Programación para Ingeniería Telemática

Curso 2018-19

Práctica 2: Clases, interfaces. Herencia y composición.

1. Objetivos de aprendizaje. Estructura de la práctica.

Objetivos

- Presentar el IDE NetBeans.
- Presentar una visión general de las prácticas correspondientes al tema de orientación a objetos.
- Repasar conceptos de algoritmia y de programación orientada a objetos (interfaces, clases, herencia, composición y *downcasting*).

Contenidos de la memoria

Esta memoria se estructura en cinco secciones con los siguientes contenidos:

Sección 1: Objetivos y contenido de la práctica.

Sección 2: Presentación de la aplicación a implementar. Las prácticas se van a presentar a partir de ahora como el progresivo desarrollo de un juego. En esta sección se describe brevemente el juego a desarrollar, sus personajes y sus variantes, así como los temas del curso relacionados con cada una de las partes de la aplicación final. Se trata de mostrar hacia dónde vamos y el camino a seguir.

Sesión 3: Proyectos con NetBeans. Las prácticas con Java se van a realizar con el IDE NetBeans. En esta parte se dan unas breves explicaciones de cómo crear y gestionar un proyecto en este entorno de desarrollo.

Sección 4: Los paquetes `common` y `game`. En esta sección se presenta el código de partida que tendrá que usarse para realizar esta práctica en concreto. Este código se irá ampliando en sucesivas prácticas.

Sección 5: Enunciado de los ejercicios a realizar. Estos ejercicios están pensados para facilitar la realización de la práctica entregable correspondiente al bloque de la asignatura dedicado a la programación orientada a objetos. Además, los ejemplos y ejercicios que se vayan mostrando en las prácticas y o bien son semejantes a los que pueden pedirse en los exámenes de la signatura o bien constituyen un paso previo para ser capaces de realizarlos.

Antes de hacer los ejercicios es aconsejable tener una visión de conjunto, por ello **se recomienda leer detenidamente el boletín de prácticas y entender el código suministrado antes de abordar la resolución de los ejercicios.**

2. Presentación de la aplicación a implementar.

En esta sección se va a presentar el juego a desarrollar en esta práctica y en las tres siguientes. En cada una de estas prácticas se irán definiendo nuevos elementos del juego o se mejorarán los ya existentes.

La programación de juegos tiene ciertas características que la hacen muy interesante para la enseñanza de la programación. En primer lugar, los juegos son programas complejos que permiten poner en práctica casi cualquier técnica de programación. En segundo lugar, esta complejidad puede abordarse de forma progresiva, empezando por un juego muy sencillo al que se le pueden ir añadiendo nuevos elementos. A medida que la complejidad del juego crece aparece de forma natural la necesidad de emplear nuevas técnicas de programación. Cada vez que se presenta un nuevo concepto o una nueva técnica de programación es posible definir una nueva característica en el juego en donde tales técnicas y conceptos se pongan inmediatamente en práctica. De esta manera, es posible alinear el desarrollo del juego con el avance de la asignatura.

Además, la programación de un juego permite poner en práctica un concepto didáctico muy interesante desde el punto de vista de la motivación: "la orientación al logro". Cada nueva característica del juego supone un reto y su logro una recompensa. Las técnicas de programación se aprenden para mejorar el juego. El juego proporciona un contexto donde esos logros se visualizan de forma casi inmediata. De alguna manera, *programar juegos es también un juego*.

2.1. El juego.

El juego consiste en un personaje, Caperucita (Little Red *Riding Hood*), que se mueve por un damero recogiendo flores de diferentes tipos, mientras evita que otros personajes, moscas, abejas y arañas, la alcancen. En la figura 1 se muestra un escenario del juego, con flores, dos arañas y Caperucita[1]. La misión de Caperucita es recoger flores y evitar ser "recogida" por las arañas u otros bichos. Cada vez que Caperucita recoge una flor aumentan sus puntos, cada vez que pasa de pantalla aumentan sus vidas. Cada vez que un bicho pilla a Caperucita, las vidas de Caperucita disminuyen. El juego acaba cuando Caperucita se queda sin vidas.

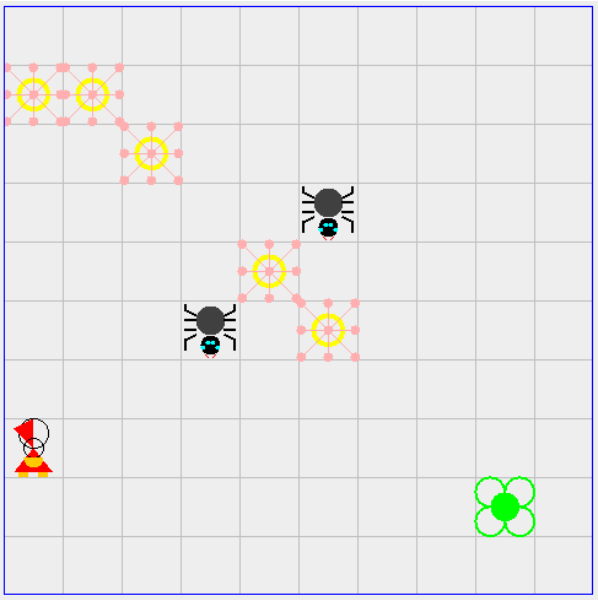


Figura 1: Escenario del juego.

[1] Obviamente, a quien no le guste el personaje de Caperucita puede cambiarlo por otro.

2.2. El desarrollo del juego en las prácticas.

- En la **Práctica 2** (esta práctica) abordaremos la definición de los personajes usando clases e interfaces. Esto nos permitirá aplicar los conceptos de herencia y composición. Adicionalmente repasaremos algunos conceptos de algoritmia a través de la implementación de métodos para serializar los objetos en formato JSON y recuperarlos. También revisaremos el concepto de *downcasting*.
- En la **práctica 3** implementaremos algunas características de los personajes y veremos cómo guardarlos en un fichero en formato texto (JSON) y posteriormente recuperarlos. La entrada y salida en fichero nos proporcionará un primer contacto con el paquete `java.io` y nos permitirá trabajar con excepciones. Lo que aprendamos en esta práctica nos servirá posteriormente para guardar y recuperar partidas e incluso para implementar una versión distribuida del juego en la última parte del curso.
- En la **práctica 4** definiremos los gráficos del juego. Primero se presentarán una serie de ejemplos de interfaces gráficas en los que se mostrará cómo definir menús y botones, cómo organizar los elementos de la interfaz, cómo capturar eventos del teclado y de un temporizador y, finalmente, cómo dibujar o volcar imágenes sobre el panel de una ventana. A continuación definiremos se pedirá que los estudiantes apliquen estos conocimientos para realizar un editor de escenarios de juego.
- En la **práctica 5** abordaremos genéricos, listas, expresiones lambda y factorías de objetos. Parte del código definido en las prácticas anteriores tendrá que ser reescrito para aprovechar la capacidad expresiva de los nuevos conceptos aprendidos en teoría. En esta práctica se definirá además una estructura básica para la programación del juego.
- En la **práctica 6** se pedirá al alumnado que termine el desarrollo el juego. Esta práctica es un entregable evaluable.

3. El IDE Net Beans

3.1. NetBeans

Un IDE (*Integrated Development Environment*) es un conjunto integrado de herramientas de desarrollo para realizar programas (en la figura 2 se muestran algunos ejemplos). Los IDE incluyen de forma coherente todo lo que hace falta para desarrollar programas (editores, compiladores, enlazadores, depuradores, gestores de proyecto, etc.). Frecuentemente, son entornos abiertos en los que pueden incorporarse herramientas desarrolladas por terceros. Cada programador elige su entorno de desarrollo en función de sus necesidades y preferencias.

IDE	Página web	Lenguajes soportados	Sistemas operativos
Eclipse	http://www.eclipse.org/	C, C++, Java, JSP, Phyton, PHP	Windows, Linux, Mac OSX
Codelite	http://www.codelite.org/	C, C++	Windows, Linux, Mac OSX
NetBeans	http://netbeans.org/index.html	C, C++, Java, JSP, PHP, Groovy	Windows, Linux, Solaris, Mac OSX
XCode	http://developer.apple.com/xcode/	C, C++, Objective-C	Mac OSX

Figura 2: Algunos IDEs de libre distribución.

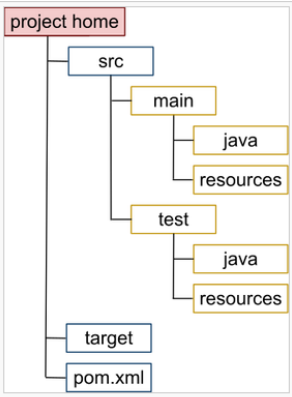
En esta asignatura vamos a utilizar **NetBeans** por su fácil integración con el gestor de proyectos **Maven** para el desarrollo de aplicaciones en Java. De hecho Maven viene integrado *de serie* con NetBeans

Descarga de NetBeans: https://netbeans.org/downloads/index.html

Descarga Java JDK actualizado: Conviene además actualizar la versión de Java (http://java.com/es/)

Maven es una herramienta para gestionar las dependencias de un proyecto, es decir para determinar qué librerías se necesitan para construir tal proyecto, encontrarlas y añadirlas al proyecto, y finalmente enlazarlas para obtener un nuevo ejecutable o una nueva librería lista para ser utilizada en otro proyecto. Maven también proporciona una estructura por defecto para la organización de los ficheros asociados a un proyecto (código fuentes, librerías, ficheros de configuración, datos, etc.) que se está convirtiendo en un estándar de facto para el desarrollo de aplicaciones en Java (Figura 3).

Directory name	Purpose
project home	Contains the pom.xml and all subdirectories.
src/main/java	Contains the deliverable Java sourcecode for the project.
src/main/resources	Contains the deliverable resources for the project, such as property files.
src/test/java	Contains the testing Java sourcecode (JUnit or TestNG test cases, for example) for the project.
src/test/resources	Contains resources necessary for testing.



The Maven software tool auto-generated this directory structure for a Java project.

Figura 3a): Estructura de directories de un Proyecto Maven

```xml
<project>
  <!-- model version is always 4.0.0 for Maven 2.x POMs -->
  <modelVersion>4.0.0</modelVersion>

  <!-- project coordinates, i.e. a group of values which
       uniquely identify this project -->

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0</version>

  <!-- library dependencies -->

  <dependencies>
    <dependency>

      <!-- coordinates of the required library -->

      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>

      <!-- this dependency is only used for running and compiling
tests -->

      <scope>test</scope>

    </dependency>
  </dependencies>
</project>
```

Figura 3b): `pom.xml` file: Maven projects are configured using a **Project Object Model**, which is stored in a **`pom.xml`-file**.

Figura 3: Estructura de un proyecto con Maven (Wikipedia).

3.2. Creación de un proyecto con NetBeans.

La figura 4 muestra un extracto de la interfaz de usuario de NetBeans, en la que se han resaltado tres componentes: el árbol de directorios del proyecto (arriba a la izquierda), el editor de Java (derecha) y una ventana (abajo izquierda) para visualizar y navegar por las variables y métodos de una clase dada.

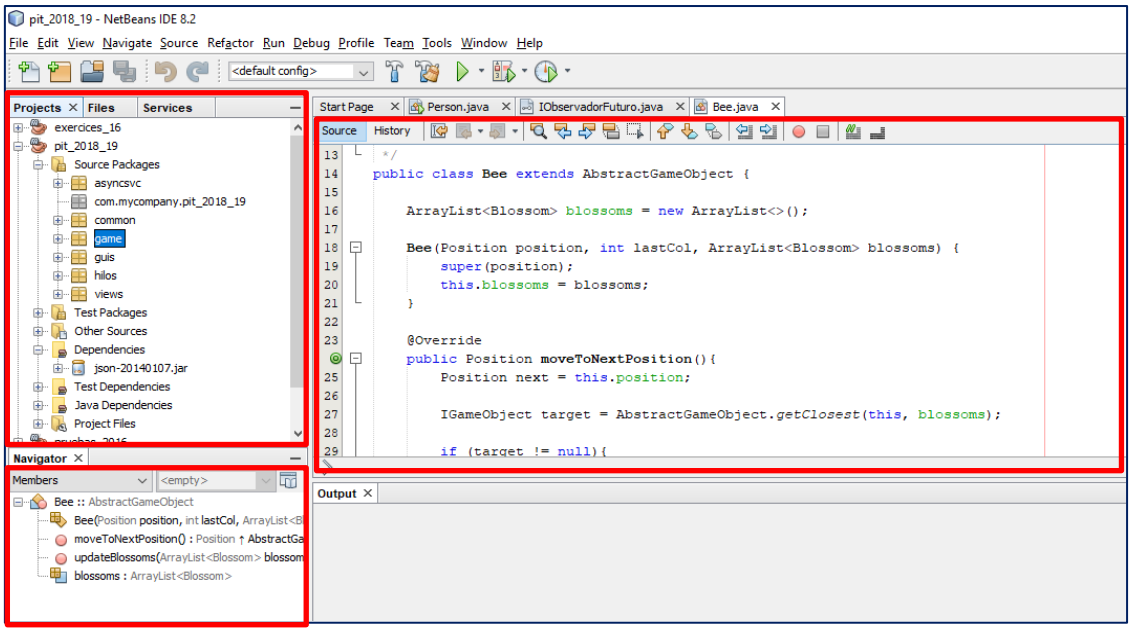


Figura 4: Extracto de la interfaz de usuario de NetBeans.

Para crear un nuevo proyecto, se selecciona File → New Project. En la ventana emergente (Figura 5) se selecciona a la izquierda *Maven* y a la derecha *Java Application*. Se pulsa entonces *Next* y aparece una nueva ventana en la que definimos el nombre del proyecto (pit2019) y su localización (en *Project Location*). Finalmente se pulsa *Finish* y se crea el proyecto con la configuración seleccionada.

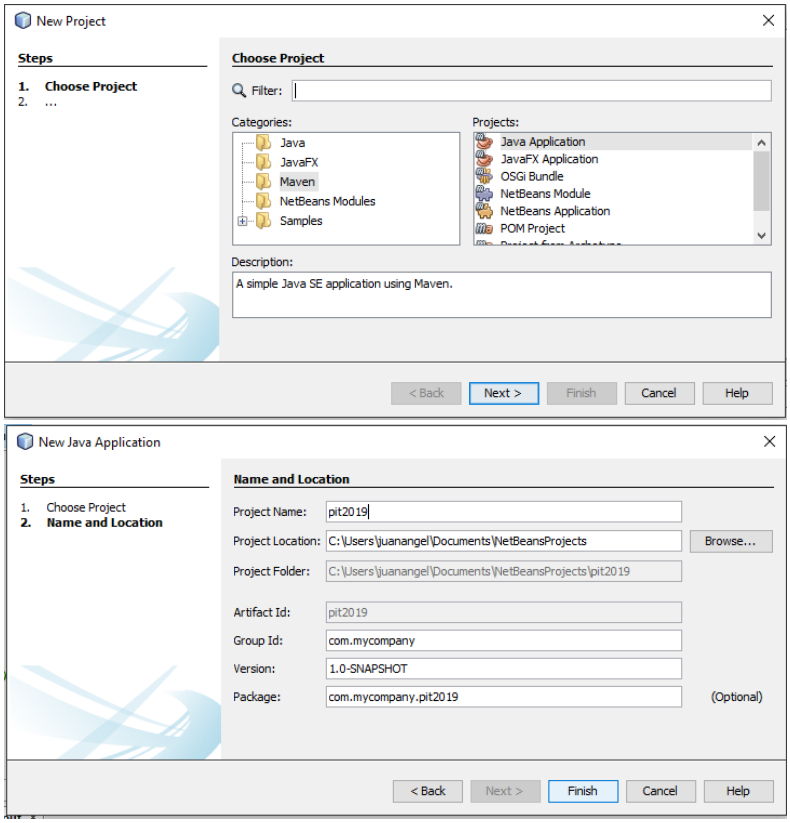


Figura 5: Creación de un proyecto de una aplicación Java gestionada con Maven

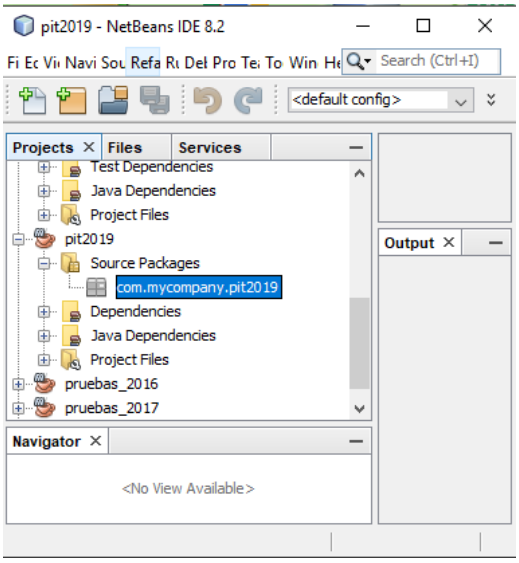
En la Figura 6, a la izquierda, puede verse el proyecto creado, con la estructura de directorios desplegada, pero vacía. Durante el curso iremos completando este proyecto a partir de los ficheros fuente proporcionadas en cada práctica y del código desarrollado en el transcurso de las prácticas. Veamos en primer lugar como incorporar al proyecto los paquetes suministrados a través del aula virtual.

4. Los paquetes common y game

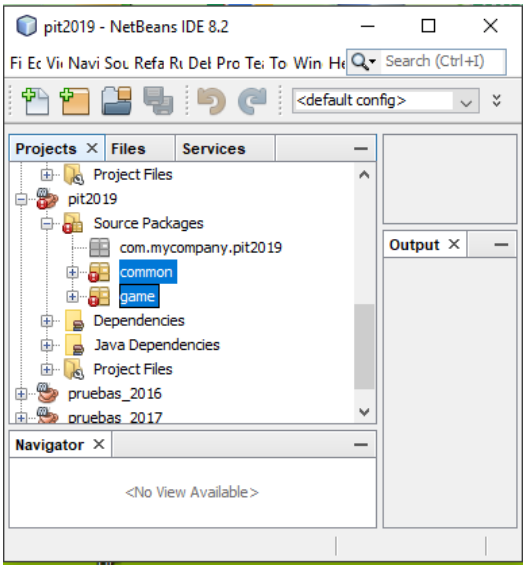
Los paquetes `common` y `game`, disponibles en el Aula Virtual, son los paquetes donde se va a ubicar el código fuente de las prácticas 2 a 6. Estos paquetes irán cambiando de práctica en práctica y será necesario descargar la versión correspondiente del aula virtual. Las actualizaciones irán incorporando el código (clases e interfaces) correspondientes tanto a las soluciones de ejercicios de prácticas anteriores como a nuevos recursos que harán falta para solucionar los nuevos ejercicios que se vayan proponiendo. En el manual de cada práctica se irán explicando estos nuevos recursos.

4.1. Integración en el proyecto de los paquetes common y game.

Copie las carpetas `common` y `game`, una vez descomprimidas, en el directorio que cuelga de *Project Location* destinado al código fuente (`../pit2019/src/main/java`). El IDE se actualiza de forma inmediata (Figura 5, a la derecha). Se puede observar también que el IDE marca la presencia de errores (círculos rojos en `common` y `game`).



Proyecto recién creado. Carpetas vacías.



Proyecto con los paquetes `common` y `game` copiados en `/src/main/java`

Figura 5: Creación del proyecto `pit2019`.

Si abrimos la carpeta `common`, vemos que el problema está en `IToJsonObject.java`: el IDE no reconoce las librerías de `org.json`[2] (Figura 6, arriba a la izquierda). Agregar estas librerías en un proyecto Maven es muy sencillo. Basta con añadir la dependencia. Para ello nos colocamos encima de la carpeta *Dependencies*, pulsamos el botón derecho del ratón y aparece un menú en el que elegimos *Add Dependency* (Figura 6, arriba a la derecha). Aparece entonces una ventana emergente en la debemos especificar el *grupo*, *artefacto* y *versión* que estamos buscando (Figura 6, abajo a la

[2] https://mvnrepository.com/artifact/org.json/json

izquierda). Estos datos se proporcionarán a los alumnos a medida que hagan falta, pero en general, no son difíciles de obtener a través de búsquedas en la web[3]. En este caso son:

```
version=20140107
groupId=org.json
artifactId=json
```

Añadimos la dependencia rellenado los campos correspondientes y pulsando *Add*. Inmediatamente se añade la librería de la que depende nuestro proyecto y los errores desaparecen.

Si no conocemos el grupo, artefacto y versión de la librería podemos actuar de otra manera y el propio IDE nos ayudará a incorporar las dependencias de forma automática. Si nos colocamos encima del círculo rojo que aparece en el editor de Java (Figura 6, derecha, línea con el `import`) y pulsamos el botón izquierdo del ratón, el sistema nos sugiere que busquemos la dependencia. Si se hace click en la sugerencia, aparece una ventana emergente con los recursos que necesita el IDE (Figura 6, abajo a la derecha). Los añadimos pulsando *Add*. Inmediatamente se añade la librería de la que depende nuestro proyecto y los errores desaparecen.

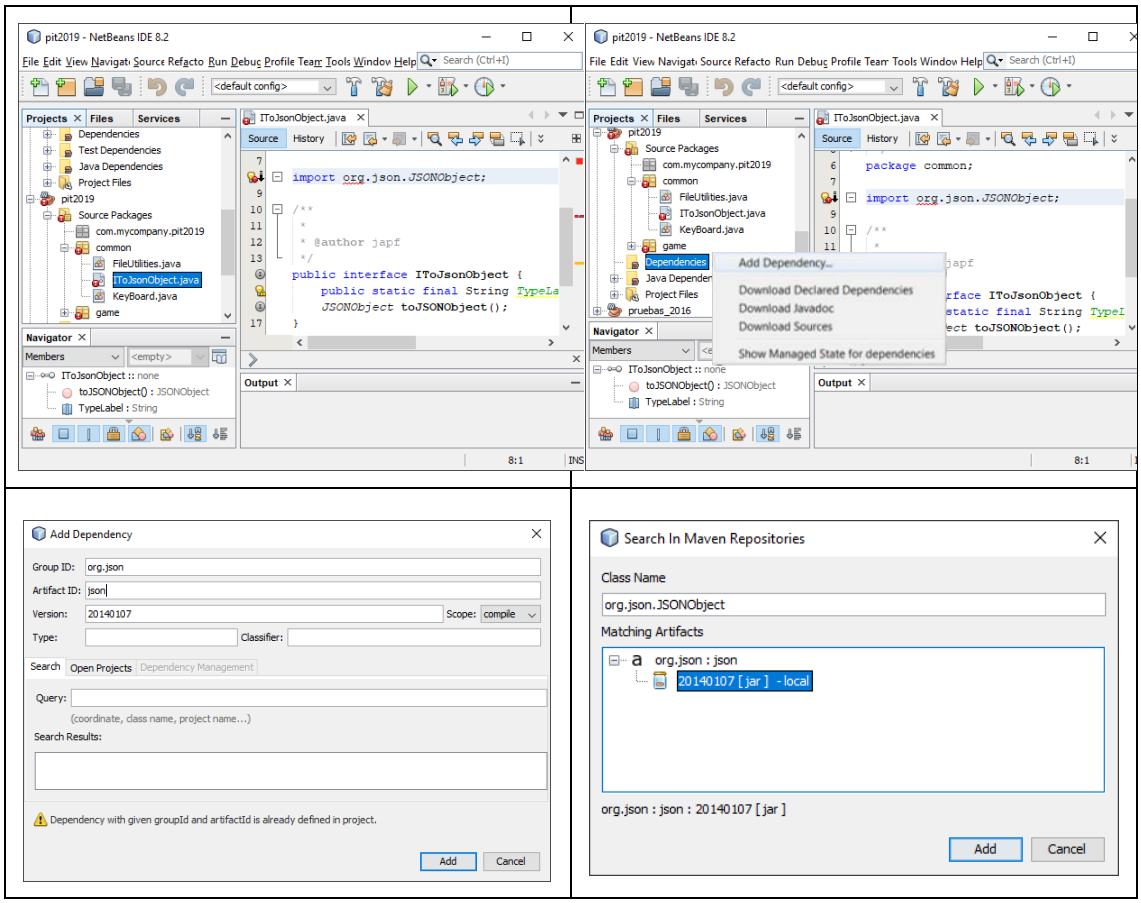


Figura 6: Adición de dependencias de json.org en proyecto `pit2019`.

[3] https://mvnrepository.com/artifact/org.json/json

4.2. El paquete common

El paquete `common` contiene una interfaz y una clase.

- La clase `KeyBoard` se corresponde con la clase `Teclado` utilizada en las prácticas de Fundamentos de Programación y consiste en una serie de métodos estáticos para leer datos de tipos primitivos y cadenas de caracteres (*strings*) introducidos desde el teclado.
- La interfaz `IToJsonObject` será implementada por todas las clases cuyos objetos puedan serializarse en formato JSON, a efectos de ser guardados en un fichero de texto o enviados a través de un socket (en las últimas prácticas del curso). La interfaz define:
  - El método `JSONObject toJSONObject()`, que devuelve un objeto del tipo `JSONObject` (definido en el paquete `org.json`) que contiene la serialización del objeto original en formato JSON. El objeto `JSONObject` proporciona métodos para acceder a los campos de la estructura JSON contenida o añadirle otros nuevos.
  - La constante `TypeLabel` que define el nombre del campo del objeto JSON donde se guardar el nombre de la clase a la que pertenece el objeto serializado.

4.3. El paquete game

El paquete `game` correspondiente a esta práctica contiene las interfaces y clases que se muestran en la Figura 7 y que se comentan brevemente a continuación (excepto `IToJsonObject` que está definida en `common`). Los detalles pueden consultarse en el propio código fuente.

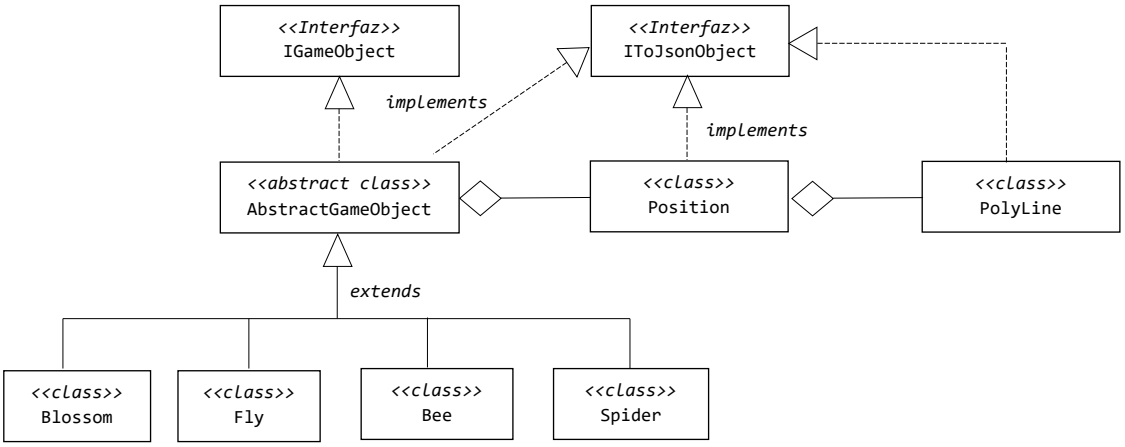


Figura 7: Tipos de datos

- La clase **`Position`** define una posición dentro de un damero (x, y), donde x se corresponde con la columna e y se corresponde con la fila (Figura 8).

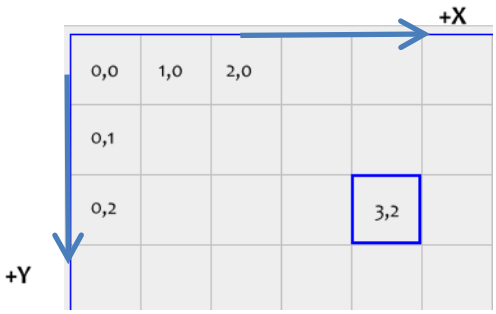


Figura 8: Modelado de posiciones.

- La clase **`Polyline`** define un conjunto de posiciones consecutivas. El objetivo de esta clase es simplemente servir de ejemplo para ayudar a la realización de los ejercicios que se piden en esta práctica, especialmente la serialización de listas y la construcción de objetos compuestos de otros objetos a partir de su representación en JSON.

- La interfaz **`IGameObject`** define una interfaz común para todos los elementos de un juego, que se describe brevemente en la Tabla 1.

Tabla 1: métodos de `IGameObject`	
Métodos para consultar o cambiar la posición del objeto en un damero (x, y)	
`Position getPosition();`	Devuelve la posición actual del objeto.
`void setPosition(Position position);`	Fija la posición del objeto.
Método para que el objeto decida su nueva posición de acuerdo a un algoritmo.	
`Position moveToNextPosition();`	El objeto actualiza su posición de acuerdo a un algoritmo.
Métodos para determinar el valor del objeto. Su significado puede variar de un juego a otro (puntos acumulados, tiempo de vida, …)	
`int getValue();`	Devuelve el valor del objeto.
`public void setValue(int value);`	Fija el valor del objeto
Métodos para determinar las vidas del objeto. Si este valor es menor que cero el objeto debería ser eliminado del juego.	
`public int getLifes();`	Devuelve el valor del objeto.
`public void incValues(int value);`	Incrementa el número de vidas según el valor del argumento, que puede ser negativo.
Método para Informar al objeto del modo de juego. Tanto si el personaje como el juego en su conjunto admiten diferentes comportamientos en función del desarrollo del juego, este método informa al objeto del modo actual, de forma que ajuste su comportamiento a dicho modo.	
`void setGameMode(int mode);`	Informa al objeto del modo de juego.

- La clase abstracta **`AbstractGameObject`** proporciona una implementación por defecto de todos los métodos de `IGameObject`, sin embargo, se marca como abstracta para que no puedan instanciarse directamente objetos de la misma, sino de sus subclases. También proporciona algunos métodos de utilidad estáticos (Tabla 2) que, como se explica más adelante deben implementarse como parte de esta práctica, así como sus constructores (Tabla 3).

Tabla 2: Métodos estáticos de `AbstractGameObject`

Método	**Descripción**
`static double` **`distance`**`(` `Position p1, Position p2)`	Obtiene la distancia entre dos posiciones.
`static double` **`getDistance`**`(` `IGameObject jObj1, IGameObject jObj2)`	Obtiene la distancia entre dos elementos del juego.
`static IGameObject` **`getClosest`**`(` `Position p, IGameObject jObjs [])`	Obtiene el elemento del juego contenido en `jObjs` más cercano a la posición `p`
`static IGameObject` **`getClosest`**`(` `IGameObject jObj, IGameObject jObjs [])`	Obtiene el elemento del juego contenido en `jObjs` más cercano al elemento `jObj`.

Tabla 3: Constructores de `AbstractGameObject`

Constructor	**Descripción**
`AbstractGameObject``()`	Constructor sin argumentos
`AbstractGameObject``(` `Position position, int value, int life)`	Constructor que toma como argumentos la posición, el valor y la vida del elemento del juego.
`AbstractGameObject``(JSONObject obj)`	Constructor que toma como argumentos la representación JSON del elemento del juego.

- Las clases concretas **`Blossom, Fly, Bee y Spider`** representan a cuatro elementos del juego. Los objetos `Blossom` representan objetos que deben recoger los personajes del juego y los objetos pertenecientes a las otras tres clases son personajes del juego que debe mover el propio juego. La implementación del personaje del juego que debe mover un jugador humano o bien moverse por sí mismo de acuerdo a un algoritmo se dejará como ejercicio.

5. Ejercicios a realizar

Ejercicio 1:
Cree una nueva clase `TestObjects`. El objetivo de esta clase es definir métodos de prueba y un método `main` en el que invocarlos. En los siguientes ejercicios se añadirán métodos a esta clase.

Ejercicio 2:
1. Implemente los constructores de las subclases de `AbstractGameObject`. Las subclases deben ofrecer los mismos constructores que la superclase.
2. Implemente el método `toJSONObject` de la clase `AbstractGameObject`.
3. Pruebe el funcionamiento de `toJSONObject`. Para ello, defina un método `testToJSON` en la clase `TestObjects` en el que se creen objetos de las subclases de `AbstractGameObject` (con el constructor sin argumentos) e invoque el método `toJSONObject` en dichos objetos. El método debe mostrar los objetos JSON obtenidos.

Ejercicio 3:
1. Implemente el constructor de `AbstractGameObject` que toma como argumento un `JSONObject`. Puede inspirarse en la implementación de Position y `Polyline`.
2. Use dicho constructor para implementar en las subclases el método constructor correspondiente.
3. Pruebe el funcionamiento de los constructores que toman como argumento un `JSONObject`. Para ello defina un método `testConstructores` en la clase `TestObjects` en el que:
    3.1. Se creen objetos de las subclases de `AbstractGameObject` mediante los constructores que no toman un `JSONObject` como argumento.
    3.2. Se obtengan las representaciones JSON de los objetos así creados.
    3.3. Se usen tales representaciones para obtener nuevos objetos.
    3.4. Se comparen los objetos obtenidos con unos y otros constructores.

Ejercicio 4:
1. Implemente los métodos estáticos de utilidad de `AbstractGameObject` (`distance`, `getDistance` y `getClosest`)
2. Pruebe el funcionamiento de dichos métodos. Para ello defina un método `testDistances` en la clase `TestObjects` en el que:
    2.1. Se cree un array con cuatro objetos de diferentes subclases de `AbstractGameObject` en posiciones conocidas.
    2.2. Se calcule y se muestre la distancia entre ellos utilizando los métodos `distance` y `getDistance`.
    2.3. Se cree un nuevo objeto de una subclase cualquiera y se muestre en consola la posición del objeto del array más cercano al mismo.

Ejercicio 5:
1. Añada a cada subclase de `AbstractGameObject` un método `printXXX`, donde `XXX` es el nombre de la subclase (`printBlossom`, `printBee`, etc.) que imprima en consola la representación JSON de cada objeto
2. Pruebe el funcionamiento de dichos métodos. Para ello defina un método `testDowncasting` en la clase `TestObjects` en el que:
    2.1. Se cree un array con cuatro objetos de diferentes subclases de `AbstractGameObject`.
    2.2. Se itere sobre el array mostrando en consola la posición de cada objeto y su representación JSON, en este último caso utilizando los métodos definidos en el apartado 1.