10 | Ajedrez

Meta

Que el alumno se familiarice con el uso de herencia en diseño orientado a objetos.

Objetivos

Al finalizar la práctica el alumno será capaz de:

1. Definir clases y métodos abstractos.
2. Programar clases derivadas de clases abstractas.
3. Dar ejemplos de jerarquías de herencia.

Antecedentes

Superclases

El mecanismo de *herencia* en orientación a objetos permite programar el código común a varias clases una sola vez y después simplemente *heredarlo*. Este mecanismo funciona por medio de la abstracción de características comunes entre una familia de clases, tanto atributos como métodos. Estas características comunes se agrupan en lo que llamaremos una *superclase*, *clase padre* o, si seguimos el acuerdo de género, *clase madre*.

Las superclases pueden ser *abstractas*, si no es posible crear objetos de ese tipo, o no serlo y entonces es posible crear objetos de ese tipo y también de sus hijas. Llevará el calificativo `abstract` al declararla. Las clases abstractas pueden tener métodos abstractos, cuyo encabezado se declara, pero no se define su contenido, en este sentido se parecen a los métodos de las interfaces, sólo que la palabra `abstract` aparece en el mismo.

Ejemplo 10.1. *Propongamos una familia de figuras, es posible calcular el área y el perímetro de toda figura, pero para cada una se hace de forma distinta. No podemos*

entonces crear objetos de tipo figura, sino hasta que digamos de qué figura se trata y especifiquemos cómo calcular estos valores. El código siguiente ejemplifica cómo programar clases con herencia.

Listado 10.1: Figura.java

```
/** No se pueden crear figuras directamente. */
public abstract class Figura {

  /** Todas las figuras tienen un nombre. */
  protected String nombre;

  /** Constructor */
  protected Figura(String nombre) {
    this.nombre = nombre;
  }

  /** Método de lectura, da acceso al valor del atributo
    * nombre */
  public String nombre() {
    return nombre;
  }

  public abstract double calculaPerímetro();
  public abstract double calculaÁrea();
}
```

Listado 10.2: Cuadrado.java

```
public class Cuadrado extends Figura {
  private double lado;

  public Cuadrado(double lado) {
    super("cuadrado");  // Invoca al constructor de Figura

    if(lado < 0) throw new IllegalArgumentException("lado < 0");
    this.lado = lado;
  }

  public double calculaPerímetro() {
    return lado * 4;
  }

  public double calculaÁrea() {
    return lado * lado / 2;
  }
}
```

Listado 10.3: Círculo.java

```
public class Círculo extends Figura {
```

```
2     private double radio;
3
4     public Círculo(double radio) {
5       super("círculo");  // Invoca al constructor de Figura
6
7       if(radio < 0) throw new IllegalArgumentException("lado␣<␣0");
8       this.radio = radio;
9     }
10
11    public double calculaPerímetro() {
12      return Math.PI * radio * 2;
13    }
14
15    public double calculaÁrea() {
16      return Math.PI * radio * radio;
17    }
18  }
```


Diagramas de clase UML

Para expresar mejor el trabajo que se realizará en esta práctica haremos uso de una notación especial: *los diagramas de clase de UML*.

UML son las siglas en inglés de *Unified Markup Language*.

Clases

Las clases se representan con cajas donde se pueden escribir:

- El nombre de la clase.
- Los atributos de la clase.
- Los métodos de la clase.

Para representar los diferentes tipos de acceso se utilizan los símbolos siguientes:

\- privado. Solamente los métodos de la clase pueden acceder a estos atributos.

\# protegido. La clase y clases que la extiendan, directa o indirectamente, pueden acceder a estos atributos.

\+ público. Todos pueden leer y escribir directamente estos atributos. Se recomienda sólo utilizarlos cuando los atributos son `final`, es decir, su valor nunca cambia.

Los accesos también afectan a los métodos e indican quién puede solicitar la ejecución de esos métodos.

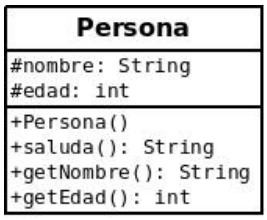


Figura 10.1 Ejemplo de diagrama de clase UML

Las variables estáticas (o de clase) se destacan por ir subrayadas. Estas son variables compartidas por todas las instancias de una clase. Una imagen dice más que mil palabras, por lo que la Figura 10.1 muestra un ejemplo.

Como verás, la notación para indicar los tipos de las variables es distinta a lo que se utiliza para programar en `Java`. En general, estos diagramas fueron diseñados para ser utilizados con cualquier lenguaje de programación, por lo que su notación es algo independiente de dichos lenguajes y hay ligeras variaciones dependiendo del software que se use para dibujarlos. Aquí seguiremos el estándar implementado por el software para dibujo de diagramas *Dia*. Es software libre y lo puedes descargar gratuitamente. Descubrirás que puede dibujar mucho más que diagramas de clase.

Generalización

La relación de herencia entre dos clases se conoce como una relación de *generalización*, pues se dice que la clase ancestra *generaliza* a sus clases descendientes. En otras palabras, la clase ancestra contiene datos que todas sus descendientes tienen (aún si no los pueden acceder directamente) y puede realizar operaciones que todas sus descendientes pueden realizar.

En UML la generalización se representa con una flecha con un triángulo vacío como punta, desde la clase que extiende hacia la clase más general [Figura 10.2].

Clases abstractas

Una superclase abstracta, aquella de la cual no es posible crear objetos, se representa escribiendo su nombre con letras itálicas. Si esta clase contiene declaraciones de métodos, sin definición, estos también se escriben con letras itálicas. [Figura 10.3]

Interfaces

En su versión más simple, las interfaces únicamente contienen declaraciones de métodos y no atributos. No todos los lenguajes de programación tienen interfaces como `Java`, por

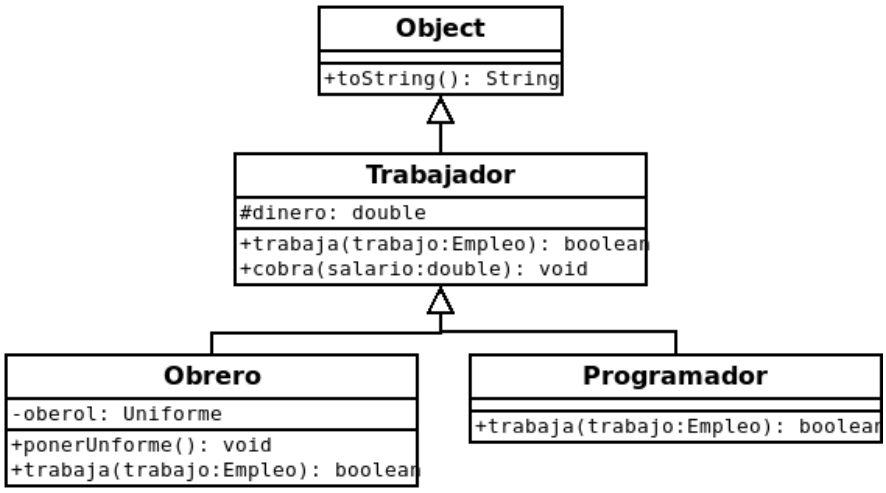


Figura 10.2 Ejemplos de generalización UML. Las clases descendientes heredan atributos y métodos de la clase ancestra, por lo que ya no los tienen que programar.

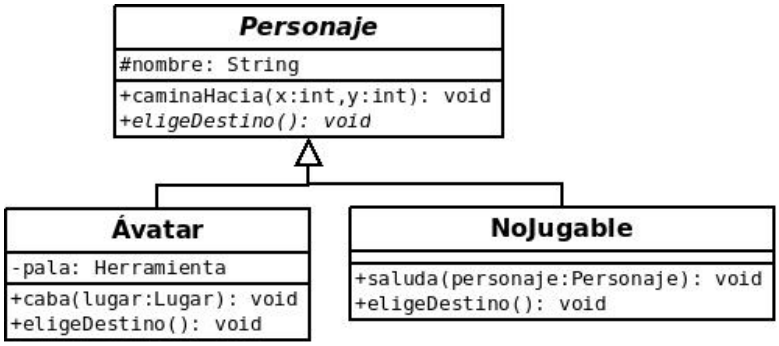


Figura 10.3 Ejemplo de clase abstracta UML. Sólo las clases que la extiendan e implementen sus métodos abtractos pueden ser usadas para crear objetos. Los métodos no abstractos se heredan.

Figura 10.4 Ejemplo de una interfaz en UML

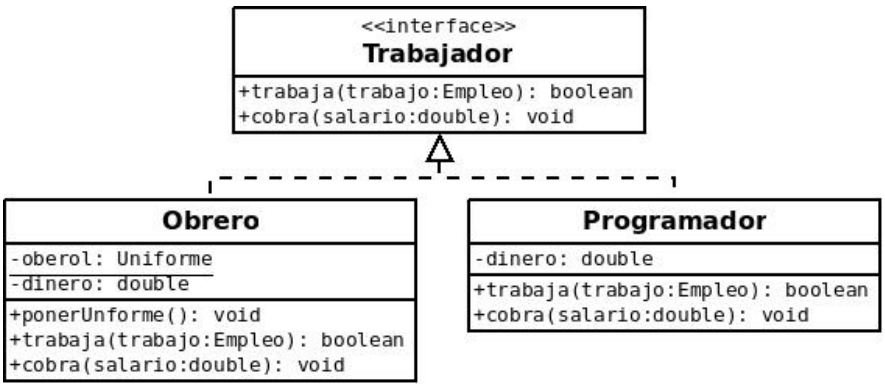


Figura 10.5 Ejemplo de realización UML. Las clases que implementan una interfaz sólo adquieren la obligación de programar los métodos declarados en la interfaz.

lo que es necesario construir un símbolo para ellas a partir de las opciones que ofrece UML. Una opción común se ilustra en la Figura 10.4.

Realización

Curiosamente, UML sí tiene un símbolo para indicar que una clase cumple con un contrato. Cuando una clase implementa una interfaz utilizaremos una flecha semejante a la de herencia, pero con la línea punteada. La Figura 10.5 muestra cómo dos clases implementan una interfaz.

Contención

Las relaciones de *contención* indican que un objeto contiene a otro (como un atributo). Se distinguen dos tipos especiales:

Agregación El objeto contenido puede existir, aunque el contenedor deje de existir. Se representa con un diamante vacío del lado de la clase contendora. Por ejemplo: un `Chofer` existe aún afuera de su coche, el `Coche` sigue existiendo aunque se haya salido su chofer. Pero cuando esté manejando el coche, el chofer estará dentro de él [Figura 10.6].

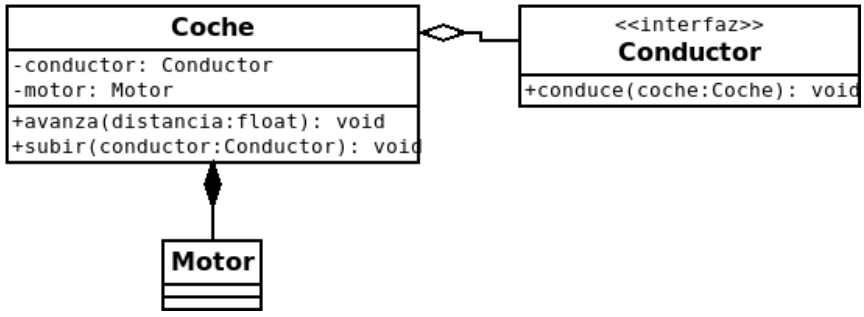


Figura 10.6 Ejemplos de contención UML. Agregación entre `Coche` y `Conductor`. Composición entre `Coche` y `Motor`.

Figura 10.7 Ejemplos de dependencia UML.

Composición El objeto contenedor está hecho de los objetos que contiene. Se representa con un diamante lleno del lado de la clase contendora. El ejemplo más tradicional son el cuerpo y las partes del cuerpo (brazos, piernas, torso, cabeza). También el coche está hecho de motor, carrocería, puertas, etc. y si falta alguno de estos componentes ya no tenemos un coche completo.

Dependencia

La relación de *dependencia* indica que una clase depende del trabajo realizado por otra (por ejemplo, de la respuesta que calcule cuando mande llamar a un método suyo) o de la estructura que define. En su forma más general, esta relación se representa con una flecha cuya punta está dibujada con dos líneas, desde la clase que ocupa el servicio hacia la que lo proveé. De hecho, la generalización y la realización son tipos particulares de relaciones de dependencia, que tienen asignada su propia variante en la notación. [Figura 10.7]

Asociación

Cuando dos clases están relacionadas de cualquier forma, se dice que están asociadas. La forma más genérica de representar una asociación es con una línea uniendo ambas clases. [Figura 10.8]

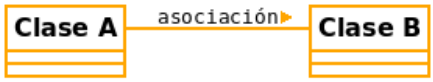


Figura 10.8 Ejemplos de dependencia UML.

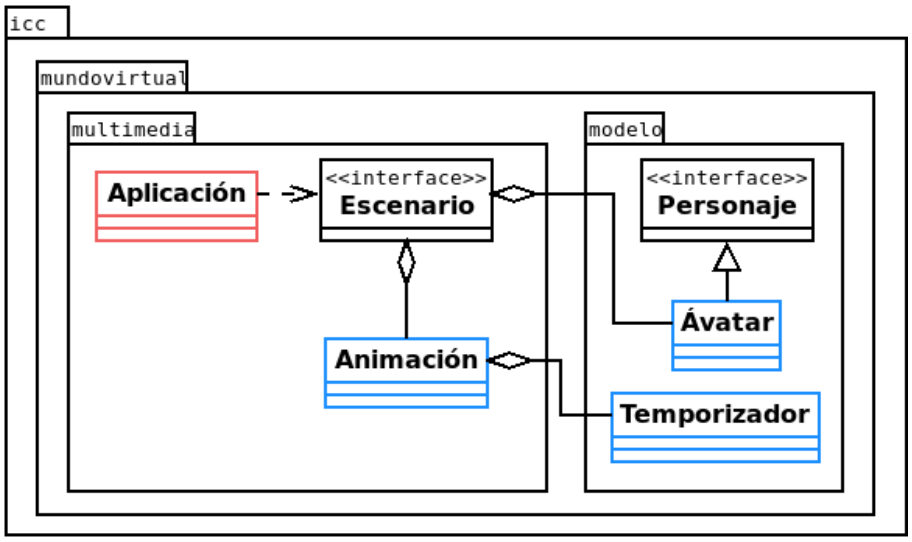


Figura 10.9 Ejemplos de paquetes UML con clases e interfaces en ellos.

Paquetes

Por último, si queremos ilustrar las relaciones entre las clases de un sistema complejo, querremos indicar la agrupación de sus componentes en paquetes con funcionalidades distintas. El símbolo para los paquetes es un folder con el nombre del paquete en su pestaña. [Figura 10.9]

El estándar de UML contiene muchos más símbolos y formas de utilizar la notación, que resultan útiles durante el proceso de ingeniería del software; sin embargo estos serán suficientes para que trabajemos ahora. Observa también que la cantidad de detalles que se incluyen en el diagrama puede facilitar o dificultar su lectura. Cuando se utilizan para documentar un proyecto es importante buscar un buen balance entre la expresividad[1] del diagrama y la claridad de lo que ilustra. Algunos paquetes de software también permiten generar cógido a partir de estos diagramas y, para que el código sea correcto, es necesario seguir las restricciones impuestas por ese paquete.

[1] Es decir, cuánta información proveé.

Listas

Un tipo de dato especial son las *colecciones*. Las colecciones son tipos de datos que nos sirven para almacenar varios objetos de algún otro tipo dado. Un ejemplo de colección es una *lista*.

Definición 10.1: Lista

Una *lista* es una secuencia de cero a más elementos **de un tipo determinado** T. Se representa como una sucesión de elementos separados por comas:

$$a_0, a_1, ..., a_{n-1} \tag{10.1}$$

donde $n \geqslant 0$ y cada a_i es de tipo T.

- Al número n de elementos se le llama *longitud* de la lista.
- a_0 es el *primer elemento* y a_{n-1} es el *último elemento*.
- Si $n = 0$, se tiene una **lista vacía**, es decir, que no tiene elementos.

(**Aho1983**)

La API de `Java` ya incluye una clase que nos permitirá crear listas. El código siguiente nos muestra cómo crear una lista, agregar objetos en ella, recuperarlos e imprimirlos en pantalla.

Listado 10.4: Listas.java

```
1  import java.util.List;
2  import java.util.LinkedList;
3
4  public class Listas {
5    public static void main(String[] args) {
6      // Crea un objeto tipo lista de cadenas
7      List<String> lista = new LinkedList<>();
8
9      // Guarda cadenas en la lista
10     lista.add("-¡Hola!␣");
11     lista.add("Buenos");
12     lista.add("␣días");
13
14     // Imprime las palabras en la lista
15     for(String palabra : lista) {
16       System.out.print(palabra);
17     }
18
19     // Lee la palabra en la posición 1
20     String unaPalabra = lista.get(1);
21     System.out.format("%nLa␣palabra␣en␣la␣posición␣1␣es␣\"%s\"%n",
```

```
                        unaPalabra);
    }
}
```

Al compilar y ejecutar este código se obtiene:

```
-¡Hola!Buenos días
La palabra en la posición 1 es "Buenos"
```

Observa que para utilizar las listas es necesario importar dos elementos desde un paquete de la API: la intefaz `List`, que define el comportamiento del tipo de dato abstracto lista y la clase `LinkedList`, que es una forma de implementar listas. También podrá llamarte la atención que se utiliza la notación <> para especificar de qué tipo son los datos que se guardarán en la lista. Esta notación corresponde al uso de una *variable de tipo* y lo que recibe como valores son, como su nombre lo indica: tipos, concretamente tipos clase.

Desarrollo

En esta práctica programarás los algoritmos para calcular las casillas a las cuales se pueden mover las piezas de un juego de ajedrez, sin considerar los casos cuando capturarán una pieza enemiga[2]. Para ello, la Figura 10.10 muestra el lenguaje estándar para indicar la posición de una pieza en el tablero de ajedrez. Para los fines de esta práctica, no necesitarás modelar el tablero, sólo las piezas. Por otro lado, la Figura 10.12 muestra los movimientos válidos para cada pieza.

Ejercicios

Trabajarás en un paquete llamado `icc.ajedrez`, todas tus clases irán dentro de él. Copia un archivo `build.xml` de alguna práctica anterior y modifícalo para que funcione con los archivos de esta práctica. Observa que tendrás más de una clase con método `main` por lo que necesitarás agregar los comandos correspondientes. Escribe en el archivo `README.md` las instrucciones para utilizar tu programa.

1. Implementa una pequeña clase `Posición` que contenga las dos cantidades: renglón y columna. Agrega un constructor y los métodos de lectura (*getters*) y escritura (*setters*) que consideres necesarios. Asegúrate de que, cada vez que se asignen valores a `renglón` o `columna`, sólamente se acepten valores válidos. Si los valores de entrada no lo son, lanza una `IllegalArgumentException`. Es decir que, para

[2]Esta práctica está inspirada en 11.1.Ajedrez (López Gaona 2012, pág. 100)

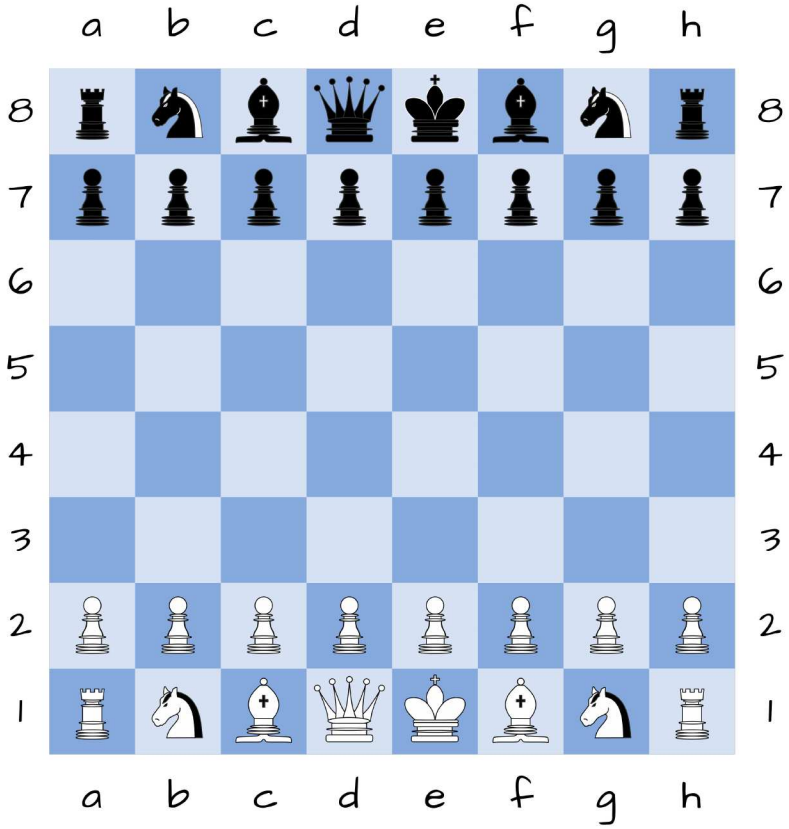


Figura 10.10 Etiquetado de las casillas en un tablero de ajedrez.

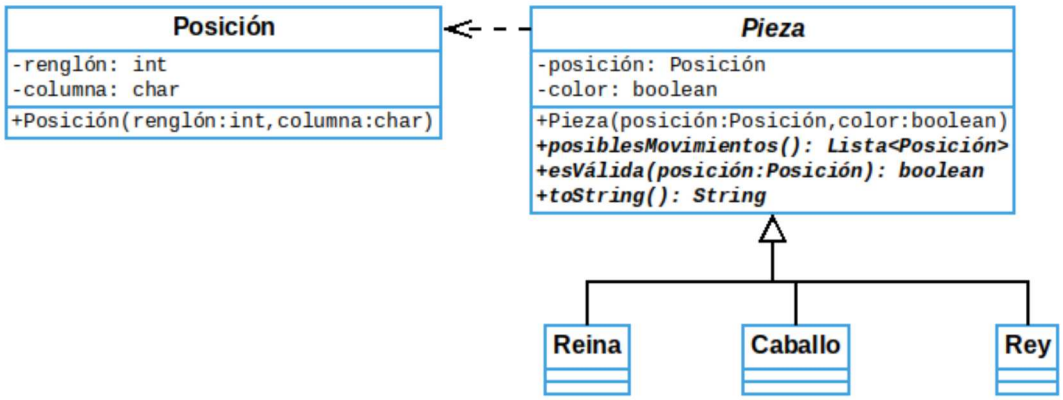
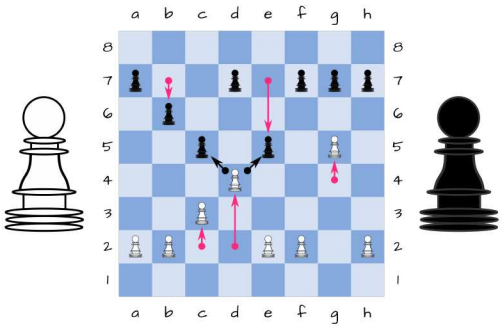


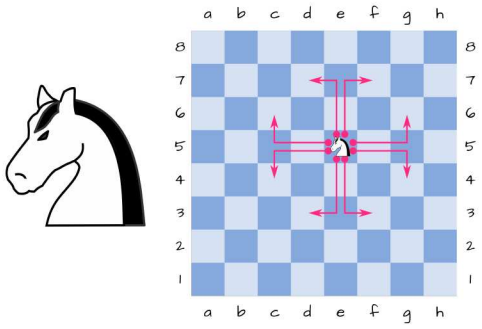
Figura 10.11 Diagrama de clases para las piezas de ajedrez.

Peón



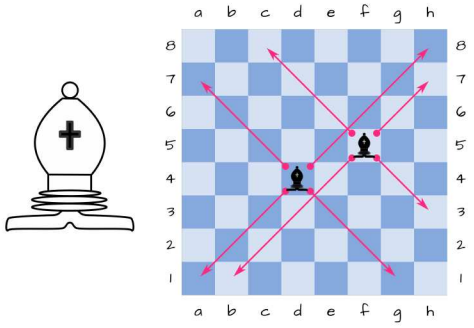
(a) Sólo avanza una casilla hacia el frente (depende del color), excepto en su primer movimiento, que puede moverse hasta dos casillas. Captura en diagonal.

Caballo



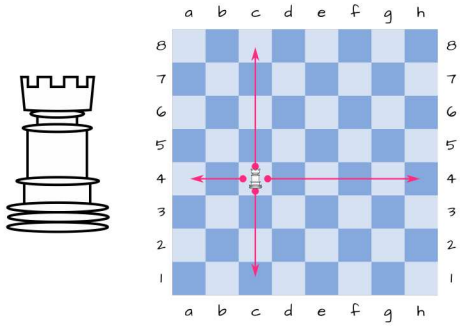
(b) Captura la pieza en la casilla al final de su movimiento. Salta otras piezas en su camino.

Alfil



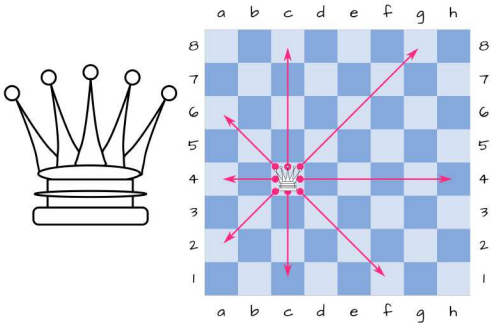
(c) Captura la pieza en la casilla al final de su movimiento. No puede saltar piezas.

Torre



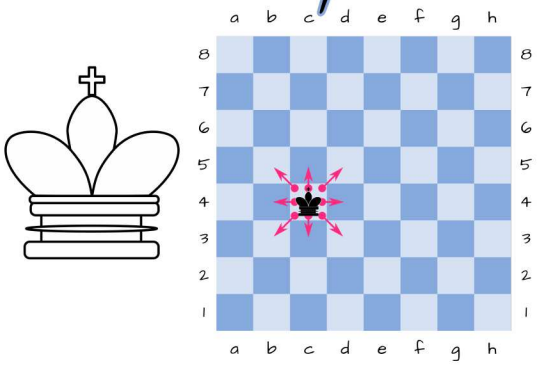
(d) Captura la pieza en la casilla al final de su movimiento. No puede saltar piezas.

Reina



(e) Captura la pieza en la casilla al final de su movimiento. No puede saltar piezas.

Rey



(f) Captura la pieza en la casilla al final de su movimiento. No puede saltar piezas.

Figura 10.12 Movimientos de las piezas de ajedrez.

renglón, los valores deben ser enteros entre 1 y 8, mientras que las columnas son los caracteres $\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{g}, \mathtt{h}$.

Crea una clase llamada `PruebaPosición`, con un método `main` donde crees objetos de tipo `Posición`. Intenta asignarle valores válidos e inválidos. Verifica que su comportamiento sea el correcto. Para los casos en que se debe lanzar una excepción, comenta la línea de código correspondiente para realizar la siguiente prueba. Cuando entregues tu código déjalas ahí, pero comentadas.

2. Implementa la clase `Pieza`, debe ser abstracta pues incluirá tres métodos abstractos:

   - `posiblesMovimientos()` Dada la posición actual de la pieza, devuelve una `Lista` con las posiciones de todas las casillas a las cuales se podría mover esa pieza. Aquí es donde la clase `Posición` resultará particularmente útil.
   - `esVálida(Posición posición)` Indica si sería válido mover a la pieza desde su posición actual hasta la posición indicada en los parámetros. Debe tomar en cuenta, en particular, que la pieza no se salga del tablero.
   - `toString()` Devuelve una representación con cadena de la pieza y su estado actual. Observa que, si lo diseñas con cuidado, puedes implementar este método en la superclase.

   Agrega los métodos de lectura y escritura que consideres necesarios. El diagrama UML de la Figura 10.11 ilustra la relación entre las clases que programarás.

   Nota: Para el tipo de dato `Lista<>` utiliza la interfaz `List<>` y su implementación `LinkedList<>` de la API de `Java`.

3. Programa la clase `Reina`. Según las reglas del juego puede desplazarse a cualquiera de las casillas en línea recta, horizontal o diagonal a partir de su posición, tanto hacia adelante como hacia atrás. En el mejor de los casos hay hasta 27 posiciones posibles.

4. Programa la clase `Caballo`. El caballo se puede desplazar en forma de L: dos casillas verticalmente y una horizontal o a la inversa. Puede tener hasta 8 posiciones a las cuales moverse.

5. Programa la clase `Rey`. El rey puede moverse una sola casilla en las ocho direcciones, por lo que hay máximo ocho posiciones.

   Para completar el programa, habría que agregar `Torre`, `Alfil` y `Peón`, pero sólo te serán útiles si de verdad quieres programar el juego y para eso aún falta ver cómo programar el tablero y determinar las capturas. De momento no haremos eso.

6. Agrega una clase de uso, `UsoAjedrez`, donde crees una reina, un caballo y una torre. Pide que genere las posiciones posibles a partir de celdas en el centro y orilla del tablero; así como que indique si la posición es válida a partir de la posición actual tanto para propuestas válidas, propuestas que no estén sobre una casilla

válida y posiciones fuera del tablero (ej $[-2, \mathrm{i}]$). Observa que, para cambiar la casilla actual de las piezas, deberás haber programado los métodos de escritura (*setters*) correspondientes y estos deben impedir que se asignen posiciones fuera del tablero.

7. Se darán dos puntos extra si programas una interfaz de usuario que permita realizar lo anterior mediante un menú para el usuario. Se otorgarán fracciones de estos puntos si la interfaz permite realizar sólo algunas de estas tareas.

Entregables

Los ejercicios de la sección anterior.