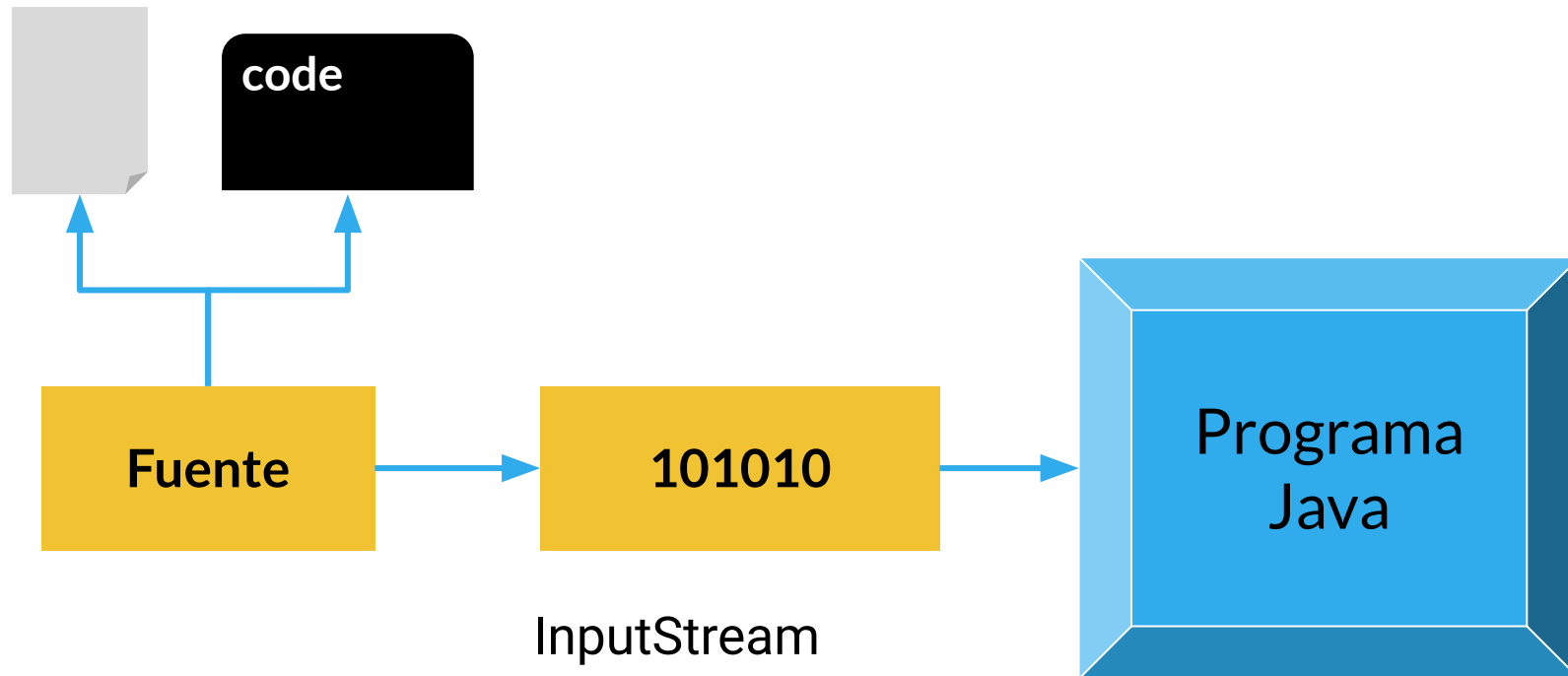
Cerrar Recursos


```
try (Connection connection = connectToDB()) {  
  
} catch (SQLException e) {  
    // TODO Auto-generated catch block  
    e.printStackTrace();  
}
```

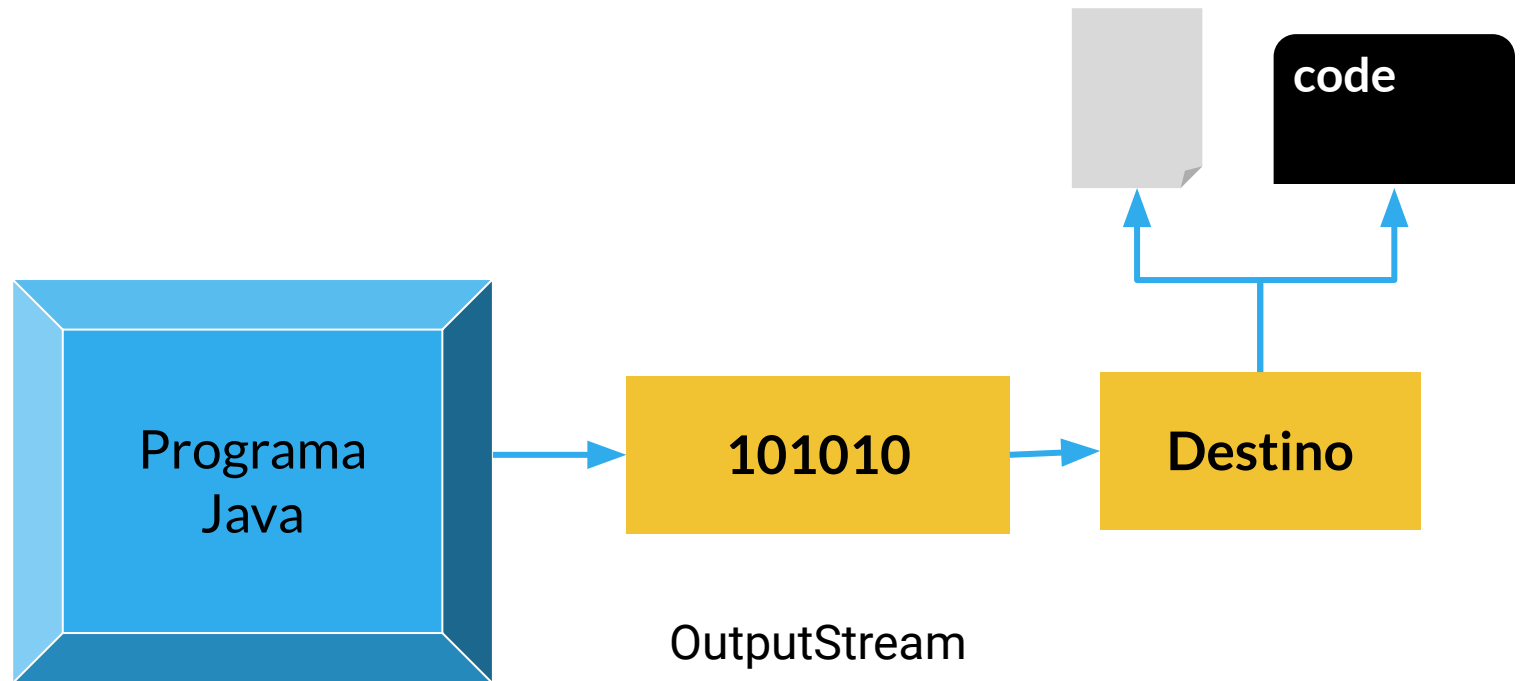
Cerrar recursos

Java I/O

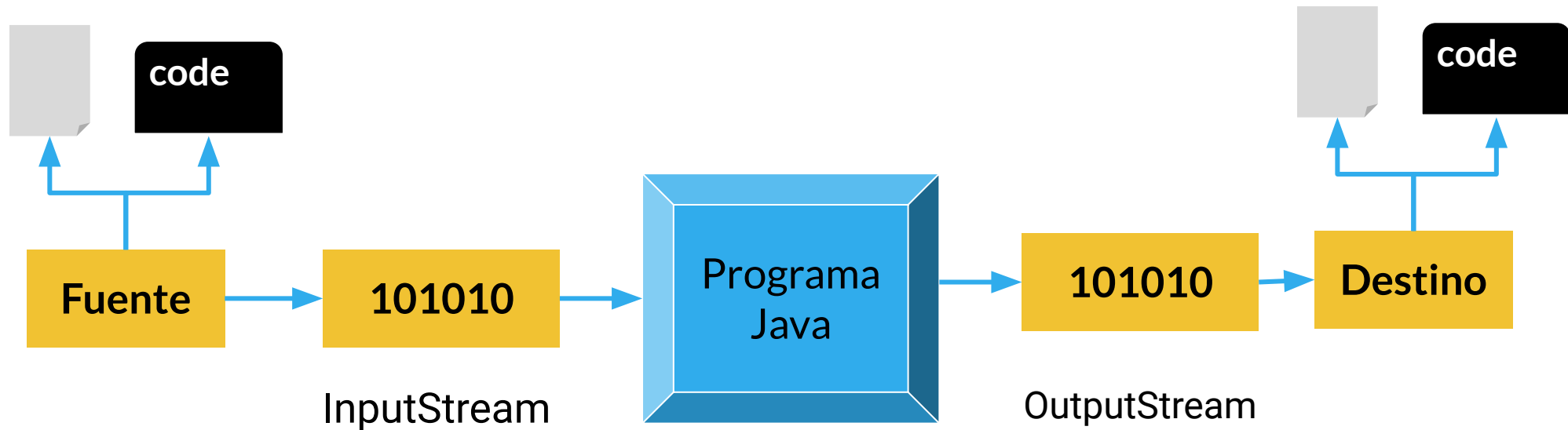
Java I/O



Java I/O



Java I/O



JDBC

JDBC

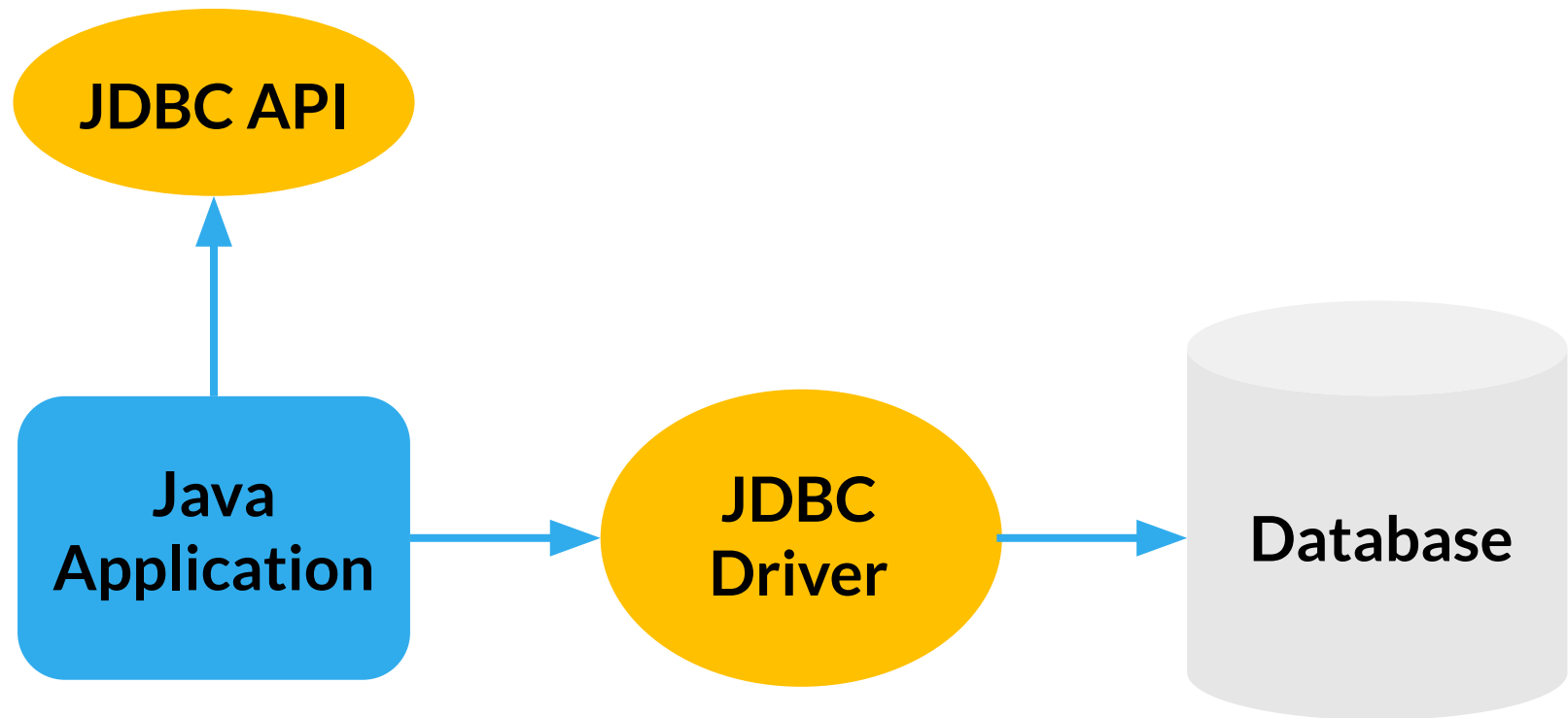
Java DataBase
Connectivity

JDBC

Es un API compuesta por
varias clases

Operaciones a base de datos

JDBC



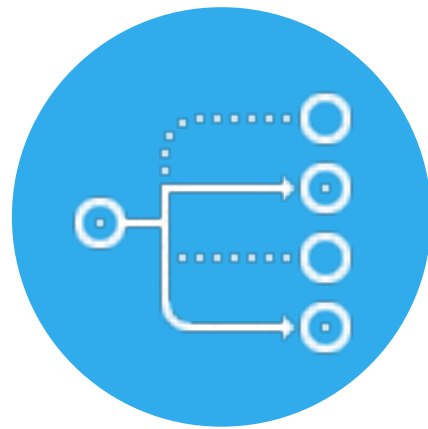
Componentes



DriverManager



Connection



Statement PreparedStatement



ResultSet

Modularización Java 9

Modularización

Proyecto Jigsaw 5años~

Modularización

Encapsulación

Modularización

Encapsulación

Detalles ocultos que provocan
una interfaz más linda

Modularización

Capas de Software

Modularización

- Agrupación de código y recursos como los **JARs**.
- **Un descriptor** que restringe el acceso a sus paquetes, y describe sus dependencias.