TEMA 1: ENCAPSULACIÓN

TIPOS DE DATOS

- Un PROGRAMA se puede entender como una **serie de instrucciones** que operan sobre un conjunto de datos de ENTRADA y generan otro conjunto de datos de SALIDA.
- Todos los lenguajes de programación utilizados en la actualidad tienen instrucciones de **selección** y **repetición**, de forma que en teoría es posible crear **cualquier** programa.
- Los **tipos de datos** disponibles en un lenguaje de programación **limitan** enormemente las posibilidades a la hora de crear programas.
  - En lenguajes de programación como C se pueden definir nuevos tipos de datos a través de **estructuras**, pero esto pone en compromiso la **integridad** de los datos, pues no es posible restringir directamente el valor que toman las variables en las distintas funciones del programa.

ENCAPSULACIÓN

- La ENCAPSULACIÓN es una de las principales características de la POO: asegura la **integridad** de los datos, limitando las porciones de código del programa que pueden acceder a determinados datos (idealmente no existe ninguna parte del código que pueda acceder a todos los datos del programa).
  - Hace uso del PRINCIPIO DE OCULTACIÓN → sólo un trozo de código puede "ver" un conjunto de datos dado.

CLASES

- En POO los tipos de datos se conceptualizan como CLASES y sus valores concretos son los OBJETOS.

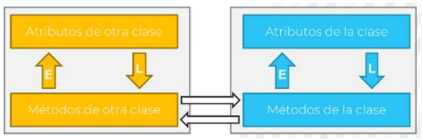
❖ **En POO todos los datos de un programa son objetos, aunque en java no sea así.**
❖ **Todos los objetos son del tipo de dato clase.**

CLASES EN JAVA

- En Java se distinguen dos especies de **tipos de datos**:
  - TIPOS DE DATOS PRIMITIVOS → están vinculados a la representación de los datos en el computador (son los mismos tipos de datos que los del lenguaje C).
  - CLASES → están vinculadas a las entidades de alto nivel del programa y no se pueden representar de forma directa en el computador.

ENCAPSULACIÓN EN LAS CLASES

- En general, las clases contienen dos **tipos de código**:
  - VARIABLES o ATRIBUTOS → son las características que definen la entidad representada por la clase.
  - FUNCIONES o MÉTODOS → acceden a los atributos para permitir su lectura, escritura y demás operaciones.



ESTRUCTURA DE LAS CLASES

```
public [final | abstract] class <nombre_clase>
{
  // Atributos
  <tipo_acceso> <tipo> <nombre_atrib>;
  <tipo_acceso> <static | final> <tipo> <nombre_atrib> = <valor>;

  // Constructores
  public <nombre_clase> (<tipo> <nombre> *) { <código> }

  // Métodos de lectura (getters)
  public <tipo_atributo> get<nombre_atributo> ( ) {
     return <nombre_atributo>;
  }

  // Métodos de escritura (setters)
  public void set<nombre_atributo> (<tipo_atributo> <nombre> * ) { <código> }

  // Otros métodos
  <tipo_acceso> <static | final> <tipo> <nombre_método> (<tipo> <nombre> *) {
     <código>
  }
}
```

Declaración de los atributos (variables) de la entidad

Reserva memoria para los atributos de la entidad

Funciones que obtienen los valores de los atributos de la entidad

Funciones que escriben los valores de los atributos

Funciones que operan sobre los atributos para obtener las funcionalidades del programa

Modificador	Clase	Paquete	Subclase	Resto
public	✓	✓	✓	✓
protected	✓	✓	✓	×
☐	✓	✓	×	×
private	✓	×	×	×

TIPO_ACCESO PÚBLICO (public)

- **Cualquier** método de cualquier clase del programa puede acceder a los atributos y métodos.

TIPO_ACCESO PRIVADO (private)

- Sólo métodos de la **misma clase** pueden acceder a los atributos y métodos.

TIPO_ACCESO A PAQUETE (☐)

- Pueden acceder métodos de las clases que pertenezcan al **mismo paquete**.

TIPO_ACCESO PROTEGIDO (protected)

- Pueden acceder métodos de las clases que pertenezcan al **mismo paquete** y también las **subclases** de la clase.

👍 Los **atributos** de una clase deben ser siempre **privados** (para que no se puedan modificar de manera incorrecta) y los **métodos**, **públicos** (para que se pueda acceder a los atributos).

👍 Se pueden usar **métodos privados** si estos existen para facilitar la codificación de otros métodos.

GETTERS Y SETTERS

- Los GETTERS son métodos que **devuelven el valor** que tienen los atributos en el momento de la invocación.
  - Sólo hay un **único** getter para cada atributo:
    $get < Nombre\_atributo >$ (OJO a la mayúscula).
  - Normalmente contienen solo el código asociado a la **devolución** del valor del atributo.
- Los SETTERS son métodos que **escriben el valor** de los atributos.
  - Sólo hay un **único** setter para cata atributo:
    $set < Nombre\_atributo >$ (OJO a la mayúscula).
  - Normalmente contiene código sobre las **condiciones** que debe cumplir el argumento para que sea un valor correcto del atributo.

👍 Los setters siempre deben incluir una condición para **evitar la escritura de null** como valor de los atributos.

👍 Normalmente, **todos los atributos deben tener setters y getters** salvo que solamente sean usados como parte de la programación de los métodos de la clase.

```
public class Continente {
   private int numeroEjercitos;
   private String nombre;
   private ArrayList<Pais> paises;
   private String color;

   public String getNombre() {
       return nombre;
   }

   public void setNombre(String nombreContinente) {
       if(nombre.equals("África") ||
          nombre.equals("América del Norte") ||
          nombre.equals("América del Sur") ||
          nombre.equals("Asia") ||
          nombre.equals("Australia") ||
          nombre.equals("Europa"))
           this.nombre= nombreContinente;
       else
           System.out.println("No existe el continente");
   }
```

CONSTRUCTORES

Al declarar un **atributo** de una clase, si este es un **tipo de dato primitivo**, se reservará memoria para él y se le asignará un valor inicial.
Sin embargo, si es una **clase**, no se reservará memoria (ya que se desconoce cuánto ocupará) y se le asignará el valor inicial ***null***.

- Si se intentase usar el atributo en este estado, se generaría un error de ejecución $NullPointerException$. Se debe reservar memoria para él invocando a los constructores con el operador new.

Al declarar una instancia de una **clase**, no se reservará memoria para dicho objeto tampoco. Si no se inicializa con el operador new, provocará un error de compilación

```
public class RiskETSE {
    public static void main(String[] args) {
        // Continente() asigna valores iniciales por defecto a los atributos
        Continente australia= new Continente();
        if(!australia.getNombre().equals("Australia"))
            australia.setNombre("Australia");
        System.out.println("Nombre del continente: " + australia.getNombre());
```

un error

```
public class RiskETSE {
    public static void main(String[] args) {
        Continente australia;
        if(!australia.getNombre().equals("Australia"))
            australia.setNombre("Australia");
        System.out.println("Nombre del continente: " + australia.getNombre());
```

- Un CONSTRUCTOR es un método que **no devuelve** ningún tipo de dato. Se invoca para { **reservar memoria** para un objeto y sus atributos. / **asignar valores** iniciales a sus atributos.
  - Su **nombre** coincide con el de la clase a la que pertenece.
  - Se **invoca** una **única** vez para cada objeto → cuando se reserva memoria para un objeto, su nombre es una referencia a la posición de memoria donde se almacenan los datos. Si se volviese a invocar al constructor, la referencia apuntaría a una nueva posición y se perdería la anterior.

```
public Continente(ArrayList<Pais> paises, String nombre, String color) {
    for(Pais pais : paises) pais.setContinente(this);
    this.paises = paises;
    this.nombre = nombre;
    this.color= color;
```

CONSTRUCTOR POR DEFECTO

- Java proporciona automáticamente este constructor cuando no se especifica ningún otro.
- No tiene **ningún argumento**.
- Inicializa los atributos a sus **valores por defecto**, es decir, todos los atributos de tipo clase tomarán el valor ***null***.

```
public Continente() {
    // No tiene ningún tipo de código
```

> Se debe evitar el constructor por defecto.
> Los constructores creados deben tener como **argumentos** los valores de los **atributos** que es **obligatorio** inicializar para crear un objeto.

MÉTODOS FUNCIONALES

- Los MÉTODOS FUNCIONALES son aquellos que utilizan los atributos de una clase para realizar las operaciones que implementan la funcionalidad de esta.
  - Se invocan desde los objetos de la clase así: $objeto.método$.

MÉTODOS SOBRECARGADOS

- Decimos que un método funcional (o un constructor) de una clase está SOBRECARGADO si tiene varias implementaciones.
- Todas las implementaciones deben tener el **mismo nombre**.
- Todas las implementaciones deben tener el **mismo objetivo**.
- Los **argumentos** deben ser **distintos** (si no, el compilador no podría distinguir cuál se está invocando).

> Se **desaconseja** el uso de **condiciones** sobre los argumentos de un método para **seleccionar** qué **código** se debe ejecutar en cada caso. Es mejor crear distintas implementaciones.

MÉTODO toSTRING

```
public String toString()
{
  return getClass().getName() + '@' + Integer.toHexString(hashCode());
}
```

⇨

```
String toString= """
        {
            nombre: %s,
            frontera: [
                        %s
                      ],
            paises: [
                        %s
                    ]
        }""".formatted(nombre, paisesFrontera, paisesContinente);
```

- $toString$ es un método que devuelve la **representación en texto** de un objeto de una clase.
- Las clases heredan este método de la **clase Object**, la superior en la jerarquía de clases. Esta implementación es muy **genérica** y no aporta información sobre el **contenido del objeto**
  - Por tanto, se recomienda **reimplementarlo** usando el operador $@Override$ de manera que devuelva una representación en texto que resuma adecuadamente el contenido del objeto.

REGISTROS

- Un REGISTRO es una clase cuyas instancias son **inmutables**, es decir, los valores de los atributos no pueden cambiar en tiempo de ejecución.
- Todos sus atributos son **privados**.
- Todos sus atributos tienen su correspondiente **getter**.
- Los atributos **no** tienen **setters** y no puede existir ningún método que modifique sus valores.
- ▷ Se pueden introducir nuevos **métodos** siempre que no alteren los valores de los atributos.
- ▷ Tienen un **CONSTRUCTOR CANÓNICO** con sus correspondientes argumentos para cada uno de los atributos.
  - Se puede **reimplementar** especificando todos los argumentos o usando una **declaración compacta**.
  - Se pueden introducir **sobrecargas** siempre que en su cuerpo se invoque al constructor canónico del registro.
- ▷ El método $toString$ incluye el **nombre de la clase** y el **nombre de los atributos** con sus correspondientes **valores**.

```
public record Continente(ArrayList<Pais> paises,
                         String nombre,
                         String color) {
    // Reimplementar constructor canónico: forma compacta
    public Continente {
        Objects.requireNonNull(paises);
        Objects.requireNonNull(nombre);
        Objects.requireNonNull(color);
    }

    // Nuevo constructor (debe invocar al canónico)
    public Continente(ArrayList<Pais> paises, String nombre) {
        this(paises, nombre, color: "Blanco");
    }

    // Método funcional: no puede escribir sobre los atributos
    public boolean esGrande() {
        if(paises.size()>10) return true;
        return false;
```

EJECUCIÓN DE UN PROGRAMA EN JAVA

- Todo programa de Java tiene una **clase principal** en la que se encuentra el método ***main***, que será el punto a partir del cual se inicia la ejecución del programa.
- La clase principal **no debe tener ningún atributo**.

```
public class RiskETSE {
    public static void main(String[] args) {
        new Menu();
    }
}
```