Métodos

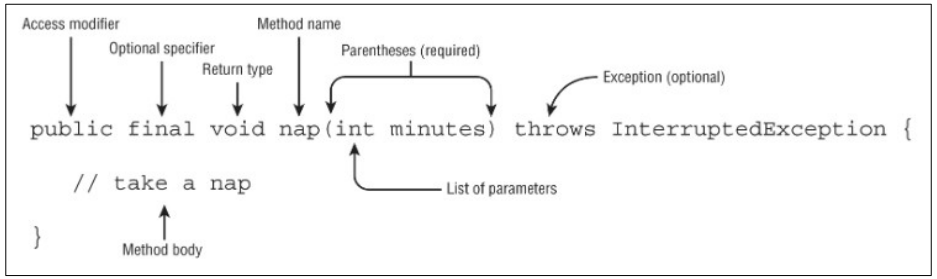

Figura 1: Estructura de un método

Element	Value in nap() example	Required?
Access modifier	public	No
Optional specifier	final	No
Return type	void	Yes
Method name	nap	Yes
Parameter list	(int minutes)	Yes, but can be empty parentheses
Method signature	nap(int)	Yes
Exception list	throws InterruptedException	No

Method body	{
	// take a nap
	}
		Yes, except for abstract methods

Figura 2: Elementos de la estructura de un método

Modificadores Opcionales (Optional specifier)

Estos modificadores pueden combinarse junto con el modificador de acceso de turno y no tienen inconvenientes de ponerse en cualquier orden (el orden por buena práctica es modificador de acceso, clase, instancia y tipo de retorno) siempre y cuando vayan antes del tipo de retorno.

Modifier	Description
static	Indicates the method is a member of the shared class object
abstract	Used in an abstract class or interface when the method body is excluded
final	Specifies that the method may not be overridden in a subclass
default	Used in an interface to provide a default implementation of a method for classes that implement the interface
synchronized	Used with multithreaded code
native	Used when interacting with code written in another language, such as C++
strictfp	Used for making floating-point calculations portable

Figura 3: modificadores opcionales

```
1 public class Exercise {
2     public void bike1() {}
3     public final void bike2() {}
4     public static final void bike3() {}
5     public final static void bike4() {}
6     public modifier void bike5() {} // modifier no es un modificador opcional valido
7     public void final bike6() {} // final no puede ir despues del tipo de retorno
8     final public void bike7() {}
9 }
```

Tipo de retorno (Return Type)

Siempre debe aparecer antes del nombre del método, indica el tipo de dato (primitivo u objeto) que el método devolverá y si no devolverá nada se indica la palabra reservada ***void***.

Los métodos que tienen ***void*** pueden tener una sentencia ***return*** sin valor de retorno (lo cual indica que en ese punto la función finalizara) u omitir la sentencia ***return*** por completo.

```
public void swin(int dist) {
    if (dist <= 0) {
        // no hacer nada
        return;
    }
    System.out.println("corriendo" + dist + " metros");
}
```

```
public class Hike {
    public void hike1() {}
    public void hike2() {
        return;
    }
    public String hike3() {
        return "";
    }
    public String hike4() {} // por su definicion esta obligado a devolver un String
     public hike5() {} // debe al menos utilizar void
     public String int hike6() {} // no puede haber 2 tipos de retorno
     String hike7(int a) {
         if (1 < 2) return "sal"; // no compila porque no hay un valor de retorno
     } // para la funcion fuera del if
}
```

```
String hike7(int a) {
    if (1 < 2) return "sal";
    return "arena"; // ahora compilara pero devolvera una advertencia
    // ya que siempre se cumplira el if y el return final nunca se ejecuta
}
```

```
public class Menos {
    int getAltura1() {
        int temp = 9;
        return temp;
    }

    int getAltura2() {
        int temp = 9 L; // no podemos insertar un LONG en un INT
        return temp;
     }

     int getAltura3() {
         long temp = 9 L; // no podemos retornar un LONG en un INT
         return temp;
     }
}
```

Nombre de métodos (Method Name)

Los métodos se pueden nombrar cumpliendo las siguientes reglas:

- solo puede tener letras, símbolos de monedas, números o _
- el primer carácter no puede ser un número o una palabra reservada
- el guion bajo solo no puede utilizarse

```
public class BeachTrip {
    public void jog1() {}
    public void 2jog() {} // no puede iniciar con un numero
    public jog3 void() {} // no puede ir void despues del nombre
    public void Jog_$() {}
    public _() {} // no puede nombrarse con solo el guion abajo
    public void() {} // no puede carecer de nombre
}
```

Lista de parámetros (Parameter List)

Este campo puede estar vació o tener diferentes tipos de variables separadas por coma.

```java
public class Educacion {
    public void run1() {}
    public void run2 {} // no tiene seccion para los parametros
    public void run3(int a) {}
    public void run4(int a; int b) {} // no se puede separar con ;
    public void run5(int a, int b) {}
}
```

Firma del método (Method Signature)

Esta compuesto por el nombre del método y la lista de parámetros (solo los tipos de datos de la lista de parámetros y respeta su orden). Sirve para que la JVM detecte el método que quieres usar y si esta permito)

```java
public class Trip {
    public void visitZoo(String a, int b) {}
    public void visitZoo(String c, int d) {} // no puede haber 2 metodos con la misma firma de metodo en la misma clase
}

public class Trip {
    public void visitZoo(String a, int b) {}
    public void visitZoo(int c, String d) {}
}
```

Lista de Excepciones (Exception List)

Es opcional declarar una excepción en la función (o una lista separada por comas)

```java
public class  Zoo{
    public void zeroException(){}
    public void oneException() throws IllegalArgumentException{}
    public void twoException() throws IllegalArgumentException, InterruptedException{}
}
```

Cuerpo del método (Method Body)

Es el bloque de código que va dentro del método


```java
public class  Zoo{
    public void zeroException(){}
    public void oneException() throws IllegalArgumentException{}
    public void twoException() throws IllegalArgumentException, InterruptedException{}
}
```


Variables locales y de instancia (Local and Instance Variables)

Las **variables locales** son definidas d**entro de los métodos o bloques de código** mientras que l**as variables de instancia** son definidas como **miembros de una clase**. Todas las variables locales se destruyen después de la ejecución del bloque pero las de instancia aun son accesibles mientras exista el objeto

```java
public class Lion {
    int humber = 4; // variable de instancia

    public int feedZoo() {
        int snack = 10; // variable local

        if (snack > 4) {
            log dinner = snack++;
        }
         hunger--;

         return snack;
    }
}
```

Modificador de variable locales (*final*)

El modificador ***final*** es el único que se puede aplicar a este tipo de variables (es útil cuando no se quiere que la variable cambie durante la ejecución del método) en la asignación de este modificador no es estricto indicar el valor que tendrá la variable pero el primero que se asigne antes de ser utilizado sera el único valido.

Observación: este modificador se puede asignar a una clase, propiedad o método

```
public void zooAnimal(boolean isWeekend) {
   final int rest;
   if (isWeekend) rest = 5; else rest = 20;
   System.out.print(rest);

   final var giraffe = new Animal();
   final int[] friends = new int[5];

   rest = 10; // ya se asigno el valor en la linea 3
    giraffe = new Animal(); // ya se asigno en la linea 6
    friends = null; // ya se asigno en la linea 7
}
```

```
public void zooAnimal(boolean isWeekend) {
   final int rest;
   if (is Weekend) rest = 5;
   System.out.print(rest); // no hay un valor de rest para el caso falso
}
```

Para los casos de objetos final permite que podamos modificar las elementos internos del objeto pero no se puede re-apuntar la variable a otro objeto.

```
public void zooAnimal() {
   final int rest = 5;
   final Animal giraffe = new Animal();
   final int[] friends = new int[5];

   giraffe.setName("George");
   friends[2] = 12;

}
```

Observación: Una variable local es efectivamente ***final*** si su valor no se modifica después de ser asignada, aunque no se le haya añadido explícitamente la palabra clave ***final***.

Modificador de variables de instancia (*final*)

Este tipo de variables pueden utilizar de forma opcional ademas del modificador ***final***, el ***volatile*** y ***transient***

Modificador	Descripción
final	Se inicializa una vez por cada instancia de clase
volatile	Indica a la JVM que el valor de la variable puede ser modificada por otro hilo
transient	Se usa para indicar que una variable de instancia no debe ser serializada con la clase (serializar es convertir un objeto en bytes para par almacenarlo o transmitirlo.

Una variable de instancia marcada como ***final*** debe ser inicializada con un valor. Esto puede hacerse en tres lugares:

- Directamente en la declaración (como ***age = 10***).
- En un bloque de inicialización de instancia (como ***{ fishEat = 10; }***).
- Dentro del constructor de la clase (como se hace con ***name***).

```
public class Polar {
   final int age = 10;
   final int fishEat;
   final String name;

   {fishEat = 100;}

   public Polar() {
      name = "Robert";
    }
}
```

Si no inicializas una variable de instancia final o intentas asignarle un valor más de una vez, el compilador generará un error.

Creando un método con Varargs

Varargs “argumento variable”. Un método con este tipo de parámetros debe cumplir:

- Un método puede tener a lo máximo un varargs
- El parámetro varargs solo puede ir como ultimo parámetro del método

```java
public class Visit {
    public void walk1(int...steps) {}
    public void walk2(int start, int...steps) {}
    public void walk3(int...steps, int start) {} // varargs no puede estar como primer parametro
    public void walk4(int...steps, int...start) {} // solo puede haber 1 varargs
}
```

Se puede pasar una matriz o una lista de elementos y Java lo creara.

Si omitimos los varargs se creara una matriz vacía
Solo necesitamos utilizar el indice de una matriz para acceder a los valores de un varargs

```java
public void walk1(int... steps){
    int[] step2 = steps;
    System.out.println(step2[1]);
}

int[] data = new int[]{1,2,3}; // pasando un array
walk1(data);

wal1(1,2,3);  // pasando una lista de valores

// mostrando un valor
walk3(1,2,3); // 2
```

Al combinar un varargs con otros parámetros

```java
public class DogWalk{
    public static void walkDog(int start, int... steps){
        System.out.println(steps.length);
    }

    public static void main(String[] args){
        walkDog(1); // 0
        walkDog(1,2); // 1
        walkDog(1,2,3); // 2
         walkDog(1, new int[]{4,5}); // 2
         walkDog(1, null); // arroja una excepcion porque no se puede ver el tamaño de un array null
    }
}
```


Modificadores de acceso

Acceso Privado (Private Access)

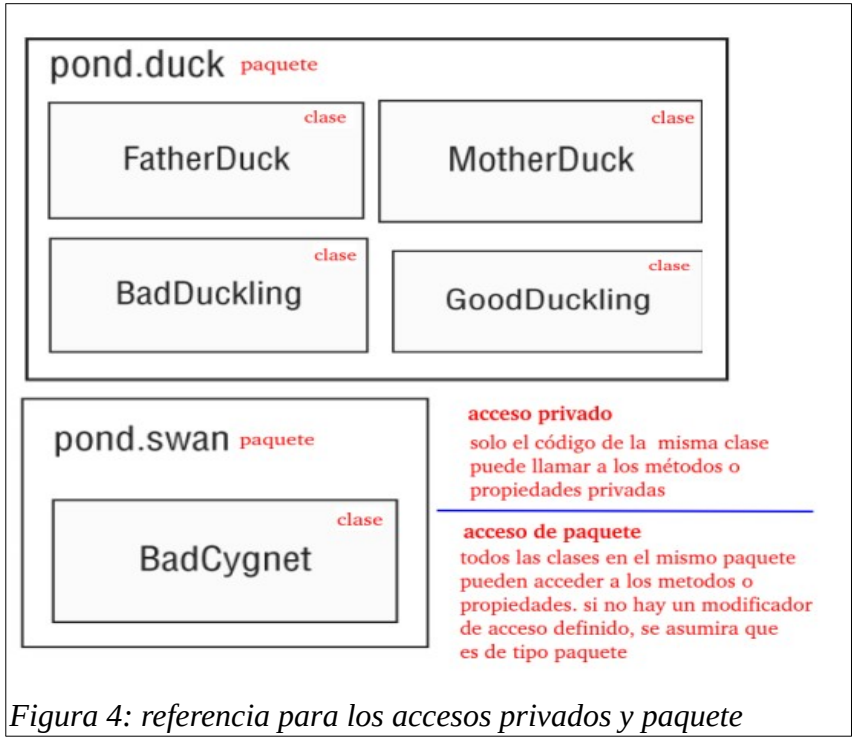

Figura 4: referencia para los accesos privados y paquete

```java
package pond.duck;

public class FatherDuck {
    private String noise = "quack";

    private void quack() {
        System.out.print(noise); // accedemos desde la misma clase
    }
}
```

```java
package pond.duck;

public class BadDuckling {
    public void makeNoise() {
        var duck = new FatherDuck();
        duck.quack(); // error porque el metodo es privado en la clase
        System.out.print(duck.noise); // error porque la propiedad es privada en la clase
    }
}
```

Acceso de paquete (Package Access)

En el siguiente código definidos una variable y un método con acceso de tipo paquete:

```java
package pond.duck;

public class MotherDuck {
    String noise = "quack";

    void quack() {
        System.out.print(noise);
    }
}
```

Ahora definimos otra clase que esta en el mismo paquete por lo cual puede acceder sin problemas a los elementos anteriores.

```java
package pond.swan;

import pond.duck.MotherDuck; // importo otro paquete

public class BadCygnet {
    public void makeNoise() {
        var duck = new MotherDuck();
        duck.quack(); // no se puede acceder desde otro paquete
        System.out.print(duck.noise); // no se puede acceder desde otro paquete
    }
}
```

```java
package pond.duck;

public class GoodDuckling {
    public void makeNoise() {
        var duck = new MotherDuck();

        duck.quack(); // acceso de paquete permitido
        System.out.print(duck.noise); // acceso de paquete permitido
    }
}
```

Acceso Protegido (Protected Access)

Este tipo de acceso permite que las clases dentro del mismo paquete y las subclases (incluso si están en otro paquete) accedan a los miembros de una clase (propiedades y métodos).

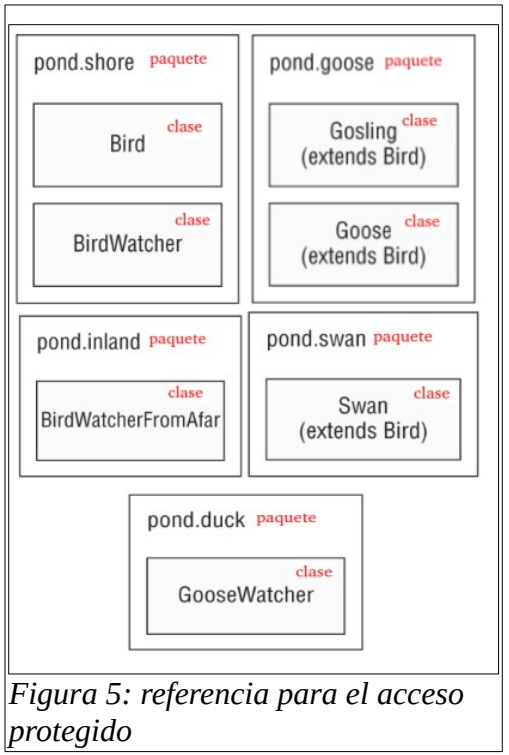

Figura 5: referencia para el acceso protegido

En este primer ejemplo vemos que la propiedad y método son protegidos y se pueden utilizar sin problemas dentro de su misma clase

```java
package pond.shore;

public class Bird {
    protected String text = "floating";

    protected void floatInWater() {
        System.out.print(text); // propiedad protegida accedida sin problemas
    }
}
```

```java
package pond.goose; // paquete diferente de Bird

import pond.shore.Bird;

public class Gosling extends Bird { // Gosling es una subclase de Bird
    public void swim() {
        floatInWater(); // por ser una subclase puedo acceder al metodo protegido de mi clase padre.
        System.out.print(text); // por ser una subclase puedo acceder a la propiedad protegida de mi clase padre.
    }

    public static void main(String[] args) {
        new Gosling().swim();
    }
}
```

```java
package pond.shore;

public class Bird {
    protected String text = "floating";

    protected void floatInWater() {
        System.out.print(text); // propiedad protegida accedida sin problemas
    }
}
```

```java
package pond.shore; // Mismo paquete que Bird

public class BirdWatcher {
    public void watchBird() {
        Bird bird = new Bird();
        bird.floatInWater(); // puede acceder al método protegido
        System.out.print(bird.text); // puede acceder a la propiedad protegida
    }
}
```

```java
package pond.inland; // esta clase se ubica en una paquete diferente a Bird

import pond.shore.Bird;

public class BirdWatcherFromAfar { // la clase no es hija de Bird
    public void watchBird() {
        Bird bird = new Bird();
        bird.floatInWater(); // no podremos utilizar un metodo protegido de Bird
        System.out.print(bird.text); // no podremos utilizar una propiedad protegida de Bird
    }
}
```

Ahora evaluamos otra validación para los tipo protegidos

```java
package pond.shore;

public class Bird {
    protected String text = "floating";

    protected void floatInWater() {
        System.out.print(text); // propiedad protegida accedida sin problemas
    }
}
```

```java
package pond.swan; // La clase se ubica en un paquete diferente al de la clase Bird

import pond.shore.Bird;

public class Swan extends Bird { // Swan es una subclase de Bird
    public void swim() {
        floatInWater(); // podra acceder al metodo protegido de Bird
        System.out.print(text); // podra acceder a la propiedad protegida de Bird
    }

    public void helpOtherSwanSwim() {
        Swan other = new Swan();
        other.floatInWater(); // podra acceder al metodo protegido de su padre
        System.out.print(other.text); // podra acceder a la propiedad protegida de su padre
    }

    public void helpOtherBirdSwim() {
        Bird other = new Bird();
        other.floatInWater(); // (1)
        System.out.print(other.text); // (2)
    }
}
// (1) (2) no se puede acceder a metodos de otra instancia de la clase padre, desde otro paquete
```

ahora revisamos otro tipo de ejemplo:

```java
package pond.goose;

import pond.shore.Bird;

public class Goose extends Bird {
    public void helpGooseSwim() {
        Goose other = new Goose();
        other.floatInWater();
        System.out.print(other.text);
    }

    public void helpOtherGooseSwim() {
        Bird other = new Goose(); // creamos un objeto hijo y lo guardamos en un tipo padre
        other.floatInWater(); // no se puede acceder al metodo de la clase padre desde un hijo
        System.out.print(other.text); // no se puede acceder a una propiedad de un padre desde un hijo
    }
}
```

en este ultimo ejemplo al no estar en el mismo paquete ni ser una extensión de Bird no podemos ejecutar el método desde un instancia de la clase ya que esto solo se permite desde dentro de las definiciones de herencia (no desde las instancias)

```java
package pond.duck;

import pond.goose.Goose;

public class GooseWatcher {
    public void watch() {
        Goose goose = new Goose();
        goose.floatInWater(); // no compila
    }
}
```

Acceso publico (Public access)

Este es el acceso mas sencillo ya que permite acceder a los métodos o propiedades de la clase desde cualquier lugar

```java
package pond.duck;

public class DuckTeacher {
    public String name = "helpful";

    public void swim() {
        System.out.print(name); // public access is ok
    }
}
```

```java
package pond.goose;

import pond.duck.DuckTeacher;

public class LostDuckling {
    public void swim() {
        var teacher = new DuckTeacher();
        teacher.swim(); // se puede acceder sin problemas porque son publicos
        System.out.print("Thanks " + teacher.name); // se puede acceder sin problemas porque son publicos
    }
}
```

Si deseamos resumir con una analogía de una frase los métodos de acceso podemos indicar

Un método en _____ puede acceder a los miembros _____

	Private	**Package**	**Protected**	**Public**
La misma clase	Si	Si	Si	Si
Otra clase en el mismo paquete	No	Si	Si	Si
Una subclase en diferente paquete	No	No	Si	Si
Una clase no relacionada en un diferente paquete	No	No	No	Si

Modificador opcional *STATIC*

Aplica a: Clases, variables y métodos

Respecto a las variables y métodos este modificación indica que ambos elementos serán parte de la clase y no cambiaran en cada instanciación de esta.

```java
public class Penguin {
    String name;
    static String nameOfTallestPenguin;

    public static void main(String[] args) {
        var p1 = new Penguin();
        p1.name = "Lily";
        p1.nameOfTallestPenguin = "Luis";

        var p2 = new Penguin();
        p2.name = "Ave";
        p2.nameOfTallestPenguin = "Wily";

        System.out.println(p1.name); // Lily
        System.out.println(p1.nameOfTallestPenguin); // Wily

        System.out.println(p2.name); // Ave
        System.out.println(p2.nameOfTallestPenguin); // Wily
    }
}
```

Ya que el método mas conocido de una clase (main) tiene este modificador podemos llamar sin problemas

```java
public class Koala {
    public static int count = 10;
    public static void main(String[] args) {
        System.out.println(count);
    }
}

public class KoalaTester {
    public static void main(String[] args) {
        Koala.main(new String[0]); // 10
    }
}
```

las variables con este modificador pueden ser accedidas llamando directamente a su clase o también desde sus instancias. En el siguiente ejemplo podremos ver que a pesar que se modifico el enlace de la variable “k” en la linea 15 aun se puede acceder la variable estática.

```java
public class Koala {
   public static int count = 10;
   public void vuela() {
      System.out.println("volar");
   }
}

public class KoalaTester {
   public static void main(String[] args) {
      System.out.println(Koala.count); // 10

      Koala k = new Koala();
      System.out.println(k.count); // 10

      k = null;

      System.out.println(k.count); // 10
   }
}
```

Otro aspecto a tener en cuenta es cuando intentamos llamar a un método de instancia desde un método estático por ejemplo:

```java
public class MantaRay {
   private String name = "sal";
   public static void first(){}
   public static void second(){}
   public void third(){System.out.println(name);}
   public static void main(String args[]){
      first();
      second();
      third(); // error por llamar a un metodo de instancia
   }
}
```

podemos arreglar volviendo “third()” un método estático pero igual nos seguirá arrojando error ya que la variable dentro de este método no es estática. Por lo cual la solución pueden ser volver

```java
public class MantaRay {
   private String name = "Sammy";
   ...
   public void third() {System.out.println(name);}
   public static void main(String args[]) {
      ...
      var ray = new MantaRay();
      ray.third();
   }
}
```

estatifica también la variable o crear una instancia de la clase y llamar al método desde esta instancia.

La misma lógica aplica para variables de instancia

```java
public class Giraffe {

   public void eat(Giraffe g) {}
   public void drink() {};
   public static void allGiraffeGoHome(Giraffe g) {}
   public static void allGiraffeComeOut() {}
}
```

Método	Puede llamar a:	
allGiraffeGoHome()	allGiraffeComeOut()	Si
allGiraffeGoHome()	drink()	No
allGiraffeGoHome()	g.eat()	Si
eat()	allGiraffeComeOut()	Si
eat()	drink()	Si
eat()	g.eat()	Si

A continuación podremos ver otro ejemplo en el cual la linea 11 no compila porque llama a un método de instancia dentro de un método estático y la linea 14 no compila porque utilizamos una variable estática como parte de una variable de instancia.

```java
public class Gorilla {
   public static int count;
   public static void addGorilla() { count++; }
   public void babyGorilla() { count++; }
   public void announceBabies() {
      addGorilla();
      babyGorilla();
   }
   public static void announceBabiesToEveryone() {
      addGorilla();
      babyGorilla(); // no compila
   }
   public int total;
   public static double average = total / count; // no compila
}
```

Una forma de definir constantes dentro de las clases es utilizar los modificadores ***static*** y ***final*** junto con un nombre de variable que sea ***totalmente en mayúsculas*** así como definir su valor al definir la constante como en el siguiente ejemplo:

```java
public class ZooPen {

   private static final int NUM_BUCKETS = 45;

   public static void main(String[] args) {

      NUM_BUCKETS = 5; // no compilara por su definición

   }
}
```

```java
public class Panda {
   final static String name = "Ronda"; // Inicialización en línea
   static final int bamboo;
   static final double height; // DOES NOT COMPILE - Falta inicialización

   static {
      bamboo = 5; // Inicialización en bloque estático
   }
}
```

En el bloque anterior la constante “height” no recibe en ninguna parte de la clase su inicialización por lo cual el compilador arrojara un error

```java
private static final int NUM_SECONDS_PER_MINUTE;
private static final int NUM_MINUTES_PER_HOUR;
private static final int NUM_SECONDS_PER_HOUR;

static {
  NUM_SECONDS_PER_MINUTE = 60;
  NUM_MINUTES_PER_HOUR = 60;
}

static {
  NUM_SECONDS_PER_HOUR
    = NUM_SECONDS_PER_MINUTE * NUM_MINUTES_PER_HOUR;
}
```

en el bloque anterior tener en cuenta que a los bloques entre las linea 5-8 y 10-13 se les denomina bloques (en este caso estáticos) y se ejecutan solo una vez cuando la clase es cargada a memoria. También indicar que en el código anterior se muestra mas ejemplos de definición de constantes.

También podemos utilizar el modificador ***static*** para realizar importaciones de variables o métodos estáticos de otras clase como el siguiente ejemplo:

```java
import java.util.List;
import static java.util.Arrays.asList; // static import

public class ZooParking {
   public static void main(String[] args) {
      List<String> list = asList("one", "two"); // No necesitamos hacer un new Array y llamar a la función
   }
}
```

A continuación se verán ejemplos de importaciones erróneas utilizando static

```
1 // 1: import static java.util.Arrays; // DOES NOT COMPILE
2 // 2: import static java.util.Arrays.asList;
3 // 3: static import java.util.Arrays.*; // DOES NOT COMPILE
4 // 4: public class BadZooParking {
5 // 5: public static void main(String[] args) {
6 // 6:  Arrays.asList("one"); // DOES NOT COMPILE
7 // 7: }
8 // 8: }
```

Línea 1: Intenta importar la clase Arrays con una importación estática, lo cual no es válido. Las importaciones estáticas son solo para miembros estáticos.

Línea 3: El orden de las palabras clave es incorrecto (***static import*** en lugar de ***import static***).

Línea 6: Aunque se importó el método ***asList()*** en la línea 2, no se importó la clase ***Arrays***. Para usar ***Arrays.asList()***, se debe importar la clase ***Arrays*** con ***import java.util.Arrays;***

Respecto a las variables tambien debemos indicar la imagen ilustra que en Java, cuando se pasa una variable a un método, se pasa una copia del valor, no la variable en sí. Las modificaciones en el método no afectan a la variable original.

```
1 public static void main(String[] args) {
2    int num = 4;
3    newNumber(num);
4    System.out.print(num); // 4
5 }
6
7 public static void newNumber(int num) {
8    num = 8;
9 }
```

En esta sección, se muestra un ejemplo de cómo se pasan objetos (en este caso, un objeto ***String***) a un método. Aunque ***String*** es un objeto y no un tipo primitivo, el comportamiento es similar al paso por valor de los primitivos.

```
1 public class Dog {
2    public static void main(String[] args) {
3       String name = "Webby";
4       speak(name);
5       System.out.print(name); // Webby
6    }
7
8    public static void speak(String name) {
9       name = "Georgette";
10    }
11 }
```

Línea 3: ***String name = "Webby"***; Se crea un objeto ***String*** con el valor "Webby" y se asigna a la variable name.

Línea 4: ***speak(name);*** Se llama al método ***speak()*** y se pasa la variable ***name*** como argumento. Se pasa una copia de la referencia al objeto "Webby".

Línea 9: ***name = "Georgette";*** Dentro del método ***speak()***, la variable ***name*** (que es una copia de la referencia) se reasigna a un nuevo objeto ***String*** con el valor "Georgette". Esto no afecta la referencia original en el método ***main().***

Línea 5: ***System.out.print(name);*** // Webby Se imprime el valor de name en el método ***main()***. El resultado es "Webby" porque la reasignación en ***speak()*** no afectó la referencia original.

En esta sección, se muestra cómo se puede modificar un objeto pasado como argumento a un método.

```
1 public class Dog {
2    public static void main(String[] args) {
3       var name = new StringBuilder("Webby");
4       speak(name);
5       System.out.print(name); // WebbyGeorgette
6    }
7
8    public static void speak(StringBuilder s) {
9       s.append("Georgette");
10    }
11 }
```

Línea 4: ***var name = new StringBuilder("Webby");*** Se crea un objeto ***StringBuilder*** con el valor "Webby" y se asigna a la variable ***name***.

Línea 5: ***speak(name);*** Se llama al método ***speak()*** y se pasa la variable ***name*** como argumento. Se pasa una copia de la referencia al objeto ***StringBuilder***.

Línea 10: ***s.append("Georgette");*** Dentro del método ***speak()***, se llama al método ***append()*** en el objeto ***StringBuilder*** al que se hace referencia. Esto modifica el objeto original.

Línea 6: ***System.out.print(name);*** // ***WebbyGeorgette*** Se imprime el valor de name en el método ***main().*** El resultado es "WebbyGeorgette" porque el método ***speak()*** modificó el objeto original.

En este ultimo ejemplo se debe tener en cuenta que ambas variables estaban referencias al mismo objeto por lo cual la modificación del metodo speak() afecto el objeto inicial.

La sentencia RETURN en Java permite devolver un primitivo u objeto y este debe ser almacenado por la variables, en el siguiente ejemplo la variable ***tickets*** no cambia ya la modificación realizada por la función no es guardada nuevamente en la variable.

```
1  public class ZooTickets {
2
3     public static void main(String[] args) {
4        int tickets = 2;              // tickets = 2
5        String guests = "abc";        // guests = abc
6
7        addTickets(tickets);          // tickets = 2
8        guests = addGuests(guests);   // guests = abcd
9
10        System.out.println(tickets + guests); // 2abcd
11    }
12
13    public static int addTickets(int tickets) {
14       tickets++;
15       return tickets;
16    }
17
18    public static String addGuests (String guests) {
19       guests += "d";
20       return guests;
21    }
22 }
```

Autoboxing y Unboxing de variables

Es el proceso automático de Java que permite convertir primitivos a Objetos (int a Integer) o viceversa

```
1  int quack = 5;
2  Integer quackquack = Integer.valueOf(quack); // Convierte int a Integer
3  int quackquackquack = quackquack.intValue(); // Convierte Integer a int
4
5
6  int quack = 5;
7  Integer quackquack = quack; // Autoboxing
8  int quackquackquack = quackquack; // Unboxing
9
10
11  Short tail = 8; // Autoboxing
12  Character p = Character.valueOf('p');
13  char paw = p; // Unboxing
14  Boolean nose = true; // Autoboxing
15  Integer e = Integer.valueOf(9);
16  long ears = e; // Unboxing, luego casting implícito
```

En el siguiente ejemplo el autoboxing y la promoción numérica (conversión implícita de un tipo primitivo a otro más grande) tienen limitaciones. Esta línea no compila porque Java no puede realizar autoboxing y promoción numérica al mismo tiempo.

```
1  Long badGorilla = 8; // DOES NOT COMPILE
```

en este caso, Java no puede decidir si convertir el int a un Integer y luego intentar asignarlo a un Long (lo cual fallaría), o si convertir el int a un long y luego intentar hacer autoboxing a un Long (lo cual también fallaría ya que para este caso requiere que se haga casting).

En el siguiente ejemplo también podemos ver que si utilizamos null este no se podrá hacer unboxing y arrojara una excepción

```
1  Character elephant = null;
2
3  char badElephant = elephant; // NullPointerException
```

En el siguiente ejemplo la linea 12 no compila porque Java no convierte automáticamente un ***long*** a un ***int*** porque es una conversión de estrechamiento que podría perder información. Se necesitaría un cast explícito: ***c.jump((int)123L);***.

```java
public class Chimpanzee {

   public void climb(long t) {}
   public void swing(Integer u) {}
   public void jump(int v) {}

   public static void main(String[] args) {
      var c = new Chimpanzee();

       c.climb(123);
       c.swing(123);
       c.jump(123L); // DOES NOT COMPILE
    }
}
```

el siguiente ejemplo, al llamar a ***g.rest(8);*** se produce un error de compilación. Java no puede realizar a la vez la conversión de ***int*** a ***long*** y el autoboxing de ***long*** a ***Long***. Se necesitaría una conversión explícita, como ***g.rest(8L);*** o g.***rest(Long.valueOf(8));***.

```java
public class Gorilla {

   public void rest(Long x) {
      System.out.print("long");
   }

   public static void main(String[] args) {
      var g = new Gorilla();
      g.rest(8); // DOES NOT COMPILE
    }
}
```

Sobrecarga de métodos (Overloading Methods)

Esto ocurre cuando un método en la misma clase se presenta varias veces pero tiene firmas de método diferentes. En el siguiente ejemplo se muestra varias implementaciones del método fly() que son totalmente aceptadas

```java
public class Falcon {

   public void fly(int numMiles) {}
   public void fly(short numFeet) {}
   public boolean fly() { return false; }
   void fly(int numMiles, short numFeet) {}
   public void fly(short numFeet, int numMiles) throws Exception {}

}
```

Ahora en el siguiente ejemplo vemos un método con tipos de retorno diferente pero la misma firma (nombre y lista de argumentos) por lo cual el segundo y tercer método, arrojan un error

```java
public class Hawk {
public void fly(int numMiles) {}
public static void fly(int numMiles) {} // DOES NOT COMPILE
public void fly(int numKilometers) {} // DOES NOT COMPILE
}
```

Tipos de referencia (Reference Types)

Cual seria la regla que Java utiliza para tomar la versión mas específica del método en una sobrecarga de métodos. En el siguiente ejemplo se imprime “string-object” identifica primero el ***string*** y después el 56 hace autoboxing a entero pero al no encontrar una firma lo asocia con el ***Object***.

```java
public class Pelican {
   public void fly(String s) {
      System.out.print("string");
   }
   public void fly(Object o) {
      System.out.print("object");
   }
   public static void main(String[] args) {
      var p = new Pelican();
       p.fly("test");
       System.out.print("-");
       p.fly(56);
    }
}
```

Para el siguiente ejemplo el resultado es "**CIO**". El primer calza en la función de la linea 9 ya que un ***String*** y ***StringBuilder*** implementan las interfaz ***CharSequence***. El segundo entra en la función de la linea 5 ya que lo que se pasa como parámetro va devolver una lista. El ultimo solo entre en la función ***Object*** ya que es lo mas genérico que hay.

```
1  import java.time.*;
2  import java.util.*;
3
4  public class Parrot {
5     public static void print(List<Integer> i) {
6        System.out.print("I");
7     }
8     public static void print(CharSequence c) {
9        System.out.print("C");
10    }
11    public static void print(Object o) {
12       System.out.print("O");
13    }
14
15    public static void main(String[] args) {
16       print("abc");
17       print(Arrays.asList(3));
18       print(LocalDate.of(2019, Month.JULY, 4));
19    }
20 }
```

Referencia en Primitivos

la referencia de primitivos funciona igual que la referencia de variables, se buscara que la función empalme de la forma mas precisa y sin perder datos.
Para el siguiente ejemplo el resultado es "***int-long***" ya que en el primero empalma con la función de tipo ***int*** y el segundo encuentra un mejor match con la función que recibe un ***long*** el cual permite que no se pierdan los datos.

Si comentáramos el método ***fly(int i)***, la salida sería "***long-long***" porque Java puede usar el método ***fly(long l)*** para un valor int, ya que ***long*** es un tipo de dato más amplio que ***int***. Sin embargo, Java prioriza la coincidencia exacta si está disponible.

```
1  public class Ostrich {
2     public void fly(int i) { System.out.print("int"); }
3     public void fly(long l) { System.out.print("long"); }
4     public static void main(String[] args) {
5        var p = new Ostrich();
6        p.fly(123);
7        System.out.print("-");
8        p.fly(123L);
9     }
10 }
```

Autoboxing

En este caso, la clase ***Kiwi*** tiene dos métodos llamados ***fly***, uno que acepta un ***int*** y otro que acepta un ***Integer***. Cuando llamas a ***fly(3)***, Java usa el primer método porque el valor 3 es un ***int*** y hay una coincidencia exacta. Java prioriza la coincidencia exacta y evita el autoboxing si es posible.

Si solo existiera el método ***fly(Integer numMiles)***, Java realizaría el autoboxing y convertiría el 3 en un ***Integer*** para poder llamar al método.

```
1  public class Kiwi {
2     public void fly(int numMiles) {}
3     public void fly(Integer numMiles) {}
4  }
```

Arrays

A diferencia del autoboxing que ocurre con tipos primitivos individuales (como int e Integer), Java no realiza autoboxing con arrays.

Esto significa que si tienes un método que acepta un array de ***int (int[])*** y otro que acepta un array de ***Integer (Integer[])***, Java no convertirá automáticamente un array de ***int*** en un array de **Integer** o viceversa. Debes pasar el tipo de array correcto al método que lo espera.

```
1  public static void walk(int[] ints) {}
2  public static void walk(Integer[] integers) {}
```

Varargs

Java trata los varargs como si fueran arrays. En este caso, ambos métodos ***fly*** tienen la misma firma, ya que ***int[]*** y ***int...*** son equivalentes para Java. Por lo tanto, el código no compila.

```
1  public static void walk(int[] ints) {}
2  public static void walk(Integer[] integers) {}
```

Cómo llamar a los métodos:

Puedes pasar un array a ambos métodos: ***fly(new int[] {1, 2, 3});***
Solo puedes llamar a la versión con varargs usando parámetros individuales: ***fly(1, 2, 3);***

Ahora veamos un ejemplo juntando todo:

```
1  public class Glider {
2     public static String glide(String s) { return "1"; }
3     public static String glide(String... s) { return "2"; }
4     public static String glide(Object o) { return "3"; }
5     public static String glide(String s, String t) { return "4"; }
6     public static void main(String[] args) {
7        System.out.print(glide("a"));
8        System.out.print(glide("a", "b"));
9        System.out.print(glide("a", "b", "c"));
10    }
11 }
```

la salida de este código sera 142 teniendo en cuenta el orden de preferencia:

- **Coincidencia exacta:** Java busca un método con parámetros del mismo tipo que los argumentos.
- **Tipo primitivo más amplio:** Si no hay coincidencia exacta, busca un método con parámetros de un tipo primitivo más amplio (por ejemplo, long en lugar de int).
- **Autoboxing:** Si no hay coincidencia con primitivos, busca un método con parámetros que sean la versión "boxed" (en clase envoltorio) de los argumentos (por ejemplo, Integer en lugar de int).
- **Varargs:** Si no hay otra opción, usa un método con varargs (parámetros variables).

En el código, ***glide("a")*** llama al primer método ***(String s)***, ***glide("a", "b")*** llama al cuarto método ***(String s, String t)***, y ***glide("a", "b", "c")*** llama al segundo método ***(String... s)***.