Universitat Oberta de Catalunya

PRÁCTICA 2
Relación entre objetos y entre clases

Diseño y programación orientada a objetos

Semestre 20191

Estudios de Informática, Multimedia y Telecomunicación

Presentación

Esta Práctica profundiza en las relaciones entre objetos y clases o, dicho de otro modo, en las relaciones de asociación y herencia (también conocida como generalización/especialización). Para conseguirlo, se usará el lenguaje de programación Java para la implementación de los diferentes programas.

Objetivos

Los objetivos que se desean lograr con el desarrollo de esta Práctica son:

- Poner en práctica a nivel de diseño y de implementación las asociaciones entre objetos.
- Entender y poner en práctica la relación de generalización/especialización (o herencia) entre clases.
- Profundizar en la creación y uso de diagramas de clase UML.
- Practicar la codificación en Java de programas basados en POO.

Enunciado

Esta Práctica está formada por 3 ejercicios. El valor de cada uno de ellos está indicado junto a cada ejercicio.

Esta Práctica está diseñada para que los ejercicios se realicen en el orden en que se presentan. Antes de empezar, lee el módulo 4 de los apuntes. También puedes leer el PDF **"El lenguaje de programación Java"** que encontrarás en el aula.

En cada ejercicio se hace una pequeña introducción que te pone en contexto a la vez que te intenta guiar para hacer el ejercicio correctamente. Esta parte introductoria está escrita en color gris (como este texto), mientras que el ejercicio en sí (i.e. aquello que se te pide y que tienes que contestar) está escrito en azul. De este modo te será más fácil localizar cada parte dentro de cada ejercicio.

Adjunto con este enunciado encontrarás una hoja de respuestas que tienes que rellenar con las respuestas de los diferentes apartados, excepto aquellos en que la respuesta es directamente uno o más ficheros `.java` y/o `.class`.

Para hacer el Ejercicio 3, donde se te pide un diagrama de clases UML, tienes varias opciones: (1) si tienes buena letra, puedes dibujar el diagrama a mano, escanearlo y insertarlo en la hoja de respuestas, (2) utilizar la herramienta de dibujo de Word, de hecho, en la hoja de respuestas te damos un ejemplo de caja de clase para que la utilices, o (3) utilizar una herramienta como ArgoUML (http://argouml.tigris.org/) o DIA (http://dia-installer.de/shapes/UML/index.html.es), exportar del diagrama a imagen (o imprimir pantalla) e insertarla en la hoja de respuestas.

Ejercicio 1 – Excepciones personalizadas (0.75 puntos)

En la clase `Team` hemos usado la clase propia de la API de Java llamada `Exception` para lanzar excepciones cuando se producía algún caso anómalo. Esto es correcto, pero cuando en un programa se dan excepciones que son particulares del problema/contexto que tratamos, es adecuado crear nuestras propias excepciones o excepciones personalizadas (en inglés, *custom exceptions* o *user-defined exceptions*) que sean adecuadas para las necesidades de nuestro programa.

> No haremos nuestras propias excepciones si las que trae Java por defecto ya nos van bien y son bastante significativas.

En nuestro caso haremos excepciones para controlar las anomalías que se puedan dar con los equipos. Evidentemente, como te puedes imaginar, Java no trae excepciones que representen anomalías que se dan con los equipos de baloncesto. Las excepciones de Java son más generalistas.

Llegados a este punto, seguramente te preguntarás*: ¿cómo hago mi propia excepción?* ¿Te suena un mecanismo que permite crear una clase a partir de otra indicando los cambios? Efectivamente, el mecanismo de herencia. Pues así (de sencillo) es como crearemos nuestras propias excepciones, creando una clase que herede de la clase `Exception`. Por lo tanto, en este ejercicio codificarás la clase `TeamException` que heredará de `Exception`. Se te pide seguir los siguientes pasos para codificar la clase `TeamException`:

1. Con Eclipse, abre el proyecto `PRAC2_ex1` que te damos.
2. Una vez abierto el proyecto, crea una clase nueva llamada `TeamException` (fichero `TeamException.java`) que herede de la clase `Exception`.
3. Dentro de la clase `TeamException` codifica su constructor por defecto. En este caso estás heredando de `Exception` y, por lo tanto, debes llamar al constructor por defecto de la clase padre, en este caso, al constructor por defecto de la clase `Exception`.

   > **Nota**: Recuerda, de los apuntes del Módulo 4, que los constructores de las clases hijas (o también llamadas *derivadas* o *subclases*) tienen que llamar de una manera especial al constructor de la clase padre (también llamada *superclase* o *base*).

   (0.25 puntos)
4. Ahora implementa un constructor público con un argumento llamado `msg` de tipo `String`. Este constructor tiene que llamar al constructor con un argumento de tipo `String` que tiene la clase padre `Exception`.

   (0.25 puntos)

5. Ahora modifica la clase `Team` que se te da dentro del proyecto `PRAC2_ex1` para que en vez de usar la clase `Exception` de Java para lanzar las excepciones use la nueva clase `TeamException`. Dicho de otro modo, donde en `Team.java` pone `Exception`, ahora debes poner `TeamException`. La clase `Team` que te proporcionamos es fruto de haber hecho la Práctica anterior.

   (0.25 puntos)

   Como verás, la clase `Team` tiene un método `main` que te servirá para comprobar que has codificado correctamente las clases `TeamException` y `Team`. Ejecuta la clase `Team`.

> **Nota**: Recuerda que un programa Java puede tener tantos métodos `main` como clases tiene. Por eso, es importante indicar a la hora de ejecutar el programa, qué método `main` es el punto de entrada al programa, es decir, por dónde empieza a ejecutarse la aplicación. Esto se entiende rápidamente cuando se ejecuta un programa Java por línea de comandos, ya que con el comando `java` debemos indicar el nombre de la clase que contiene el `main` a usar, p.ej. `java Team`. Con Eclipse se debe configurar con el menú *Run → Run Configurations… → Java → Java Application → Main class*. Si sólo hay un `main` en todo el programa, Eclipse normalmente lo detecta y no hace falta configurar nada.

Si has realizado el paso 5, ahora el `catch` que hay en el método `main` del fichero `Team.java` debería capturar excepciones `TeamException`, en vez de tipo `Exception`. Si pusieras `Exception` (como antes del paso 5), en lugar de `TeamException`, también te funcionaría, pues `TeameException` hereda de `Exception` y `Exception` captura todas las excepciones que hereden de ella. Por eso es importante que si ponemos varios `catch`, el último capture `Exception`, pues en caso contrario una excepción propia nunca sería capturada por su `catch`, al ser la excepción propia que hemos creado hija de `Exception`.

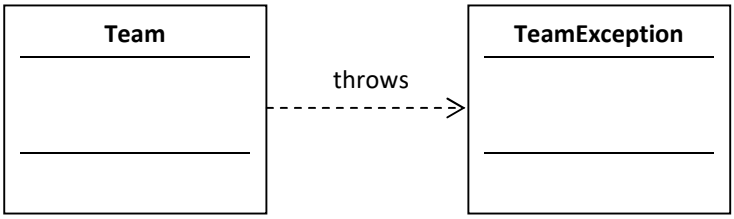
Para saber si has hecho bien este ejercicio, tienes que ver que cuando se lanza una excepción, la instrucción `e.printStackTrace()` indique por pantalla (i.e. Console) que el tipo de excepción es `TeamException` en vez de `java.lang.Exception`.

Si quieres saber más sobre las excepciones, lee los siguientes recursos:

- http://www.javatpoint.com/exception-propagation

- https://examples.javacodegeeks.com/java-basics/exceptions/java-custom-exception-example/
- https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html

Quizás te estás preguntando cómo se representa la relación entre la clase `Team` y `TeamException` en un diagrama de clases UML. Pues bien, para representar que una clase lanza una excepción, habitualmente, se utiliza un tipo de relación que no hemos visto en los apuntes llamada *dependencia*. La *dependencia* se dibuja de la siguiente manera:

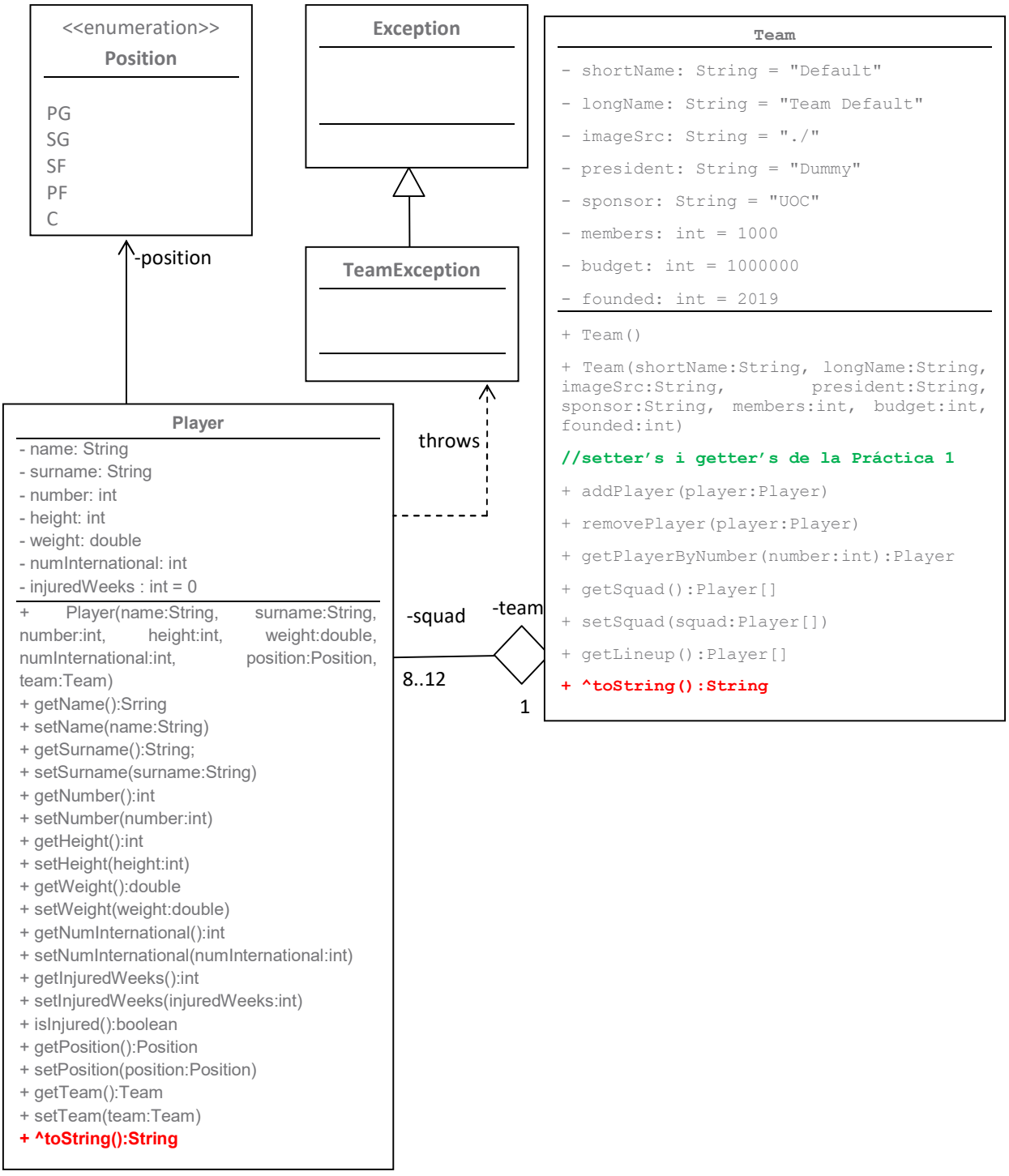


Se trata de una línea discontinua que va de la clase `Team` hacia la clase `TeamException`. Para enfatizar que se trata de una excepción algunas personas ponen la palabra/rol/etiqueta `throws` sobre la línea. La relación/asociación de *dependencia* indica que una clase A (p.ej. `Team`) utiliza el servicio de otra clase B (p.ej. `TeamException`) para poder funcionar en su totalidad. Es la clase A (p.ej. `Team`) la que conoce la existencia de la clase B (p.ej. `TeamException`). Se parece bastante a una asociación binaria unidireccional, pero a nivel de codificación, como la relación de *dependencia* indica el uso de un servicio de la clase B por parte de la clase A, se traduce como que la clase A instancia la clase B pero no guarda la instancia en un atributo de la clase. Esto es precisamente lo que sucede con las excepciones, creamos la instancia de tipo `Exception` (o derivada de `Exception`, como `TeamException`) y la usamos llamando a la instrucción `throw`, pero no guardamos la instancia de tipo `Exception` (o clase derivada de `Exception`) en un atributo de la clase que la instancia.

Te comentamos la existencia de este tipo de asociación llamada *dependencia* para que te des cuenta de que en esta asignatura tratamos una pequeña parte (la más importante y utilizada) de los diagramas de clases, pero que hay mucho más. De hecho, los diagramas de clases son uno de los tipos de representación del lenguaje UML, pero hay más: diagrama de casos de uso, diagrama de secuencia, etc.

Ejercicio 2 – Equipos y jugadores (5.5 puntos)

Cómo ya vimos en la Práctica anterior, el software tendrá que gestionar los equipos (`Team`). Cada equipo tendrá jugadores. Así pues, debemos incorporar la clase `Player` a nuestro programa. La relación entre las clases `Team` y `Player` quedaría así:



Como puedes ver en el diagrama de clases, **la relación entre los objetos de las clases `Team` y `Player` es una asociación de agregación**, ya que cada equipo tiene jugadores, pero si el equipo desaparece los jugadores no desaparecen, en todo caso, son reubicados a otros equipos. Lo mismo ocurre si un jugador se va del equipo, el equipo no desaparece… ¡¡entonces no existirían los fichajes!! Fíjate que en la relación hemos usado dos etiquetas (en verdad se llaman roles) que se llaman `-squad` y `-team`. Esto indica que en la clase `Team` habrá un atributo privado (de aquí el símbolo menos) llamado `squad` del tipo `Player` y al mismo tiempo en la clase `Player` habrá un atributo privado llamado `team` de tipo `Team`. Si no hubiéramos puesto ningún rol (i.e. etiqueta), se tendría que haber interpretado lo mismo. El objetivo de poner el rol/etiqueta es dejar más claro cómo se llamarán los atributos, puesto que podríamos haber elegido unos nombres diferentes a como se llaman las clases a las que pertenecen. Además, fíjate que dentro de `Team` no está el atributo `squad`, puesto que sería redundante, y lo mismo ocurre en el caso de la clase `Player`. Por último, fíjate que la navegabilidad de la relación/asociación es bidireccional (cuando no hay flechas se entiende que es como si estuvieran en los dos extremos).

A continuación explicaremos cada clase por separado para que te sea más fácil la comprensión y resolución de este ejercicio.

Clase `TeamException`

Esta clase no se ve alterada respecto al diseño e implementación realizados en el Ejercicio 1 de esta Práctica. Por lo tanto, tan sólo cópiala en el proyecto `PRAC2_ex2`.

Clase `Player`

1) Atributos

Un jugador tiene un nombre (`name`), apellido (`surname`), un dorsal (`number`) que identifica unívocamente al jugador dentro del equipo, una altura (`height`), un peso (`weight`), el número de veces que ha sido internacional con su selección (`numInternational`) y el número de semanas que le quedan para estar recuperado de una lesión (`injuredWeeks`).

Finalmente, como hemos comentado, la clase `Player` tiene un atributo privado (indicado como un rol) llamado `team` que guarda el equipo en el que juega.

2) Métodos

La clase `Player` tiene un constructor con argumentos. Éste inicializa todos los atributos con el valor indicado por el parámetro homónimo y el atributo `injuredWeeks` con el valor indicado por defecto en el UML.

El valor pasado como parámetro al método `setHeight` debe ser igual o mayor a `100` (cm), en caso contrario debe lanzar un error de tipo `Exception`: `[ERROR] Height must be equal or greater than 100 cm!!`

El valor pasado como parámetro al método `setWeight` debe ser igual o mayor a `30` (kg), en caso contrario debe lanzar un error de tipo `Exception`: `[ERROR] Weight must be equal or greater than 30 kg!!`

El valor pasado como parámetro al método `setNumInternational` debe ser igual o mayor a `0`, en caso contrario debe lanzar un error de tipo `Exception: "[ERROR] Num. international must be equal or greater than 0!!"`

El valor pasado como parámetro al método `setInjuredWeeks` debe ser igual o mayor a `0`, en caso contrario debe lanzar un error de tipo `Exception`: `"[ERROR] Weeks that a player is injured must be equal or greater than 0!!"`. El método `isInjured` devolverá `true` cuando el atributo `injuredWeeks` sea mayor a cero.

Por otro lado, para cada jugador se debe indicar la posición en la que juega de manera habitual. Para ello utilizaremos un atributo privado llamado `position` que será del tipo `enum` (o `enumeration`), concretamente, del `enumeration Position` que tendrá cinco valores: `PG`, `SG`, `SF`, `PF` y `C`. La posición se asigna mediante el constructor y se puede modificar y consultar a través de los métodos `setPosition` y `getPosition`, respectivamente.

Como ya sabes, el `enumeration` no es una clase, sino un tipo especial que permiten los lenguajes de programación y que se utiliza cuando queremos restringir los posibles valores que puede tomar una variable/atributo o estos valores conforman un dominio concreto y conocido, p.ej. días de la semana, meses del año, planetas del Sistema Solar, etc. En Java, no obstante, el tipo `enum` es más potente que en otros lenguajes como por ejemplo C. Por defecto, Java define cada `enum` como una clase y, de hecho, en su declaración se pueden incluir atributos y métodos y, a la vez, esta "clase" incluye por defecto algunos métodos, como por ejemplo `values()`, que devuelve todos los valores del `enum`. De hecho, la manera ideal de declarar un `enum` es poniendo su código dentro de un fichero `.java` que tenga el mismo nombre que el `enum`. No obstante, un `enum` se puede declarar dentro de una clase ya existente. Puedes encontrar más información sobre el *enum type* en la documentación oficial que hay en la web de Java: https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html.

En cualquier caso, la manera de representar un tipo `enum` en UML (recordamos que es independiente del lenguaje de programación) es como se indica en el diagrama anterior, es decir, mediante una asociación binaria unidireccional y usando el estereotipo `<<enumeration>>`.

Clase `Team`

Esta clase se ve alterada respecto al diseño e implementación realizados en el Ejercicio 1 de esta Práctica. Por lo tanto, lo primero que debes hacer es copiar el fichero `Team.java` del Ejercicio 1 en el proyecto `PRAC2_ex2`. Puedes también eliminar el `main` que contiene, puesto que ya no lo usaremos.

Verás en el diagrama de clases que ahora esta clase tiene un atributo privado llamado `squad`. A nivel de implementación será un *array* de como máximo 12 casillas, puesto que tiene que permitir guardar hasta 12 jugadores (recuerda hacer esta inicialización en los constructores de la clase). Cada casilla del *array* será un objeto de tipo `Player`. ¿Cómo sabemos que el atributo `squad` es una colección de objetos (i.e. un *array*)? Pues porque la cardinalidad es mayor a 1; en este caso es 8..12. Además la cardinalidad del diagrama nos está diciendo que un equipo no permite más de 12 jugadores ni menos de 8. En el constructor de la clase `Team` tendremos que inicializar a 12 dicho *array*.

Por otro lado, hay que prestar especial atención a los métodos `getSquad`, `getLineup`, `addPlayer` y `removePlayer`. La visibilidad de cada uno de ellos la encontrarás en el diagrama de clases anterior.

- `getSquad` i `setSquad`: *getter* y *setter* respectivamente del atributo `squad`. En el caso del *setter*, éste debe controlar que la longitud del *array* pasado por parámetros tiene entre 8 y 12 jugadores. En caso contrario, debe lanzar la excepción de tipo `TeamException` correspondiente: `"[ERROR] Your team cannot have less than 8 players!!"` o `"[ERROR] Your team cannot have more than 12 players!!"`, respectivamente.
- `getLineup`: devuelve un *array* con los cinco primeros jugadores del *array* `squad`. Si hay menos de cinco jugadores, entonces devuelve el máximo número de jugadores que hay (que será menos de cinco).
- `getPlayerByNumber`: devuelve el jugador cuyo atributo `number` es igual al valor indicado por el parámetro `number`. Si el valor del parámetro `number` es negativo, entonces el método `getPlayerByNumber` debe lanzar la siguiente excepción (usando `TeamException`): “`[ERROR] The number is incorrect. The number must be 0 or positive!!`”. Del mismo modo, si no se encuentra ningún jugador con el `number` indicado, el método debe lanzar la siguiente excepción (usando `TeamException`): “`[ERROR] The player you want to retrieve does not exist in your team!!`”.

- `addPlayer`: este método ubica el objeto de tipo `Player` pasado por parámetros en la primera posición (i.e. casilla) libre del *array* `players`. Una casilla está libre si es igual a `null`. Si todas las posiciones/casillas están ocupadas, lanza la siguiente excepción (usando `TeamException`): "`[ERROR] Your team cannot have more than 12 players!!`". Si el jugador ya existe, entonces deberá lanzar la siguiente excepción (usando `TeamException`): "`[ERROR] The player who you want to add is already in the team!!`". Consideramos que dos jugadores son el mismo si sus objetos apuntan a la misma zona de memoria, es decir, si son la misma referencia, el mismo objeto.
- `removePlayer`: este método recibe el jugador que queremos borrar del equipo. Para borrar una copia del *array*, lo único que debes hacer es asignarle a su casilla/posición del *array* `squad` el valor `null` (poniendo a `null` todas las referencias a un objeto, Java lo elimina de memoria; veremos esto con detalle más adelante). En caso de que el jugador no exista, el método debe lanzar la siguiente excepción (usando `TeamException`): "`[ERROR] The player you want to remove does not exist in your team!!`".

¿Qué se pide en este ejercicio?

A partir de aquí:

Abre el proyecto `PRAC2_ex2` que se te da con el enunciado y copia/importa las clases `Team` y `TeamException` que has implementado en el ejercicio anterior, eliminando el `main` que venía en `Team.java`. En el proyecto `PRAC2_ex2` encontrarás el fichero `Position.java` que incluye la codificación del `enum` comentado en el enunciado. También encontrarás el fichero `Check.java` que te tiene que permitir comprobar que todo (o casi todo) lo que codifiques sea correcto. Evidentemente, te recomendamos que hagas tus propias pruebas dentro de este fichero de test.

Ahora, se te pide que siguiendo las indicaciones del diagrama de clases UML anterior y las especificaciones descritas en el enunciado:

Nota: Para los apartados a) y b) no tengas en cuenta el método `toString()` que aparece tanto en `Team` como en `Player` en el diagrama de clases. Tampoco hace falta que documentes con Javadoc lo nuevo que desarrolles.

a) Codifiques la clase `Player`.

(1.5 puntos: 0.30 puntos los atributos; 0.15 puntos `setHeight`; 0.15 puntos `setWeight`; 0.15 puntos `setNumInternational`;

0.15 puntos `setInjuredweeks`; 0.6 puntos el resto de getters/setters y el constructor;)

b) Codifiques los nuevos miembros de la clase `Team`.

(2.5 puntos: 0.25 puntos `getLineup`; 0.25 puntos `setSquad`; `getPlayerByNumber`, `addPlayer` y `removePlayer` valen 0.6 puntos cada uno; 0.2 puntos el resto de la implementación de la clase)

Te habrás dado cuenta de que, en el nuevo diagrama de clases de este enunciado, **tanto la clase `Team` como la clase `Player` tienen un método denominado `toString()`**. Este método es heredado de la clase `Object`, de la cual heredan automáticamente todas las clases en Java. Por lo tanto, toda clase en Java puede llamar al método `toString()`. Este método devuelve un `String` (i.e. cadena de caracteres) que consiste en una breve descripción de la instancia. Por defecto (i.e. si no se cambia), esta descripción suele ser el nombre de la clase a la que pertenece el objeto y la posición de memoria que ocupa. Por este motivo, lo más habitual es que el programador sobrescriba el método `toString()` en sus clases para personalizar el `String` que devuelve. Si se sobrescribe un método de la clase padre en una clase hija, este método sobrescrito tiene que aparecer explícitamente como método de la clase hija en el diagrama de clases, es por eso que en el nuevo diagrama de clases aparece `toString()` en las clases `Team` y `Player`. Como la clase padre (`Object`) no aparece porque es de la API de Java, le hemos añadido el símbolo `^` delante del nombre del método para indicar que es un método sobreescrito procedente de una clase padre. El símbolo `^` la mayoría de veces no se pone, pero el estándar de UML lo recoge.

En el caso de la clase `Player`, el método `toString()` tiene que devolver un `String` con el siguiente formato:

```
#. N S | TSN | H m. | W kg. | NI | P | IW
```

Donde `#` es el valor del atributo `number`, `N` es el nombre, `S` el apellido, `TSN` es el nombre corto/abreviado del equipo, `H` es la altura en metros (recuerda que el valor de `height` está en centímetros), `W` es el valor del atributo `weight`, `NI` indica el número de veces que ha sido internacional, `P` la posición e `IW` el número de semanas que le quedan lesionado.

```
5. Rafael Jofresa Prats | Barça | 1.83 m. | 82 kg. | 75 | Point Guard | 0
```

En el caso de la clase `Team`, el método `toString()` tiene que devolver el nombre del equipo y, entre paréntesis, el nombre corto/abreviado, el año de fundación, el nombre del presidente, el número socios, el presupuesto del que dispone, el nombre del sponsor y los jugadores que forman parte de la plantilla (nunca tiene que devolver `null`). El formato sería el siguiente:

```
F.C. Barcelona (Barça)
```

```
Founded: 1928
President: Josep Lluís Nuñez Clemente
Members: 5.000
Budget: 1.555.000.000 ptas.
Sponsor: Banca Catalana
Squad:
      5. Rafael Jofresa Prats | Barça | 1.83 m. | 82.0 kg. | 75 | Point Guard | 0
      12. Roberto Dueñas Hernández | Barça | 2.19 m. | 127.0 kg. | 84 | Center | 0
```

Como se ve en el ejemplo, el listado de las jugadores está tabulado respecto al resto del texto. No es necesario que los jugadores los muestres ordenados por el atributo `number`.

Nota: Para hacer el formato de texto anterior, te recomendamos que utilices los elementos `\n` (salto de línea) y `\t` (tabulación) dentro de las cadenas de caracteres según sea necesario. A la vez te recomendamos que leas sobre `String`, `StringBuilder` y `StringBuffer`, y utilices la clase `StringBuilder` en las clases `Team` y `Player`, aunque se puede utilizar cualquiera de las tres clases:

http://www.dosideas.com/noticias/java/339-string-vs-stringbuffer-vs-stringbuilder.html

Para hacer que el valor del número de socios y del presupuesto se muestre utilizando puntos como separador, debes investigar sobre el método estático `format` de la clase `String` de Java.

El método `toString()` de cualquier clase se puede llamar explícitamente como cualquier otro método (i.e. `nombreInstancia.toString()`); o bien implícitamente usando el objeto dentro de los métodos `println` o `print`. El símbolo `+` también invoca implícitamente el método `toString()`. Una manera de ejemplificar esta explicación sería:

```
Clase1 c1 = new Clase1();
System.out.println(c1.toString()); //llamada explícita
System.out.println(c1); //println llama implícitamente toString
String s1 = "El operador '+' llama implícitamente toString:"+c1:
System.out.println(s1);
```

Nota: Cómo ya sabes, para escribir por pantalla, en Java se usa `System.out.println` (o `System.out.println`). Lo que no sabemos es si te has dado cuenta de que se trata de la llamada a un objeto estático y público (denominado `out`) de tipo `PrintStream` (una clase de Java) que se encuentra dentro de la clase `System` (otra clase de Java). Es porque `out` es estático y público, que no hay que crear un objeto de tipo `System` para poder acceder a `out`. La clase `PrintStream` (y, por lo tanto, todos los objetos

declarados como tales) tienen dos métodos públicos llamados `println` y `print` (la diferencia entre uno y el otro es que el primero, además de escribir por pantalla/consola, añade un "enter de teclado", i.e. un salto de línea). Por curiosidad, mira la documentación de la clase `System` que nos proporciona la web oficial de Oracle: https://docs.oracle.com/javase/7/docs/api/java/lang/System.html

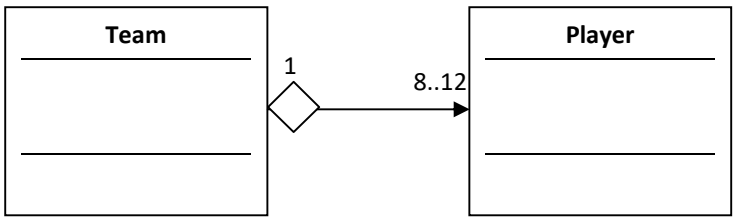
Por cierto, ¿te has dado cuenta de que la documentación de la clase `System` está hecha con Javadoc? Y no sólo la clase `System`, sino que toda la documentación de la API de Java (i.e. clases que ya trae incorporadas el JDK) que encontramos en la web oficial de Oracle está hecha con Javadoc.

c) Sobrescribe el método `toString()` en las clases `Player` y `Team`.

Nota: Antes de la firma del método `toString()` pon la anotación `@Override` para indicar al compilador y a ti, que estás sobrescribiendo el método. No es obligatorio ponerla, pero, tal y como se indica en los apuntes, sí que es práctico y, por lo tanto, recomendable.

(1 punto; 0.8 puntos el de `Team` y 0.2 puntos el de `Player`)

d) **Teoría**: Como puedes apreciar en el diagrama de clases, la navegabilidad en la relación entre las clases `Team` i `Player` es bidireccional. Explica qué cambiaría en la implementación si la relación fuera unidireccional tal y como se muestra a continuación. ¿Qué consecuencias tendría en el comportamiento del programa este cambio? ¿Qué atributos y/o métodos del diagrama original no tendrían sentido que estuvieran con esta nueva navegabilidad? Razona tu respuesta.



(0.5 puntos)

Ejercicio 3 – El cuerpo técnico (3.75 puntos)

Como ya sabes, en un equipo hay un cuerpo técnico que entrena a los jugadores. PC BalUOCesto no podía ser menos y define la clase `Coach`. Una vez sabemos que vamos a tener entrenadores, vemos que éstos comparten cierta información con los jugadores. Así pues, estamos ante un claro caso de herencia o generalización/especialización. A la nueva clase padre la llamaremos `Person`.

Person

Tanto los jugadores como los entrenadores tienen los atributos `name`, `surname` y `nick` (apodo). También tienen una fecha de nacimiento (`birthdate`). Asimismo, tanto jugadores como entrenadores tienen una nacionalidad (`country`) los valores de la cual vendrán delimitados por un `enum` llamado `Country`. Además, tanto jugadores como entrenadores tienen un contrato que está formado por tres elementos: un salario (`salary`), una cláusula de rescisión (`cancellationClause`) y los años de contrato que quedan (`contractYears`). En el caso de los entrenadores la cláusula de rescisión siempre es `0` ptas. y los contratos son anuales, por lo que el valor de `contractYears` siempre es `0`. Ninguno de los tres elementos que forman el contrato puede ser negativo. En caso de querer asignar un valor negativo tendremos que lanzar la excepción (`Exception`): `[ERROR] XXX must be equal or greater than 0!!`", siendo `XXX`: "Salary", "Cancellation clause" o "Contract years" según sea el caso. Finalmente, decir que los jugadores y los entrenadores, obviamente, guardan una referencia al equipo al que pertenecen (`team`). La signatura del constructor de la clase `Person` tiene que ser:

```
public Person(String name, String surname, String nick,
LocalDate birthdate, Country country, int salary, int
cancellationClause, int contractYears, Team team);
```

Ten en cuenta:

- Cada atributo tiene sus *getter* y *setter* públicos.
- Piensa cómo modelar el contrato de cada persona.
- Sobrescribe el método `toString()` de manera que devuelva un `String` con el siguiente formato:

  `N S 'A' | B | C | T`

  Donde `N` es el nombre, `S` el apellido, `A` es el apodo si no es `null`, `B` es la fecha de nacimiento con el formato `dd de mes de yyyy`, `C` es el nombre del país de nacimiento de la persona y `T` el nombre

corto/abreviado del equipo. Recuerda poner la anotación `@Override` antes de la firma de `toString`.

Coach

Cada entrenador tiene un atributo que indica el número máximo de jugadores que es capaz de entrenar a la vez (`maxNumPlayers2Train`). Para este atributo definiremos un *getter* público y un *setter* privado. En el caso que queramos asignarle un valor igual o inferior a `0`, el *setter* debe lanzar una excepción (`Exception`) con el mensaje: "`[ERROR] Number of players that a coach can train cannot be 0 or negative!!`". El constructor tiene la siguiente firma:

```
public Coach(String name, String surname, String nick, LocalDate birthdate, Country country, int salary, Team team, int maxNumPlayers2Train)
```

Debes sobrescribir el método `toString()` para que devuelva el siguiente `String` (mostramos el formato con los valores en negrita, el resto es texto):

`Surname, Name (maxNumPlayers2Train) | salary` `mill.`

Recuerda poner la anotación `@Override` antes de la firma de `toString`.

Player

Los jugadores, por su parte, se comportan como hemos definido hasta ahora a lo largo de las Prácticas. Haz las modificaciones que creas necesarias debido a la introducción de la clase `Person` y ten en cuenta que ahora la firma del constructor debe cambiar por la siguiente firma:

```
public Player(String name, String surname, String nick, LocalDate birthdate, Country country, int salary, int cancellationClause, int contractYears, Team team, int number, int height, double weight, int numInternational, Position position);
```

Nota: Si para un atributo, método o argumento no se indica su tipo y/o su visibilidad (i.e. nivel de acceso), tendrás que ser tú quien decida qué es lo que mejor se ajusta según las características del problema/programa que estamos resolviendo. Lo mismo si hay algún detalle en la implementación de un constructor y/o método.

A partir de la explicación anterior, se te pide:

a) Dibujar el diagrama de clases UML que cumple las especificaciones descritas en el enunciado de este ejercicio. Además, explica las

decisiones que has tomado a la hora de hacer el diagrama de clases UML que presentas.

> **Nota**: Para la clase `Team` no hace falta que pongas en el diagrama UML que entregues los atributos y métodos aparecidos en los ejercicios anteriores.

(1.5 puntos)

Para los siguientes apartados ten en cuenta que:

> **Nota**: Tendrás que abrir/importar el proyecto `PRAC2_ex3` que se te facilita con este enunciado y copiar en él las clases `Team`, `TeamException`, `Player` y `Position` del Ejercicio 2. Para probar que casi todo es correcto, se te facilita el fichero `Check.java`. Evidentemente, tendrás que hacer tus propias pruebas dentro de este fichero.

b) Codifica la clase `Person`.

(0.75 puntos)

c) Codifica la clase `Coach`.

(0.75 puntos)

d) Codifica los cambios necesarios en la clase `Player`.

(0.75 puntos)

Recursos

Esta Práctica se centra especialmente en los conceptos estudiados en el **"Módulo 4 – Asociación y herencia"**, pero inevitablemente trabaja también los conceptos vistos en los módulos anteriores. También dispones en el aula de una guía sobre Java denominada **"El lenguaje de programación Java"**.

Es muy normal que surjan dudas cuando se programa, aunque llevemos muchos años programando con un lenguaje y un paradigma concretos. Es por eso que hay que tener muy asimilada la competencia de buscar información.

Hoy en día, el mejor lugar donde encontrar información y de manera rápida es Internet. En el caso de Java, tienes toda la documentación de su API (i.e. librerías con clases ya hechas) en la siguiente web: https://docs.oracle.com/en/java/javase/12/docs/api/index.html. También puedes utilizar foros como *stackoverflow*, tanto en inglés como en español:

- Inglés: http://stackoverflow.com/questions/tagged/java
- Español: http://es.stackoverflow.com/questions/tagged/java

O utilizar directamente el buscador Google: www.google.com

Si quieres ampliar conocimientos, siempre puedes comprar libros sobre POO y Java en alguna librería o pedirlos prestados en cualquier biblioteca (incluyendo la de la UOC, http://biblioteca.uoc.edu).

Si a pesar de buscar información, no solucionas tus problemas, puedes hacer preguntas teóricas, pedir aclaraciones de los enunciados o preguntar sobre aspectos técnicos (p.ej. Java) en el foro del aula. **No puedes usar los foros ni ningún otro medio para pedir a otro compañero la solución de lo que se pide en los enunciados. Tanto la persona que pide como quien proporciona la solución, serán penalizadas siguiendo la normativa académica de la UOC.**

Criterios de valoración

El reparto de los puntos de esta Práctica es el siguiente:

- Ejercicio 1: 0.75 puntos
- Ejercicio 2: 5.5 puntos
- Ejercicio 3: 3.75 puntos

El detalle del reparto de los puntos en cada ejercicio está en el enunciado.

Formato y fecha de entrega

Se tiene que entregar un fichero *.zip, cuyo nombre debe seguir este patrón: `loginUOC_PRAC2.zip`. Por ejemplo: *dgarciaso_PRAC2.zip*. Este fichero comprimido tiene que incluir los siguientes elementos:

- Hoja de respuestas completada en formato PDF. El nombre del fichero tiene que ser: `loginUOC_PRAC2.pdf`.

- El proyecto `PRAC2_ex1` completado siguiendo las peticiones y especificaciones del Ejercicio 1.

- El proyecto `PRAC2_ex2` completado siguiendo las peticiones y especificaciones del Ejercicio 2.

- El proyecto `PRAC2_ex3` completado siguiendo las peticiones y especificaciones del Ejercicio 3.

Recuerda añadir/explicar comentarios normales (no Javadoc) en las partes de código que creas importante destacar/comentar. Para esta Práctica no es necesario poner comentarios Javadoc ni generar la documentación en formato web.

El último día para entregar esta Práctica es el **10 de noviembre de 2019 a las 23:59**. Cualquier Práctica entregada más tarde será considerada como no presentada.