

Sesión 11: Tipos Abstractos de Datos

Hoja de problemas

Programación 2

Ángel Herranz

aherranz@fi.upm.es

Universidad Politécnica de Madrid

2021-2022

📄 **Ejercicio 1.** 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**.

🔍 **Ejercicio 2.** Busca en internet un tutorial o un manual de la herramienta javadoc. Intenta entender al menos las opciones `-d directorio`, `-author` o `-use`.

📄 **Ejercicio 3.** 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 (API¹), 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.
- **quease:** La operación quease 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.

¹Application Public Interface

- **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 **javadococumentar** todo el código de SIM.java.

📖 **Ejercicio 4.** 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 5.** Nunca olvides la siguiente frase del gran Edsger W. Dijkstra:

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 6.** Llegó la hora de implementar el simulador. Lo que tienes que hacer es escribir un programa 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 7.** 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 8.** 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 9.** 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 10.** Ahora ya puedes implementar la clase `ListSIM` utilizando los arrays nativos de Java y otros atributos (índices o **nulles**) que consideres oportunos.