# Índice general

# Capítulo 1

# Contexto y evaluación

## 1.1. Conocimientos y destrezas a adquirir por los estudiantes

Se espera que, una vez finalizadas las prácticas, el estudiante sea capaz de:

1. Manejar bucles (tanto simples como anidados) para resolver problemas.
2. Manejar *Arrays* multidimensionales.
3. Implementar una estructura de datos.
4. Utilizar estructuras de datos.
5. Implementar algoritmos recursivos.
6. Construir software mediante desarrollo dirigido por pruebas (TDD).
7. Realizar un análisis básico de complejidad de algoritmos.

## 1.2. Material a entregar por los estudiantes

Para las dos primeras prácticas, se deberán entregar bien sea el `makefile`, bien sea el proyecto Maven, con, al menos, las opciones de compilar y generar el Javadoc.

Para la última práctica será necesario entregar el proyecto Maven, incluyendo también las pruebas realizadas a través de JUnit. Además de las opciones de compilar y generar el Javadoc, también se incluirá la opción de ejecutar las pruebas.

Además de lo indicado en el párrafo anterior, para cada práctica, se entregará el siguiente material:

- Fichero `README.md` en formato *markdown* con las notas oportunas tanto para los usuarios del programa como para los desarrolladores. Tales notas han de incluir el análisis de complejidad de cada método.
- Diagramas UML en el formato de Umbrello.
- El fichero `.gitignore` del repositorio.

## 1.3. Criterios de evaluación

Tanto el peso que tiene en la evaluación de la asignatura como el umbral de los distintos aspectos de la práctica (documentación, uso de herramientas de soporte al desarrollo y producto funcionando) están publicados en la guía docente. Los pesos de las prácticas serán los que se muestran a continuación:

1. Práctica 1: 5 % del total de las prácticas.
2. Práctica 2: 35 % del total de las prácticas.
3. Práctica 3: 60 % del total de las prácticas.

Se aplicarán los siguientes criterios de evaluación:

1. El producto debe funcionar correctamente para alcanzar la puntuación de aprobado.
2. Tendrán importancia en la evaluación los siguientes aspectos del código: legibilidad, eficiencia, extensibilidad, etc.
3. Se valorará tanto la documentación en `Javadoc` como el `README.md` y, en caso de que sean necesarios, los comentarios adicionales, por ejemplo, aquéllos que están dentro de los cuerpos de los métodos.
4. Para el caso de la práctica 3, se valorará que las pruebas sean las adecuadas para el sistema. Deberán estar documentadas y su código deberá satisfacer los criterios de calidad expuestos en los puntos anteriores para el software a desarrollar.
5. Se valorará la gestión de repositorios. Para ello, el profesor podrá consultar los *commits* realizados a los repositorios utilizados.

## 1.4. Forma de entrega

Los materiales deberán ser subidos a un repositorio remoto (al que debe tener acceso el profesor). **NO** se corregirá material que no haya sido subido al repositorio.

Las fechas de las entregas serán publicadas por el profesor en el portal del alumno. **NO** se corregirá a partir del último *commit* dentro del plazo de entrega.

**NO** se permitirán formatos propietarios, por ejemplo, RAR.

## 1.5. Licencias

Al principio de cada fichero de código fuente se ha de incluir el comentario que se muestra a continuación, reemplazando los corchetes por la información específica:

```
Copyright [año] [nombre del propietario del copyright]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing,
software distributed under the License is distributed on an
"AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,
either express or implied. See the License for the specific
language governing permissions and limitations under the
License.
```

# Capítulo 2

# Prácticas

## 2.1. Práctica 1. Obtención de una aproximación al número pi

Implemente un programa que permita obtener una aproximación al número pi mediante el método de Montecarlo[1]. Se propone el siguiente pseudocódigo (tenga en cuenta que el radio del círculo elegido es 1):

**Algoritmo 1:** Aproximación a pi mediante Montecarlo

**Input:** *puntosTotales*: cantidad de puntos a generar
**Output:** Número real que es una aproximación al número pi
$aciertos \leftarrow 0$;
$areaCuadrado \leftarrow 4$;
**for** $i \leftarrow 1$ **to** *puntosTotales* **do**
    Obtener un número aleatorio $x$ entre -1 y 1;
    Obtener un número aleatorio $y$ entre -1 y 1;
    **if** $d((x, y), (0, 0)) \leq 1$ **then** $aciertos \leftarrow aciertos + 1$;
**end**
**return** $areaCuadrado \cdot (aciertos/puntosTotales)$ ;

El cuadrado en que está inscrito el círculo tiene como vértices los puntos $(-1, -1), (-1, 1), (1, 1), (1, -1)$. El círculo tiene como centro el punto $(0, 0)$. Además, tal y como explica el vídeo al que se ha hecho referencia anteriormente, se asume que la proporción de puntos que caen dentro del círculo con respecto al total de puntos generados es aproximadamente la misma que la proporción entre el área del círculo y la del cuadrado en que está inscrito.

El pseudocódigo está escrito para que el estudiante pueda relacionarlo con el método explicado en el vídeo. No obstante, se pueden realizar simplificaciones para que el programa final ocupe menos líneas de código sin cambiar su comportamiento.

[1] `https://www.youtube.com/watch?v=ELetCV_wX_c`

El algoritmo estará implementado en la clase `Matematicas.java` de acuerdo con la siguiente estructura:

```
package mates;

public class Matematicas{
    /**
     * Genera una aproximación al número pi mediante el método de
     * Montecarlo. El parámetro 'pasos' indica el número de puntos
     * generado.
     */
    public static double generarNumeroPi(long pasos){
        return 0; // Este código hay que cambiarlo.
    }
}
```

El programa principal para mostrar el resultado es el siguiente:

```
package aplicacion;

import mates.Matematicas;

public class Principal{
    public static void main(String[] args){
        System.out.println("El número PI es " + Matematicas.
            generarNumeroPi(Integer.parseInt(args[0])));
    }
}
```

## 2.2. Práctica 2. El juego de la vida

Implemente una versión del juego de la vida[2] con las siguientes características:

1. Las reglas son las habituales:
   a) Si una célula está viva y dos o tres de sus vecinas también lo están, entonces continúa viva en el estado siguiente.
   b) Si una célula está muerta y tres de sus vecinas están vivas, entonces pasa a estar viva en el estado siguiente.
   c) El resto de células pasan a estar muertas en el estado siguiente.
2. Asuma un tablero con 30 celdas (células).
3. El estado inicial del tablero estará almacenado en un fichero (llamado `matriz`) con 30 filas y 30 columnas en que cada celda será un uno o un cero. Por ejemplo,

[2] `https://www.youtube.com/watch?v=ouipbDkwHWA`

```
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000001000000000000000000000000
00000101000000000000000000000000
00000110000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

4. Se implementará la clase `Tablero.java` con la siguiente estructura:

```
package dominio;

/**
 * Esta clase es responsable de leer el tablero de un
 * fichero en forma de ceros y unos, ir transitando de
 * estados e ir mostrando dichos estados.
 */
public class Tablero{
    private static int DIMENSION = 30;
    private int[][] estadoActual; //matriz que representa el
                                  // estado actual.
    private int[][] estadoSiguiente
        = new int[DIMENSION][DIMENSION]; // Matriz que
                                         // representa el
                                         // estado
                                         // siguiente.

    /*******************************************************
     * Lee el estado inicial de un fichero llamado 'matriz'.
     *******************************************************/
```

```
    public void leerEstadoActual(){}
    // La secuencia de ceros y unos del fichero es guardada
    // en `estadoActual` y, utilizando las reglas del juego
    // de la vida, se insertan los ceros y unos
    // correspondientes en `estadoSiguiente`.

    /*******************************************************
     * Genera un estado inicial aleatorio. Para cada celda
     * genera un número aleatorio en el intervalo [0, 1). Si
     * el número es menor que 0,5, entonces la celda está
     * inicialmente viva. En caso contrario, está muerta.
     *******************************************************/
     public void generarEstadoActualPorMontecarlo(){}
     // La secuencia de ceros y unos generada es guardada
     // en `estadoActual` y, utilizando las reglas del juego
     // de la vida, se insertan los ceros y unos
     // correspondientes en `estadoSiguiente`.

    /*******************************************************
     * Transita al estado siguiente según las reglas del
     * juego de la vida.
     *******************************************************/

    public void transitarAlEstadoSiguiente(){}
    // La variable `estadoActual` pasa a tener el contenido
    // de `estadoSiguiente` y, éste útimo atributo pasar a
    // reflejar el siguiente estado.

    /*******************************************************
     * Devuelve, en modo texto, el estado actual.
     * @return el estado actual.
     *******************************************************/
    @Override
    public String toString(){
        return ""; // Esta línea hay que modificarla.
    }

}
```

5. El programa principal para mostrar el resultado es el que se presenta a continuación:

```
import dominio.Tablero;

import java.util.concurrent.TimeUnit;
import java.lang.InterruptedException;

public class Principal{
    public static void main(String[] args){
        try
        {
            Tablero tablero = new Tablero();
            System.out.println("SIMULACIÓN CON TABLERO LEÍDO
                ");
```

```
                tablero.leerEstadoActual();
                System.out.println(tablero);
                for(int i = 0; i <= 5; i++)
                {
                    TimeUnit.SECONDS.sleep(1);
                    tablero.transitarAlEstadoSiguiente();
                    System.out.println(tablero);
                }

                System.out.println("SIMULACIÓN CON TABLERO
                    GENERADO MEDIANTE MONTECARLO");
                tablero.generarEstadoActualPorMontecarlo();
                System.out.println(tablero);
                for(int i = 0; i <= 15; i++)
                {
                    TimeUnit.SECONDS.sleep(1);
                    tablero.transitarAlEstadoSiguiente();
                    System.out.println(tablero);
                }
            }catch(InterruptedException e)
            {
                System.out.println(e);
            }
        }
    }
```

A continuación se muestra el resultado del método `toString()` de la clase `Tablero.java` para el tablero de la matriz de ceros y unos presentada anteriormente:

```
 x
x x
 xx
```

## 2.3. Práctica 3. Búsqueda de un camino en un grafo

Implemente, siguiendo un desarrollo dirigido por pruebas, una estructura de datos de grafo según el siguiente esquema:

```
package pr2;

public class Graph<V>{

    //Lista de adyacencia.
    private Map<V, Set<V>> adjacencyList = new HashMap<>();

    /**********************************************************************

     * Añade el vértice `v` al grafo.
     *
     * @param v vértice a añadir.
     * @return `true` si no estaba anteriormente y `false` en caso
```

```
     *         contrario.
     ************************************************************/

    public boolean addVertex(V v){
        return true; //Este código hay que modificarlo.
    }

    /************************************************************

     * Añade un arco entre los vértices `v1` y `v2` al grafo. En
     * caso de que no exista alguno de los vértices, lo añade
     * también.
     *
     * @param v1 el origen del arco.
     * @param v2 el destino del arco.
     * @return `true` si no existía el arco y `false` en caso
         contrario.
     ************************************************************/

    public boolean addEdge(V v1, V v2){
        return true; //Este código hay que modificarlo.
    }

    /************************************************************

     * Obtiene el conjunto de vértices adyacentes a `v`.
     *
     * @param v vértice del que se obtienen los adyacentes.
     * @return conjunto de vértices adyacentes.
     ************************************************************/

    public Set<V> obtainAdjacents(V v) throws Exception{
        return null;  //Este código hay que modificarlo.
    }

    /************************************************************

     * Comprueba si el grafo contiene el vértice dado.
     *
     * @param v vértice para el que se realiza la comprobación.
     * @return `true` si `v` es un vértice del grafo.
     ************************************************************/

    public boolean containsVertex(V v){
        return true; //Este código hay que modificarlo.
    }

    /************************************************************

     * Método `toString()` reescrito para la clase `Grafo.java`.
     * @return una cadena de caracteres con la lista de
     * adyacencia.
     ************************************************************/
```

```
    @Override
    public String toString(){
        return ""; //Este código hay que modificarlo.
    }

    /*******************************************************
     * Obtiene, en caso de que exista, un camino entre `v1` y
     * `v2`. En caso contrario, devuelve `null`.
     *
     * @param v1 el vértice origen.
     * @param v2 el vértice destino.
     * @return lista con la secuencia de vértices desde `v1` hasta
     * `v2` * pasando por arcos del grafo.
     *******************************************************/
    public List<V> onePath(V v1, V v2){
        return null;  //Este código hay que modificarlo.
    }
}
```

Para implementar el método `onePath()` puede seguir el siguiente pseudocódigo:

**Algoritmo 2:** Búsqueda de un camino entre dos vértices

**Input:** El vértice de inicio $v_1$ y el vértice de fin $v_2$
**Output:** Secuencia de vértices desde $v_1$ hasta $v_2$ a través de arcos del grafo

```
Cree una tabla llamada traza;
Cree una pila llamada abierta;
abierta.push(v1);
traza.annadir(v1, null);
encontrado ← falso;
while ¬abierta.esVacia() ∧ ¬encontrado do
    v ← abierta.pop();
    encontrado pasa a ser verdadero si v es igual a v2;
    if ¬encontrado then
        for s ∈ adyacentes(v) do
            abierta.push(s);
            traza.annadir(s, v);
        end
    end
end
if encontrado then
    Reconstruir el camino que hay en traza y devolverlo
else
    Devolver un indicador (por ejemplo, null) de que no se ha
     encontrado el camino
end
```

La *traza* almacena los nudos que se van encontrando en la exploración

| Vértice | Padre |
|---|---|
| $A$ | *null* |
| $B$ | $A$ |
| $C$ | $A$ |
| $D$ | $B$ |

Tabla 2.1: Ejemplo de contenido de la *traza*.

con su padre correspondiente. Por ejemplo, la tabla 2.1 muestra que $A$ es el vértice de inicio (que no tiene ningún padre en la búsqueda), $B$ y $C$ son los sucesores directos de $A$, y $D$ es sucesor de $B$. En caso de que $D$ fuera el vértice destino, se podría reconstruir el camino $A$, $B$, $D$.

La pila *abierta* contiene aquellos vértices del árbol de búsqueda que han sido generados y que están pendientes de ser examinados.

El código debe pasar, al menos, la siguiente prueba:

```
/**
 * Este test comprueba que el método `onePath(V v1, V v2)`
 * encuentra un camino entre `v1` y `v2` cuando existe.
 */
@Test
public void onePathFindsAPath(){
    System.out.println("\nTest onePathFindsAPath");
    System.out.println("----------------------");
    // Se construye el grafo.
    Graph<Integer> g = new Graph<>();
    g.addEdge(1, 2);
    g.addEdge(3, 4);
    g.addEdge(1, 5);
    g.addEdge(5, 6);
    g.addEdge(6, 4);

    // Se construye el camino esperado.
    List<Integer> expectedPath = new ArrayList<>();
    expectedPath.add(1);
    expectedPath.add(5);
    expectedPath.add(6);
    expectedPath.add(4);
    //Se comprueba si el camino devuelto es igual al esperado.
    assertEquals(expectedPath, g.onePath(1, 4));
}
```