

UT 3: Utilización de objetos.

1. Introducción	3
2. Objetos	3
3. Clases	4
4. Declaración de miembros de una clase	5
4.1. Atributos	5
4.2. Métodos.....	5
4.2.1. Ámbito de una función.....	6
4.2.2. Paso de parámetros	6
4.3. Constructores.....	7
4.3.1. Referencias.....	7
4.4. Sobrecarga de métodos	8
4.5. Miembros estáticos o de clase	8
4.6. Paquetes	9
5. Métodos especiales.....	9
5.1. set/get	9
5.2. toString	9
5.3. Equals.....	10
5.4. hashCode	12
6. Composición/Agregación	13
7. Herencia y polimorfismo.....	15
7.1. Compatibilidad de tipos.....	16
8. Clase abstracta	17
9. Interfaces.....	18
10. Modificadores y características	18
10.1. Clase	18

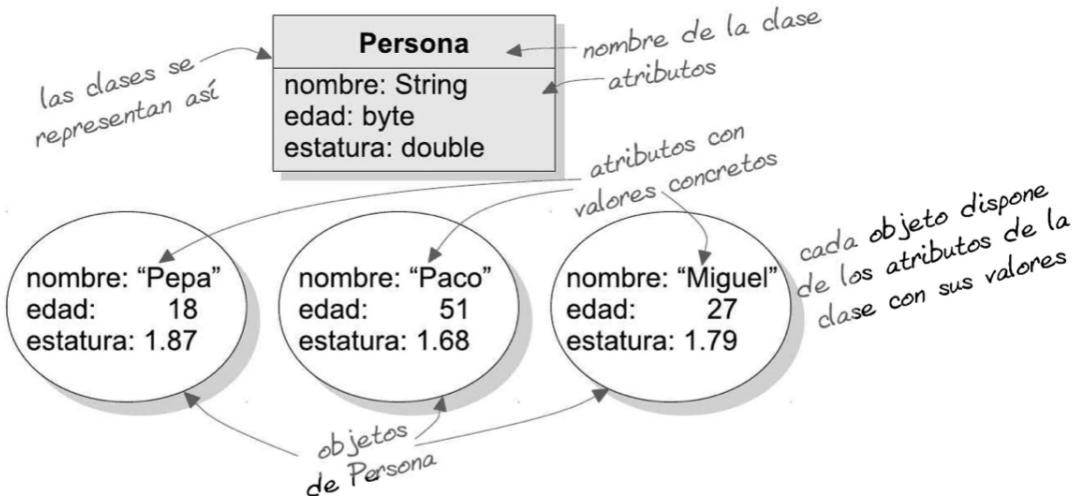
10.1.1.	Combinaciones	19
10.2.	Interfaz	19
10.2.1.	Características de una Interfaz.....	19
10.3.	Métodos y atributos	19
10.4.	Constructores	20
11.	Clase Object.....	20
12.	Clases wrapper	21
12.1.	Trabajando con String.....	22
13.	Garbage collector	23

1. Introducción

Un programa orientado a objetos es, esencialmente, una colección de objetos que se crean, interaccionan entre sí y dejan de existir cuando ya no son útiles durante la ejecución de un programa.

Cualquier programa escrito en Java está compuesto por una serie de clases, que en realidad son los “moldes” o “plantillas” que permiten crear objetos.

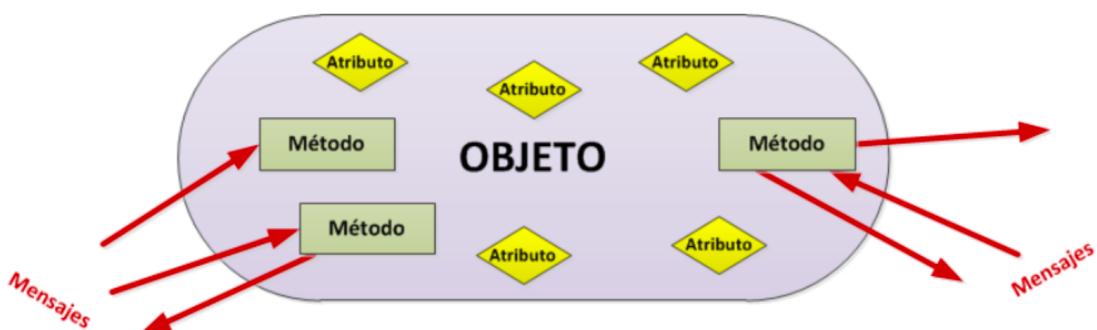
En realidad un objeto solo existe en tiempo de ejecución y una clase es algo estático, que existe en tiempo de codificación.



Para poder crear un objeto, hay que instanciar una clase.

2. Objetos

Un objeto siempre dispone de una serie de atributos, expresados de forma similar a las variables, y un comportamiento, que se definen como métodos.



Al conjunto de métodos y atributos se denomina “miembros” del objeto.

3. Clases

La sintaxis básica de una clase, en la mayoría de lenguajes de programación, tiene la siguiente estructura:

```
class Nombre {  
    <atributos>  
    <métodos>  
}
```

Por ejemplo:

```
public class Cliente {  
  
    //Atributos  
    private String dni;  
    private String nombre;  
    private long telefono;  
  
    //métodos  
    > public Cliente(String dni, String nombre, long telefono) {  
        this.dni = dni;  
        this.nombre = nombre;  
        this.telefono = telefono;  
    }  
  
>     public String getDni() {  
        return dni;  
    }  
  
>     public void setDni(String dni) {  
        this.dni = dni;  
    }  
  
>     public String getNombre() {  
        return nombre;  
    }  
  
>     public void setNombre(String nombre) {  
        this.nombre = nombre;  
    }  
  
>     public long getTelefono() {  
        return telefono;  
    }  
  
>     public void setTelefono(long telefono) {  
        this.telefono = telefono;  
    }  
}
```

Para poder utilizar la clase, se debe instanciar:

```
public class PruebaPersona {  
    public static void main(String[] args) {  
  
        Persona per = new Persona("70707070M", "Fran", 666888777);  
        System.out.println("El nombre de la persona es: " + per.getNombre());  
    }  
}
```

4. Declaración de miembros de una clase

4.1. Atributos

Los objetos almacenan información en sus atributos, es decir, los atributos almacenan “sus características”. La definición de un atributo es similar a una variable, aunque puede llevar información adicional (modificadores).

modificadorVisibilidad tipo nombreVariable;

Ejemplo:

```
public class Persona {  
    private String name;  
    private int edad;  
    private String dni;  
    private String correo;  
}
```

4.2. Métodos

Cualquier objeto tiene un comportamiento que viene determinado por sus métodos, o dicho de otro modo, cualquier acción que realiza un objeto se produce por la ejecución de uno de sus métodos.

También se denominan métodos de instancia puesto que para poder usarlos, hay que instanciar la clase previamente. Tienen la siguiente sintaxis:

**modificadorVisibilidad tipoValorDevuelto nombreMetodo (parámetros) {
 cuerpo método
}**

Al conjunto formado por el modificado, el tipo devuelto, nombre y los parámetros se denomina **prototipo o signatura** del método.

Un ejemplo concreto sería:

```

public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}

```

Los parámetros en realidad son las variables por las que los métodos reciben valores.

Para invocar a un método se debe indicar el nombre del mismo seguido de los parámetros que dispone, en el orden indicado en su definición.

4.2.1. Ámbito de una función

El cualquier programa que haga uso de métodos y funciones puede distinguir entre dos tipos de variables:

- **Variable local:** Es aquella definida dentro de un método o parte del mismo y su uso se limita a ese método o zona, de forma que ese mismo nombre se puede reutilizar en otro lugar del código.
- **Variable global:** Es aquella que se declara para que pueda ser utilizada por más partes que un solo método.

```

void func2() {
    double a;
    ...
    if(...) {
        int x;
        ...
    } //del if
} //de func2

```

4.2.2. Paso de parámetros

A la acción de pasar valores a un método se denomina paso de parámetros. Existen dos tipos de pasos de parámetros:

- **Paso por valor:** Los valores pasados solo sirven de entrada. El método recibe una copia del valor de la variable.
- **Paso por referencia:** Los valores pasados sirven de entrada, se modifican con la ejecución de todas las sentencias de la función/procedimiento, y por último, sirven de salida.

Para poder pasar un parámetro de referencia se debería disponer de un “puntero” a memoria y en el caso de Java ese concepto no existe. Por tanto, las variables pasadas por un programa en Java se pasan siempre por valor.

En caso de que la variable haga referencia a un objeto, este sí se podría modificar dentro del método, es decir, es una especie de paso por referencia.

Algunas clases como la clase *String* en Java son inmutables, esto significa que al manipular el objeto se devuelve una nueva instancia de la clase en vez de modificar la original.

4.3. Constructores

Los constructores son métodos especiales que permite instanciar una clase. Cuando se crea un objeto, automáticamente se invoca al constructor. En caso de no haber implementado un constructor concreto, muchos lenguajes, como por ejemplo Java, utilizan un constructor por defecto (no hay que escribirlo).

La función del constructor es inicializar el objeto.

```
public class Coche {  
    private String matricula;  
    private String color;  
  
    public Coche(String matricula, String color) {  
        this.matricula = matricula;  
        this.color = color;  
    }  
}
```

Como se puede observar, la clase *Coche* dispone de atributos con el mismo nombre que los parámetros del constructor. Para poder distinguir un atributo de un parámetro se utiliza la palabra reservada “*this*”, que hace referencia a los miembros de la propia clase. *This* permite acceder tanto a atributos como métodos dentro de la propia clase, de forma que facilita su legibilidad.

4.3.1. Referencias

Cuando se instancia un objeto, la variable contiene la referencia a dicho objeto. Es posible encontrar otros tipos de referencias.

- **null**: es posible declarar una variable de un objeto, antes de la instancia. En el momento de la declaración, la variable contendrá valor nulo (**null**).

```
Persona persona; //persona contiene valor null  
persona = new Persona("70707070M", "Fran", 666888777);
```

- **Referencias compartidas:** se podría dar el caso de que un objeto esté referenciado con varias variables. Estas variables “sobrantes” son alias, es decir, contienen la misma dirección de memoria y por tanto, podrían modificar el mismo objeto.

```

16     Persona persona; //persona contiene valor null
17     persona = new Persona("70707070M", "Fran", 666888777);
18     Persona persona2 = persona;
19
20     System.out.println(persona);
21     System.out.println(persona2);
22

```

The screenshot shows an IDE interface with several tabs at the top: Problems, Javadoc, Declaration, Search, Console, and Servers. The Console tab is active, displaying the output of a Java application named 'PruebaPersona'. The application prints two lines of text: 'Persona: dni=70707070M, nombre=Fran, telefono=666888777' and 'Persona: dni=70707070M, nombre=Fran, telefono=666888777'. This demonstrates that both 'persona' and 'persona2' reference the same object in memory.

4.4. Sobrecarga de métodos

La sobrecarga es un mecanismo por el cual se pueden crear métodos o constructores con el mismo nombre pero diferentes parámetros. Es especialmente útil con los constructores, ya que permiten inicializar más o menos atributos según se necesiten.

```

public class Coche {
    private String matricula;
    private String tipoRueda;

    >   public Coche(String matricula) {
        this.matricula = matricula;
    }
    >   public Coche(String matricula, String tipoRueda) {
        this.matricula = matricula;
        this.tipoRueda = tipoRueda;
    }
}

```

4.5. Miembros estáticos o clase

Son miembros especiales que pertenecen a la clase y no a un objeto particular. Cuando se define un miembro estático quiere decir que solo se va a crear uno, independientemente de si la clase se instancia 0 o N veces. Para definir un miembro estático se utiliza la cláusula “static”:

```

public class Coche {
    public static String matricula;
    private static String color;

    >   public static void mostrarColor() {
        System.out.println("El color es: " + color);
    }
}

```

Para hacer uso de los miembros estáticos no hay que instanciar la clase que los incluye, simplemente se indica el nombre de la clase, el operador punto y se sigue del miembro estático:

```
Coche.matricula = "1234-AAA";
Coche.mostrarColor();
```

Aunque existen ciertas restricciones en el uso de métodos *static*:

- Los métodos *static* no pueden usar la referencia “this”.
- Los métodos *static* no pueden acceder a miembros que no sean *static*.

4.6. Paquetes

Un paquete es un contenedor que agrupa un conjunto de clases relacionadas entre sí. La agrupación no se realiza siguiendo criterios de herencia, simplemente se organizan bajo un criterio que definido por el programador.

El objetivo del uso de paquetes es clarificar la estructura estática del programa y es fundamental con el uso de patrones de diseño.

5. Métodos especiales

5.1. set/get

Permiten acceder a los atributos privados de la clase.

```
public class Persona {
    private String dni;
    private String nombre;
    private long telefono;
}

    public String getDni() {
        return dni;
    }

    public void setDni(String dni) {
        this.dni = dni;
    }
```

5.2. toString

Muestra información sobre el contenido de los atributos de un objeto. Si no se implementa este método, nos mostrará el paquete y el id del objeto en la ejecución:



En caso de implementarlo, el resultado dependerá del formato que se desee, pero sería algo así:

```
<terminated> PruebaPersona [Java Application] /Library/Java/JavaVirtualMachine/...
Persona: dni=70707070M, nombre=Fran, telefono=666888777
```

La implementación puede ser similar a esta:

```
@Override
public String toString() {
    return "Persona: dni=" + dni + ", nombre=" + nombre + ", telefono=" + telefono;
}
```

Nota: Si no se pone el @Override, funcionaría igual, pero de esta forma te evitar problemas de sintaxis, puesto que quiere decir que heredas ese método y le estás sobreescribiendo. Si por ejemplo, pusieramos toStri(), al disponer de la cláusula @Override, nos avisaría que no es un método heredado.

```
40
41 @Override
42 public String toStri() {
43     return "Persona: dni=" + dni + ", nombre=" + nombre + ", telefono=" + telefono;
44 }
45
```

Problems X @ Javadoc Declaration Search Console Servers

1 error, 0 warnings, 0 others

Description	Resource
Errors (1 item)	Persona.java
The method toStri() of type Persona must override or implement a supertype method	

5.3. Equals

Compara dos objetos y decide si son iguales, devolviendo true en caso afirmativo y false en caso contrario. Es importante destacar que el operador “==” es válido para comparar tipos primitivos, pero no sirve para comparar objetos, ya que solo examina sus referencias, sin fijarse en su contenido.

```

3  public class PruebaPersona {
4
5  public static void main(String[] args) {
6
7
8      Persona per = new Persona("70707070M", "Fran", 666888777);
9
10     Persona per2 = new Persona("70707070M", "Fran", 666888777);
11
12     System.out.println("Comparación con == -> " + (per==per2));
13     System.out.println("Comparación con equals -> " + (per.equals(per2)));
14

```

Problems @ Javadoc Declaration Search Console Servers
<terminated> PruebaPersona [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/bin/java (11)
Comparación con == -> false
Comparación con equals -> true

Ejercicio: Intenta replicar este ejemplo.

Para que este ejercicio funcione se debe implementar el método *equals* y el método *hashCode* del propio objeto.

La mejor implementación de *equals* es la que se presenta a continuación:

```

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    VehiculoAlquilado other = (VehiculoAlquilado) obj;
    if (anioAlquiler != other.anioAlquiler)
        return false;
    if (cliente == null) {
        if (other.cliente != null)
            return false;
    } else if (!cliente.equals(other.cliente))
        return false;
    if (diaAlquiler != other.diaAlquiler)
        return false;
    if (mesAlquiler != other.mesAlquiler)
        return false;
    if (totalDiasAlquiler != other.totalDiasAlquiler)
        return false;
    if (vehiculo == null) {
        if (other.vehiculo != null)
            return false;
    } else if (!vehiculo.equals(other.vehiculo))
        return false;
    return true;
}

```

Además de comprobar que el objeto sea igual a él mismo, que no sea nulo y que la clase sea igual, compara uno a uno los atributos. Ojo, para comparar atributos que sean objetos, primero se debe comprobar que no sean nulos, para evitar una excepción.

Esta implementación es algo compleja y por eso, la clase *EqualsBuilder* de la librería *commons-lang* nos lo pone fácil:

```

@Override
public boolean equals(Object obj) {
    return EqualsBuilder.reflectionEquals(this, obj);
}

```

Para poder usar esa clase deberíamos incluir la librería, por ejemplo esta referencia:

```
<!-- https://mvnrepository.com/artifact/org.apache.commons/commons-lang3 -->
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.12.0</version>
</dependency>
```

Lo más fácil es buscar *commons-lang* en el repositorio de Maven.

Aunque es una solución cómoda, no funciona correctamente en caso de utilizar *ORM (Object-Relational Mapping)* como *Hibernate*, *JPA*, *JDO*, etc. debido a la “carga perezosa” o *lazy-load*, que carga los datos a destiempo, en ciertas ocasiones.

Tenemos la opción de una solución intermedia, que hace uso de la clase *Objects* (no confundir con *Object*):

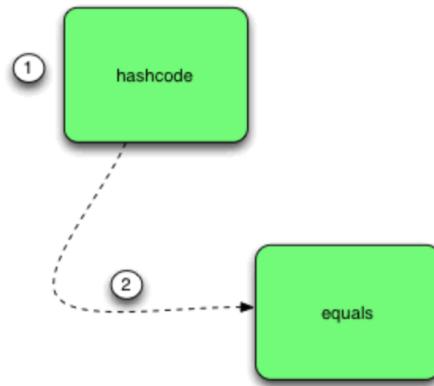
```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    VehiculoAlquilado other = (VehiculoAlquilado) obj;
    return super.equals(other) &&
        Objects.equals(this.anioAlquiler, other.anioAlquiler) &&
        Objects.equals(this.cliente, other.cliente) &&
        Objects.equals(this.diaAlquiler, other.diaAlquiler) &&
        Objects.equals(this.mesAlquiler, other.mesAlquiler) &&
        Objects.equals(this.totalDiasAlquiler, other.totalDiasAlquiler) &&
        Objects.equals(this.vehiculo, other.vehiculo);
}
```

5.4. hashCode

Este método viene a complementar al método *equals* y sirve para comparar objetos de una forma más rápida en estructuras Hash ya que únicamente nos devuelve un número entero.

Cuando Java compara dos objetos en estructuras de tipo hash (*HashMap*, *HashSet*, etc.) primero invoca al método *hashcode* y luego el *equals*.

- Si los métodos *hashcode* de cada objeto devuelven diferente hash no seguirá comparando y considerará a los objetos distintos.
- En el caso en el que ambos objetos compartan el mismo *hashcode* Java invocará al método *equals()* y revisará a detalle si se cumple la igualdad. De esta forma las búsquedas quedan simplificadas en estructuras hash.



Hay varias formas de implementar el método *hashCode*:

```

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((dni == null) ? 0 : dni.hashCode());
    result = prime * result + ((nombre == null) ? 0 : nombre.hashCode());
    // >>> es un operador de desplazamiento de bits en binario
    result = prime * result + (int) (telefono ^ (telefono >>> 32));
    return result;
}
  
```

Esta versión permite calcular el hash de cada campo, en función de su tipo:

Por cada campo usado en el método *equals* se debe obtener un *hash code* (int) realizando:

- Si el campo es un *boolean* se debe calcular (*f* ? 1 : 0).
- Si el campo es un *byte*, *char*, *short* o *int* se debe calcular (*int*) *f*.
- Si el campo es un *long* se debe calcular (*int*) (*f* ^ (*f* >>> 32)).
- Si el campo es un *float* se debe calcular *Float.floatToIntBits(f)*.
- Si el campo es un *double* se debe calcular *Double.doubleToLongBits(f)* y calcular el *hash* del *long* obtenido en el paso para los tipos *long*.
- Si el campo es una referencia a un objeto y el método *equals* de esta clase compara recursivamente invocando el método *equals* del campo, invocar su método *hashCode*. si el valor de campo es nulo se debe retornar una constante que tradicionalmente es 0.
- Si el campo es un *array* se debe tratar individualmente cada elemento aplicando estas reglas a cada elemento. Si cada elemento del array es significativo se puede usar *Arrays.hashCode*.
- Combinar los *hash code* obtenidos de la siguiente forma, *result* = 31 * *result* + *c*.

Esta forma es bastante tediosa y se pueden producir errores. Para evitarlo, a partir de Java 7 se utiliza la clase *Objects* (no confundir con *Object*):

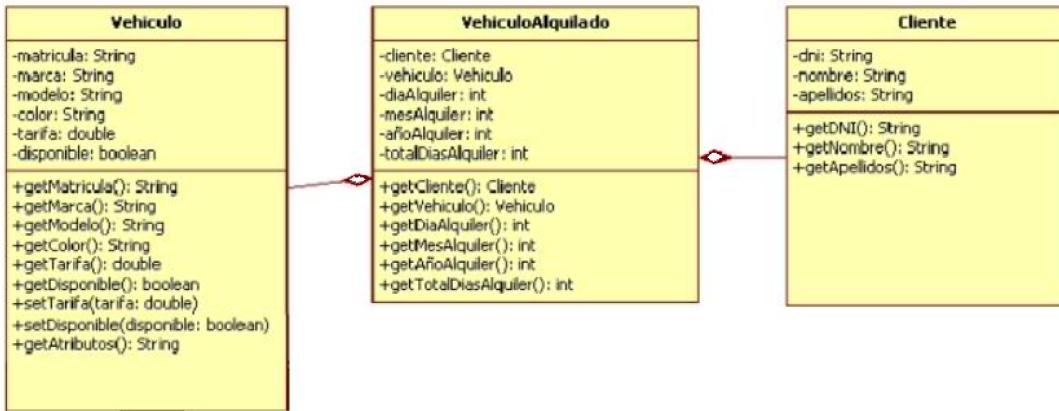
```

@Override
public int hashCode() {
    return Objects.hash(dni, nombre, telefono);
}
  
```

6. Composición/Agregación

Este mecanismo permite crear clases que contendrán objetos ya definidos. El caso más simple de composición es aquel en el que se definen atributos de tipo *String*.

Vamos a ver un ejemplo de agregación:



```

public class Vehiculo {
    private String matricula;
    private String marca;
    private String modelo;
    private String color;
    private double tarifa;
    private boolean disponible;

    public Vehiculo(String matricula, String marca, String modelo, String color, double tarifa, boolean disponible) {
        this.matricula = matricula;
        this.marca = marca;
        this.modelo = modelo;
        this.color = color;
        this.tarifa = tarifa;
        this.disponible = disponible;
    }

    public String getMatricula() {
        return matricula;
    }
    public void setMatricula(String matricula) {
        this.matricula = matricula;
    }
    public String getMarca() {
        return marca;
    }
    public void setMarca(String marca) {
        this.marca = marca;
    }
    public String getModelo() {
        return modelo;
    }
    public void setModelo(String modelo) {
        this.modelo = modelo;
    }
    public String getColor() {
        return color;
    }
}

public class Cliente {
    private String dni;
    private String nombre;
    private String apellidos;

    public Cliente(String dni, String nombre, String apellidos) {
        this.dni = dni;
        this.nombre = nombre;
        this.apellidos = apellidos;
    }

    public String getDni() {
        return this.dni;
    }
    public String getNombre() {
        return this.nombre;
    }
    public String getApellidos () {
        return this.apellidos;
    }
}
  
```

```

public class VehiculoAlquilado {
    private Cliente cliente;
    private Vehiculo vehiculo;
    private int diaAlquiler;
    private int mesAlquiler;
    private int anioAlquiler;
    private int totalDiasAlquiler;

    public VehiculoAlquilado(Cliente cliente, Vehiculo vehiculo, int diaAlquiler, int mesAlquiler, int anioAlquiler, int totalDiasAlquiler) {
        this.cliente = cliente;
        this.vehiculo = vehiculo;
        this.diaAlquiler = diaAlquiler;
        this.mesAlquiler = mesAlquiler;
        this.anioAlquiler = anioAlquiler;
        this.totalDiasAlquiler = totalDiasAlquiler;
    }

    public Cliente getCliente() {
        return this.cliente;
    }

    public Vehiculo getVehiculo() {
        return this.vehiculo;
    }

    public int getDiaAlquiler() {
        return this.diaAlquiler;
    }

    public int getMesAlquiler() {
        return this.mesAlquiler;
    }

    public int getAnioAlquiler() {
        return this.anioAlquiler;
    }

    public int getTotalDiasAlquiler() {
        return this.totalDiasAlquiler;
    }
}

```

En caso de que fuera composición, en VehículoAlquilado se deberían crear los objetos cliente y vehículo, por ejemplo dentro del constructor.

7. Herencia y polimorfismo

La herencia entre clases es similar a la herencia entre padres e hijos de la vida real. Un hijo puede heredar ciertas características del padre, como el color de los ojos, e incluso alguno de sus gestos, por ejemplo la forma de andar. En el caso de las clases es similar, la herencia permite jerarquizar un grupo de clases de forma que se pueden aprovechar propiedades y métodos sin tenerlos que implementar de nuevo.

Hay que distinguir entre dos tipos de clases:

- **Clase base o padre o superclases:** Son las que se encuentran más arriba en la jerarquía y de las que se pueden aprovechar funcionalidad y características.
- **Clase derivada o hija:** Es la clase que hereda. Puede darse el caso de ser una clase padre y derivada a la vez.

También existen varios tipos de herencia:

- **Herencia simple:** Cada clase solo tiene una clase padre.
- **Herencia múltiple:** Una clase puede derivar de más de una clase padre. Este tipo de herencia depende del lenguaje concreto, por ejemplo C++ sí la permite pero Java no (aunque las últimas versiones la simulan con el uso de interfaces).

Para heredar de una clase, en la definición de la clase derivada se debe utilizar la cláusula *extends*. La clase derivada solo podrá acceder a métodos y atributos cuyo modificar de visibilidad sea *public* o *protected*.

```

public class Coche extends Vehiculo{
    private String matricula;

    public Coche(String matricula, String propietario) {
        super.propietario = propietario;
        this.matricula = matricula;
    }
}

```

Para acceder a los miembros de la superclase se utiliza “super”.

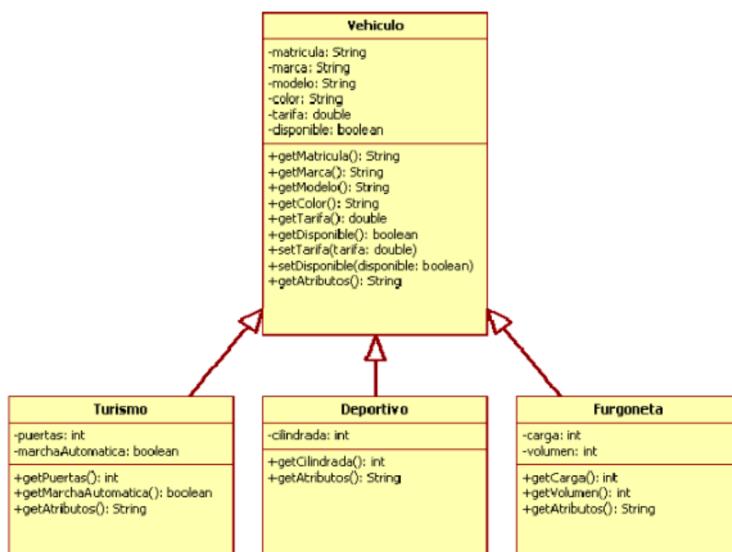
IMPORTANTE: si se accede al constructor de la superclase, como en este caso, siempre se debe indicar en la primera línea del constructor (super(...)).

Por su parte, el polimorfismo se refiere a la posibilidad de definir clases diferentes que tienen métodos o atributos denominados de forma idéntica, pero que se comportan de manera distinta. Por ejemplo, una clase Rectángulo y otra Triángulo podrán disponer de un método que sea calcularÁrea cuya implementación es distinta.

El polimorfismo es muy importante cuando se relaciona con la herencia, ya que podemos heredar un método de una clase base y redefinirlo en la clase derivada para que tenga.

7.1. Compatibilidad de tipos

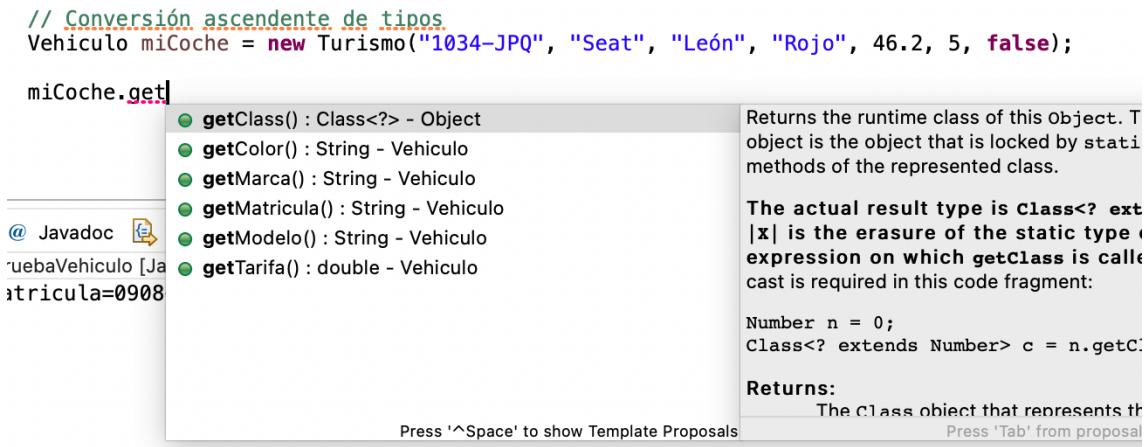
En una relación de tipo herencia, un objeto de la superclase puede almacenar un objeto de cualquiera de sus subclases. Por ejemplo, un objeto de la clase Vehiculo puede almacenar un objeto de la clase Turismo, Deportivo o Furgoneta. Dicho de otro modo, cualquier referencia de la clase Vehiculo puede contener una instancia de la clase Vehiculo o bien una instancia de las subclases Turismo, Deportivo o Furgoneta.



Si nos vamos a la instanciación:

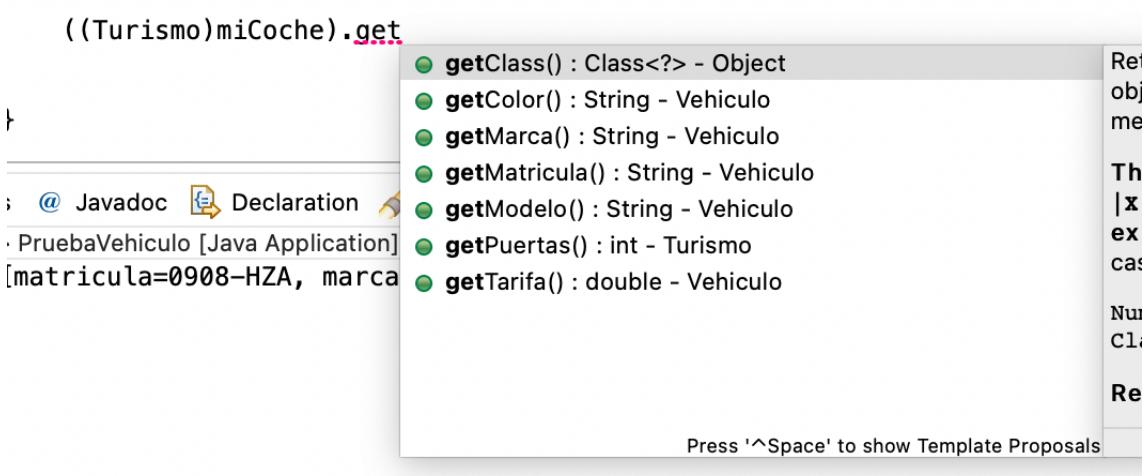
```
// Conversión ascendente de tipos
Vehiculo miCoche = new Turismo("1034-JPQ", "Seat", "León", "Rojo", 46.2, 5, false);
```

También hay que tener en cuenta que al acceder a los métodos no están disponibles los de la subclase:



En este caso faltarían los que hacen referencia al número de puertas y si es automático.

Para poder acceder a ellos, deberíamos utilizar un casting de la subclase:



8. Clase abstracta

Una clase abstracta es aquella en la que alguno de sus métodos (al menos uno), están definidos como abstractos, es decir, está marcado como *abstract* y no está implementado, solo tiene su signatura.

Para declarar una clase como abstracta simplemente se debe poner la cláusula *abstract* y debe tener al menos un método abstracto:

```
public abstract class FactoryDAO {  
    public abstract DAO getDAO();  
}
```

Una clase abstracta no se puede instanciar, por tanto, dependen de clases derivadas para que se puedan utilizar sus métodos y atributos.

Las clases abstractas se suelen utilizar para definir una estructura y evitar repetir código, ya que sus métodos y atributos corresponden con las características y comportamientos comunes de sus clases hijas.

Las clases hijas que no sean abstractas deberán implementar todos los métodos abstractos.

9. Interfaces

Una interfaz es una clase en la que ninguno de sus métodos está implementado, solo está su signatura.

Para la definición de una interfaz se utiliza la palabra reservada “*interface*”. Para implementar una interfaz se utiliza la palabra “*implements*”.

```
public class FactoryOracle implements FactoryDAO{  
    @Override  
    public DAO getDAO() {  
        // TODO Auto-generated method stub  
        return null;  
    }  
}  
public interface FactoryDAO {  
    public DAO getDAO();  
}
```

Todas las clases derivadas deben implementar todos sus métodos.

10. Modificadores y características

Una propiedad muy importante de la POO es el encapsulamiento. El encapsulamiento es un mecanismo que permite controlar el acceso a los elementos de una clase desde el exterior. Oculta la estructura interna de un objeto al exterior, es decir, se accede a los métodos desde fuera sin saber como están implementados internamente.

El control del acceso se hace en la definición de la clase y debe indicarse por cada elemento de la clase el tipo de acceso que se permite.

10.1. Clase

Las clases disponen de 2 modificadores de acceso:

- **default**: Solo tienen acceso a esa clase las clases del mismo paquete, por tanto, solo se pueden instanciar clases del mismo paquete.
- **public**: Las clases con este modificador son accesibles desde cualquier paquete de la aplicación.

Hay otros modificadores que no hacen referencia al acceso:

- **final**: Indica que la clase no puede ser extendida, es decir, no puede tener subclases.
- **abstract**: Clase que contiene uno o más métodos abstractos. Nunca puede ser instanciada y las clases que la extiendan deben implementar todos los métodos abstractos (salvo que la clase hija sea también abstracta).

10.1.1. Combinaciones

- default o public + final
- default o public + abstract
- **NUNCA**: final + abstract

10.2. Interfaz

Utiliza los modificadores de acceso:

- default
- public

Se pueden utilizar “abstract”, aunque no aporta nada.

Nunca utilizar “final”.

10.2.1. Características de una Interfaz

- Todos sus métodos son public y abstract, nada más.
- Las variables siempre son public, static y final => Constantes.
- Una interfaz puede extender de una o más interfaces.
- Una interfaz NO puede:
 - Implementar otra clase o interfaz.
 - Extender de una clase.

10.3. Métodos y atributos

Modificadores de acceso:

- **public**: Cualquier clase puede usar ese método o variable siempre y cuando haya instanciado su clase.

- **private:** Solo accesibles desde la propia clase.
- **default:** Solo pueden acceder a esos métodos o variables las clases del mismo paquete. También hay que instanciar la clase.
- **protected:** Método y variables que pueden ser accedidos desde cualquier clase del mismo paquete o desde cualquier subclase de cualquier paquete. La clase debe ser instanciada antes.

Otros modificadores:

- **final**
 - Método que no puede ser sobrescrito.
 - Variable que no se puede modificar su valor.
- **abstract:** Método que no se ha implementado. NO es posible utilizar “abstract” con final o private o static.
- **static:** Métodos y variables que se pueden usar sin instanciar la clase a la que pertenecen.
- **synchronized:** Métodos que aseguran atomicidad => Solo puede ser accedido por un hilo a la vez.
- **transient:** Variables que no pueden ser serializadas.

NOTA: Las variables que son constantes (public static final String VAR = “hola”) siempre deben ser inicializadas.

10.4. Constructores

- Nunca tiene valor de retorno, ni siquiera void.
- No pueden ser static, final o abstract.
- Visibilidad: public o private o protected o default (sin valor).
- Se deben llamar igual que la clase (la primera letra con mayúscula).

11. Clase Object

Todas las clases en Java son descendentes (directos o indirectos) de la clase *Object*. Esta clase define los estados y comportamientos básicos que deben tener todos los objetos.

Entre los métodos que incorpora la clase *Object* y que por tanto hereda cualquier clase en Java dispones de:

Principales métodos de la clase Object

Método	Descripción
<code>Object ()</code>	Constructor.
<code>clone ()</code>	Método clonador : crea y devuelve una copia del objeto ("clona" el objeto).
<code>boolean equals (Object obj)</code>	Indica si el objeto pasado como parámetro es igual a este objeto.
<code>void finalize ()</code>	Método llamado por el recolector de basura cuando éste considera que no queda ninguna referencia a este objeto en el entorno de ejecución.
<code>int hashCode ()</code>	Devuelve un código hash para el objeto.
<code>toString ()</code>	Devuelve una representación del objeto en forma de String .

12. Clases wrapper

Hay ciertas estructuras de datos, como las colecciones, que no pueden trabajar con tipos primitivos de datos. Para poder usarlas, se han implementado unas clases especiales, denominadas **clases envoltorio** o **wrapper**, que “envuelven” a los tipos primitivos.

Por cada tipo primitivo hay una clase wrapper.

Tipo primitivo	Clase envoltorio
<code>byte</code>	<code>Byte</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>boolean</code>	<code>Boolean</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>
<code>char</code>	<code>Character</code>

Todas las clases de este tipo, salvo `Character`, heredan de la clase abstracta `Number`.

Para poder utilizarlas, simplemente se crea un objeto pasando su tipo primitivo al constructor.

Método constructor a partir de un valor de tipo simple

```
Character letra = new Character('A');
Integer numero = new Integer(10);
```

También es posible crearlas sin ni siquiera invocar al constructor, similar a lo que hacemos con `String`. Este mecanismo se llama *autoboxing*.

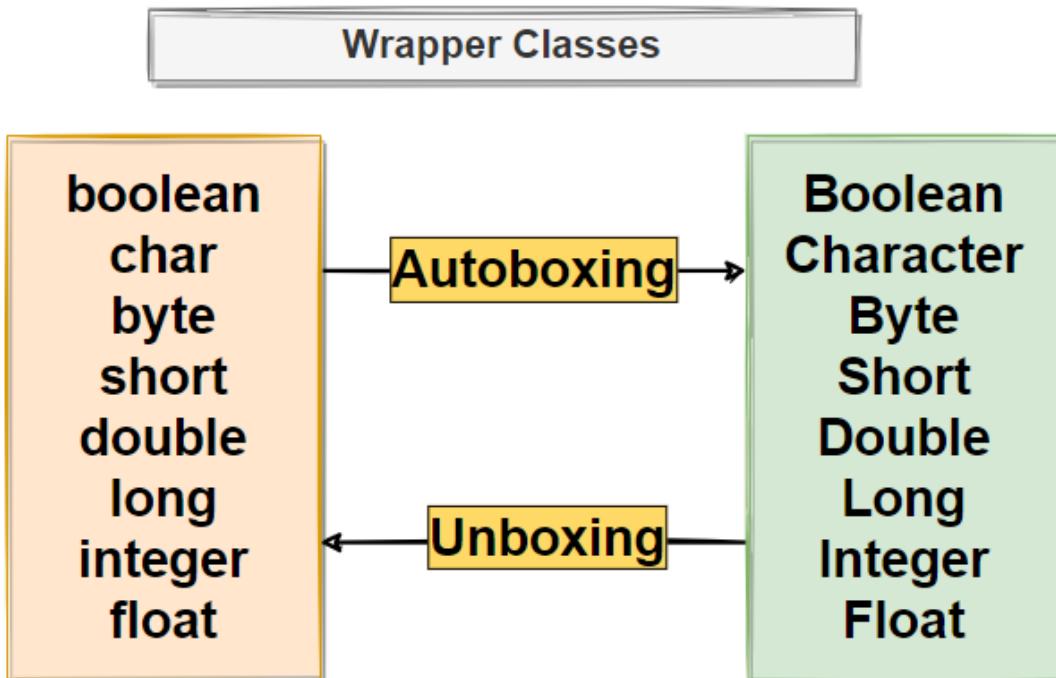
```
public static void main(String[] args) {

    Integer entero = new Integer(123);
    Integer entero2 = 456;

    Integer x = 3, y = 5;
```

El proceso de conversión de una clase *wrapper* a un tipo primitivo se llama *unboxing*.

```
int entero3 = entero;
int entero4 = entero2.intValue();
```



Desde Java 9, no se recomienda utilizar el constructor para la creación de un objeto *Wrapper*, sino que se recomienda el uso de *valueOf*.

```
Integer entero5 = Integer.valueOf(56);
```

12.1. Trabajando con String

Para pasar de un *String* a un objeto *Wrapper* se utiliza el método *parse* o *valueOf*:

```
Integer entero6 = Integer.parseInt("876");
Integer entero7 = Integer.valueOf("876");
```

Para pasar de un *Wrapper* a un *String* se utilizan *valueOf* y *toString*.

```
String enteroCadena = String.valueOf(1234);
String enteroCadena2 = entero7.toString();
```

13. Garbage collector

En Java existen dos formas de que un objeto no esté referenciado:

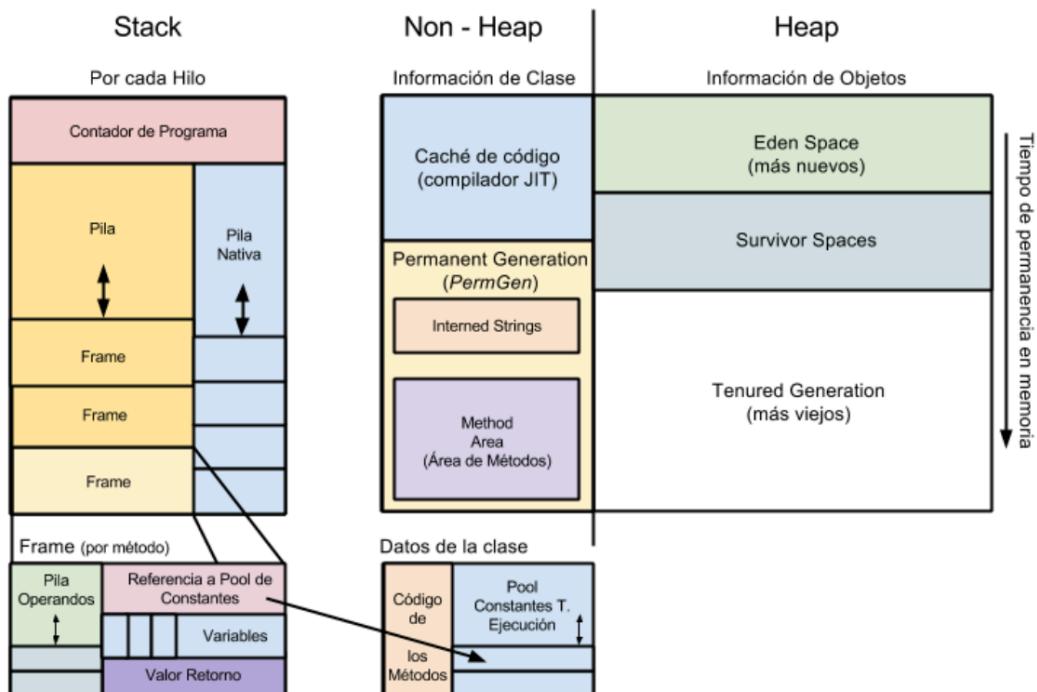
- No asignar una variable en la creación de la instancia. Esta opción no tiene ningún sentido:

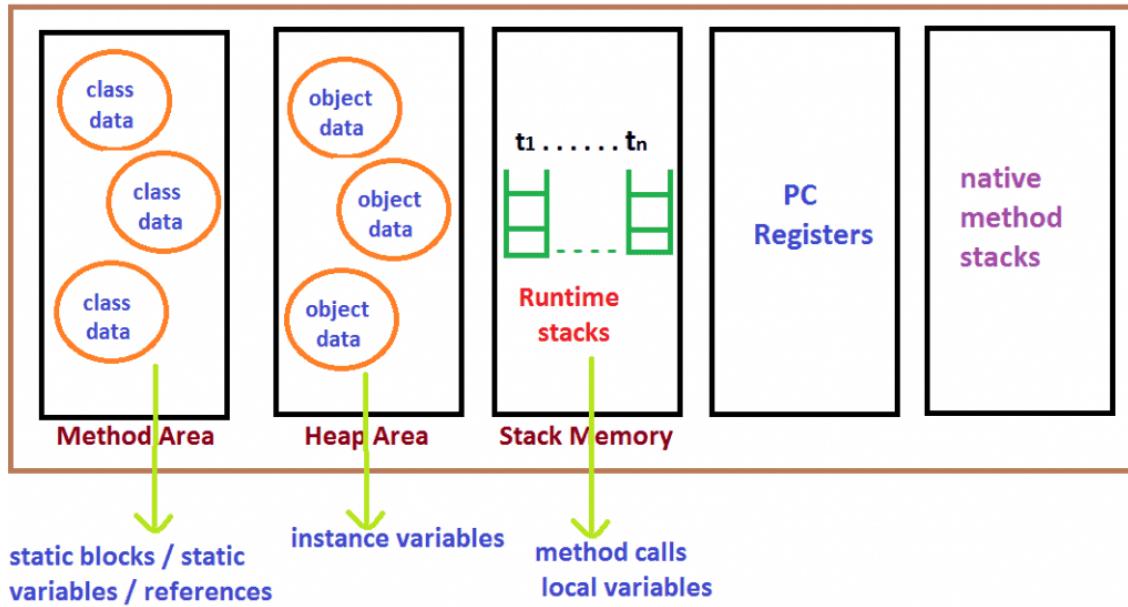
```
new Persona();
```

- Asignar null a las variables que contenían objetos.

En los dos casos, el objeto queda perdido en memoria y no se puede acceder a él. Con el tiempo, se pueden ir almacenando muchos objetos “des-referenciados” y saturar la memoria.

Para evitar este problema Java dispone de un mecanismo conocido como recolector de basura o *garbage collector (GC)*, que se ocupa de comprobar si los objetos que están en el *heap* tienen o no referencia. En caso de no tenerla, los destruye y libera memoria.





El *stack area* es único por cada proceso (hilo), sin embargo, el *method area* (almacena información a nivel de clase como su nombre, nombre de la clase que la contiene, métodos, variables, variables estáticas, etc.) y el *heap area* (almacena información de todos los objetos en memoria) son recursos compartidos, o dicho de otra forma, hay uno por JVM.

Por tanto, como el *heap* es un área compartida, el GC puede revisarlo de forma global.