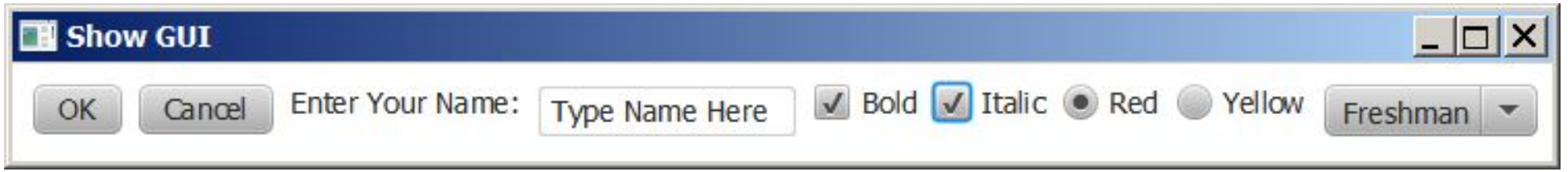
Capítulo 9

Objetos y Clases

Motivaciones

Después de lo aprendido en los capítulos anteriores, podemos resolver muchos problemas de programación mediante selecciones, bucles, métodos y matrices. Sin embargo, estas características de Java no son suficientes para desarrollar interfaces gráficas de usuario y sistemas de software a gran escala. Supongamos que deseamos desarrollar una interfaz gráfica de usuario como se muestra a continuación. ¿Cómo lo programariamos?



Objetivos

- Describir objetos y clases, y usar clases para modelar objetos.
- Demostrar cómo definir clases y crear objetos.
- Crear objetos usando constructores.
- Acceder a objetos a través de variables de referencia de objeto.
- Definir una variable de referencia usando un tipo de referencia.
- Acceder a los datos y métodos de un objeto utilizando el operador de acceso (.).
- Definir campos de datos de tipos de referencia y asignar valores por defecto para los campos de datos de un objeto.
- Distinguir entre las variables de referencia del objeto y las variables de datos primitivos.
- Distinguir entre instancia y variables estáticas y métodos.
- Definir campos de datos privados con los métodos de obtención y configuración apropiados.
- Encapsular campos de datos para hacer que las clases sean fáciles de mantener.
- Desarrollar métodos con argumentos de objeto.
- Almacenar y procesar objetos en matrices.
- Determinar el alcance de las variables en el contexto de una clase.
- Usar la palabra clave this para referirse al mismo objeto que llama.
- Usar las clases de biblioteca de Java LocalDate y Random.
- Usar las clases contenedoras numéricas: Integer y Double.
- Estructuras dinámicas Colas y Pilas.

Conceptos de programación OO

La programación orientada a objetos (POO) implica la programación utilizando objetos.

Un *objeto* representa una entidad en el mundo real que puede identificarse claramente. Por ejemplo, un estudiante, un escritorio, un círculo, un botón e incluso un préstamo se pueden ver como objetos.

Un objeto tiene una identidad, estado y comportamiento únicos. El *estado* de un objeto consiste en un conjunto de *campos de datos* (también conocidos como *propiedades*) con sus valores actuales. El *comportamiento* de un objeto está definido por un conjunto de métodos.

Objetos

Un objeto tiene un estado y un comportamiento. El estado o estructura define el objeto, y el comportamiento define lo que hace el objeto.

ESTRUCTURA

- Nombre
- Fecha nacimiento
- Email
- Avatar
- ...

COMPORTAMIENTO

- Login
- Logout
- Consultar productos
- Realizar compras
- Pagar
- ...

Clases

Las *clases* son construcciones que definen objetos del mismo tipo. Podemos ver una clase como una plantilla o modelo para crear objetos.

Una clase Java usa variables para definir campos de datos y métodos para definir comportamientos. Además, una clase proporciona un tipo especial de métodos, conocidos como constructores, que se invocan para construir objetos de la clase.

Las clases son un nuevo tipo de datos, pudiendo definir variables de ese tipo e inicializarlas mediante instanciación.

La definición e implementación de una clase en Java se hace en un mismo archivo (por ahora, una única clase por fichero).

Definición de Clases

Sintaxis de la Definición de una clase

```
<modificador> class NombreDeLaClase {

        //propiedades
        int propiedad1;
        String propiedad2;
        float propiedad3;
        //…

        //metodos
        void metodo1() {
                //…
        }

        //…
}
```

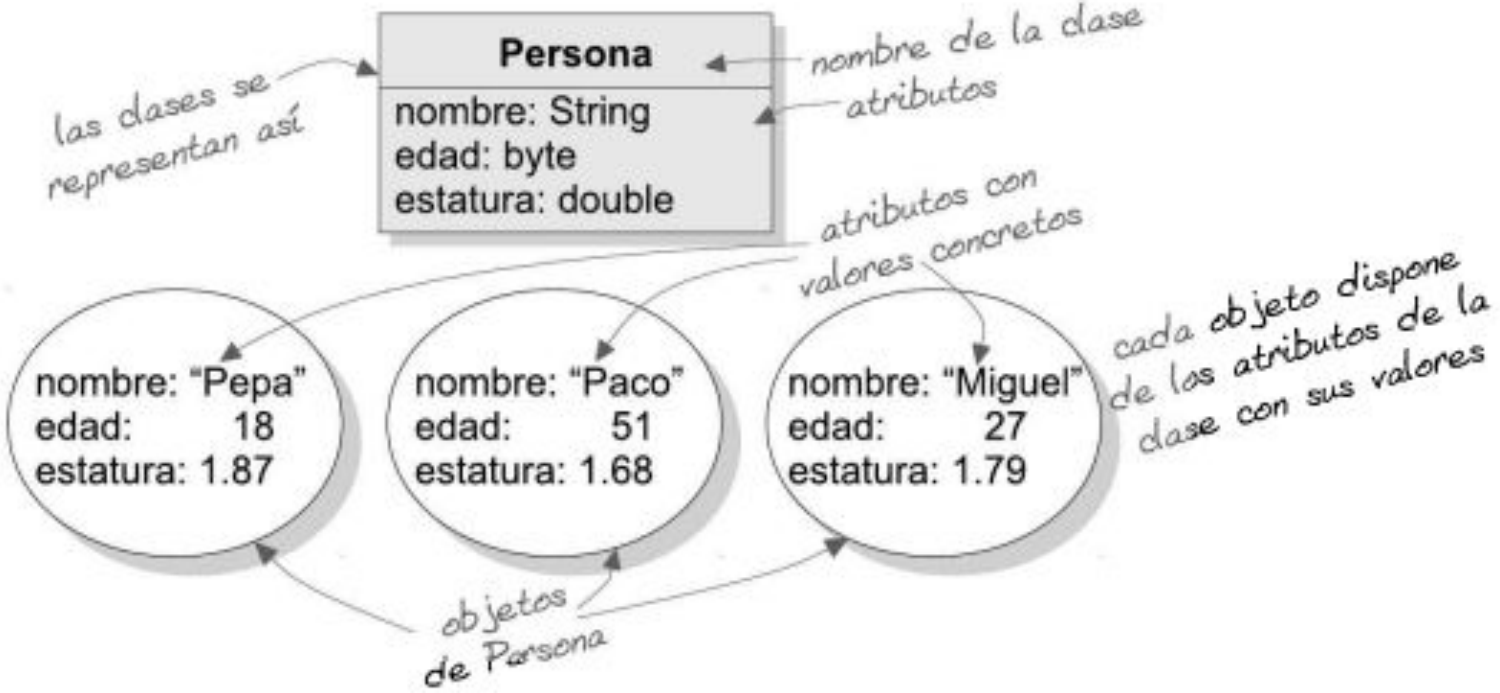
Ejemplo1: Definición de una clase sólo con atributos

```
class Persona {
   String nombre;
   byte edad;
   double estatura;
}
```

Definición de Objetos

Los elementos que pertenecen a una clase se denominan instancias u objetos.

Todos los objetos que se creen a partir de una clase tendrán los mismos componentes, aunque cada uno tendrá sus propios valores.



Creación de Objetos con new

Antes de construir objetos necesitamos declarar variables cuyo tipo sea una clase.

Clase nombreVariable;

Ejemplo:

Persona p; *//p es una variables de tipo Persona*

La forma de crear objetos es mediante el operador new.

Ejemplo:

p = new Persona(); *//p contiene la referencia al objeto*

Una vez creado el objeto podemos asignar valores a sus atributos.

Ejemplo:

```
p.nombre = "Pepa";
p.edad = 18;
p.estatura = 1.87;
```

Creación de Clases, cont

Continuamos con la creación de clases, añadiéndoles comportamiento mediante la definición de métodos.
Ejemplo:

```
public class Persona {
   String nombre;
   byte edad;
   double estatura;

   void saludar() {
      System.out.println("Hola. Mi nombre es " + nombre);
      System.out.println("Encantando de conocerte");
   }
   void cumplirAños() {
      edad++; //incrementamos la edad en 1
   }
   void crecer(double incremento) {
      estatura += incremento; //la estatura aumenta cierto incremento
   }
} //de la clase
```

Miembros de un objeto

Los miembros de un objeto son normalmente atributos y métodos, que se definen en la clase.

Los atributos son variables cuyo ámbito es el del objeto al que pertenecen. Por tanto están disponibles durante el ciclo de vida de dicho objeto. Cuando se crea un objeto, se crea dentro de él, una copia de cada uno de los atributos.

Los métodos son como las funciones, pero se ejecutan en el contexto del objeto con el que son invocados. Dentro de los métodos se puede hacer referencia a cualquier otro miembro del objeto (atributos y métodos). Pueden usar y modificar todos los atributos del objeto.

Pautas de diseño de clase

Cohesión.-
Una clase debe describir una sola entidad, y todas las operaciones de clase deben encajar lógicamente entre sí para apoyar un propósito coherente. Por ejemplo, podemos usar una clase para estudiantes, pero no debemos combinar estudiantes y personal en la misma clase, porque los estudiantes y el personal tienen entidades diferentes.

Consistencia.-
Siga el estilo de programación Java estándar y las convenciones de nomenclatura.
Un estilo muy usado es colocar la declaración de datos antes del constructor y colocar constructores antes de los métodos e inicialice las variables para evitar errores de programación.

Pautas de diseño de clase

En general, debe proporcionar un constructor público no-arg. Si una clase no es compatible con un constructor no-arg, documente el motivo. Si no se definen explícitamente constructores, se supone un constructor no-arg predeterminado público con un cuerpo vacío.
Si desea evitar que los usuarios creen un objeto para una clase, puede declarar un constructor privado en la clase, como es el caso de la clase Math

Encapsulación.-
Una clase debe usar el modificador privado para ocultar sus datos del acceso directo. Esto hace que la clase sea fácil de mantener. Los objetos conocen solamente su estructura, no la de los demás.

Pautas de diseño de clase

Normalmente, los atributos de un objeto se deben consultar o editar a través de métodos. Deben ser privados para proteger los datos y para hacer que el código sea fácil de mantener.

El trato entre objetos se realiza a través de los métodos. Proporcione un método getter solo si desea que el campo de datos sea legible, y proporcione un método setter solo si desea que el campo de datos sea actualizable.

Ejemplo: definición de clases y creación de objetos

El signo + indica un modificador público.

TV
-channel: int -volumeLevel: int -on: Boolean -static nuemroTVs: int
+TV() +turnOn(): void +turnOff(): void +getChannel():int +setChannel(newChannel: int): void +getVolume():int +setVolume(newVolumeLevel: int): void +getOn():Boolean +setOn(newOn: Boolean):void +getNuemroTV():int +channelUp(): void +channelDown(): void +volumeUp(): void +volumeDown(): void

El canal actual (1 a 120) de este TV.

El nivel de volumen actual (1 a 7) de este TV.

Indica si este TV está encendido / apagado.

Construye un objeto de TV predeterminado.

Enciende este TV.

Apaga este televisor

Establece un nuevo canal para este TV.

Establece un nuevo nivel de volumen para este televisor.

Aumenta el número de canal en 1.

Reduce el número de canal en 1.

Aumenta el nivel de volumen en 1.

Disminuye el nivel de volumen en 1.

Campos de datos o Atributos

Los atributos pueden ser de cualquier tipo, primitivos u otras clases de tipo de referencia. Pero tienen que tener el tipo definido, no se puede usar **var**.

Si un campo de datos de un tipo de referencia no hace referencia a ningún objeto, el campo de datos tiene un valor literal especial, null.

Por ejemplo, la siguiente clase Student contiene un nombre de campo de datos de tipo String.

```
public class Student {
  String name; // name por defecto inicializado como null
  int age; // age por defecto inicializado a 0
  boolean isScienceMajor; // isScienceMajor por defecto false
  char gender; // c su valor por defecto es '\u0000'
}
```

Valor predeterminado para un campo de datos

El valor predeterminado de un campo de datos es ***null*** para un tipo de referencia, ***0*** para un tipo numérico, ***falso*** para un tipo booleano y '\ ***u0000***' para un tipo de carácter. Sin embargo, Java no asigna ningún valor predeterminado a una variable local dentro de un método.

```
public class Test {
  public static void main(String[] args) {
    Student student = new Student();
    System.out.println("name? " + student.name);
    System.out.println("age? " + student.age);
    System.out.println("isScienceMajor? " +
student.isScienceMajor);
    System.out.println("gender? " + student.gender);
  }
}
```

Alcance de variables y atributos

- El alcance de la instancia y las variables estáticas es toda la clase. Se pueden declarar en cualquier lugar dentro de una clase.
- El alcance de una variable local comienza desde su declaración y continúa hasta el final del bloque que contiene la variable. Una variable local debe inicializarse explícitamente antes de poder usarse.
- Cualquier atributo o método definido en una clase podrá ser utilizado en cualquier lugar de la misma.

La palabra clave this

La palabra clave this es el nombre de una referencia que se refiere a un objeto en sí mismo. Un uso común de la palabra clave this hace referencia a campos de datos ocultos de una clase. Y sirve para diferenciar variables que se llaman igual.

```
public class Ambito {
   int edad; //atributo entero

   void metodo() {
      double edad; //oculta el atributo edad (entero)

      edad = 20.0; //variable local, no el atributo
      this.edad = 30; //atributo de la clase
   }
```

Constructores

Los constructores son un tipo especial de métodos que se invocan para construir objetos e inicializar los campos de datos. El operador new facilita esta tarea mediante los constructores.

Los constructores tienen una sintaxis parecida a la de los métodos pero también tienen diferencias importantes.

Su sintaxis es

```
public NombreClase (lista_de_parámetros) {
    acciones;
}
```

El modificador public es opcional y sirve para que se pueda usar el constructor desde cualquier clase. De momento lo usaremos siempre así.

Ejemplo Constructor

Creamos una clase Coche con los atributos(marca, modelo y anio) y método(arrancar), en la aplicación (clase que contiene el main) creamos un objeto con el constructor y llamamos al método arrancar():

```
package orientacionobjetos;

public class Coche {

    private String marca;
    private String modelo;
    private int anio;

    public Coche(String marca, String modelo, int anio) {
        this.marca = marca;
        this.modelo = modelo;
        this.anio = anio;
    }

    public void arrancar() {
        System.out.println("El coche %s %s %d ha arrancado"
                    .formatted(marca, modelo, anio));
    }


}
```

```
package orientacionobjetos;

public class App {

    public static void main(String[] args) {

        Coche coche = new Coche("Ford", "Focus", 2020);
        coche.arrancar();

    }

}
```

Ejemplo Constructor

Creamos una clase Producto con los atributos(nombre y precio) y métodos(aplicarDescuento y mostraInformacion). En la aplicación (clase que contiene el main) creamos un objeto con el constructor y llamamos al método aplicarDescuento() y una vez ejecutado llamamos a mostrarInformacion():

```
public class Producto {
    String nombre;
    double precio;
    // Constructor
    public Producto(String nombre, double precio) {
        this.nombre = nombre;
        this.precio = precio;
    }
    public void aplicarDescuento(double porcentaje) {
        this.precio = this.precio - (this.precio * porcentaje / 100);
    }
    public void mostrarInformacion() {
        System.out.println("Producto: " + this.nombre + ", Precio: $" + this.precio);
    }
}

// Uso de la clase
public class Main {
    public static void main(String[] args) {
        Producto producto = new Producto("Laptop", 1000);
        producto.aplicarDescuento(10);
        producto.mostrarInformacion(); // Output: Producto: Laptop, Precio: $900.0
    }
}
```

Crear objetos usando constructores

Los constructores no devuelven nada. En realidad, cuando se crea un objeto con el operador new(), éste ejecuta el constructor y devuelve la referencia al objeto creado.

```
new ClassName();
```

El nombre tiene que coincidir exactamente con el de la clase.
La lista de parámetros se define igual que en los métodos.

Un constructor sin parámetros se conoce como constructor no-arg, se proporciona automáticamente solo si no hay constructores explícitamente definidos en la clase.

Los constructores se pueden sobrecargar, es decir se pueden definir varios constructores siempre que difieran en sus listas de parámetros.

Ejemplo Constructores

Vamos a implementar un constructor para Persona, que asigne los valores iniciales de los atributos.
Definición de la clase:

```
class Persona {
   ...
   Persona (String nombre, int edad, double estatura) {
      this.nombre = nombre;
      this.edad = edad;
      this.estatura = estatura;
   }
   ...
}
```

Los valores de los parámetros de entrada se especifican al crear el objeto con new en la llamada al constructor.
Creación de un objeto Persona:

```
Persona p = new Persona("Claudia", 8, 1.20); //creamos el objeto
                                             //y lo inicializamos mediante el constructor
```

Ejemplo Constructores

También podríamos conocer el nombre de la persona, fijando valores arbitrarios para el resto de atributos o valores por defecto. Definición del constructor sobrecargado:

```
class Persona {

    //constructor sobrecargado que solo asigna el nombre
    Persona (String nombre) {
       this.nombre = nombre;
       estatura = 1.0; //valor arbitrario para la estatura
       //al no asignar la edad se inicializa por defecto: a 0
    }
}
```

Ahora disponemos de un segundo constructor que se utilizará de la forma:

```
Persona b = new Persona("Dolores");
```

Constructores, cont.

Si no especificamos ningún constructor en una clase, el compilador crea implícitamente uno por defecto (sin parámetros) y con el cuerpo vacío.

Los constructores no se consideran miembros del objeto, ya que una vez creado el objeto, ya no se puede llamar al constructor para ese objeto.

Las acciones se usan normalmente para inicializar los valores del objeto recién creado.

En Java no existen los destructores, se destruyen automáticamente por la JVM.

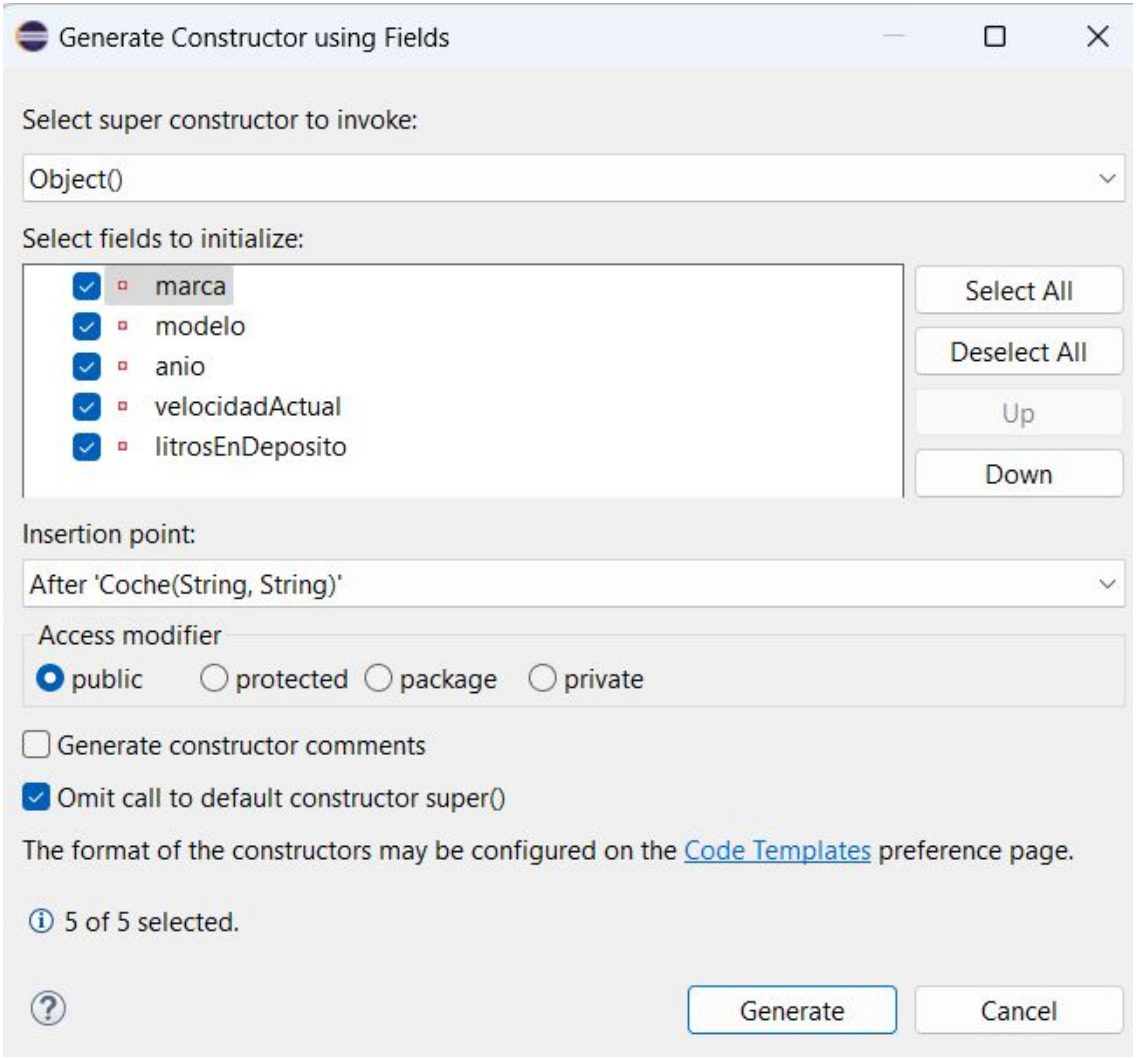
Ejemplo Constructor

Creamos una clase Animal con distintos constructores y el método(mostraInformacion).

```
public class Animal {
    String especie;
    int edad;
    // Constructor vacío
    public Animal() {
        this.especie = "Desconocida";
        this.edad = 0;
    }
    // Constructor con parámetros
    public Animal(String especie, int edad) {
        this.especie = especie;
        this.edad = edad;
    }
    public void mostrarInformacion() {
        System.out.println("Especie: " + especie + ", Edad: " + edad + " años");
    }
}

// Uso de la clase
public class Main {
    public static void main(String[] args) {
        Animal animal1 = new Animal();
        Animal animal2 = new Animal("Gato", 3);

        animal1.mostrarInformacion(); // Output: Especie: Desconocida, Edad: 0 años
        animal2.mostrarInformacion(); // Output: Especie: Gato, Edad: 3 años
    }
}
```

Generación automática del Constructor.

Generamos el constructor automáticamente desde Eclipse:



Constructor genérico this()

Cuando una clase dispone de un conjunto de constructores sobrecargados, es posible que un constructor invoque a otro constructor y así reutilice su funcionalidad.

El problema es que los constructores no se invocan directamente por su nombre desde dentro de su clase sino con el constructor this() .

La forma de distinguir los distintos constructores, igual que cualquier método sobrecargado, es mediante el número y el tipo de parámetros de entrada, tienen que tener alguna diferencia.

En el caso de utilizar this(), tiene que ser la primera instrucción de un constructor.

Ejemplo Constructor con this()

```
class Persona {
   ...
   //constructor que asigna valores a todos los atributos
   Persona (String nombre, int edad, double estatura) {
      this.nombre = nombre;
      this.edad = edad;
      this.estatura = estatura;
   }

   //constructor sobrecargado que solo asigna el nombre
   Persona (String nombre) {
      this(nombre, 0, 1.0); //invoca al primer constructor
      //la edad se pone a 0 y la estatura a 1.0
   }
}
```

Paquetes

En Java usamos los paquetes para controlar la accesibilidad de unas clases desde otras por razones de seguridad y eficiencia.

Los paquetes se organizan de forma jerárquica a modo carpeta. Los paquetes son contenedores que permiten guardar las clases en comportamientos separados y a través de la importación poder decidir las clases que queramos tener visibles desde una clase.

Se importan las clases, no los paquetes. Si queremos importar todas las clases públicas de un paquete en una sola sentencia, se usa el asterisco.

```
import paquete1.*; //C puede utilizar cualquier clase visible en paquete1

class C {
   ...
}
```

Modificadores de acceso para clases

Por defecto, la clase, atributo o método puede ser accedido por cualquier clase en el mismo paquete. Sin embargo, existen modificadores de acceso que alteran la visibilidad permitiendo que se muestre u oculte.

Dos clase cualesquiera se pueden definir como:

- `Clases vecinas`

  Ambas están definidas en el mismo paquete. Todas las clases que pertenecen a un paquete son vecinas entre sí.

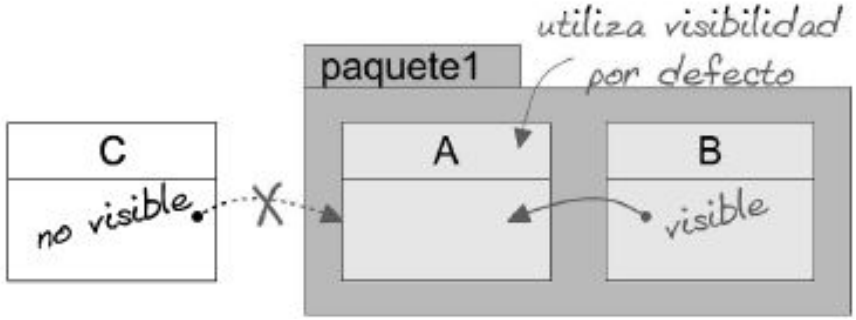
- `Clases externas`

  En caso de que ambas no estén definidas en el mismo paquete.

Modificadores de acceso para clases

Las clases se rigen por el lema *si lo ves, puedes usarlo*. Puedes crear objetos en ella o acceder a sus miembros estáticos.

- `Visibilidad por defecto`



- `Visibilidad total`

  Utilizando el modificador *public* se da visibilidad total a la clase. La clase externa C sólo necesitaría <u>importar</u> la clase A para acceder.

	Visible desde...	
	clases vecinas	clases externas
sin modificador	✓	
`public`	✓	✓

Modificadores acceso para miembros

Por defecto, cualquier miembro es visible dentro de su propia clase. Dentro de una clase siempre tendremos acceso a todos sus atributos y podremos invocar cualquiera de sus métodos.

- `Visibilidad por defecto`

  Un miembro es visible desde las clases vecinas, pero invisible desde las clases externas.

- `Modificador de acceso private y public`

  Solo se puede acceder a los datos o métodos por la clase declarante.

  - `private` Sólo se puede acceder a los datos o métodos por la clase declarante, pero no fuera de ella. Fuera es invisible.

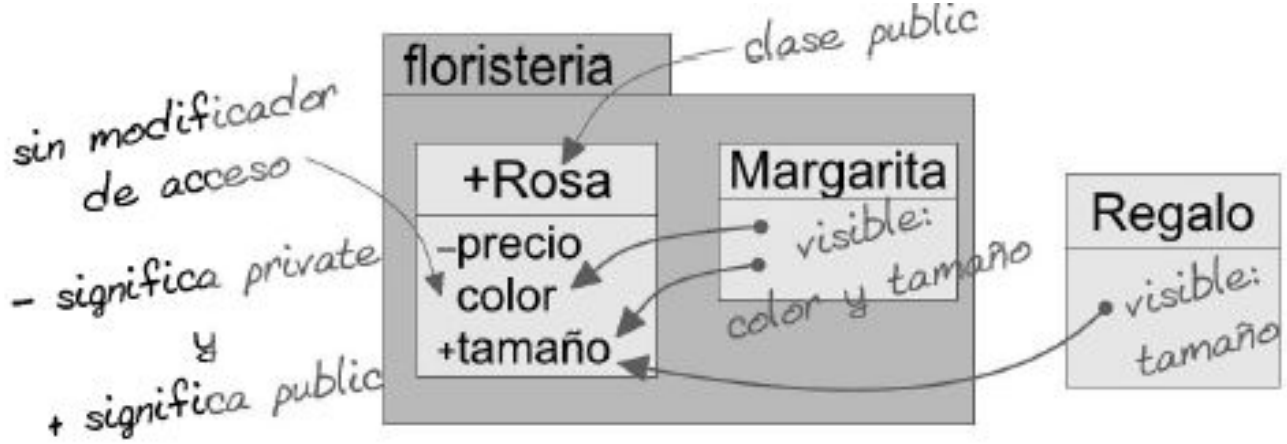
Modificadores acceso para miembros

- `public` Un miembro puede ser visible para cualquier clase externa.

Resumen alcance de la visibilidad según modificador:

	Visible desde...		
	la propia clase	clases vecinas	clases externas
`private`	✓		
sin modificador	✓	✓	
`public`	✓	✓	✓

Ejemplo visibilidad de private, public y por defecto:



Buenas Prácticas I

La mayoría de las clases que se crean son públicas.

Cada fichero .java tendrá solamente una clase pública, con el mismo nombre del fichero.

```
public class MiClase {
    //propiedades
    //…

    //metodos
    //…
}
```

Buenas Prácticas II

La mayoría de los atributos de una clase serán privados.

Solamente algunas constantes, o casos muy particulares, tendrán otro modificador de acceso.

```
public class MiClase {
    //propiedades
    private int numero;
    private String nombre;

    //metodos
    //…
}
```

Buenas Prácticas III

Si una clase tiene atributos, seguramente tenga métodos públicos.

Los métodos privados son interesantes para cálculos auxiliares o parciales (solo se pueden invocar desde la propia clase).

```
public class MiClase {
    //propiedades
    private int numero;
    private String nombre;
    //metodos
    public int getNumero { … }
}
```

Métodos get y set

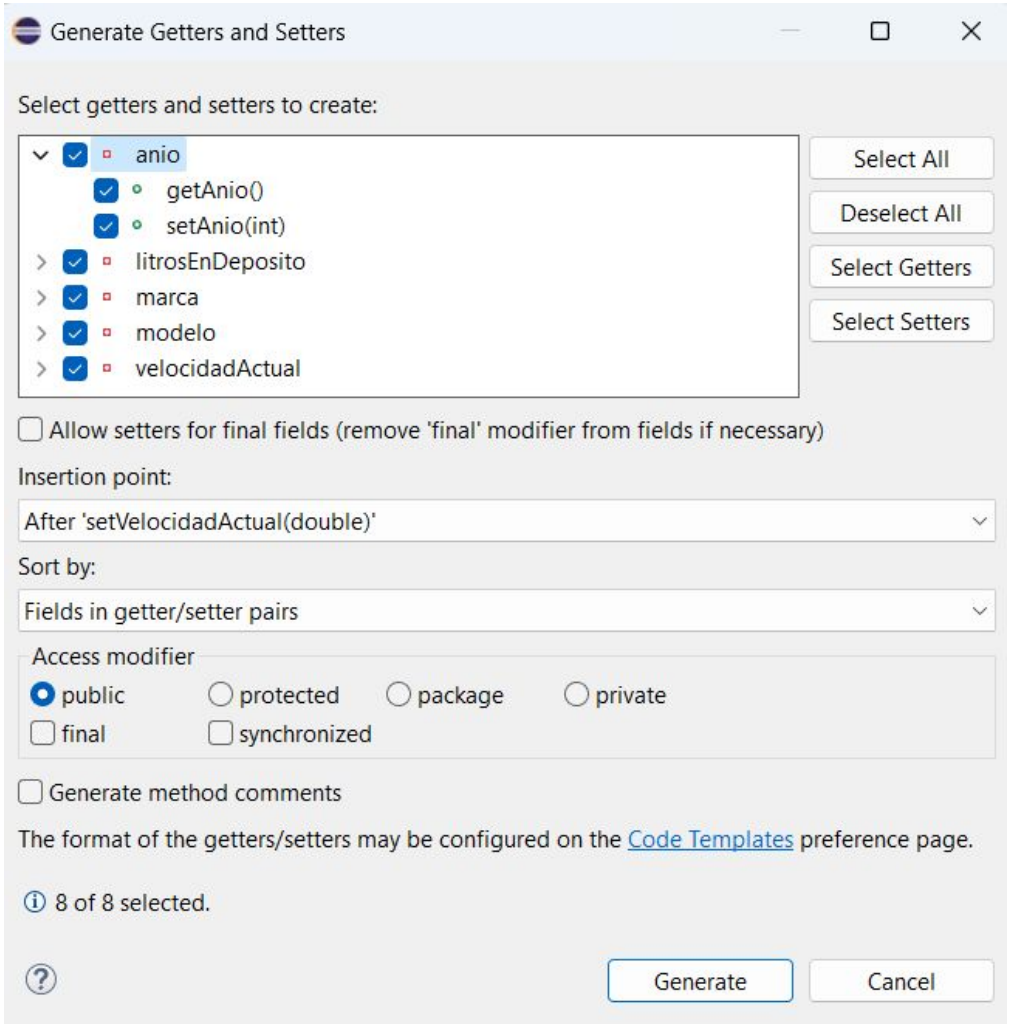
Los métodos get y set se utilizan para leer y modificar propiedades privadas, permitiendo controlar el valor asignado.

El método set asigna un valor al atributo, controlando el rango válido de valores.
El método get, devuelve el atributo, permite conocer su valor.

```
class Persona {
   private int edad;
   ...

   public void setEdad(int edad) {
      if (edad >= 0) //solo los valores positivos tienen sentido
         this.edad = edad;
      } // en caso contrario no se modifica la edad
   }

   public int getEdad() {
      return edad;
      }
   }
```

Creación automática los Métodos get y set

Los métodos get y set se pueden autogenerar desde la opción de Eclipse del menú Source.



Métodos especiales

La mayoría de las clases comparten algunos métodos comunes.

- toString
- equals
- hashCode

Estos tres métodos tienen siempre el mismo nombre y una misión específica.

La implementación de los métodos equals y hashCode va de la mano.

toString

Método útil para obtener una representación de un objeto.
Si no está implementado, y hacemos System.out.println(obj) aparece una representación de la referencia del objeto.

Métodos especiales

equals

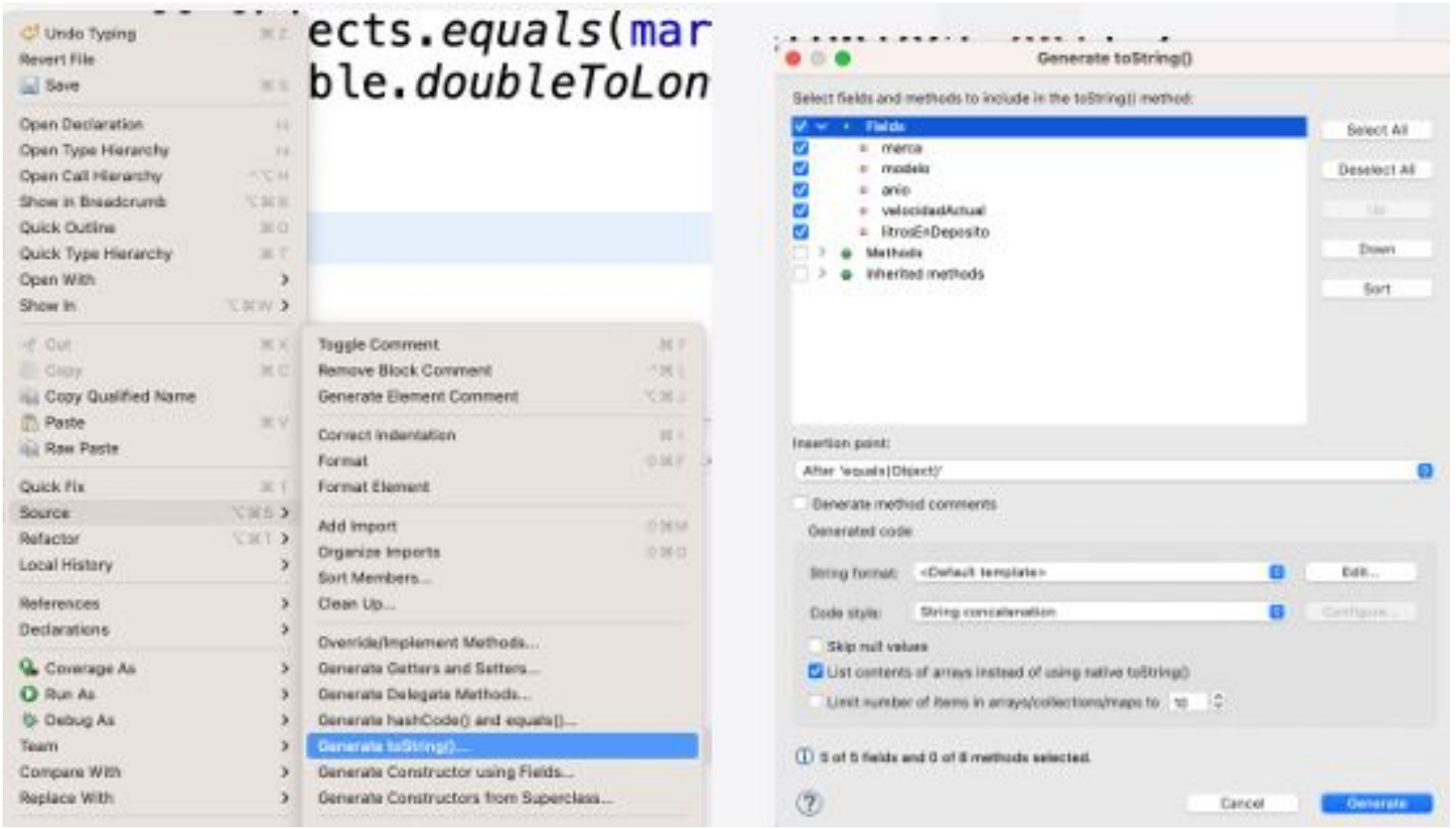
Método para comparar dos objetos.
Podemos customizar cómo se va a realizar la comparación:
Si son el mismo objeto, devuelve true.
Si el objeto con el que comparamos es null, devuelve false.
Si los objetos son de clases diferentes, devuelve false.
Si todos sus atributos son iguales devuelve true.

hashCode

Método que devuelve un valor hash (int) para el objeto.
Será útil tenerlo cuando se trabaje con colecciones p.e: HashMap.
La especificación de Java deja abierta su implementación teniendo en cuenta que en la misma ejecución de un programa, el método hashCode debe devolver el mismo valor para el mismo objeto. No es obligatorio que sea así en diferentes ejecuciones del programa.

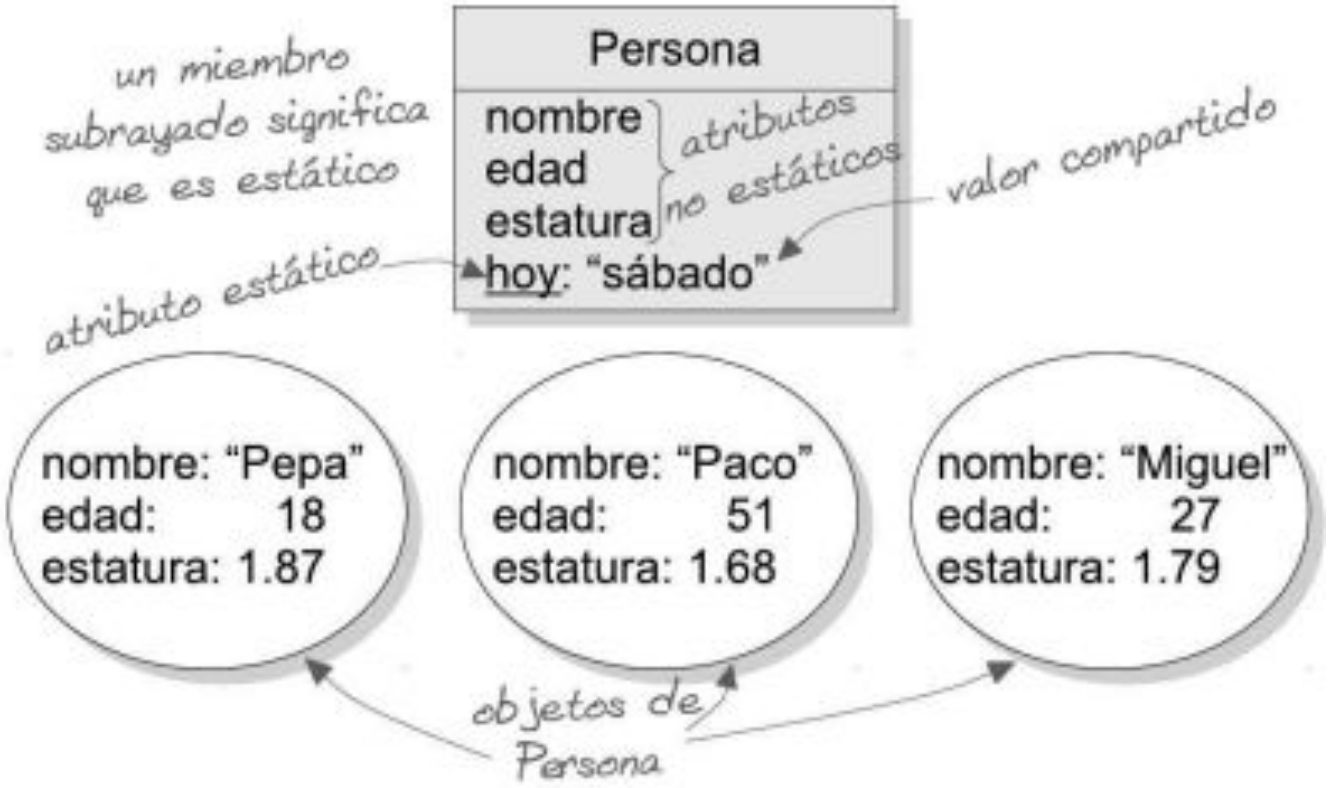
Creación automática de los Métodos especiales

Los métodos toString, equals y hashCode se pueden autogenerar desde la opción de Eclipse del menú Source.



Atributos y métodos estáticos

Un atributo estático o de clase es aquel del que no existe una copia en cada objeto. Todos los objetos de una misma clase comparten un valor.



Atributos y métodos estáticos

Un atributo estático mediante la palabra reservada static.

```
class Persona {
   ...
   static String hoy;
}
```

Para acceder a un atributo estático es posible utilizar el nombre de la clase o cualquier objeto:

```
Persona p = new Persona();

Persona.hoy = "domingo";
System.out.println(p.hoy); //mostrará "domingo"
```

Un atributo estático se inicializa una sóla vez y se hace antes de crear ningún objeto, para asignar el valor inicial:

```
class Persona {
   ...
   static String hoy = "lunes"; //valor inicial
}
```

Atributos y métodos estáticos

De igual forma se declaran métodos estáticos, que no requieren de ningún objeto para ejecutarse y no pueden usar ningún atributo que no sea estático.

Ejemplo: Método que actualice el atributo a partir de un entero

```
static void hoyEs(int dia) {
   switch (dia) {
      case 1: hoy = "lunes";
              break;
      case 2: hoy = "martes";
              break;
      . . .

      case 7: hoy = "domingo";
              break;
   }
}

Persona.hoyEs(2); //martes        p.hoyEs(2); //martes
```

Tipos de datos Enumerados

Sirven para definir grupos de constantes como los únicos valores posibles de una variables.
Son parecidos a las clases. Se definen también dentro de un paquete con botón derecho opción: New/Java Enum.
Usamos la palabra clave enum en vez de class.
Podemos definirlos como una clase o directamente dentro de una clase:

```
class Cliente {
   enum Sexo {HOMBRE, MUJER} //definición del tipo
   Sexo sexo; //declaración del atributo

   Cliente(String dni, String nombre, int edad, Sexo sexo) {
      ...
   }
}
```

Para acceder al tipo Sexo desde fuera de la clase Cliente:

```
Cliente c = new Cliente("123", "Nuria", 41, Cliente.Sexo.MUJER);
```

Tipos de datos Enumerados

Normalmente cuando tengamos que introducir por teclado un valor de tipo enumerado, escribiremos una cadena como “MUJER” y no Sexo.MUJER. Para convertir la cadena a enum usaremos el método *valueOf()*.

Como hemos definido el enum dentro de la propia clase Cliente, al hacer la llamada al constructor tendremos que hacer la conversión de String a Sexo antes de llamar al constructor de la siguiente manera:

```
Scanner sc = new Scanner(System.in);
String sexoCliente = sc.nextLine();
Cliente c = new Cliente( ..., Sexo.valueOf(sexoCliente));
```

Los tipos enumerados se pueden definir en paquetes distintos a donde se vayan a usar. En este caso, habrá que definirlos public e importarlos al igual que si fueran clases.

Ejercicio Cuentas bancarias

- Crea una clase llamada Cuenta que tendrá los siguientes atributos: titular y cantidad (puede tener decimales).
- El titular será obligatorio y la cantidad es opcional. Crea dos constructores que cumpla lo anterior.
- Crea sus métodos getters y setters.
- Tendrá los métodos especiales: toString y equals.
- Tendrá todos métodos propios:
  - ingresar(double cantidad): se ingresa una cantidad a la cuenta, si la cantidad introducida es negativa, no se hará nada.
  - retirar(double cantidad): se retira una cantidad a la cuenta, si restando la cantidad actual a la que nos pasan es negativa, la cantidad de la cuenta pasa a ser 0.

- Crea una clase llamada App desde la que hagas la lógica para probar el comportamiento completo de la clase Cuenta, mediante la creación de diferentes objetos.

Ejercicios Alumnos

Ejercicio1

- Se necesita crear una clase que representará a un Alumno de un centro educativo.
- De cada alumno se quiere conocer el nombre, apellidos, email, edad y teléfono.
- Se necesitan diferentes constructores.
  - Sin argumentos.
  - Nombre, apellidos y edad.
  - Todos los atributos.
- Se deben implementar los métodos getters/setters.
- Se deben implementar los métodos equals, hashCode y toString.
- Se debe añadir un método que nos indique si el alumno es mayor de edad (>= 18 años).
- Implementa un método main donde se creen un par de objetos de tipo Alumno y se muestren por consola, así como el resto de comportamiento de la clase.

Ejercicio2

- Utilizando la clase Alumno del ejercicio1, crea un programa en Java que tenga un array de objetos de esta clase y nos indique cuál es el alumno más joven de todos.

Ejercicio Usuario

Realiza una aplicación que cree una matriz de usuarios (5 usuarios por teclado). Implementa en la aplicación 2 métodos para carga la matriz (que llamará a grabar usuario para cada usuario) y otro para mostrar todos los usuarios insertados.
En la aplicación, para insertar la contraseña se deben de escribir dos veces y comprobar que coinciden.

La clase usuario tiene la siguiente definición.

Usuario
- email: String - password: String
+ Usuario( ) + Usuario(em:String, pass:String) + getEmail ( ):String + setEmail (em:String):void + setPassword (pass:String):void - Validar (pass:String):boolean

Se deben usar métodos para grabar/modificar usuarios, que tengan en cuenta que se valide si la contraseña tiene un tamaño mínimo.

Ejercicio Empleado

Crea una clase Empleado que tenga los siguientes atributos privados:

- Nif.
- Nombre.
- Sueldo base.
- Horas extra realizadas en el mes.
- Tipo de IRPF (%).
- Casado o no.
- Número de hijos.
- Importe de la hora extra. Este será un atributo static o atributo de clase.

Los objetos Empleado se podrán crear con un constructor por defecto o con un constructor con un solo parámetro correspondiente al DNI.

Además de los métodos getter/setter la clase Empleado tendrá estos métodos:

Método para el cálculo del complemento correspondiente a las horas extra realizadas.

Método para calcular el sueldo bruto (sueldo base + complemento por horas extras)

Método para calcular las retenciones por IRPF. El porcentaje de IRPF se aplica sobre el sueldo bruto, teniendo en cuenta que el porcentaje que hay que aplicar es el tipo menos 2 puntos si el empleado está casado y menos 1 punto adicional por cada hijo que tenga.

Método toString() para mostrar los datos de los empleados de la siguiente forma:

- *12345678A Lucas Guerrero Arjona*
- *Sueldo Base: 1150.0*
- *Horas Extras: 4*
- *tipo IRPF: 15.0*
- *Casado: S*
- *Número de Hijos: 2*

Una vez creada la clase Empleado, la utilizaremos en un programa que lea empleados y los guarde en un array estático. El número total de empleados se pide por teclado. El número máximo de empleados es de 20.

Después de leer los datos de los empleados se pedirá que se introduzca el importe correspondiente al pago por hora extra asignándoselo al atributo estático de la clase.

A continuación el programa mostrará:

- El empleado que más cobra y el que menos
- El empleado que cobra más por horas extra y el que menos.
- Todos los empleados ordenados por salario de menor a mayor.

Clase Empleado

- nif: String
- nombre: String
- sueldoBase: double
- horasExtras: int
- tipoIRPF: double
- casado: char (S ó N)
- numeroHijos: int
- static pagoPorHoraExtra: double

+ Empleado()
+ Empleado(String nif)
+ getCasado():char
+ setCasado(char casado): void
+ getHorasExtras(): int
+ setHorasExtras(int horasExtras): void
+ getNif(): String
+ setNif(String nif): void
+ getNombre(): String
+ setNombre(String nombre): void
+ getNumeroHijos(): int
+ setNumeroHijos(int numeroHijos): void
+ getSueldoBase(): double
+ setSueldoBase(double sueldoBase): void
+ getTipoIRPF(): double
+ setTipoIRPF(double tipoIRPF): void
+ static getPagoPorHoraExtra(): double
+ static setPagoPorHoraExtra(double pagoPorHoraExtra): void
+ calcularImporteHorasExtras(): double
+ calcularSueldoBruto(): double
+ calcularRetencionIrpf(): double
+ calcularSueldo(): double
+ toString(): String

Asociaciones entre objetos

Los objetos pueden asociarse de diferentes formas:

- Uso
- Agregación
- Composición
- Herencia

Cada tipo de asociación determina algunas características, como el nivel de interacción, el ámbito de los objetos, …

(Vemos las tres primeras, la herencia la estudiaremos en detalle en la siguiente unidad)

Asociación de uso

También conocida como asociación simple.

Sucede cuando un objeto utiliza otro objeto como parte de su funcionamiento.

Habitualmente vendrá determinada porque en una clase A tengamos al menos un método que reciba como argumentos una instancia de otra clase B.

Es una relación de utilización más que una relación estructural. B no forma parte de la estructura de A, pero es necesaria para su funcionamiento.

Agregación y composición

Son relaciones estructurales. Se da cuando una clase A tiene un atributo cuyo tipo de dato es otra clase B.

Son muy parecidas, se diferencian en la intensidad de la asociación. A veces se conoce a la agregación como composición débil, y a la composición como composición fuerte. Sin embargo, muchos textos no las diferencian.

AGREGACIÓN
Una clase tiene en su estructura objetos de otro tipo de clase.

Los objetos del otro tipo de clase pueden existir de forma independiente.

Agregación y composición

Por ejemplo: CestaCompra y Producto.

- La cesta de la compra puede tener una colección de productos.
- Los productos pueden existir sin estar incluídos en una cesta de la compra.

COMPOSICIÓN

Una versión más restrictiva que la agregación.

Representan una relación de propiedad entre objetos.
Si A está compuesto de B, las instancias de B no pueden existir si no están asociadas a una instancia de A.

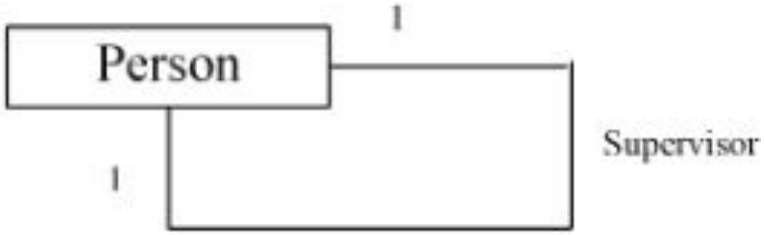
Por ejemplo: una Casa está compuesta de Habitaciones.

Agregación entre la misma clase

La agregación puede existir entre objetos de la misma clase.

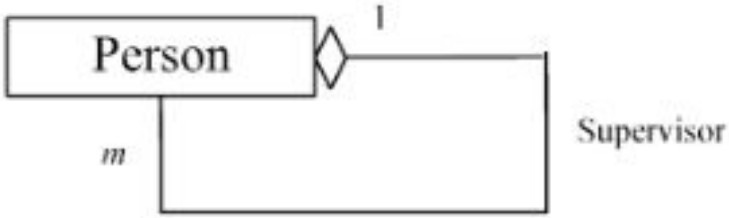
Por ejemplo, una persona puede tener un supervisor

```
public class Person {
  // El tipo de los dato es la propia clase
  private Person supervisor;
  ...
}
```



Por ejemplo, una persona puede tener varios supervisores

```
public class Person {
   ...
   private Person[] supervisors;
}
```



Tipos de Clases existentes

- Clases para el tratamiento de fechas:
  - LocalDate
  - DateTimeFormatter
- Clase Randon
- Clases contenedoras numéricas:
  - Integer
  - Double
- Clases de estructuras dinámicas:
  - Pila
  - Cola

Ejemplo clase fecha propia

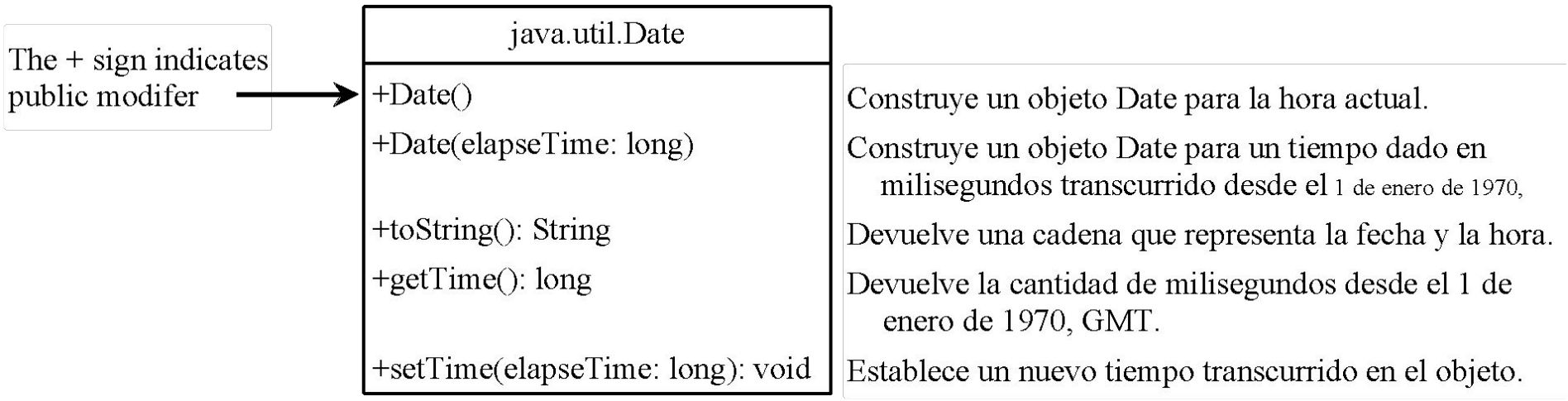
```
public class Student {
  private int id;
  private BirthDate birthDate;

  public Student(int ssn,
      int year, int month, int day) {
    id = ssn;
    birthDate = new BirthDate(year, month, day);
  }

  public int getId() {
    return id;
  }

  public BirthDate getBirthDate() {
    return birthDate;
  }
}
```

```
public class BirthDate {
  private int year;
  private int month;
  private int day;

  public BirthDate(int newYear,
      int newMonth, int newDay) {
      year = newYear;
      month = newMonth;
      day = newDay;
  }

  public void setYear(int newYear) {
    year = newYear;
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    Student student = new Student(111223333, 1970, 5, 3);
    BirthDate date = student.getBirthDate();
    date.setYear(2010); // Now the student birth year is changed!
  }
}
```

Clase Date

Java proporciona una encapsulación de fecha y hora independiente del sistema en la clase java.util.Date.
Podemos usar la clase Date para crear una instancia para la fecha y hora actual y usar su método toString para devolver la fecha y la hora como una cadena.



Por ejemplo, el siguiente código

```
java.util.Date date = new java.util.Date();
System.out.println(date.toString());
```

Clases LocalDate, LocalTime y LocalDateTime

Desde Java8 se proporcionan paquetes con clases para facilitar más el trabajo con las fechas y horas. Documentación Clases Fechas

El paquete **java.time** contiene las clases relativas al tiempo (LocalDate, LocalTime y LocalDateTime) que nos permiten trabajar con fechas y horas.

Ejemplo uso de LocalDate y LocalDateTime

```
import java.time.*; //Contiene LocalDate y LocalDateTime
public class Fechas {
    public static void main(String[] args) {
        LocalDate hoy = LocalDate.now();
        System.out.println("Hoy es: " + hoy);
        LocalDateTime ahora = LocalDateTime.now();
        System.out.println("Ahora es: " + ahora);
    }
}
```

Clase DateTimeFormatter

El paquete **java.time.format** que contiene la clase DateTimeFormatter, que permite formatear las fechas y horas.

DateTimeFormatter f = DateTimeFormatter	.ISO_LOCAL_DATE; .ISO_LOCAL_TIME; .ISO_LOCAL_DATE_TIME;	
	.ofLocalizedDate .ofLocalizedTime .ofLocalizedDateTime	(FormatStyle.SHORT); (FormatStyle.MEDIUM);
	.ofPattern("dd MM yyyy");	

Ejemplo uso de LocalDateTime y DateTimeFormatter

```
import java.time.*; // Este paquete contiene LocalDateTime entre otras
import java.time.format.*; // Este paquete contiene DateTimeFormatter
public class Fechas {
    public static void main(String[] args) {
        LocalDateTime fecha = LocalDateTime.now();
        DateTimeFormatter isoFecha = LateTimeFormatter.ISO_LOCAL_DATE;
        System.out.println(fecha.format(isoFecha));
    }
}
```

Repaso de la Clase Random

Hasta ahora hemos usado Math.random() para obtener un valor doble aleatorio entre 0.0 y 1.0 (excluyendo 1.0). Se proporciona un generador de números aleatorios más útil en la clase java. Sin embargo la clase java.util.Random proporciona un generador de números aleatorios más útil.

java.util.Random	
+Random()	Construye un objeto aleatorio con la hora actual.
+Random(seed: long)	Construye un objeto aleatorio con un valor especificado.
+nextInt(): int	Devuelve un valor int aleatorio.
+nextInt(n: int): int	Devuelve un valor int aleatorio entre 0 y n (exclusivo).
+nextLong(): long	Devuelve un valor largo aleatorio.
+nextDouble(): double	Devuelve un valor doble aleatorio entre 0.0 y 1.0 (exclusivo).
+nextFloat(): float	Devuelve un valor flotante aleatorio entre 0.0F y 1.0F (exclusivo).
+nextBoolean(): boolean	Devuelve un valor booleano aleatorio.

Ejemplo de la Clase Random

Si dos objetos Random tienen la misma semilla, generarán secuencias idénticas de números. Por ejemplo, el siguiente código crea dos objetos aleatorios con la misma semilla 3.

```
Random random1 = new Random(3);
System.out.print("From random1: ");
for (int i = 0; i < 10; i++)
  System.out.print(random1.nextInt(1000) + " ");
Random random2 = new Random(3);
System.out.print("\nFrom random2: ");
for (int i = 0; i < 10; i++)
  System.out.print(random2.nextInt(1000) + " ");
```

From random1: 734 660 210 581 128 202 549 564 459 961
From random2: 734 660 210 581 128 202 549 564 459 961

Las clases contenedoras

Debido al rendimiento, los valores de tipo de datos primitivos no son objetos en Java. La sobrecarga de objetos de procesamiento, afectaría negativamente al rendimiento de Java si los valores de tipo de datos primitivos se tratarán como objetos. Sin embargo, muchos métodos Java requieren el uso de objetos como argumentos.

Java ofrece una forma de incorporar, o envolver, un tipo de datos primitivo en un objeto (por ejemplo, envolviendo int en la clase Integer, envolviendo doble en la clase Double y envolviendo char en la clase Character).

Al usar una clase contenedora, puede procesar valores de tipo de datos primitivos como objetos. Java proporciona las clases contenedoras Boolean, Character, Double, Float, Byte, Short, Integer y Long en el paquete **java.lang** para tipos de datos primitivos. La clase Boolean ajusta un valor booleano verdadero o falso.

Las Clases `Integer` y `Double`

java.lang.Integer
-value: int +MAX_VALUE: int +MIN_VALUE: int
+Integer(value: int) +Integer(s: String) +byteValue(): byte +shortValue(): short +intValue(): int +longVlaue(): long +floatValue(): float +doubleValue():double +compareTo(o: Integer): int +toString(): String +valueOf(s: String): Integer +valueOf(s: String, radix: int): Integer +parseInt(s: String): int +parseInt(s: String, radix: int): int

java.lang.Double
-value: double +MAX_VALUE: double +MIN_VALUE: double
+Double(value: double) +Double(s: String) +byteValue(): byte +shortValue(): short +intValue(): int +longVlaue(): long +floatValue(): float +doubleValue():double +compareTo(o: Double): int +toString(): String +valueOf(s: String): Double +valueOf(s: String, radix: int): Double +parseDouble(s: String): double +parseDouble(s: String, radix: int): double

Las clases contenedoras Integer y Double

Esta sección usa Integer y Double como ejemplos para presentar las clases contenedoras numéricas.

- Boolean
- Character
- Short
- Byte
- Integer
- Long
- Float
- Double

NOTE: Las instancias de todas las clases contenedoras son inmutables, es decir, sus valores internos no se pueden cambiar una vez que se crean los objetos.

Las clases contenedoras no tienen constructores non-arg. Se puede construir un objeto contenedor desde un valor de tipo de datos primitivo o desde una cadena que representa el valor numérico.
Los constructores para Integer y Double son :

```
public Integer(int value)
public Integer(String s)
public Double(double value)
public Double(String s)
```

Las clases contenedoras Integer y Double

Cada clase contenedora numérica tiene las constantes MAX_VALUE y MIN_VALUE.

MAX_VALUE representa el valor máximo del tipo de datos primitivo correspondiente.

Para Byte, Short, Integer, and Long, MIN_VALUE representa el mínimo valor de cada tipo primitivo.

Para Float and Double, MIN_VALUE representa el mínimo valor positivo float y double.

Cada clase contenedora numérica implementa los métodos abstractos doubleValue, floatValue, intValue, longValue y shortValue, que se definen en la clase Number. Estos métodos "convierten" los objetos en valores de tipo primitivo.

Ejemplo: La Clase Pila

Una pila es una estructura dinámica de datos donde los elementos se insertan (se apilan) y se retiran (se desapilan) siguiendo la norma de que el último que se apila será el primero en desapilarse.

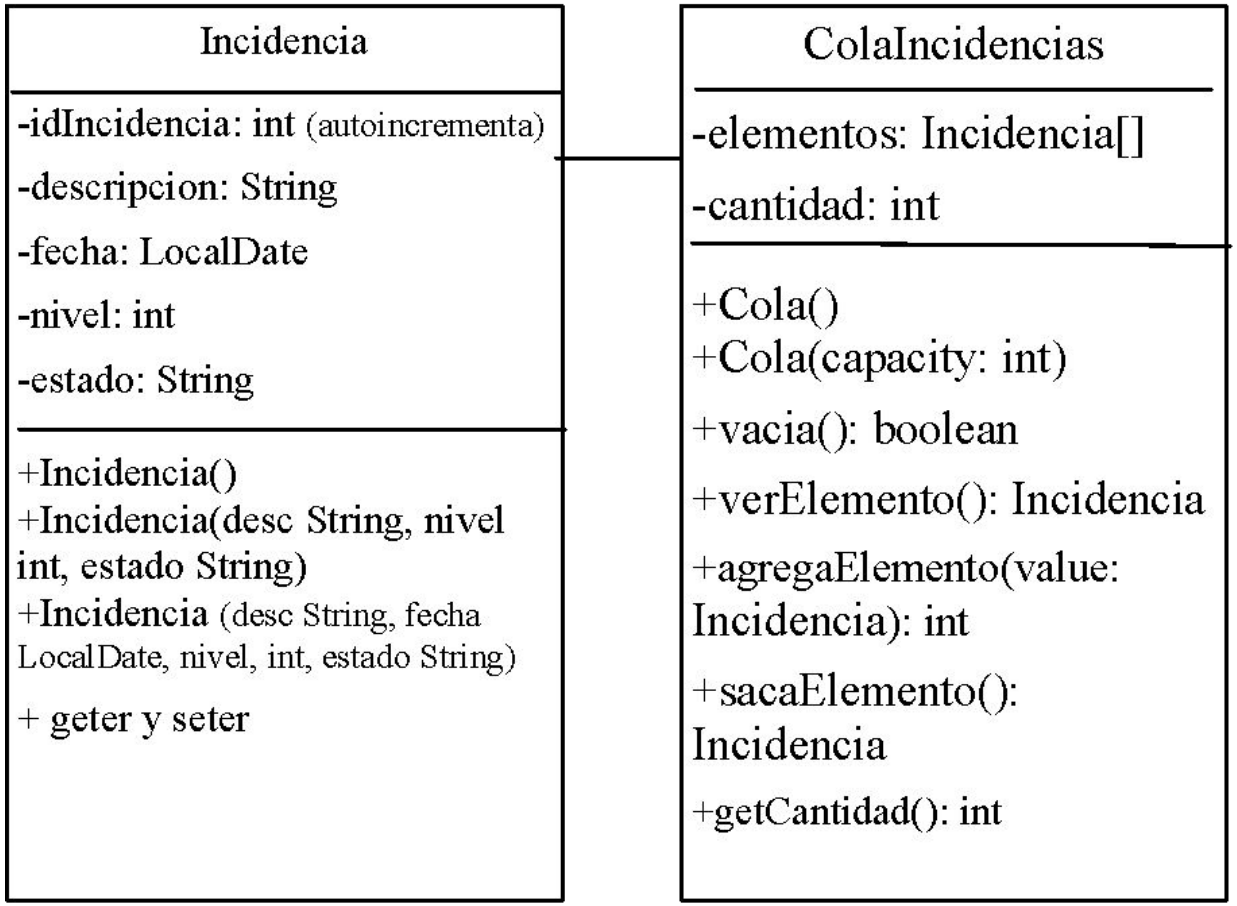
Pila	
-elementos: int[]	Una matriz para almacenar enteros en la pila..
-cantidad: int	El número de enteros en la pila..
+Pila()	Construye una pila vacía con una capacidad predeterminada de 16.
+Pila(capacidad: int)	Construye una pila vacía con una capacidad especificada.
+vacia(): boolean	Devuelve true si la pila está vacía.
+verElemento(): int	Devuelve el entero en la parte superior de la pila sin eliminarlo de la pila
+apilaElemento(value: int): void	Almacena un entero en la parte superior de la pila.
+retiraElemento(): int	Elimina el entero en la parte superior de la pila y lo devuelve.
+getCantidad(): int	Devuelve la cantidad de elementos en la pila.

Ejemplo: La Clase Cola

Una cola es una estructura dinámica de datos donde los elementos se insertan (se encolan) y se retiran (se desencolan) siguiendo la norma de que el primero que se añada será el primero en salir.

Cola	
-elementos: int[]	Una matriz para almacenar enteros en la cola..
-cantidad: int	El número de enteros en la cola..
+Cola()	Construye una cola vacía con una capacidad predeterminada de 16.
+Cola(capacity: int)	Construye una cola vacía con una capacidad especificada.
+vacia(): boolean	Devuelve true si la cola está vacía.
+verElemento(): int	Devuelve el primer entero de la cola sin eliminarlo
+agregaElemento(value: int): int	Almacena un entero en la parte de atrás de la cola.
+sacaElemento(): int	Elimina el entero en la parte delantera de la cola y lo devuelve .
+getCantidad(): int	Devuelve la cantidad de elementos en la cola.

Ejercicio aplicación de cola: Gestor de incidencias



Crear un programa de gestión de incidencias de una empresa. Las incidencias se van resolviendo por orden de llegada.

Menú incidencias:

1.-Añadir incidencia

2.-Ver próxima incidencia

3.-Eliminar incidencia.

4.-Número de incidencias

5.-Salir