MÓDULO PROFESIONAL PROGRAMACIÓN

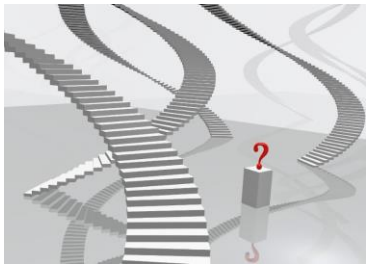
UD 7 – Introducción POO. Desarrollo de clases

Desarrollo de clases

Hasta ahora has utilizado un buen número de clases en las unidades anteriores. Las clases String, Random, Math, Scanner, System, Integer, Double, Character, etc. son un ejemplo de ello.

Esta unidad didáctica trata de que aprendas a diseñar tus propias clases en el desarrollo de la solución a un problema además de usar las clases proporcionadas por Java.

Diseñar una clase puede ser un reto porque no siempre es fácil saber cómo empezar, o saber si el resultado es de buena calidad.



POO: clase y objeto

Un programa realizado en Java está formado por un conjunto de clases que en tiempo de ejecución del programa permiten construir objetos.

Por lo tanto, un programa Java, una vez instalado en la memoria RAM del ordenador, es un conjunto de objetos que interactúan entre si para desempeñar una tarea.

POO: clase y objeto

Los objetos se crean a partir de una clase.

Para explicar el concepto de clase, se podría decir que una clase constituye el diseño de un objeto. Es una **plantilla** para crear objetos.

Una clase describe la **estructura y comportamiento** de los objetos que se crean a partir de ella.

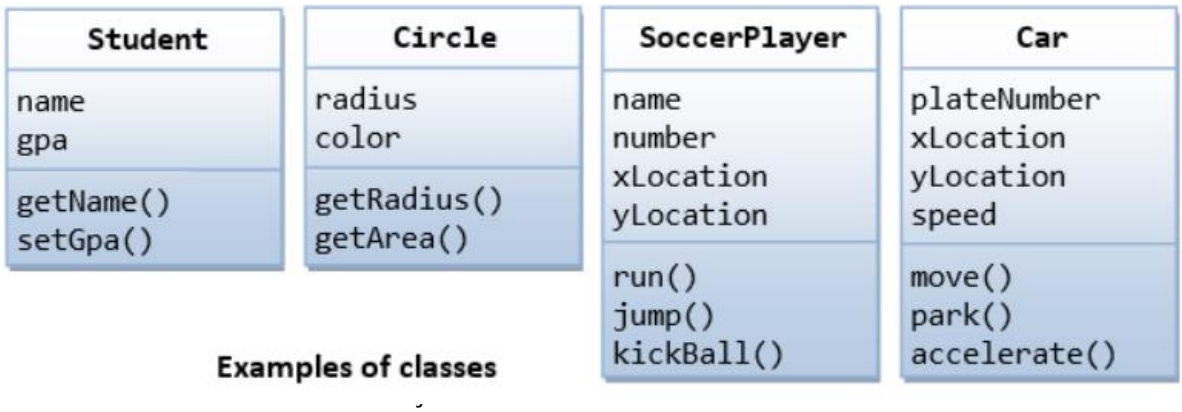
POO: clase y objeto

Nos podemos imaginar a una clase como un molde para crear objetos.

Cada vez que **instanciemos una clase**, o lo que es lo mismo usemos el molde, estamos fabricando un objeto idéntico al original que determinó el molde, es decir, la clase.

Has aprendido a utilizar objetos de clases existentes. En esta unidad didáctica empezarás a implementar tus propias clases

Clases



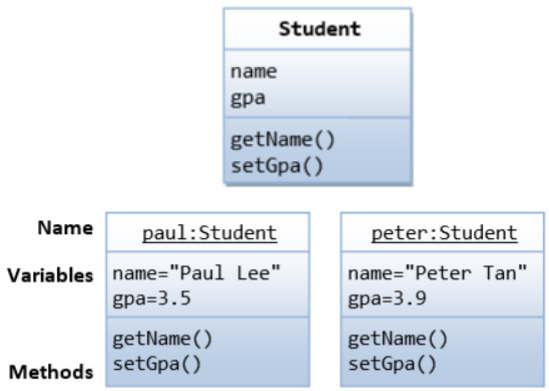
Examples of classes

- **Métodos (o comportamientos, funciones, operaciones):** contiene los comportamientos de la clase.

Clases

Durante la ejecución del programa se producirá la instanciación de la clase, es decir la creación de los objetos.

La figura muestra instancias de la clase Student (Estudiante).



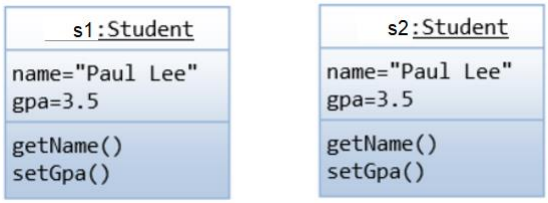
Two instances - paul and peter - of the class Student

GPA (**Grade Point Average**): es un término utilizado para asignar un valor numérico a las calificaciones acumuladas por un estudiante en el sistema estadounidense.

Clases

Aunque dos objetos tengan los mismos valores en sus atributos, son objetos diferentes.

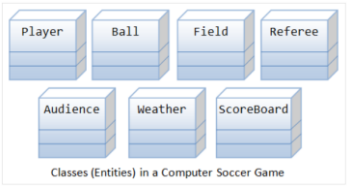
¿Puede ocurrir esto?



Clases

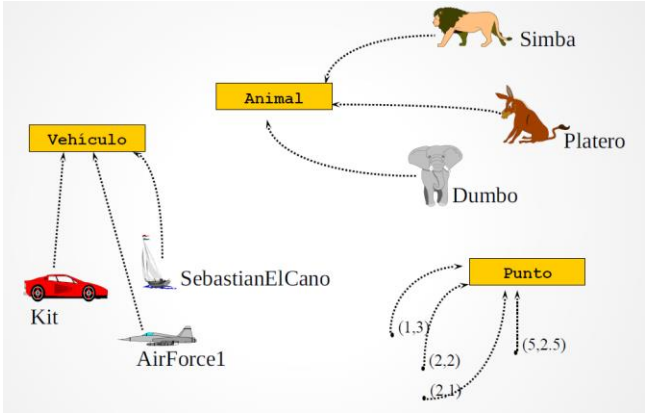
Supone que deseas programar **juegos de fútbol** (que considero una aplicación compleja). Es bastante difícil modelar el juego en lenguajes procedimentales. Pero al usar lenguajes **orientados a objetos**, se puede modelar fácilmente el programa de acuerdo con las "cosas reales" que aparecen en los juegos de fútbol:

- **Jugador**: los atributos incluyen nombre, número, ubicación en el campo y etc. y las operaciones incluyen correr, saltar, patear la pelota, etc.
- **Balón** (Ball)
- **Árbitro** (Referee)
- **Campo** (Field)
- **Espectadores** (Audience)
- **Clima** (Weather)



Lo más importante es que algunas de estas clases (como Balón y Espectadores) se pueden reutilizar en otra aplicación, por ejemplo, un juego de baloncesto de computadora, con poca o ninguna modificación.

Las clases y los objetos están en todas partes



Definición de una clase

La sintaxis para definir una clase en Java es:

```
[modificadores] class NombreClase {
   // el cuerpo de la clase contiene atributos y métodos
   ......
}
```

Más adelante explicaremos los modificadores como el público, privado, abstract, static, etc.

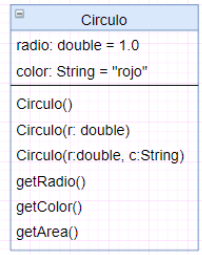
Definición de una clase

```
[modificadores] class NombreClase {
   // el cuerpo de la clase contiene atributos y métodos
   ......
}
```

La implementación de una clase, por ejemplo la clase Circulo, en Java se realizaría en un fichero con nombre Circulo.java y su contenido podría ser el siguiente:

```
public class Circulo { //nombre de la clase
    //atributos
    private double radio;
    private String color;

    //constructores
    Circulo() {...}
    Circulo(double r) {...}
    Circulo(double r, String) {...}

    //métodos
    public double getRadio() {...}
    public String getColor() {...}
    public double getArea() {...}

}
```

Definición de la clase:



Definición de una clase

Otro ejemplo

```
public class JugadorFutbol {  // nombre de la clase
   int numero;               // atributos
   String nombre;
   int x, y;

   void correr() { ..... }     // métodos
   void patearPelota() { ..... }
}
```

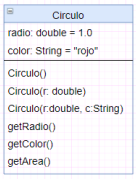
Clase: nomenclatura

Convención de **nomenclatura** de clases:

- Un nombre de clase debe ser un **sustantivo** o una frase nominal compuesta de varias palabras.
- Todas las palabras deben estar en mayúscula inicial (UpperCamelCase).
- Usa un nombre singular para el nombre de la clase.
- Elige un nombre de clase significativo y auto-descriptivo. Por ejemplo, JugadorFutbol, CuentaBancaria, ServidorArchivos, etc.

Creando objetos

Definición de la clase:



Para crear una **instancia de una clase (un objeto)** debes:

- **Declarar** una referencia a un objeto de una clase particular.
- **Construir** la instancia (es decir, asigne almacenamiento para la instancia e inicialice la instancia) utilizando el **operador "new"**.
- **Conectar** el objeto con la referencia mediante el operador de asignación

Por ejemplo, supongamos que ya tenemos definida una clase llamada Circulo, podemos crear instancias de Circulo de la siguiente manera:

```
//Declaro 3 instancias de la clase Circulo: c1, c2 y c3
Circle c1, c2, c3;

//Construyo las instancias(objetos) con el operador new y
//las vinculo con las referencias

c1 = new Circulo();             //1º constructor
c2 = new Circulo(2.0);          //2º constructor
c3 = new Circulo(3.0, "verde"); //3º constructor

//Puedes declarar y construir en la misma sentencia
Circulo c4 = new Circulo();
```

Cuando una instancia se declara pero no se construye, tiene un valor especial llamado **null**.

El operador punto

Los atributos y métodos que pertenecen a un objeto se denominan formalmente **atributos y métodos miembro. Significa que solo están disponibles cuando un objeto es creado.**

Para hacer referencia a un atributo o método miembro, debes:

- Primero, identificar el objeto que quieres usar y luego
- Usar el **operador punto (.)** para hacer referencia al atributo o método miembro deseado.

El operador punto

Volvamos con la clase **Circulo**, con dos atributos miembro: radio y color; y dos métodos miembros: getRadio () y getArea().

Hemos creado **dos instancias** de la clase Circulo, a saber, c1 y c2.

Para invocar el método getArea(), primero debes identificar la instancia de interés, por ejemplo c1, luego usar el operador de punto, en forma de c1.getArea(). Por ejemplo,

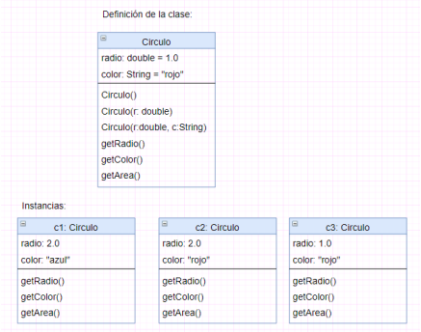
```
//Supongamos que la clase circulo tiene los atributos radio y
//color y los métodos getArea() y getRadio()
//Declaro y construyo las instancias c1 y c2 de la clase
//circulo

c1 = new Circulo();
c2 = new Circulo();

//Invoco los métodos miembros de la clase c1 con el operador
//punto
System.out.println(c1.getArea());
System.out.println(c1.getRadio());

//referencio los atributos miembros de la clase c2 con el
//operador punto
c2.radio=5.0;
c2.color="blue";
```

Atributos: tipos

Los atributos pueden ser de cualquiera de los tipos básicos de Java (boolean, char, byte, short, int, double, etc) o variables de referencia a otros objetos. Existen tres tipos de atributos:

- Atributos **miembro** o variables de instancia
- Atributos **estáticos** o variables de clase
- Atributos **finales**

Generalmente, cuando hablamos de atributos nos referimos a **atributos miembro**, de lo contrario aclararemos de qué otro tipo se trata.



Atributos: declaración

Los atributos de una clase se declaran dentro de esta, al comienzo de la misma, fuera de cualquier método. Solamente se declaran, opcionalmente se inicializan o instancian. La sintaxis básica es la siguiente:

```
[modificadores] tipo nombreAtributo [= valorInicial];
```

Ejemplos,

```
private double radio;
public int largo = 1, ancho = 1;
```

Atributos: nomenclatura

Convención de **nomenclatura** de atributos:

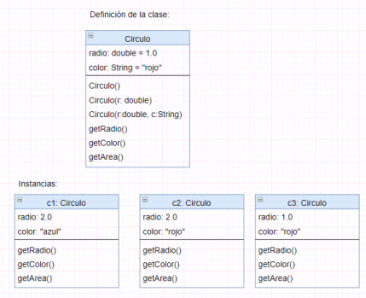
- Un nombre de atributo debe ser un **sustantivo** o una frase de sustantivo formada por varias palabras.
- La primera palabra está en **minúsculas** y el **resto de las palabras están en mayúscula inicial**. Por ejemplo, numeroHabitacion.

Métodos: tipos

Los métodos son funciones definidas dentro de una clase.
Pueden ser de cuatro tipos:

- Métodos **miembro** o de instancia.
- Métodos **estáticos** o de clase.
- Métodos **finales**. (los estudiaremos más adelante).
- Métodos **abstractos**. (los estudiaremos más adelante).

Generalmente, cuando hablamos de métodos nos referimos a **métodos miembro**, de lo contrario aclararemos de qué otro tipo se trata.



Método: ¿qué es?

Un método:

- recibe argumentos de la parte del programa que lo llama,
- realiza las operaciones definidas en el cuerpo del método, y
- devuelve un resultado (o nada) al programa que lo llama.

Métodos: declaración

La sintaxis para la declaración de métodos en Java es la siguiente:

```
[modificadores] tipoDevuelto nombreMetodo ([listaParametros]) {
   // cuerpo del método o implementación
   ......
}
```

Ejemplo,

```
// Devuelve el área de un objeto Circulo
public double getArea() {
   return radio * radio * Math.PI;
}
```

Métodos: cabecera

```
[modificadores] tipoDevuelto nombreMetodo ([listaParametros]) {
   // cuerpo del método o implementación
   ......
}
```

- **Modificadores**: public, private, … definen quien puede acceder a ellos. También static, final, abstract… Lo estudiaremos más adelante
- **Tipo devuelto**: al finalizar el método, este puede devolver un valor, este puede ser un tipo primitivo o un objeto de una clase, o nada (void)
- **Nombre**: es el identificador del método.
- **Lista de parámetros**: puede tener cero o más parámetros que se declaran indicando su tipo, su identificador y si son más de uno se separan por comas.

Métodos: cabecera

Ejemplos de parámetros:

```
public void setEdad(int nuevaEdad)
public void setEdad(int anios, double meses)
```

Los parámetros del método se definen mediante su tipo y su identificador, este **identificador no tiene por que coincidir con el identificador original,** si lo tuviese.

Métodos: sobrecarga

En Java podemos definir más de un mismo método con el mismo identificador (nombre) pero que realice operaciones diferentes.

La condición para poder **sobrecargar** un método es que todos ellos posean el mismo tipo de dato devuelto y que tengan distintos parámetros. Es decir que sólo se diferencien en su declaración en el número o tipo de argumentos que reciben:

```
public void setEdad(int nuevaEdad)
public void setEdad(int anios, double meses)
```

La sobrecarga de métodos significa que el mismo nombre de método puede tener diferentes implementaciones (versiones). Sin embargo, las diferentes implementaciones deben ser distinguibles por su lista de parámetros.

Métodos: nomenclatura

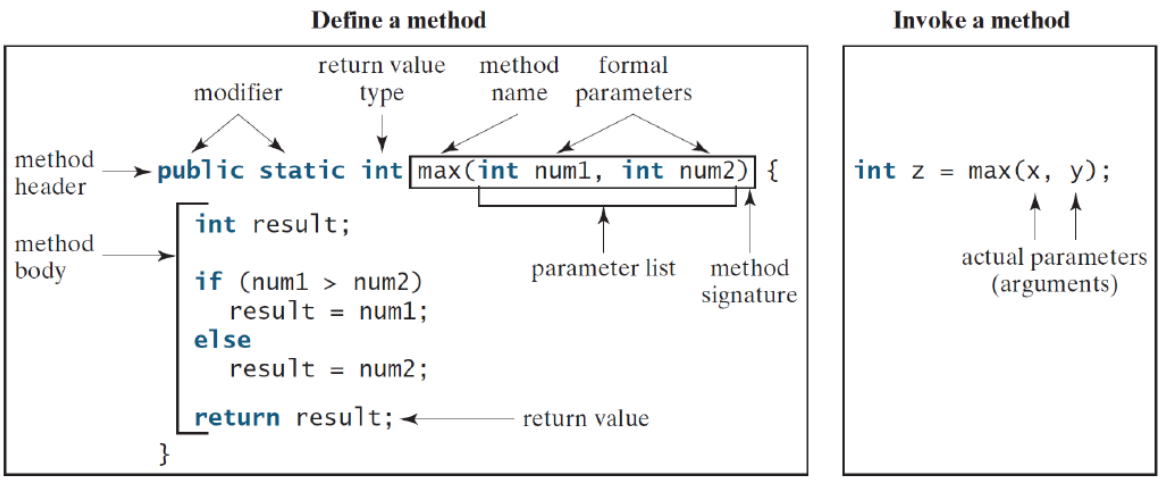
Convención de **nomenclatura** de métodos:

- Un nombre de método debe ser un **verbo** o una frase verbal compuesta de varias palabras.
- La primera palabra está en minúsculas y el resto de las palabras están en mayúscula inicial. Por ejemplo, getArea(), setRadio(), existeEmpleado().

Métodos: cuerpo

```
// Devuelve el área de un objeto Circulo
public double getArea() {
   return radio * radio * Math.PI;
}
```

- Va delimitado dentro de llaves, {}
- Podemos realizar cualquier tipo cálculo, y operación de las vistas hasta ahora incluido condicionales, bucles, operaciones de E/S …
- Dentro de un método podemos utilizar los atributos y métodos del objeto, los parámetros de la cabecera y además crear todas las variables que necesitemos **(variables locales)**
- Todas las variables que se definan dentro del mismo se eliminaran al acabar al terminar este.
- Finalmente podemos devolver un valor utilizando **return** y el valor a devolver
- Si no queremos devolver ningún valor el tipo devuelto por el método será void.



Métodos sentencia return

La sentencia return es el punto final del método.

- Cuando se invoca a un método el control vuelve al código que lo invoca cuando:
  - El método completa todas las sentencias (sin declaración de retorno, por ejemplo, main())
  - Se alcanza una sentencia return
  - Se lanza una excepción (lo veremos más adelante).

Nótese que la sentencia de retorno **no está necesariamente al final del método.**

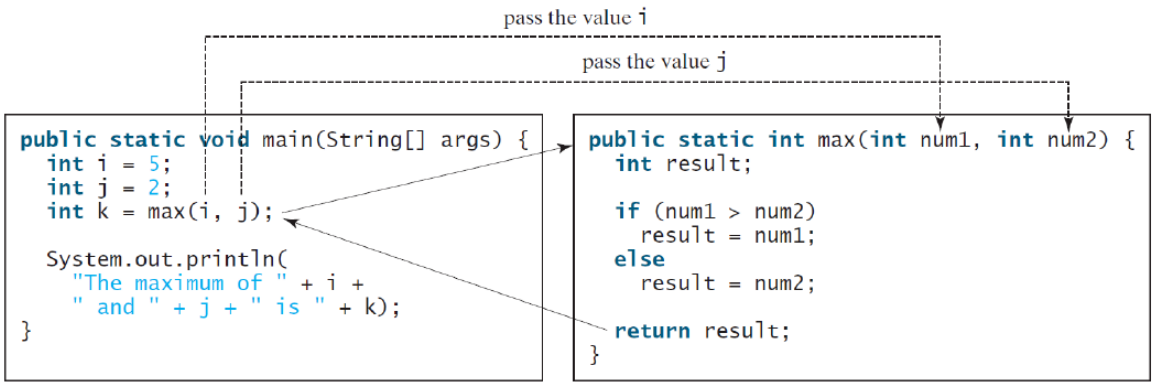
Métodos sentencia return

Una vez que se define el tipo de retorno (excepto void), el método debe devolver un valor o un objeto de ese tipo. **Sin embargo, tu método debe asegurarse de que la sentencia return esté disponible para todas las condiciones.**

```
public int fun (int x){
    if (x > 0)
        return x;
}
```

¿Y si x es menor qué cero?

Este código no compila, "missing return statement" es el error.



Tenga en cuenta que los parámetros de entrada son una especie de variables declaradas dentro del método.

Al llamar al método, hay que proporcionar argumentos, que deben coincidir con los parámetros en orden, número y tipo compatible, como definidos en la firma del método.

Métodos: paso de parámetros

Algunos lenguajes de programación pueden utilizar dos formas alternativas de pasar los parámetros al llamar a un método/función:

- Paso por valor: se copia el dato original que se pasa como argumento. Los cambios en los parámetros no afectan al dato original
- Paso por referencia: se pasa una referencia al dato original. Cualquier modificación en el parámetro implica un cambio en el dato original.

Métodos: paso de parámetros

El paso de parámetros en Java es solo por valor. Se hace una copia del argumento.

Pero OJO,

- En Java las variables de tipo objeto contienen referencias a objetos
- Si pasamos como argumento un objeto, pasamos una copia de la referencia a él
- Los cambios en los objetos afectan a todas sus referencias.
- El objeto que se pasa como argumento se puede modificar dentro del método

Ámbito de una variable

El **ámbito de la variable** es el bloque donde se puede referenciar la variable.

Las variables pueden ser declaradas a nivel de clase, a nivel de método y a nivel de estructura de control de flujo. En general, las llaves {} definen un ámbito particular.

Uno puede declarar variables con el mismo nombre en diferentes niveles de ámbitos. Sin embargo, uno no puede declarar las variables con el mismo nombre en el mismo ámbito.

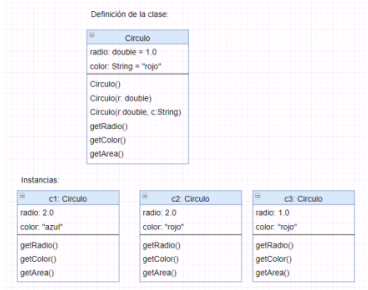
```
public class Ambito {

    static int i;

    public static void metodo(){
        i = i + 1 ;
        System.out.println(i); //imprime 1
    }

    public static void main(String[] args){

        System.out.println(i); //imprime 0

        int i=2;
        i++;

        System.out.println(i); //imprime 3

        metodo();

        System.out.println(i); //imprime 3
    }
}
```

Juntando todo en un ejemplo

Se define una clase llamada **Circulo** como se muestra en el diagrama de clases. Contiene:

- dos atributos miembro privados: radio (de tipo double) y color (de tipo String) y
- tres métodos miembro públicos: getRadios(), getColor(), y getArea().

Se construirán tres instancias de Circulo, llamadas c1, c2 y c3, con sus respectivos miembros de datos, como se muestra en los diagramas de instancia.



Juntando todo en un ejemplo

Copia el código fuente de Circulo.java:

Observa que la clase Circulo no tiene un método main(). Por lo tanto, NO es un programa independiente y no puedes ejecutar la clase Circulo por sí misma.

La clase Circulo está destinada a ser un bloque de construcción para ser utilizado en otros programas.

```java
/**
 * Modela un círculo con un radio y un color
 */
public class Circulo {    // Guardado como "Circulo.java"

   // atributos miembro privados
   private double radio;
   private String color;

   // contructores sobrecargados
   /**
    * Construye un objecto Círculo con valores para radio y color
    * por defecto
    */
   public Circulo() {
      radio = 1.0;
      color = "rojo";
   }

   /**
    * Construye un objeto Circulo con el radio pasado como parámetro
    */
   public Circulo(double unRadio) {
      radio = unRadio;
      color = "rojo";
   }

   /**
    * Construye un objeto Circulo con el radio y color pasados
    * como parámetros
    */
   public Circulo(double unRadio, String unColor) {
      radio = unRadio;
      color = unColor;
   }

   /**
    * Construye un objeto Circulo a partir de otro objeto Circulo
    * pasado como parámetro
    */
   public Circulo(Circulo unCirculo) {
      radio = unCirculo.getRadio();
      color = unCirculo.getColor();
   }

   // Métodos públicos
   /** Devuelve el radio */
   public double getRadio() {  // getter para radio
      return radio;
   }

   /** Devuelve el color */
   public String getColor() {   // getter para color
      return color;
   }

   /** Devuelve el área */
   public double getArea() {
      return radio * radio * Math.PI;
   }

   @Override
   public String toString() {
      return "Circulo{" + "radio=" + radio + ", color=" + color + '}';
   }
}
```

Juntando todo en un ejemplo

Ahora escribiremos otra clase llamada PruebaCirculo, que utiliza la clase Circulo.

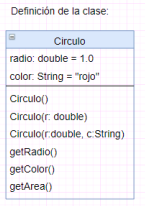
La clase PruebaCirculo tiene un método main() y se puede ejecutar.

```java
/**
 * Clase de prueba para la clase Circulo
 */
public class PruebaCirculo {    // guardada como "PruebaCirculo.java"
   public static void main(String[] args) {

      // Declara y construye una instancia de Circulo llamada c1
      Circulo c1 = new Circulo(2.0, "azul");  // Usa el tercer constructor
      System.out.println("El radio es: " + c1.getRadio());
      System.out.println("El color es: " + c1.getColor());
      System.out.printf("El área es: %.2f%n", c1.getArea());

      // Declara y construye otra instancia de Circulo llamada c2
      Circulo c2 = new Circulo(2.0);  // Usa el segundo constructor
      System.out.println("El radio es: " + c2.getRadio());
      System.out.println("El color es: " + c2.getColor());
      System.out.printf("El área es: %.2f%n", c2.getArea());

      // Declara y construye otra instancia de Circulo llamada c3
      Circulo c3 = new Circulo();  // Usa el primer constructor
      System.out.println("El radio es: " + c3.getRadio());
      System.out.println("El color es: " + c3.getColor());
      System.out.printf("El área es: %.2f%n", c3.getArea());
   }
}
```

Constructores

Un **constructor** se diferencia de un método ordinario en los siguientes aspectos:

- Un constructor se parece a un método especial que tiene el mismo nombre que el nombre de la clase.
- Los constructores son un tipo particular de métodos cuya ÚNICA función es inicializar los atributos de un objeto
- Los constructores **nunca devuelven un valor**



Constructores

Los constructores solo pueden ser invocados por el operador new. Solo se puede utilizar una vez para inicializar el objeto construido. Una vez que se construye el objeto, ya no se puede llamar al constructor.

Si al definir la clase, no se crea ningún constructor, Java proporciona automáticamente un constructor por defecto, que no recibe argumentos y realiza una inicialización por defecto de los atributos.

Constructores

Los constructores solo pueden ser invocados por el operador **new**. Solo se puede utilizar una vez para inicializar el objeto construido. Una vez que se construye el objeto, ya no se puede llamar al constructor.

```
//Construyo las instancias(objetos) con el operador new y
//las vinculo con las referencias
c1 = new Circulo();             //1º constructor
c2 = new Circulo(2.0);          //2º constructor
c3 = new Circulo(3.0, "verde"); //3º constructor
```

Parámetros

Conexión del objeto con su referencia

Creación del objeto

Constructores

A pesar de esto es **MUY** recomendable definir siempre nuestros propios constructores, especialmente cuando se trabaje con referencias a objetos como atributos.

Al igual que con el resto de los métodos, los constructores también se pueden sobrecargar, es decir podemos definir distintos constructores, siempre que se diferencien en el tipo o número de sus argumentos.

Constructores

Generalmente, se recomienda la construcción de 3 tipos de constructores: constructor por defecto, constructor con parámetros y constructor de copias, Ejemplo:

```
//Constructor por defecto
public Circulo() {...}

//Constructor por parámetros
public Circulo(double unRadio, String unColor) {...}

//Constructor de copias
public Circulo(unCirculo) {...}
```

Constructores

Constructor por defecto

Un constructor sin parámetros se denomina constructor por defecto. Inicializa los atributos miembro a un valor predeterminado.

```
/**
 * Construye un objecto Círculo con valores para radio y color
 * por defecto
 */
public Circulo() {
   radio = 1.0;
   color = "rojo";
}
```

Constructores

Constructor por parámetros

```
/**
 * Construye un objeto Circulo con el radio pasado como parámetro
 */
public Circulo(double unRadio) {
   radio = unRadio;
   color = "red";
}

/**
 * Construye un objeto Circulo con el radio y color pasados
 * como parámetros
 */
public Circulo(double unRadio, String unColor) {
   radio = unRadio;
   color = unColor;
}
```

Constructores

Constructor de copias

```
/**
 * Construye un objeto Circulo a partir de otro objeto Circulo
 * pasado como parámetro
 */
public Circulo(Circulo unCirculo) {
   radio = unCirculo.getRadio();
   color = unCirculo.getColor();
}
```

Finalización

Cuando finaliza el uso de un objeto, es necesario la realización de ciertas tareas antes de su destrucción, principalmente la liberación de la memoria solicitada durante su ejecución. Esto se realiza en C++ y otros lenguajes O.O. en los denominados destructores.

Estos **destructores** son métodos especiales (como los constructores) que se invocan cuando se quiere eliminar el objeto y liberar el espacio que ocupa en memoria.

Sin embargo en Java la liberación de memoria se realiza de manera automática, por lo tanto la necesidad de este tipo de operaciones no existe en la mayoría de los casos.

Finalización

Cuando un objeto ya no es referenciado por nadie porque las referencias a ese objeto:

- han cambiado a null
- apuntan a otro objeto
- se sale de su ámbito y desaparecen

entra en juego el **GC (Garbage Collector)** que recoge la memoria que ya no es accesible (basura). Es activado en forma automática por la JVM

Principios OO

La estrategia de ocultar información no es exclusiva de la programación informática: se utiliza en muchas disciplinas de la ingeniería.

Pensemos en el **módulo de control electrónico** que está presente en todos los coches modernos. Es un dispositivo que controla la sincronización de las bujías y el flujo de gasolina en el motor. Si le pregunta a su mecánico qué hay dentro del módulo de control electrónico, probablemente se encogerá de hombros.

Principios OO

El módulo es una caja negra, algo que mágicamente hace lo suyo. Un mecánico de coches nunca abriría el módulo de control: contiene piezas electrónicas que sólo se pueden reparar en la fábrica.

En general, los ingenieros utilizan el término **"caja negra"** para describir cualquier dispositivo cuyo funcionamiento interno está oculto.

Principios OO

Ten en cuenta que una caja negra no es totalmente misteriosa. Su interfaz con el mundo exterior está bien definida. Por ejemplo, el mecánico de automóviles entiende cómo el módulo de control electrónico debe estar conectado con los sensores y las piezas del motor.

El proceso de ocultar los detalles de la implementación mientras se publica una interfaz se llama se llama encapsulación.

Principios OO

Del mismo modo, **un programador que utiliza una clase no se ve agobiado por detalles innecesarios**, lo sabes por tu propia experiencia.

En la unidad didáctica 6, utilizaste la clase String sin preocuparte como implementa esta clase los métodos, substring o indexOf , por poner un ejemplo.

Una de las ventajas de la encapsulación es que permite cambiar la implementación de una clase sin tener que decírselo a los programadores que utilizan la clase.

Principios OO

En la unidad didáctica 5 aprendiste a ser un **usuario de objetos.** Viste cómo crear objetos y cómo manipularlos en un programa.

En esa unidad trataste los objetos como cajas negras. Tu papel era más o menos análogo al del mecánico de coches que arregla un coche conectando un nuevo módulo de control electrónico.

Principios OO

En esta unidad didáctica, pasarás a **desarrollar tus propias clases**. Ahora tu papel es análogo al de un fabricante de piezas de automóvil que monta un módulo de control electrónico a partir de transistores, condensadores y otras piezas electrónicas.

Aprenderá las técnicas de programación Java necesarias para que sus objetos tengan el comportamiento deseado **ocultando los detalles de como implementan ese comportamiento.**

En esta sección, discutiremos el proceso de especificar la **interfaz pública** de una clase.

Encapsulamiento y ocultamiento de la información

Existen cuatro niveles de acceso diferentes para controlar la visibilidad de un atributo o método dentro de una clase. Son las restricciones que tiene un elemento para ser accedido:

- public: accesible y disponible para todas las demás clases en el sistema.
- private: accesible y disponible solo dentro de esta clase.
- protected: lo explicaremos más adelante
- no especificado: lo explicaremos más adelante

Encapsulamiento y ocultación de la información

Las atributos miembro de una clase generalmente están ocultos al exterior (es decir, las otras clases), con el especificador de acceso private.

El acceso a los atributos miembro se proporciona a través de métodos de acceso públicos, por ejemplo, getColor() y getRadio().

```
public class Circulo {

   private double radio;
   private String color;

   public Circulo() {
      radio = 1.0;
      color = "rojo";
   }

   public Circulo(double unRadio, String unColor) {
      radio = unRadio;
      color = unColor;
   }

   public double getRadio() {
      return radio;
   }

   public String getColor() {
      return color;
   }

   public double getArea() {
      return radio * radio * Math.PI;
   }
}
```

Encapsulamiento y ocultación de la información

Esto sigue el principio de ocultación de la información. Es decir, los objetos se comunican entre sí utilizando interfaces bien definidas (métodos públicos).

Los objetos no tienen permitido conocer los detalles de implementación de otros. Los detalles de la implementación están ocultos o encapsulados dentro de la clase. La ocultación de la información facilita la reutilización de la clase.

```
public class Circulo {

   private double radio;
   private String color;

   public Circulo() {
      radio = 1.0;
      color = "rojo";
   }

   public Circulo(double unRadio, String unColor) {
      radio = unRadio;
      color = unColor;
   }

   public double getRadio() {
      return radio;
   }

   public String getColor() {
      return color;
   }

   public double getArea() {
      return radio * radio * Math.PI;
   }
}
```

Encapsulamiento y ocultación de la información

Regla de oro: no hagas público ningún atributo, a menos que tengas una buena razón.

Getters y setters públicos para atributos privados

Para permitir que otras clases lean el valor de un atributo privado, **xxx**, proporcionamos un método **get** (o getter o método de acceso) denominado **getXxx**(). Estos métodos no modifican la variable.

Para permitir que otras clases modifiquen el valor de un atributo privado como **xxx**, proporcionamos un método **set** (o setter o método de establecimiento) llamado **setXxx**().

```
public double getRadio() {
   return radio;
}
public void setRadio(double unRadio){
   radio = unRadio;
}
```

El operador THIS

En Java, cuando se necesita hacer referencia al objeto actual se utiliza la palabra reservada **this.**

this es una referencia al objeto actual, cuyo método está siendo invocado. Puedes usar la palabra clave this para evitar conflictos de nombres en métodos/constructores.

Este inconveniente se podría resolver cambiando el nombre al parámetro del método/constructor.

```
public class Circulo {

   private double radio;
   private String color;

   public Circulo() {
      radio = 1.0;
      color = "rojo";
   }

   public Circulo(double radio, String color) {
      this.radio = radio;
      this.color = color;
   }

   public double getRadio() {
      return radio;
   }

   public void setRadio(double radio){
      this.radio = radio;
   }

   public String getColor() {
      return color;
   }

   public void setColor(double color){
      this.color = color;
   }

   public double getArea() {
      return radius * radius * Math.PI;
   }
}
```

El operador THIS

Mas sobre this...

- this.nombreAtributo se refiere al nombreAtributo de este objeto
- this.nombreMetodo (...) invoca nombreMetodo(...) de este objeto
- En un constructor, podemos usar this(...) para llamar a otro constructor de esta clase.
- Dentro de un método, podemos usar la declaración "devolver this" para devolver este objeto al llamador

```
public Circulo() {
   radio = 1.0;
   color = "rojo";
}

public Circulo(double radio, String color) {
   this.radio = radio;
   this.color = color;
}
```

```
public Circulo() {
   this(1.0,"rojo");
}

public Circulo(double radio, String color) {
   this.radio = radio;
   this.color = color;
}
```

El método toString()

Toda clase de Java bien diseñada debe tener un método público llamado **toString()** que devuelva una descripción del objeto, es decir, el valor de sus **atributos**.

```
@Override
public String toString() {
     return "Circulo{" + "radio=" + radio + ", color=" + color + '}';
}
```

El método toString()

Puedes invocar el método toString() explícitamente llamando a un unObjeto.toString(), o implícitamente a través de println() o el operador de concatenación de cadenas '+`.

```
Circulo c1 = new Circulo(2.0, "azul");
System.out.println(c1);
```

```
Circulo{radio=2.0, color=azul}
```

Es decir, la ejecución del método **println(unObjeto)** invoca el método toString() de ese objeto de forma implícita.

Constantes

Las constantes son variables definidas con el modificador final. Una variable final solo puede asignarse una vez y su valor no puede modificarse una vez asignada. Por ejemplo,

```
public static final int MAX_ITEMS = 100;
public static final String COMPANY_NAME = "Acme Inc.";
```

error: no se puede asignar un valor a la variable final.

Una constante se declara normalmente como una variable final y estática. Esto significa que la variable no se puede cambiar una vez que se le asigna un valor, y que se puede acceder a ella sin crear una instancia de la clase que la contiene. Sin embargo, no es estrictamente necesario que una constante en Java se declare como estática, **pero es la práctica común hacerlo.**

Constantes: nomenclatura

Convención de **nomenclatura** constante:

- Un nombre constante es un sustantivo, o una frase nominal compuesta de varias palabras.
- Todas las palabras están en mayúsculas separadas por los guiones bajos '_', por ejemplo,

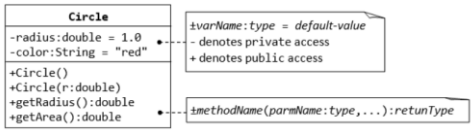
  X_REFERENCE,

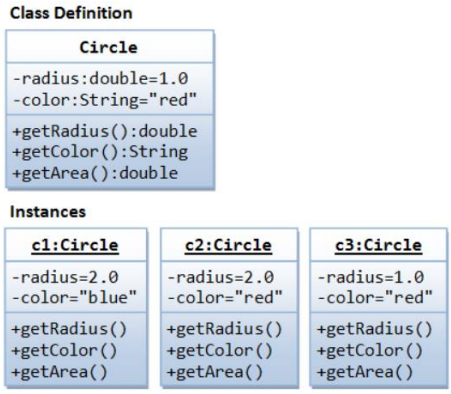
  MAX_INTEGER y

  MIN_VALUE.

Actividad: clase Circulo

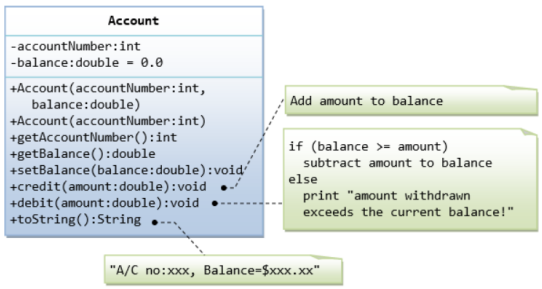
Ya se hizo





Diagramas de clase y objetos del Lenguaje de modelado unificado (UML): Los diagramas de clase anteriores se dibujan de acuerdo con las notaciones UML. Una clase se representa como un cuadro de 3 compartimentos, que contiene nombre, variables y métodos, respectivamente. El nombre de la clase se muestra en negrita y centralizado. Un objeto (instancia) también se representa como un cuadro de 3 compartimentos, con el nombre del objeto: nombre de clase y subrayado.

Actividad: clase Cuenta



Una clase llamada Account, que modela una cuenta bancaria, está diseñada como se muestra en el diagrama de clases. Contiene:

- Dos atributos miembro privados: accountNumber (int) y balance(double) que mantiene el saldo de la cuenta corriente.
- Dos constructores (sobrecargados).
- Getters y Setters para los atributos miembro privados. No hay un setter para accountNumber ya que no está diseñado para ser cambiado.
- Los métodos públicos credit() y debit(), que suman/restan el monto dado a/de el saldo, respectivamente.
- El método toString(), que devuelve "A/C no: xxx, Balance = $ xxx.xx", con saldo redondeado a dos decimales.

Escribe la clase de Account y una TestAccount para probar todos los métodos públicos.

Niveles de protección: paquetes

Un paquete en Java es un contenedor de clases que permite agrupar las distintas clases de un programa que cumplen una tarea en común. El uso de paquetes proporciona las siguientes ventajas:

- Agrupamiento de clases con características comunes.
- Reutilización de código.
- Mayor seguridad al existir niveles de acceso.
- En Java todas las clases agrupadas dentro de un mismo paquete se denominan clases amigas ya que van a compartir una serie de privilegios a la hora de accederse unas a otras.

Niveles de protección: paquetes

Dentro de un fichero de código Java se utiliza la palabra reservada **package** para especificar a qué paquete pertenece la clase en cuestión. Suele indicarse en la primera sentencia del fichero:

```
package prog.figuras;

public class Circulo {

   private double radio;
   private String color;
   ...
```

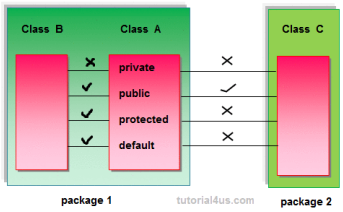
Un fichero Java no puede encontrarse dentro de más de un paquete a la vez, pero si es posible que el paquete en el que se encuentra sea un subpaquete de otro paquete, esta jerarquía se indica mediante el operador punto.

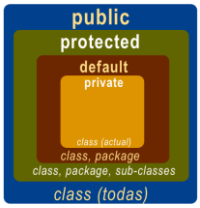
Protección de miembros de una clase

Protección de miembros de una clase:

- Los métodos o atributos **públicos** pueden ser accedidos por cualquiera, no tienen ninguna limitación de acceso.
- Los métodos o atributos **privados** sólo pueden ser accedidos por la propia clase, nadie puede acceder a ellos desde fuera de la propia clase.
- Los métodos o atributos **protegidos** permiten ser accedidos además de por la propia clase, por las clases amigas (del mismo paquete) y por las clases que heredan de ellas (lo estudiaremos más adelante).
- Los métodos o atributos **por defecto** permiten ser accedidos demás de por la propia clase, por las clases amigas (del mismo paquete).

Por lo general se **desaconseja el uso de la protección por defecto,** tanto por claridad como por funcionalidad.





Protección de una clase

Por **protección de clases** entendemos un nivel superior del ocultamiento de la información. Es decir, se trata de especificar qué clases pueden ser utilizadas y cuales no, y por quien.

- Para la protección de clases sólo están permitidos los modificadores public y por defecto.
- Una clase pública podrá ser utilizada por cualquier otra clase, mientras que una clase por defecto, sólo podrá utilizarse por las clases del mismo paquete.
- Existe un nivel más de ocultación con las clases interiores y anidadas, pero se consigue el mismo efecto con la jerarquía de directorios y los modificadores por defecto y públicos

OBJETOS COMO ATRIBUTOS DE OTRAS CLASE

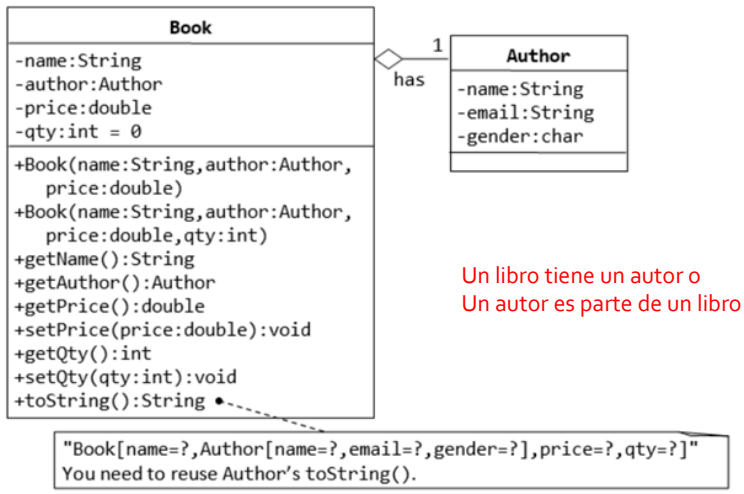
Como se vio en apartados anteriores, los atributos de una clase además de tipos primitivos, también pueden ser objetos de otras clases.

¿Qué cambia esto en nuestra forma de trabajar?

- En los constructores habrá que inicializar estos atributos llamando a sus respectivos constructores.
- En los métodos de la clase podremos invocar a los métodos y atributos del objeto que se ha declarado como atributo.

Generalmente sólo podremos acceder a los métodos de esta clase y no a los atributos directamente.

OBJETOS COMO ATRIBUTOS DE OTRAS CLASES



Un libro tiene un autor o
Un autor es parte de un libro

Aplicaciones orientadas a objetos

En una aplicación orientada a objetos debe existir una clase que represente la propia aplicación. Este sería el punto donde comienza la ejecución de la misma.

En un lenguaje orientado a objetos puro como Java esta clase de aplicación es obligatoria.

La máquina virtual Java se encarga de instanciar esta clase y llamar al método principal de esa: **main**. La existencia de este método principal es lo que caracteriza a la clase de la aplicación.

La clase de aplicación debe de ser publica y no tener ningún constructor y en la medida de lo posible no más métodos que le main:

```
public static void main(String[] args)
```

Aplicaciones orientadas a objetos

Aunque en el resto de clases también se puede definir un método **main** se desaconseja su uso, excepto para probar el correcto funcionamiento del resto de métodos y atributos de la clase.

De hecho la mayoría de los autores recomiendan una vez comprobado el correcto funcionamiento de la clase, borrar el método main (o al menos comentarlo) por motivos de seguridad.

Implementación de clases

¿Qué hace que una clase sea buena? Lo más importante es que una clase debe representar un único concepto de un dominio del problema. Algunas de las clases que has visto representan conceptos de las matemáticas:

- Punto
- Rectángulo
- Elipse

Otras clases son abstracciones de entidades de la vida real:

- CuentaBancaria
- CajaAhorros

Implementación de clases

Para estas clases, las propiedades de un objeto típico son fáciles de entender. Un Rectángulo tiene un ancho y un alto. Dado un objeto CuentaBancaria se puede depositar y retirar dinero.

Generalmente, los conceptos de la parte del universo a la que un programa como la ciencia, los negocios o un juego, son buenas clases.

El nombre para una clase de este tipo debe ser un **sustantivo** que describa el concepto. De hecho, una simple regla depara empezar a diseñar clases es buscar sustantivos en la descripción del problema.

Implementación de clases

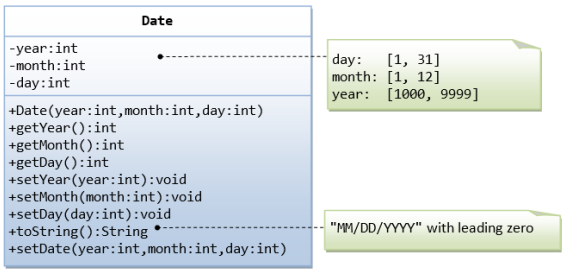
Una categoría útil de clases puede describirse como actores. Los objetos de una clase actor llevan a cabo ciertas tareas por ti. Ejemplos de actores son la clase Scanner que escanea un flujo en busca de números y cadenas. La clase Random, que genera números aleatorios.

Es una buena idea elegir nombres de clase para los actores que terminen en "-er" o "-or". (Un nombre mejor para la clase **Random** podría ser **RandomNumberGenerator**)

Implementación de clases

Por último, se han visto clases con solo un método, el main. Su único propósito es iniciar un programa. Desde una perspectiva de diseño, estos son ejemplos algo degenerados de clases.

Actividad: clase Fecha

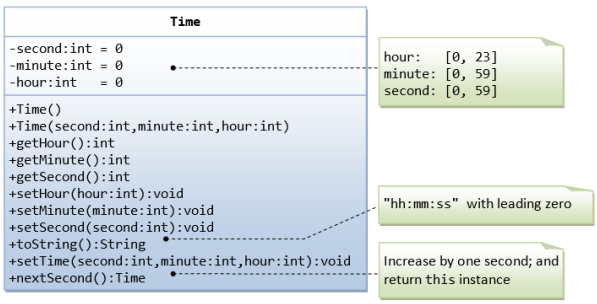


Una clase de Date modela una fecha de calendario con día, mes y año, está diseñada como se muestra en el diagrama de clase. Contiene:

- Tres atributos miembro privados día, mes y año.
- Constructores, getters y setters públicos para los atributos miembro privados.
- Un método setDate(), que establece el día, mes y año.
- Un método toString (), que devuelve "DD/MM/AAAA", con cero a la izquierda para DD y MM si corresponde.

Escribe la clase Date y la clase TestDate para probar todos los métodos públicos. No se requieren validaciones de entrada para día, mes y año.

Actividad: clase Hora



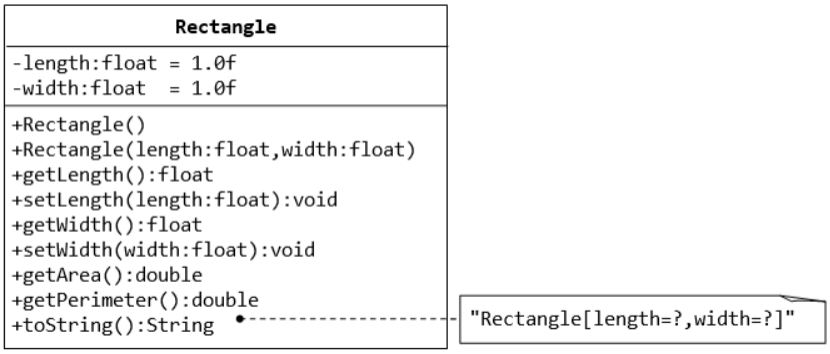
Una clase llamada Time , que modela una instancia de tiempo con hora, minuto y segundo, está diseñada como se muestra en el diagrama de clase. Contiene:

- Tres atributos miembro privados: hora, minuto y segundo.
- Constructores, setters y getters.
- Un método setTime() para establecer horas, minutos y segundos.
- Un método toString() que devuelve "hh: mm: ss" con cero a la izquierda si corresponde.
- Un método nextSecond () que avanza esta instancia en un segundo. Devuelve esta instancia para admitir operaciones de encadenamiento (en cascada), por ejemplo, t1.nextSecond ().nextSecond (). Tenga en cuenta que el nextSecond () de 23:59:59 es 00:00:00.

Escribe la clase Time y una clase TestTime para probar todos los métodos públicos. No se requieren validaciones de entrada.

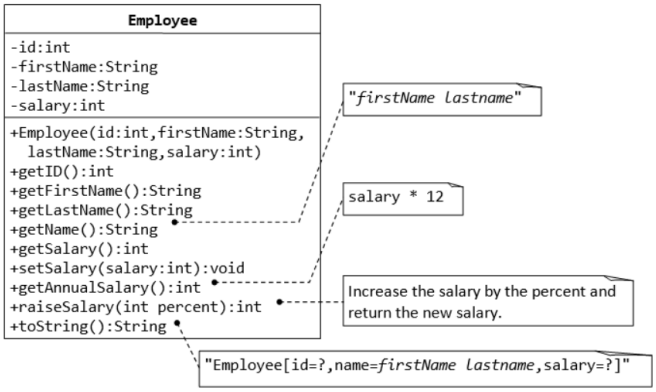
Actividad: clase Rectángulo

Escribe la clase de Rectangle y una TestRectangle para probar todos los métodos públicos.



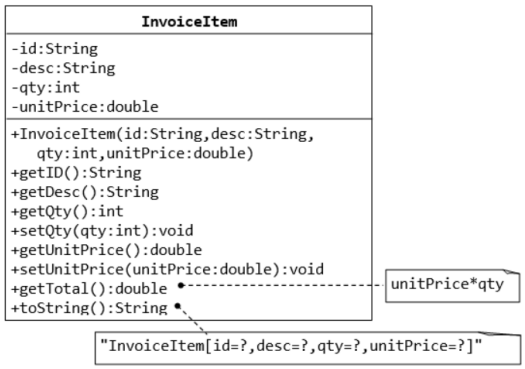
Actividad: clase Empleado

Escribe la clase de Employee y una TestEmployee para probar todos los métodos públicos.



Actividad: clase LíneaFactura

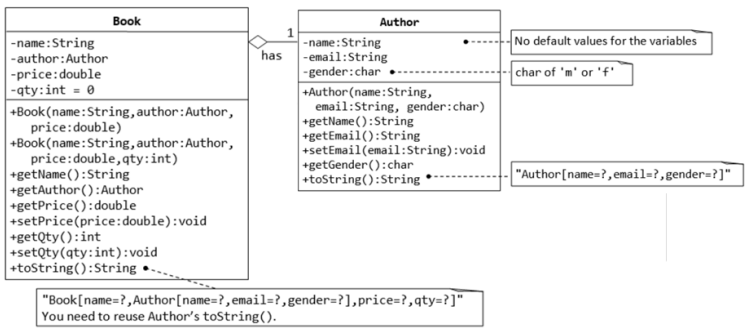
Escribe la clase de InvoiceItem y una TestInvoiceItem para probar todos los métodos públicos.



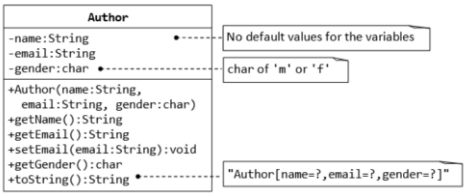
Actividad: clases Autor y Libro

Una introducción a la composición OO

- Comenzaremos primero con la clase
- Autor y luego la clase Libro



Actividad: clase Autor



Una clase llamada **Author** (se muestra en el diagrama de clase) está diseñada para modelar el autor de un libro. Contiene:

- Tres atributos miembro privados: nombre (String), correo electrónico (String) y género (carácter de 'm' o 'f');
- Un constructor para inicializar el nombre, correo electrónico y género con los valores dados; public Author (String name, String email, char gender) {.....}

(No hay un constructor por defecto para el Autor, es decir, no hay valores predeterminados para el nombre, correo electrónico y género).

- getters / setters públicos: getName (), getEmail (), setEmail () y getGender ();

(No hay setters para nombre y género, estos atributos no se pueden cambiar).

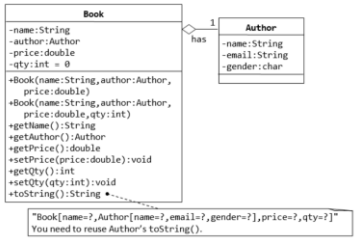
- Un método toString() que devuelve "Autor [nombre =, Correo electrónico =, Género =?]", Por ejemplo, "Autor [nombre = Tan Ah Teck, correo electrónico = ahTeck@somewhere.com, género = m]".

Actividad: clase Autor

Escribe la clase Author. También escriba el controlador de prueba llamado TestAuthor para probar todos los métodos públicos, por ejemplo:

```
Author ahTeck = new Author("Tan Ah Teck", "ahteck@nowhere.com", 'm'); // Test the constructor
System.out.println(ahTeck);  // Test toString()
ahTeck.setEmail("paulTan@nowhere.com");  // Test setter
System.out.println("name is: " + ahTeck.getName());     // Test getter
System.out.println("eamil is: " + ahTeck.getEmail());   // Test getter
System.out.println("gender is: " + ahTeck.getGender()); // Test gExerciseOOP_MyPolynomial.pngetter
```

Actividad: clase Libro



Una clase llamada **Book** está diseñada (como se muestra en el diagrama de clase) para modelar un libro escrito por un autor. Contiene:

- Cuatro atributos miembro privados: nombre (String), autor (de la clase Author que acaba de crear, suponga que un libro tiene un solo autor), precio (doble) y qty (int);
- Dos constructores:
  public Book (String name, Author author, double price) { ..... }
  public Book (String name, Author author, double price, int qty) { ..... }
- Métodos públicos getName (), getAuthor (), getPrice (), setPrice (), getQty (), setQty ().
- Un método toString() que devuelve "Libro [nombre =, Autor [nombre =, correo electrónico =, género =?], Precio =, cantidad =?". Debe reutilizar el toString () del autor.

Actividad: clase Libro

Escribe la clase Book (que usa la clase Author escrita anteriormente). También escribe un controlador de prueba llamado TestBook para probar todos los métodos públicos en la clase Book. Ten en cuenta que se debe crear una instancia de Author antes de poder crear una instancia de Book. P.ej.,

```
// Construct an author instance
Author ahTeck = new Author("Tan Ah Teck", "ahteck@nowhere.com", 'm');
System.out.println(ahTeck);  // Author's toString()

Book dummyBook = new Book("Java for dummy", ahTeck, 19.95, 99);  // Test Book's Constructor
System.out.println(dummyBook);  // Test Book's toString()

// Test Getters and Setters
dummyBook.setPrice(29.95);
dummyBook.setQty(28);
System.out.println("name is: " + dummyBook.getName());
System.out.println("price is: " + dummyBook.getPrice());
System.out.println("qty is: " + dummyBook.getQty());
System.out.println("Author is: " + dummyBook.getAuthor());  // Author's toString()
System.out.println("Author's name is: " + dummyBook.getAuthor().getName());
System.out.println("Author's email is: " + dummyBook.getAuthor().getEmail());

// Use an anonymous instance of Author to construct a Book instance
Book anotherBook = new Book("more Java",
      new Author("Paul Tan", "paul@somewhere.com", 'm'), 29.95);
System.out.println(anotherBook);  // toString()
```

Actividad: clases Autor y Libro

Ten en cuenta que las clases de Book y Author tienen un atributo miembro nombre. Sin embargo, se puede diferenciar a través de la variable de referencia

Para una instancia de Book, digamos aBook, aBook.name se refiere al nombre del libro; mientras que para la instancia de un Author digamos anAuthor, anAuthor.name se refiere al nombre del autor.

No es necesario (y no se recomienda) llamar a las variables bookName y authorName.

Actividad: clases Autor y Libro

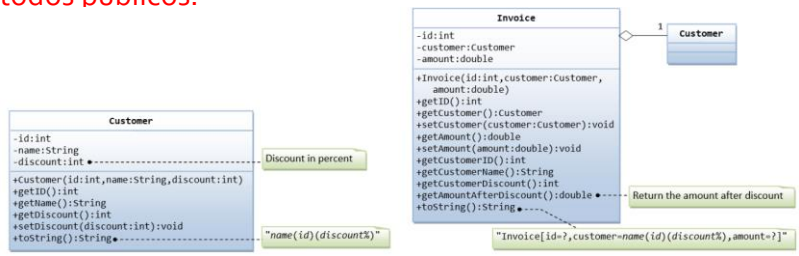
PRUEBA ESTO:

- Imprime el nombre y correo electrónico del autor desde una instancia de Book. (Sugerencia: aBook.getAuthor ().getName (), aBook.getAuthor ().getEmail ()).
- Introduce nuevos métodos llamados getAuthorName(), getAuthorEmail(), getAuthorGender() en la clase Book para devolver el nombre, correo electrónico y género del autor del libro. Por ejemplo,

```
public String getAuthorName() {
   return author.getName();
      // cannot use author.name as name is private in Author class
}
```

Actividad: clases Cliente y Factura

Una clase llamada Cliente, que modela un cliente en una transacción, está diseñada como se muestra en el diagrama de clases. También se muestra una clase llamada Factura, que modela una factura para un cliente en particular y compone una instancia de Cliente como atributo. Escribe las clases Cliente y Factura. Escribe las clases Customer, Invoice y copia la clase TestInvoiceCustomer para probar todos los métodos públicos.



Actividad: clases Cliente y Factura

```
public class TestInvoiceCustomer {
   public static void main(String[] args) {
      // Test Customer class
      Customer c1 = new Customer(88, "Tan Ah Teck", 10);
      System.out.println(c1);  // Customer's toString()

      c1.setDiscount(8);
      System.out.println(c1);
      System.out.println("id is: " + c1.getID());
      System.out.println("name is: " + c1.getName());
      System.out.println("discount is: " + c1.getDiscount());

      // Test Invoice class
      Invoice inv1 = new Invoice(101, c1, 888.8);
      System.out.println(inv1);

      inv1.setAmount(999.9);
      System.out.println(inv1);
      System.out.println("id is: " + inv1.getID());
      System.out.println("customer is: " + inv1.getCustomer());  // Customer's toString()
      System.out.println("amount is: " + inv1.getAmount());
      System.out.println("customer's id is: " + inv1.getCustomerID());
      System.out.println("customer's name is: " + inv1.getCustomerName());
      System.out.println("customer's discount is: " + inv1.getCustomerDiscount());
      System.out.printf("amount after discount is: %.2f%n", inv1.getAmountAfterDiscount());
   }
}
```

Resumen de modificadores que podemos usar por ahora

```
public MiClase {

    private tipo atributo1;
    public/private final static tipo CONS;
    public/private static tipo atributo2;


    public/private tipo metodo1(...){...}
    public/private static tipo metodo3(...){...}

    //para eviar abusar de métodos estáticos
    //por ahora solo pueden ser declarados
    //cuándo el enunciado lo solicite
}
```

	público	privado
atributo	viola el encapsulamiento	fuerza el encapsulamiento
método	provee servivios a otras clases	da soporte a otros métodos de la clase

Principio KISS

El principio KISS establece que la mayoría de sistemas funcionan mejor si se mantienen **simples** que si se hacen complejos; por ello, la simplicidad debe ser mantenida como un objetivo clave del diseño, y cualquier complejidad accidental debe ser evitada.

Posiblemente tiene su origen en el marketing y las presentaciones de ventas, para ser utilizado después en el desarrollo de sistemas, sobre todo para evitar que los sucesivos desarrollos en los diseños se complicaran

¿Qué es simple?

Simple no es fácil…

Fácil es la mínima cantidad de esfuerzo para producir un resultado.

Simple es eliminar todo excepto lo que importa.

Mejores prácticas para el diseño de clases

- **Encapsulación**: Mantiene oculto el estado interno de la clase y proporciona métodos para acceder y modificar el estado.
- **Principio de responsabilidad única:** que cada clase debe tener una única responsabilidad, y que esta debe estar contenida únicamente en la clase.
- **Principio Abierto-Cerrado:** Una clase debe estar abierta a ampliaciones, pero cerrada a modificaciones. Expresa la necesidad de que ante un cambio de los requisitos, el diseño de las entidades existentes permanezca inalterado, recurriéndose a la extensión del comportamiento de dichas entidades añadiendo nuevo código, pero nunca cambiando el código ya existente.
- **Principio de sustitución de Liskov:** las subclases deben poder sustituir a su clase padre sin que ello afecte al programa. (lo veremos más adelante)

Mejores prácticas para el diseño de clases

- **Principio de inversión de dependencia:** Las clases deben depender de abstracciones, no de implementaciones concretas. (lo veremos más adelante)
- **Simplicidad**: La clase debe ser pequeña y centrarse en una tarea específica. Evite complejidades innecesarias y asegúrese de que la clase es fácil de entender y mantener.
- **Sigue las convenciones de nomenclatura:** Siga las convenciones de nomenclatura de Java para clases, variables y métodos para que el código sea coherente y fácil de entender.
- **Utiliza interfaces:** Utiliza interfaces para definir el contrato de una clase e implementarlo en varias clases diferentes. Esto permite una mejor reutilización y flexibilidad del código. (lo veremos más adelante)
- Utilice patrones de diseño: Utilice patrones de diseño bien establecidos para resolver problemas comunes y hacer que su código sea más fácil de mantener (lo veremos más adelante)

Recursos

- Curso youtube : Java desde 0
- Libro Java 9. Manual imprescindible. F. Javier Moldes Teo. Editorial Anaya
- App SoloLearn: Aprende a Programar. Curso Java

PRINCIPIOS SOLID

- https://devexperto.com/principio-responsabilidad-unica/
- https://devexperto.com/principio-open-closed/
- https://devexperto.com/principio-de-sustitucion-de-liskov/
- https://devexperto.com/principio-de-segregacion-de-interfaces/
- https://devexperto.com/principio-de-inversion-de-dependencias/