Objetivos:

- Tipos Genéricos
- Excepciones, Declaración y propagación (cláusula *throws*)
- Captura, Manejador de excepciones.
- Incorporar información a excepciones.
- Introducción a las IU. División en capas.

En la práctica usaremos algunas clases definidas en las prácticas anteriores. Es por ello que se recomienda finalizar dichas prácticas antes de comenzar con la presente.

Ejercicio 1.

Una fábrica de juegos de PC desea modelar un sistema de selección aleatoria de jugadores (`RandomSelector`). Dado que la selección aleatoria se usa en muchos y variados juegos (p.e: selección aleatoria de números, de cartas, de colores, de fichas, etc.) Se detalló que el `RandomSelector` debería ser diseñado de tal forma que se pueda usar con cualquier tipo de objetos.

Nota: *El funcionamiento debería ser el siguiente: Una vez creado el* `RandomSelector` *con el tipo de Objeto que va a seleccionar este permanece así. Al* `RandomSelector` *se le pueden agregar* ***N*** *objetos a seleccionar. Cuando se le dice* `selectNext()` *retorna un objeto seleccionado de manera aleatoria.*

Ejercicio 2.

Implementar una clase genérica llamada `Par` que represente un par de elementos, donde cada elemento puede ser de un tipo diferente. La clase debe tener métodos para obtener el primer y segundo elemento del par, así como un método para imprimir la información del par.

Además, crea una aplicación que utilice la clase `Par` para instanciar pares con diferentes tipos de datos, como `Integer` y `String`, y muestre la información de cada par.

```
Par<Integer, String> par1 = new Par<>(42, "Hola, mundo");
```

Ejercicio 3.

Dado el siguiente método:

```
public int dividir (int a, int b){
        return a/b;
}
```

a) Realice pruebas con diferentes valores de a y de b.
b) ¿Qué pasa cuando b es 0?
c) Modifique el método para que ahora cuando b sea 0, el valor de retorno sea 0.
d) ¿Cuál sería la excepción más adecuada para utilizar en este caso?

Nota: en el ejercicio d) debe capturar la excepción con un `try - catch`

Ejercicio 4.

Cree la clase `ColaDeTrabajo` que permite encolar diversos trabajos. O sea, los trabajos a encolar deben implementar la interfaz `Trabajo`. Defina en la clase `ColaDeTrabajo` un método `sacar()` que retorna el próximo trabajo a procesar. Además, agregue en dicha clase los atributos *nombre* y *lista* que representan el nombre de la cola y si está lista o no para retornar trabajos. Tenga presente, que cuando no existan trabajos en la cola o cuando la misma no esté lista se debe lanzar las siguientes excepciones: `NoListaException` y `SinTrabajoEnColaException`

A continuación, se detalla cada una de las excepciones lanzadas por el método `sacar()` de `ColaDeTrabajo`

```
public class NoListaException extends Exception {

	private String nombre;
	private long cantidadTrabajos;

	public NoListaException (String nom, long s) {
		nombre = nom;
		cantidadTrabajos = s;
	}

	@Override
	public String getMessage() {
		return "La Cola de Trabajo: " + nombre + " no está
		disponible. Cantidad de trabajos a procesar : " +
		cantidadTrabajos;
	}
}


public class SinTrabajoEnColaException extends Exception {

	private String nombre;

	 public SinTrabajoEnColaException (String nom) {
		nombre = nom;
 }
```

```
        @Override
        public String getMessage() {
            return "La cola " + nombre + " no tiene trabajos para
            procesar. ";
        }
    }
```

Se pide:

a) Implementar la clase `ColaDeTrabajo` y definir el método `sacar()` en dicha clase.
b) ¿Cómo se lanzan las excepciones anteriores dentro del método?
c) ¿Cómo se capturan las excepciones al llamar al método `sacar()`?

Nota: Utilizar la interfaz `Queue<T>` y `LinkedList<>` para definir la cola:

```
private Queue<Trabajo> lista = new LinkedList<>();
```

Ejercicio 5.

Modifique la clase `RandomSelector` del Ejercicio 1 para lanzar la excepción `IndexOutOfBoundsException` al momento de solicitar un elemento del selector y no disponer de ninguno en su colección para escoger.

Ejercicio 6.

Defina en Java la clase `DataBag`, la cual tiene un número máximo de elementos y permite almacenar **cualquier tipo** de objetos.

a) Implemente en JAVA la clase *DataBag*.
b) Defina el método `add()` para permitir agregar elementos a la bolsa y en el caso de que la misma esté llena dispare la excepción `FullDataBagException`.
c) Defina el método `remove()` para remover elementos de la bolsa y en el caso de que la misma esté vacía se dispare la excepción `EmptyDataBagException`

Nota: Tenga en cuenta que las dos excepciones `FullDataBagException` **y** `EmptyDataBagException` son excepciones chequeadas que deben ser creadas por usted como subclase de `Exception`.

Ejercicio 7.

Dada la clase `PruebaExcepcion`**:**

```
public class PruebaExcepcion {

	public static void main(String st[]){
		PruebaExcepcion t1 = new PruebaExcepcion();
		t1.metodo(5,0);
	}
	public void metodo(int a, int b){
		try {
			int c = a/b;
			System.out.println("Después de la división");
		} catch (ArithmeticException ae) {
			System.out.println("Excepción Aritmética");
		}
		catch (Exception e) {
			System.out.println("Otra Excepción");
		}
		finally {
			System.out.println("Bloque Finally");
		}

		System.out.println("Después del bloque finally");
	}
}
```

a) ¿Qué retorna su ejecución?
b) ¿Qué hace la cláusula `finally`?
c) Proponga ejemplo de casos en donde se podría llegar a usar la cláusula `finally`.

Ejercicio 8.

Use el **Ejercicio 2 del Práctico 1**, modifique las clases que considere para que lancen las excepciones:

- `SuperaLimiteMinimoException`
- `SuperaCantidadExtraccionesException`

Ejercicio 9.

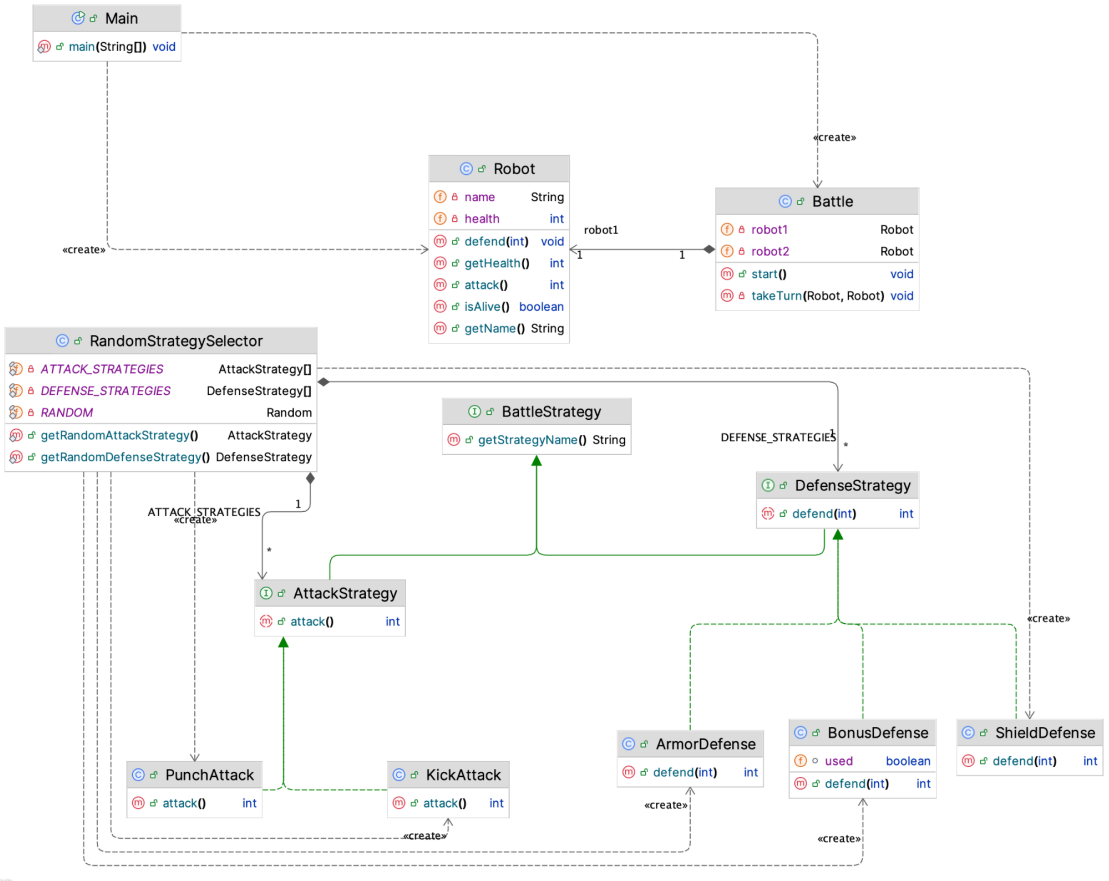
Vamos a implementar una simulación de una **guerra entre dos robots** utilizando el patrón de diseño `Strategy` en Java. Para ello, definiremos varias estrategias de ataque

y defensa, y cada robot tendrá varias estrategias de ataque y defensa. Cada vez que un robot ataca o se defiende, se restará puntos de vida según la estrategia utilizada.
Las estrategias de ataque y defensa se deberán seleccionar de manera random,

El juego se desarrollará por turnos hasta que uno de los robots llegue a 0 puntos de vida.

Clases Necesarias:

1. **Interfaz AttackStrategy**: Define la estrategia de ataque.
2. **Interfaz DefenseStrategy**: Define la estrategia de defensa.
3. **Clases Concretas de cada Estrategia**: Implementaciones específicas de las estrategias de ataque y defensa.
4. **Clase Robot**: Representa a un robot con sus puntos de vida y sus estrategias de ataque y defensa.
5. **Clase Battle**: Gestiona el flujo de la batalla entre dos robots.
6. **Clase RandomStrategySelector**: Se encargará de seleccionar estrategias aleatorias.



Código Existente:

```
// Interfaces que definen las estrategias de ataque y defensa
public interface BattleStrategy {
   default String getStrategyName() {
       return getClass().getSimpleName();
   }
}

public interface DefenseStrategy extends BattleStrategy{
   int defend(int attack);
}

public interface AttackStrategy extends BattleStrategy {
   int attack();
}

// Clase principal que hace luchar a los robots
public class Battle {
   private Robot robot1;
   private Robot robot2;

   public Battle(Robot robot1, Robot robot2) {
       this.robot1 = robot1;
       this.robot2 = robot2;
   }

   public void start() {
       System.out.println("La batalla comienza entre "
            + robot1.getName()
            + " y " + robot2.getName());

      // Mientras los dos tengan vidas seguimos luchando
      while (robot1.isAlive() && robot2.isAlive()) {
           takeTurn(robot1, robot2);
           if (robot2.isAlive()) {
               takeTurn(robot2, robot1);
           }
       }

       if (robot1.isAlive()) {
           System.out.println(robot1.getName()
                   + " gana la batalla con " + robot1.getHealth()
                   + " puntos de vida restantes.");
       } else {
           System.out.println(robot2.getName()
                   + " gana la batalla con " + robot2.getHealth()
                   + " puntos de vida restantes.");
       }
   }
```

```
    private void takeTurn(Robot attacker, Robot defender) {
        int damage = attacker.attack();
        defender.defend(damage);
        System.out.println(attacker.getName()
                    + " ataca a "
                    + defender.getName()
                    + " causando "
                    + damage
                    + " puntos de daño.");

        System.out.println(defender.getName()
                    + " tiene " + defender.getHealth()
                    + " puntos de vida restantes.");
    }
}

public class Robot {
    private String name;
    private int health;

    public Robot(String name) {
        this.name = name;
        this.health = 1000;
    }

    public int attack() {
        AttackStrategy attackStrategy =
RandomStrategySelector.getRandomAttackStrategy();
        System.out.println("Using Attack-> " +
attackStrategy.getStrategyName());
        return attackStrategy.attack();
    }

    public void defend(int damage) {
        DefenseStrategy defenseStrategy =
RandomStrategySelector.getRandomDefenseStrategy();
        System.out.println("Using Defense-> " +
defenseStrategy.getStrategyName());
        int damageReduced = defenseStrategy.defend(damage);
        health -= Math.max(damageReduced, 0);
    }

    public boolean isAlive() {
        return health > 0;
    }

    public int getHealth() {
        return health;
    }

    public String getName() {
        return name;
    }}
```

```
public class Main {
   public static void main(String[] args) {

        Robot robot1 = new Robot("Robot1");
        Robot robot2 = new Robot("Robot2");

        Battle battle = new Battle(robot1, robot2);
        battle.start();
   }
}

public class RandomStrategySelector {

      private static final AttackStrategy[] ATTACK_STRATEGIES = {
             new PunchAttack(),
             new KickAttack()
      };

      private static final DefenseStrategy[] DEFENSE_STRATEGIES = {
            new ShieldDefense(),
            new ArmorDefense(),
            new BonusDefense()

      }

      // Completar el código faltante

}
```

Se pide que:

1. Complete todo el código faltante
2. Complete las estrategias de **Ataque** donde:
    - *PunchAttack: Resta 10 puntos de vida.*
    - *KickAttack: resta 40 puntos de vida*
    - *LaserAttack: Resta 90 puntos de vida.*
    - *MissileAttack: Resta 120 puntos de vida.*
    - *HammerAttack: Resta 80 puntos de vida.*
    - *FlameThrowerAttack: Resta 100 puntos de vida.*
    - *ElectricShockAttack: Resta 110 puntos de vida.*
3. Complete las estrategias de **Defensa** donde:
    - *ShieldDefense: Reduce el daño recibido en 30 puntos*
    - *ArmorDefense: Reduce el daño recibido en 50 puntos*
    - *BonusDefense: Reduce el daño recibido en su totalidad*
    - *EvadeDefense: Reduce 40 puntos de daño.*
    - *AbsorbDefense: Reduce 60 puntos de daño.*
    - *CounterDefense: Reduce 50 puntos de daño*
4. ¿Qué cambios hay que hacer para agregar múltiples robots?

Interfaces de Usuario

*La **interfaz gráfica de usuario**, conocida también como GUI (del inglés graphical user interface) es un programa informático que actúa de interfaz de usuario, utilizando un conjunto de imágenes y objetos gráficos para representar la información y acciones disponibles en la interfaz. Su principal uso, consiste en proporcionar un entorno visual sencillo para permitir la comunicación con el sistema operativo de una máquina o computador.*

Ejercicio 10. **–** *Nuestro primer programa Swing* **–**

En este ejercicio se pide crear nuestra primera interfaz gráfica en Swing, haciendo el habitual *Hola, Mundo*. El resultado de ejecutar esta aplicación es la siguiente ventana:

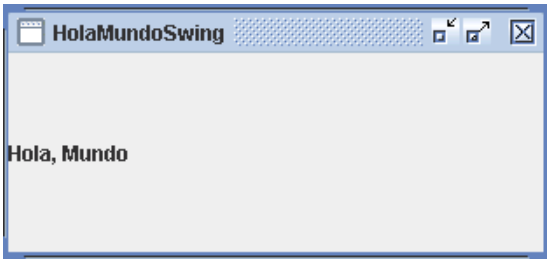


Figura 1