

# Laboratorio 6a: Tipos Abstractos de Datos

## Hoja de problemas

### Programación 2

Ángel Herranz


aherranz@fi.upm.es


Universidad Politécnica de Madrid


2023-2024

Este laboratorio va a girar alrededor del mundo de los **tipos abstractos de datos** y de las **pruebas unitarias** y el desarrollo basado en pruebas (*TDD: test driven development*).

**Nota:** Observarás que de vez en cuando aparecen algunos iconos. Aquí tienes un pequeño diccionario:

 *leer, convenciones*

 *programar*

 *pensar, observar*


---


**Se pide:** Tendrás que entregar cinco ficheros:


- TestSim.java: Pruebas unitarias de la clase Sim.
- Sim.java: Implementación de la clase Sim.
- TestListSim.java: Pruebas unitarias de la clase ListaSim.
- ListSim.java: Implementación de la clase Sim.


Dichos programa deberá poder compilarse y ejecutarse desde la línea de comandos usando estos comandos:

```
C:\Users\UPM\lab02> javac TestSim.java
C:\Users\UPM\lab02> java -ea TestSim.java
C:\Users\UPM\lab02> javac TestListSim.java
C:\Users\UPM\lab02> java -ea TestListSim.java
```

 **Ejercicio 1.** Ten a mano las transparencias de la sesión 11 de tipos abstractos de datos. Antes de empezar el laboratorio el profesor hará una mini-presentación de las mismas.

 **Ejercicio 2.** Termina el test para los Sims (TestSim.java) enriqueciendo aún más los tests elaborados durante la clase. Recuerda, **aún no hemos empezado con la implementación de Sim.java** así que tendrás que empezar con una versión *vacía*, es decir, una versión que simplemente compile pero que no haga nada de momento.

 **Ejercicio 3.** Busca en internet un tutorial o un manual de la herramienta javadoc. Intenta entender al menos las opciones `-d directorio`, `-author` o `-use`.

 **Ejercicio 4.** Ya puedes empezar a programar Sim.java. Recuerda que la clase Sim es un **tipo abstracto de datos**:


- tiene un **nombre**,
- una serie de **operaciones** públicas (API<sup>1</sup>), y
- cada operación tiene una **semántica**.

Como puedes ver, nada se menciona de la implementación, por que en un tipo abstracto de datos nos **abstraemos** de la implementación, nos abstraemos de las estructuras de datos (atributos) usados para representar los datos.

Antes de comenzar recordemos la semántica de las operaciones:

- Sim y nombre: Cada Sim tiene un nombre que se le da cuando se crea.
- haciendo: La operación haciendo dice qué actividad está haciendo el Sim.
- simular: Un Sim no cambia lo que está haciendo hasta que se invoca el método simular.
- simular: La simulación consiste en seguir haciendo lo que está haciendo el Sim durante las horas indicadas y luego cambiar de actividad.
- simular: Un Sim, cuando está durmiendo, tiene que hacerlo durante al menos 8 horas.
- hacerAmigo y amigo: Se puede hacer que un Sim (*a*) tenga a otro Sim (*b*) como su *más mejor amigo* (pero eso no significa que *a* sea el más mejor amigo de *b*).
- estadística: Dada una actividad dice cuántas horas ha dedicado el Sim a dicha actividad.

Este ejercicio consiste en **javadoc** todo el código de Sim.java.

 **Ejercicio 5.** Ahora sí. Ahora ya llegó el momento: **implementa la clase Sim**. A medida que vayas completando la clase tienes que ir compilando y ejecutando el programa de test para que puedas tener cierta confianza en que lo estás haciendo bien.

 **Ejercicio 6.** Nunca olvides la siguiente frase del gran Edsger W. Dijkstra:

---

<sup>1</sup>Application Public Interface

*Program testing can be used to show the presence of bugs, but never to show their absence!*


*Edsger W. Dijkstra*

En español, por si el inglés fuera un problema:

**¡El testing se puede usar para demostrar la presencia de errores pero nunca para demostrar su ausencia!**


Edsger W. Dijkstra

- ☐ **Ejercicio 7.** Llegó la hora de implementar el simulador. Lo que tienes que hacer es escribir un programar principal que crea unos cuantos Sims y va ejecutando `simular(1)` (una hora cada vez) sobre cada uno de ellos. En cada paso sería conveniente decir lo que le pasa a cada Sim. Se puede limitar el simulador a  $N$  horas y finalmente imprimir las estadísticas de cada Sim

 **Ejercicio 8.** En la asignatura vamos a ver tipos abstractos de datos que representan colecciones acotadas y no acotadas. Los nombres de dichos tipos serán **listas** (*lists*), **colas** (*queues*) y **pilas** (*stacks*).

Vamos a centrarnos en el tipo de las listas. Dicho tipo, además de su nombre `List`, tiene las siguientes operaciones: `add`, `get`, `size`, `set`, `indexOf`, `remove` ( $\times 2$ ) y `sublist`. Su **semántica**:

- **void** `add(int index, E elem)`: Coloca un nuevo elemento `elem` en la posición `index` de la lista.
- **E** `get(int index)`: Devuelve el elemento de la lista en la posición `index`.
- **int** `size()`: Devuelve el número de elementos en la lista.
- **void** `set(int index, E elem)`: Coloca el elemento `elem` en la posición `index` (sobre-escribiendo el elemento que ocupara dicha posición).
- **int** `indexOf(E elem)`: Devuelve la posición ocupada por el primer elemento de lista igual a `elem` (se usa `equals` para hacer la comparación).
- **void** `remove(int index)`: Elimina de la lista el elemento que ocupa la posición `index`.
- **void** `remove(E elem)`: Elimina de la lista el primer elemento que sea igual a `elem` (se usa `equals` para hacer la comparación).
- **public List** `subList(int inicio, int fin)`: Devuelve una vista de la porción de la lista **this** entre las posiciones `inicio` y `fin - 1`.

 **Ejercicio 9.** Implementa una versión “vacía” de `ListSim` siguiendo el API anterior (sustituye `E` por `Sim`). Cuando escribimos una versión “vacía”, queremos decir que simplemente compile y que no haga nada, pero que esté **documentado**. Por ejemplo:

```
public class ListSim {
    /**
     * Crea una lista con la capacidad máxima indicada.
     */
    public ListSim(int capacidad) {
    }
    /**
     * Devuelve el Sim de la lista en la posición getIndex.
     *
     * @return Sim que ocupa la posición getIndex
     */
    public Sim get(int getIndex) {
        return null;
    }
    // ETC.
}
```

- 📄 **Ejercicio 10.** Implementa unos tests para las listas de Sims: `TestListSim.java`. Dicho programa tiene que comprobar ciertas propiedades que esperas que las listas cumplan. Puedes empezar con estos, aunque son realmente tontos:

```
public class TestListSim {
    public static void main(String[] args) {
        ListSim lista = new ListSim();

        assert lista.size() == 0 : "Una lista recién debe tener 0 elementos";

        lista.add(0, new Sim("Ángel"));

        assert lista.size() == 1
            : "Tras añadir un elemento la lista debe tener 1 elemento";

        assert "Ángel".equals(lista.get(0).nombre())
            : "Error en el elemento almacenado";
    }
}
```

- 📄 **Ejercicio 11.** Ahora ya puedes implementar la clase `ListSim` utilizando los arrays nativos de Java y otros atributos (índices o **nulles**) que consideres oportunos.