UNIVERSIDADE DA CORUÑA

# Diseño Software

# Práctica 1 (2025-2026)

## INSTRUCCIONES COMUNES A TODAS LAS PRÁCTICAS:

- **Grupos de prácticas**
  - Los ejercicios se realizarán preferentemente por parejas (pueden hacerse en solitario pero no lo recomendamos) y ambos miembros del grupo serán responsables y deben conocer todo lo que se entregue en su nombre. Recomendamos realizar la práctica con técnicas de "programación en pareja".
  - El nombre del equipo de prácticas será el nombre del grupo de prácticas al que pertenecen los miembros (con un prefijo "`DS-`") seguido por sus correspondientes *logins* de la UDC separados por guiones bajos, por ejemplo: `DS-12_jose.perez_francisco.garcia`[1].
  - En caso de pertenecer a grupos de prácticas distintos anteponer los dos grupos al inicio, siendo el primer grupo el que corresponde al primer login, como en el siguiente ejemplo: `DS-12-32_jose.perez_francisco.garcia`.
- **Entrega**
  - Los ejercicios serán desarrollados mediante la herramienta IntelliJ IDEA (versión *Community*) que se ejecuta sobre Java.
  - Los ejercicios se entregarán usando el sistema de control de versiones Git utilizando el servicio *GitHub Classroom*.
  - Tendremos una clase de prácticas dedicada a explicar Git, su uso en IntelliJ, GitHub Classroom y a cómo entregar las prácticas usando este sistema. Hasta entonces podéis ir desarrollando las prácticas en local.
  - Para la evaluación de la práctica sólo tendremos en cuenta aquellas contribuciones hechas hasta la fecha de entrega en el correspondiente repositorio de GitHub Classroom, los envíos posteriores no serán tenidos en cuenta.
- **Evaluación**
  - **Importante: Si se detecta algún ejercicio copiado en una práctica, ésta será anulada en su totalidad (calificación cero), tanto el original como la copia**.

[1] Si tenéis un *login* muy largo podéis tratar de acortarlo poniendo solo un apellido, siempre y cuando no exista confusión con algun compañero vuestro.

## INSTRUCCIONES PRÁCTICA 1:

**Fecha límite de entrega: 10 de octubre de 2025 (hasta las 23:59)**.

- **Realización de la práctica**
  - Los ejercicios se entregarán usando *GitHub Classroom*. En concreto en el *assignment* `2526-P1` del *classroom* `GEI-DS-614G010152526-OP1`.
  - Se deberá subir al repositorio un único proyecto IntelliJ IDEA para la práctica con el nombre del grupo de prácticas más el sufijo "`-P1`" (por ejemplo `DS-12_jose.perez_francisco.garcia-P1`).
  - Se creará un paquete por cada ejercicio de la práctica usando los siguientes nombres: `e1`, `e2`, etc.
  - Es importante que sigáis detalladamente las instrucciones del ejercicio, ya que persigue el objetivo de probar un aspecto determinado de Java y la orientación a objetos.
- **Comprobación de la ejecución correcta de los ejercicios con JUnit**
  - En la asignatura usaremos el framework JUnit 5 para comprobar, a través de pruebas, que el funcionamiento de las prácticas es el correcto.
  - En esta primera práctica os adjuntaremos algunos de los tests JUnit que deben pasar los ejercicios para ser considerados válidos.
  - **IMPORTANTE: No debéis modificar los tests que os pasemos**, sí podréis añadir nuevos tests para esos ejercicios si lo consideráis necesario. **Si un ejercicio no tiene tests es responsabilidad vuestra desarrollar los tests para el mismo**. Valoraremos la calidad de las pruebas realizadas.
  - En el seminario de JUnit os daremos información detallada de como ejecutar los tests y calcular la cobertura de los mismos y en el seminario de Git os comentaremos su integración con GitHub Classroom.
- **Evaluación**
  - Esta práctica corresponde a un 20 % de la nota final de prácticas.
  - **Criterios generales**: que el código compile correctamente, que no de errores de ejecución, que se hayan seguido correctamente las especificaciones, que se hayan seguido las buenas prácticas de la orientación a objetos explicadas en teoría, etc.
  - **Pasar correctamente nuestros tests es un requisito importante en la evaluación de esta práctica**.
  - Aparte de criterios fundamentales habrá criterios de corrección específicos que detallaremos en cada ejercicio.
  - No seguir las normas aquí indicadas significará una penalización en la nota.

1. **Substitution Cipher**

   Crearemos una clase denominada `SubstitutionCipher` cuyo objetivo sea realizar un cifrado de sustitución. El cifrado de sustitución consiste en desplazar una letra un número dado de posiciones en el alfabeto. Por ejemplo con un desplazamiento de 3 letras "`CESAR`" se convierte en "`FHVDU`". Para ello incluye los siguientes métodos públicos y estáticos en la clase:

   - `String encode(int key, String text)`. Devuelve un `String` que es el resultado de codificar el texto `text`, usando el desplazamiento indicado por `key`
   - `String decode(int key, String text)` Devuelve un `String` que es el resultado de decodificar el texto `text` que ha sido codificado con el desplazamiento `key`

   Por simplicidad, podemos usar el alfabeto inglés ("`ABCDEFGHIJKLMNOPQRSTUVWXYZ`" en mayúsculas y "`abcdefghijklmnopqrstuvwxyz`" en minúsculas. Si la letra original estaba en mayúsculas la letra cifrada deberá estar en mayúsculas y análogamente con las minúsculas. Si una letra no pertenece a dicho alfabeto se deja tal y como está (las "`Ñ`", los carácteres acentuados, interrogaciones, etc.). Así el texto "`¡¡Cañonazo al Balón!!`" se codificaría en clave 10 como "`¡¡Mkñyxkjy kv Lkvóx!!`"

   Tema: Bucles, sentencias condicionales, clase `String`, tipo básico `char`

   De utilidad: Para resolver este ejercicio os serán de mucha utilidad las siguientes clases y sus correspondientes métodos (revisarlos en la documentación de Java):

   - `String`. Representa una cadena de caracteres inmutable (no se puede modificar). Incluye métodos para el manejo de la misma y sus caracteres.
   - `Character`. Incluye métodos para el manejo y análisis de elementos de tipo `char`.
   - `StringBuilder`. Similar a `String` pero mutable, le puedes añadir o modificar los contenidos.

2. **Descenso de pendientes**

En una estación de esquí las pendientes que están fuera de pista están plagadas de árboles. Hemos hecho el mapeo de una de estas pendientes y el resultado es el siguiente (donde un punto "`.`" representa un espacio libre y una almohadilla "`#`" representa un árbol).

```
. . # # . . . . . . .
# . . . # . . . # . .
. # . . . . # . . # .
. . # . # . . . # . #
. # . . . # # . . # .
. . # . # # . . . . .
. # . # . # . . . . #
. # . . . . . . . . #
# . # # . . . # . . .
# . . . # # . . . . #
. # . . # . . . # . #
```

Tenemos un esquiador dispuesto a bajar una de estas pendientes. Nuestro esquiador es habilidoso pero un poco limitado, ya que una vez que decide una estrategia de descenso sigue con ella sin importar los árboles que se encuentre por el camino.

A la hora de ejecutar las estrategias de descenso tenemos que tener en cuenta que:

- Las estrategias de descenso se definen por dos números (`right` y `down`) que definen el movimiento que hace el esquiador hacia la derecha y hacia abajo en el mapa.
- El esquiador siempre empieza en la esquina superior izquierda y acaba en el momento en que abandona la última fila.
- El esquiador siempre hace primero el movimiento `right` y luego el movimiento `down`.
- Cuando el esquiador llega al límite derecho del mapa sigue por el izquierdo en la misma fila (como si el mapa fuese un mapa infinito que repite el mismo patrón constantemente).

Nuestro esquiador decide probar con una estrategia (`right=3` y `down=1`) para bajar la pendiente. A continuación mostramos el recorrido que hace, marcando los puntos iniciales/finales del movimiento con corchetes "`[]`" y los puntos intermedios con paréntesis "`()`". Como vemos siempre hace primero el movimiento a la derecha y, en caso de llegar a la última columna, continúa por la primera columna dentro de la misma fila. El resultado es el siguiente:

```
[.]  (.)  (#)  (#)   .    .    .    .    .    .    .
 #    .    .   [.]  (#)  (.)  (.)   .    #    .    .
 .    #    .    .    .    .   [#]  (.)  (.)  (#)   .
(.)  (.)   #    .    #    .    .    .    #   [.]  (#)
 .   [#]  (.)  (.)  (.)   #    #    .    .    #    .
 .    .    #    .   [#]  (#)  (.)  (.)   .    .    .
 .    #    .    #    .    #    .   [.]  (.)  (.)  (#)
(.)  (#)  (.)   .    .    .    .    .    .    .   [#]
 #    .   [#]  (#)  (.)  (.)   .    #    .    .    .
 #    .    .    .    #   [#]  (.)  (.)  (.)   .    #
(.)   #    .    .    #    .    .    .   [#]  (.)  (#)
```

Nuestro esquiador se ha encontrado con 17 árboles en el camino. Decide cambiar la estrategia y seguir una de *saltos*, es decir, el movimiento será igual que antes (primero a la derecha y luego abajo) pero se moverá ejecutando un salto, por lo que solo tendremos en cuenta, a la hora de chocar con los árboles, aquellos que se encuentren en los puntos iniciales y finales del recorrido. De esta forma solo se encontrará con 7 árboles, como se muestra a continuación:

```
[.]   .    #    #    .    .    .    .    .    .    .
 #    .    .   [.]   #    .    .    .    #    .    .
 .    #    .    .    .    .   [#]   .    .    #    .
 .    .    #    .    #    .    .    .    #   [.]   #
 .   [#]   .    .    .    #    #    .    .    #    .
 .    .    #    .   [#]   #    .    .    .    .    .
 .    #    .    #    .    #    .   [.]   .    .    #
 .    #    .    .    .    .    .    .    .    .   [#]
 #    .   [#]   #    .    .    .    #    .    .    .
 #    .    .    .    #   [#]   .    .    .    .    #
 .    #    .    .    #    .    .    .   [#]   .    #
```

Implementa las funciones `downTheSlope()` y `jumpTheSlope()`:

```java
/**
* Traverses the slope map making the right and down movements and
* returns the number of trees found along the way.
* @param slopeMap A square matrix representing the slope with spaces
*                 represented as "." and trees represented as "#".
* @param right Movement to the right
* @param down Downward movement
* @return Number of trees
* @throws IllegalArgumentException if the matrix is incorrect because:
*     - It is not square.
*     - It has characters other than "." and "#"
*     - right >= number of columns or right < 1
*     - down >= number of rows of the matrix or down < 1
*/
public static int downTheSlope(char[][] slopeMap, int right, int down) { /* */}

/**
* Traverses the slope map making the right and down movements and
* returns the number of trees found along the way.
* Since it "jumps" from the initial position to the final position,
* only takes into account the trees on those initial/final positions.
*
* Params, return value and thrown expections as in downTheSlope...
*/
public static int jumpTheSlope(char[][] slopeMap, int right, int down) { /* */}
```

**Criterios**: Manejo de *arrays* en Java.

3. **Rectángulos**

Escribe la clase `Rectangle` según la especificación que se muestra en el código. Ten en cuenta que la orientación del rectángulo es irrelevante a efectos de igualdad de objetos; es decir, un rectángulo se considera igual a sí mismo girado 90 grados. Ten en cuenta también que ciertos métodos pueden lanzar excepciones en su funcionamiento (los constructores o los métodos *set*).

```
// Represents a rectangle
public class Rectangle {

    // Initializes a new rectangle with the values passed by parameter.
    // Throws IllegalArgumentException if a a negative value is passed to any of
    // the dimensions.
    public Rectangle(int base, int height) { }

    // Copy constructor
    public Rectangle(Rectangle r) { }

    // Getters
    public int getBase() { }
    public int getHeight() { }

    // Setters. Throw IllegalArgumentException if the parameters are negative.
    public void setBase(int base) { }
    public void setHeight(int height) { }

    // Return true if the rectangle is a square
    public boolean isSquare() { }

    // Calculate the area of the rectangle.
    public int area() { }

    // Calculate the perimeter of the rectangle.
    public int perimeter() { }

    // Calculate the length of the diagonal
    public double diagonal() { }

    // Turn this rectangle 90 degrees (changing base by height).
    public void turn() { }

    // Ensure that this rectangle is oriented horizontally (the base is greater
    // or equal than the height).
    public void putHorizontal() { }

    // Ensure that this rectangle is oriented vertically (the height is greater
    // or equal than the base).
    public void putVertical() { }

    // Two rectangles are equal if the base and the height are the same.
    // It does not take into account if the rectangle is rotated.
    public boolean equals(Object obj) { }

    // It complies with the hashCode contract and is consistent with the equals.
    public int hashCode() { }
}
```

**Criterios**:

- Instanciación de objetos.
- Encapsulamiento.
- Manejo de excepciones.
- Contratos del equals y el hashCode.

4. **Semáforos y cruces**

Un semáforo tiene las siguientes características:

- Tiene tres colores: rojo, ambar y verde. Solo uno puede estar encendido a la vez.
- El color ambar puede parpadear, pero los otros no.
- La secuencia de cambio de colores es siempre la misma: de verde a ambar, luego a rojo y de vuelta de nuevo al verde.
- Los semáforos tienen un tiempo determinado para estar en verde y en amarillo, luego automáticamente pasan al siguiente color de la secuencia.
- Por simplificar supondremos que todos los semáforos tienen los mismos tiempos: `15` segundos en verde y `5` segundos en amarillo.
- El tiempo que pasan en rojo dependerá de cuántos semáforos hay en un cruce, pero como mínimo será de `20` segundos (`15`+`5`).
- El color ambar con parpadeo es un color especial que, en nuestro modelo, se usa para indicar que el semáforo está desactivado. No tiene un tiempo en concreto en el que estar en dicho estado (puede ser indefinidamente).
- Todo semáforo tendrá un nombre que lo identifique.

En base a eso crea una clase `TrafficJunction` que represente a un cruce controlado por semáforos que tenga los métodos que se indican a continuación:

```
public class TrafficJunction {
    /**
     * Creates a trafic junction with four traffic lights named north, south,
     * east and west. The north traffic light has just started its green cycle.
     */
    public TrafficJunction() { /* ... */ }
    /**
     * Indicates that a second of time has passed, so the traffic light with
     * the green or amber light should add 1 to its counter. If the counter
     * passes its maximum value the color of the traffic light must change.
     * If it changes to red then the following traffic light changes to green.
     * The order is: north, south, east, west and then again north.
     */
    public void timesGoesBy() { /* ... */ }
    /**
     * If active is true all the traffic lights of the junction must change to
     * blinking amber (meaning a non-controlled junction).
     * If active is false it resets the traffic lights cycle and started again
     * with north at green and the rest at red.
     * @param active true or false
     */
    public void amberJunction(boolean active) { /* ... */ }
    /**
     * Returns a String with the state of the traffic lights.
     * The format for each traffic light is the following:
     * - For red colors: "[name: RED]"
     * - For green colors: "[name: GREEN counter]"
     * - For yellow colors with blink at OFF: "[name: YELLOW OFF counter]
     * - For yellow colors with blink at ON: "[name: YELLOW ON]
     * Examples:
     * [NORTH: GREEN 2][SOUTH: RED][EAST: RED][WEST: RED]
     * [NORTH: AMBER OFF 5][SOUTH: RED][EAST: RED][WEST: RED]
     * [NORTH: AMBER ON][SOUTH: AMBER ON][EAST: AMBER ON][WEST: AMBER ON]
     * @return String that represents the state of the traffic lights
     */
    @Override
    public String toString() { /* ... */ }
}
```

**Criterios**:

- El problema debe resolverse haciendo uso de valores enumerados en Java. Qué tipos deben ser y con qué diseño es tarea vuestra.
- Se valorará especialmente que no se usen enumerados simples sino que sean enumerados complejos (con constructores, métodos, etc.).
- También se valorará que los enumerados se usen eficientemente, haciendo uso de sus propios métodos (como `values()`) o de clases diseñadas para manejarlos (como `EnumSet` o `EnumMap`).
- Los semáforos nunca pueden ser puestos en un estado inconsistente (en ningún color, en más de un color, en verde parpadeando, etc.)
- Se valorará también que `TrafficJunction` sea una clase sencilla que delegue la mayoría del funcionamiento en otras clases (cambio de colores en el semáforo, tiempo en un determinado color, etc.).