

# Funciones

Diseño estructurado

Verónica E. Arriola-Rios

Facultad de Ciencias, UNAM

1 de octubre de 2025



## Definiciones

- 1 Definiciones
- 2 Polimorfismo I: Sobrecarga
- 3 Bibliografía

# Temas

## 1 Definiciones

- Funciones
  - Declarar y definir una función
  - Métodos
  - Parámetros formales y actuales

# Funciones

- Las *funciones* son segmentos de código que implementan un algoritmo con una funcionalidad concreta.
- Al igual que los algoritmos requieren:
  - *Parámetros*, que corresponden a los argumentos del algoritmo.
  - Revisión de precondiciones.
  - Código que ejecute el algoritmo en un lenguaje dado.
  - Si la función debe devolver un valor, la última línea que ejecute debe devolver ese valor.
  - Documentación donde se explique claramente lo que hace la función y cómo satisface los requerimientos o postcondiciones.

# Funciones

## Sintaxis (Función)

```

<función>      ::= <acceso> <modificador> <tipo de regreso> <nombre>(<pará
    ↪metros>)
                <enunciado>*
                <return>
<acceso>      ::= public | protected | private | ∅
<modificador> ::= final | static | ∅
<parámetros>  ::= <tipo> <identificador> (, <tipo> <identificador>)* | ∅
<return>      ::= return <valor> | return | ∅
  
```

# Convenciones

- Los nombres de funciones inician con **minúscula**.
- Si se utiliza más de una palabra para su nombre, las palabras van pegadas y cada palabra después de la primera se escribe en mayúsculas. A esto se le llama ***camel case***.  
Ej: `hacerAlgo`
- El nombre de la función debe ser **descriptivo** de la operación que realiza.

# Temas

## 1 Definiciones

- Funciones
- Declarar y definir una función
- Métodos
- Parámetros formales y actuales

# Declarar y definir

- Las funciones se *declaran* con su *encabezado*:

## Sintaxis (Encabezado)

```
<función> ::= <acceso> <modificador> <tipo de regreso> <nombre>(<pará  
    ↪ metros>)
```

```
1 public static int minutosASegundos(int minutos)
```

- Las funciones se *definen* cuando se especifica su contenido.  
Es decir, cuando se escribe el código que implementa el algoritmo en el *cuerpo* de la función.



# Ejemplo

## Código: Una función simple con todo

```
1 public class Tiempo {
2     /**
3      * Convierte los minutos indicados a segundos.
4      * @param minutos Cualquier cantidad de minutos positivos.
5      * @return Los segundos equivalentes.
6      * @throws IllegalArgumentException si se pasan minutos negativos.
7      */
8     public static int minutosASegundos(int minutos) {
9         if(minutos < 0) throw new IllegalArgumentException("No hay minutos ↪ negativos");
10        return minutos * 60;
11    }
12 }
```

# Temas

## 1 Definiciones

- Funciones
- Declarar y definir una función
- **Métodos**
- Parámetros formales y actuales

# Métodos

## Definición (Métodos)

Los *métodos* son funciones que pertenecen a clases y operan sobre los atributos de un objeto `this`.

# Ejemplo de método

```
1  /** Clase para representar círculos.*/
2  public class Círculo {
3      private double radio;
4      /** Constructor. */
5      public Círculo(double radio) {
6          this.radio = radio;
7      }
8      /** Devuelve al área de este círculo. */
9      public double calculaÁrea() {
10         return Math.PI * radio * radio;
11     }
12     /** Indica si este círculo es más grande que <code>otro</code>. */
13     public boolean másGrande(Círculo otro) {
14         if(otro == null)
15             throw new NullPointerException("No hay con quien comparar.");
16         return radio > otro.radio;
17     }
18 }
```

# Temas

## 1 Definiciones

- Funciones
- Declarar y definir una función
- Métodos
- Parámetros formales y actuales

# Parámetros formales y actuales

- Para utilizar una función hay que *mandarla llamar*.
- Al momento de llamarla se indican los valores concretos que deben tomar sus parámetros.
- Los nombres de la variable que reciben los parámetros de la función son los *parámetros formales*.
- Cuando la función es mandada llamar y se indican los valores concretos con que será ejecutado el código, estos valores se convierten en los *parámetros actuales*.
- Cada vez que se manda llamar un función sus parámetros actuales cambian.

# Ejemplo

## Código: Usando una función

```
1 public class Tiempo {
2     // ... sigue de arriba
3     public static void main(String[] args) {
4         int segundos = minutosASegundos(15);
5         System.out.println("15 minutos son " + segundos + " segundos.");
6
7         int unosMinutos = 10;
8         int enSegundos = minutosASegundos(unosMinutos);
9         System.out.println("" + unosMinutos + " minutos son " +
10             segundos + " segundos.");
11     }
12 }
```

**Nota:** evaluación ávida (*greedy*) los argumentos se evalúan antes de pasar el valor.

# Ejemplo

## Código: Usando una función estática en otra clase

```
1 public class UsoTiempo {
2
3     public static void main(String[] args) {
4         int segundos = Tiempo.minutosASegundos(15);
5         System.out.println("15 minutos son " + segundos + " segundos.");
6
7         int unosMinutos = 10;
8         int enSegundos = Tiempo.minutosASegundos(unosMinutos);
9         System.out.println("" + unosMinutos + " minutos son " +
10                             segundos + " segundos.");
11     }
12 }
```



# Polimorfismo I: Sobrecarga

- 1 Definiciones
- 2 Polimorfismo I: Sobrecarga
- 3 Bibliografía

# Temas

## 2 Polimorfismo I: Sobrecarga

- Firma de un método
  - Sobrecarga

# Firma

- De cada función o *método*, nos interesa también la *firma*, que es sólo una parte de su **encabezado**.

## Sintaxis (Firma)

```
<firma> ::= <nombre>(<tipos>)  
<tipos> ::= <tipo> (, <tipo>)* | ∅
```

```
1 minutosASegundos(int)
```

# Sobrecarga

- Java nos permite definir varias funciones con el mismo **nombre**, dentro de la misma **clase**, siempre y cuando tengan una **firma** distinta.
- **OJO:** como el tipo de regreso no es parte de la firma, cambiar el tipo de regreso no cuenta.

# Ejemplo

```
1 public class Trivial {
2     public static void imprime(int n1) {
3         System.out.println("Un entero: " + n1);
4     }
5
6     public static void imprime(double n1) {
7         System.out.println("Un flotante de doble precisión: " + n1);
8     }
9 }
```

- Al utilizar un método *sobrecargado*, el compilador sabrá cuál llamar dependiendo de los tipos de los parámetros actuales.

```
1 public class Trivial {  
2     // ...sigue de antes  
3  
4     public static void main(String[] args) {  
5         imprime(9.0);  
6         imprime(9);  
7     }  
8 }
```

# Ejemplo con orientación a objetos

```
1 package polimorfismo;
2
3 import static java.lang.System.out;
4
5 public class Sobrecarga {
6     public int métodoSobrecargado(int param) {
7         return param * 2;
8     }
9
10    public String métodoSobrecargado(double x) {
11        return "Un_double_" + x;
12    }
13
14    public static void main(String[] args) {
15        Sobrecarga s = new Sobrecarga();
16        out.println("Sobrecarga_double:_" + s.métodoSobrecargado(Math.PI));
17        out.println("Sobrecarga_int:_" + s.métodoSobrecargado(89));
18    }
19 }
```

Esto se ejecuta así:

---

```
1 $ java polimorfismo.Sobrecarga
2 Sobrecarga double: Un double3.141592653589793
3 Sobrecarga int: 178
```


---



# Bibliografía

- 1 Definiciones
- 2 Polimorfismo I: Sobrecarga
- 3 Bibliografía**

# Bibliografía I

-  Viso, Elisa y Canek Peláez V. (2012). *Introducción a las ciencias de la computación con Java*. 2a. Temas de computación. Las prensas de ciencias. 571 págs. ISBN: 978-607-02-3345-6.

# Licencia

Creative Commons  
Atribución-No Comercial-Compartir Igual

