

Tema 1

Introducción

1. Programación orientada a objetos	3
1.1. Fundamentos de Java	3
1.2. Tipos de datos y operadores	5
1.3. Clases, métodos y funciones	6
1.3.1. Clases	6
1.3.2. Métodos	7
1.3.3. Funciones	8
1.3.4. La sentencia import	8
2. Estructuras de control	9
2.1. Estructuras de selección	9
2.1.1. If	10
2.1.2. Switch	11
2.2. Estructuras de repetición	11
2.2.1. While	11
2.2.2. Do-while	12
2.2.3. For	12
2.3. Estructuras de salto	12
2.3.1. Break y continue	12
2.3.2. Return	13
3. Programación orientada a objetos: conceptos avanzados	13
3.1. Clase Object	13
3.1.1. Clone()	13
3.1.2. Equals()	13
3.1.3. toString()	14

3.2. Miembros estáticos. Métodos de clase e instancia	14
3.3. Constructores	14
3.3.1. <i>Por defecto</i>	15
3.3.2. <i>Con parámetros</i>	15
3.3.3. <i>De copia</i>	15
3.4. Interfaces	15
3.5. Herencia	16
3.6. Sobrecarga y sobrescritura de métodos	16
3.7. Polimorfismo	17
4. Estructuras de datos	19
4.1. Array	19
4.2. Colecciones	20
4.2.1. <i>HashSet</i>	20
4.2.2. <i>ArrayList</i>	21
4.2.3. <i>HashMap</i>	21
5. Recursos bibliográficos y web	21

1. Programación orientada a objetos

La programación orientada a objetos es un paradigma de programación muy popular en la actualidad que emplea el diseño de objetos y su comportamiento para resolver problemas y crear programas y aplicaciones informáticas.

Bajo el manto de este paradigma se encuentran múltiples conceptos que dotan a los lenguajes que lo emplean de una flexibilidad y versatilidad muy interesantes para un desarrollador de aplicaciones: polimorfismo, encapsulamiento, herencia...

Para poder aprovechar al máximo todas estas utilidades para el lenguaje Java, se utilizará este tema introductorio como un repaso progresivo de todos los conceptos impartidos en el módulo de Programación de 1º de DAM.

1.1. Fundamentos de Java

Java es un lenguaje de programación particular, ya que utiliza una máquina virtual intermedia entre el lenguaje de programación que utiliza el programador humano y el lenguaje binario que emplea el ordenador. Es por esto por lo que Java es un lenguaje de programación excelente para el desarrollo multiplataforma, ya que este código intermedio estandariza las instrucciones para que pueda ejecutarse en cualquier equipo.

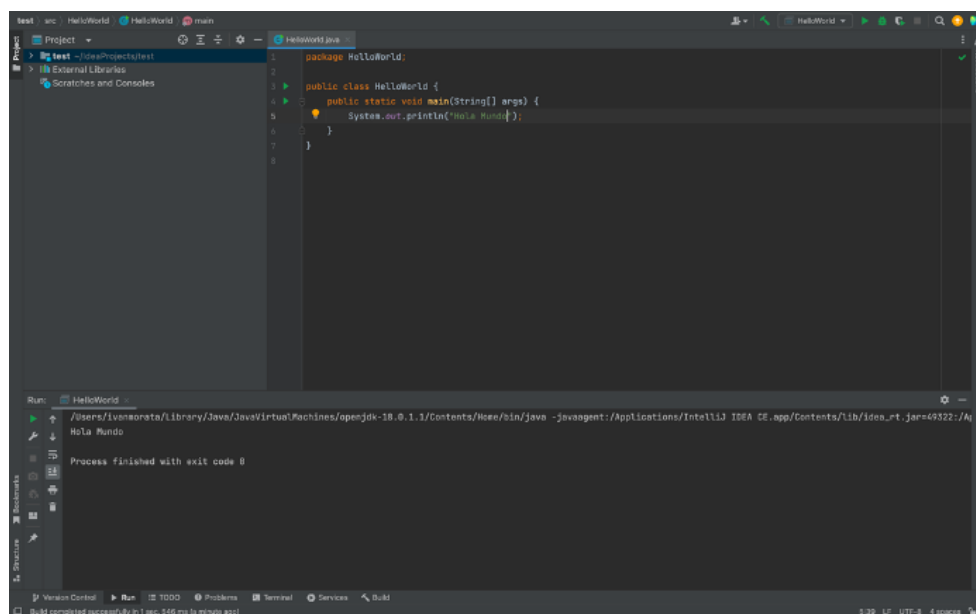
Para programar en Java, es necesario descargar e instalar el Kit de Desarrollo Java, o JDK. Este paquete es imprescindible, ya que contendrá la máquina virtual, compiladores, enlazadores, depuradores... Es habitual que los IDE soliciten el JDK durante el proceso de instalación.

Una vez instalado y acoplado el JDK en el IDE, podrán escribirse y ejecutarse programas. Todo programa en Java tiene un bloque principal, denominado main, que almacenará el núcleo de ejecución del programa.

```
public class HolaMundo {  
    /*programa holamundo*/  
    public static void main (String[] args){  
        //Lo único que hace este programa es mostrar "Hola mundo" por pantalla  
        System.out.println("Hola mundo");  
    }  
}
```

- Este código representa el programa HolaMundo.
- El conjunto `public static void main (String[] args)` es el bloque `main`:
 - Tiene visibilidad pública (`public`).
 - Es estático (`static`), por lo que solo puede haber una instancia de este bloque en ejecución.
 - No devuelve información ninguna al finalizar su ejecución (`void`).
 - Admite una lista de argumentos en formato de texto (`String[] args`).
- Presenta comentarios que no se tendrán en cuenta a la hora de ejecutar el programa.
 - Multilínea (`/* ... */`).
 - Línea singular (`//`).
- Manda a la salida estándar (la pantalla) el texto “Hola mundo” a través de la orden `System.out.println()`.
- Toda línea de código que no abra o cierre otro bloque debe terminar en `;` o dará error.

El resultado de ejecutar dicho bloque de código será similar al siguiente:



```
1 package HelloWorld;
2
3 public class HelloWorld {
4     public static void main(String[] args) {
5         System.out.println("Hola Mundo");
6     }
7 }
8
```

Run: HelloWorld

/Users/ivanorata/Library/Java/JavaVirtualMachines/openjdk-18.0.1.3/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA CE.app/Contents/lib/idea_rt.jar=49322:/...

Hola Mundo

Process finished with exit code 0

Build completed successfully in 1 sec, 546 ms (a minute ago)

1.2. Tipos de datos y operadores

Cuando es necesario almacenar un dato para su uso en un programa, se guardan en variables, que serán de diferentes tipos en función del tipo de dato. Estos son los tipos primitivos (básicos) de Java:

Tipo de datos	Información representada	Rango	Descripción
byte	Datos enteros	-128 +127	Se utilizan 8 bits (1 byte) para almacenar el dato.
short	Datos enteros	-32768 +32767	Dato de 16 bits de longitud (independiente de la plataforma).
int	Datos enteros	-2147483648 +2147483647	Dato de 32 bits de longitud (independiente de la plataforma).
long	Datos enteros	-9223372036854775808 +9223372036854775807	Dato de 64 bits de longitud (independiente de la plataforma).
char	Datos en coma y caracteres	0 65535	Este rango es para representar números en unicode, los ASCII se representan con valores 0 al 127.
float	Datos en coma flotante de 32 bits	Precisión aproximada de 7 dígitos	Dato en coma flotante de 32 bits en formato IEEE754 (1 bit signo, 8 exponente, 24 mantisa).
double	Datos en coma flotante de 64 bits	Precisión aproximada de 16 dígitos	Dato en coma flotante de 64 bits en formato IEEE754 (1 bit signo, 11 exponente, 52 mantisa).
boolean	Valores booleanos	True/false	Utilizado para evaluar si el resultado de una expresión booleana es verdadero o falso.

Para emplear un valor constante, puede utilizarse la palabra reservada `final` antes del tipo de dato durante la declaración de dato.

Con respecto a las operaciones básicas que permite Java, existen de diversos tipos: aritméticos (+, -, *, /, %), relacionales (<, >, <=, >=, ==, !=), lógicos (&&, ||, !) y otros tantos. A diferencia de otros lenguajes como C++, no se permite la sobrecarga de operadores, pero pueden utilizarse métodos y funciones como alternativa equivalente.

1.3. Clases, métodos y funciones

Todo código Java se escribe en ficheros .java, pero únicamente se ejecuta todo lo contenido en aquel fichero .java que contenga el bloque Main que hemos visto anteriormente. ¿Cómo, entonces, deberá organizarse el código para facilitar la lectura y correcta ejecución del código en el bloque main?

1.3.1. Clases

Cuando programamos, estamos trasladando un problema existente en el mundo real a una solución que un equipo informático pueda procesar apoyado en un lenguaje de programación. A este proceso se le denomina abstracción, y es la piedra angular del paradigma orientado a objetos.

Las abstracciones que realizamos sobre el mundo real las implementamos sobre clases, que son los distintos ficheros .java que podemos programar. Cada clase puede tener una serie de tipos de datos o atributos, que definirán las características de la clase; y una serie de métodos, que definirán el comportamiento de dicha clase.

Veámoslo con un ejemplo:

```
public class Empleado {  
    private String nombre;  
    private int edad;  
    private double sueldo;  
  
    public double cobrarExtra(double extra){  
        System.out.println("El empleado " + nombre + ", " + edad + " años,  
        cobra " + sueldo + " euros y una extra de ." + extra + "euros.");  
        return sueldo+extra;  
    }  
}
```

Esta es una abstracción enormemente simplificada de un empleado en un fichero Empleado.java. Vemos que las características que definen a todo empleado son una cadena de caracteres con el nombre, y la edad y el sueldo que cobran como un número entero y decimal, respectivamente. Además, el único comportamiento implementado es la acción de cobrar, codificada en el método cobrarExtra(), que notifica por salida estándar una breve descripción del empleado y devuelve la cantidad que cobrará.

Para utilizar el código de una clase desde el main, suele ser habitual declarar un objeto de esa clase. Esta declaración funciona de forma muy similar a la declaración de variables con tipos de datos primarios.

```
public class HolaEmpleado {  
    public static void main (String[] args){  
        Empleado eufrasio;  
        eufrasio = new Empleado();  
        System.out.println(eufrasio.cobrarExtra(499.99));  
    }  
}
```

Siempre es necesario inicializar todo objeto que se vaya a utilizar con la palabra clave new y una referencia a la clase. Este código llama al constructor de la clase para dotar de valores iniciales válidos a los atributos de la clase, y evitar incoherencias. Más adelante, analizaremos los diversos tipos de constructores que hay.

1.3.2. Métodos

Un método siempre se implementa de la siguiente manera:

visibilidad [*static*] *retorno* *nombre_método*(*[tipo_argumento nombre_argumento, ...]*)

- *Visibilidad* denota desde donde es posible llamar al método: public es el más permisivo, garantizando que el método se pueda llamar desde cualquier otro fichero .java que esté al alcance de la clase sobre la que está implementado. Por el contrario, private solo permite que se llame al método desde dentro de la propia clase.
- *static* es una palabra reservada que indica que el método implementado es de instancia, y no se le puede llamar desde un objeto de la clase, sino a partir de la propia clase.
- *Retorno* indica qué devolverá el método al ejecutarse, siempre a través de la palabra clave return. Si no se devuelve ningún tipo de dato, se indica con void.
- *nombre_método* es el identificador con el que se denotará el método.
- *tipo_argumento nombre_argumento* define el tipo de dato que el método espera recibir cuando se le llama desde cualquier punto del código. Se le puede pasar el tipo de dato de diversas formas: como un literal, a través de una constante, con una variable, o incluso con el retorno de otro método.

1.3.3. Funciones

Las funciones actúan de forma muy similar a los métodos. La principal diferencia entre ambos radica en la ubicación en la que se programan.

Un método siempre va asociado a una clase, y es un representa un comportamiento característico de la clase sobre la que se programa. Una función se programa sobre el método main, y su ámbito de ejecución está limitado al fichero main.java y todo lo que se utilice en el mismo.

Se definen de la misma forma que los métodos, aunque en este caso la palabra static no es opcional, ya que no se hará uso de los métodos a través de objetos, sino con una llamada directa.

```
public static void saluda(){
    System.out.println("Hello from the other side!");
}

public class HolaEmpleado {
    public static void main (String[] args){
        saluda();
    }
}
```

1.3.4. La sentencia import

Las clases en Java suelen estructurarse en paquetes. Cada paquete se organiza según una temática o serie de funcionalidades comunes, y queda al criterio del programador en cuantos paquetes dividir un proyecto (o siquiera si utilizar paquetes).

Sin embargo, esto supone que cada paquete es una unidad separada del resto, por lo que si queremos utilizar una clase de un paquete distinto al actual, será necesario incluirlo de alguna manera.

Para ello, se utiliza la sentencia import, que nos permite añadir tanto clases individuales de un paquete como todas que estén contenidas en el mismo:

```
import unPaquete.Empleado; //Se importa solo la clase Empleado del paquete unPaquete
import otroPaquete.*; //Se importan todas las clases presentes en el paquete otroPaquete
```


Cabe mencionar que Java incluye numerosos paquetes con una gran cantidad de funcionalidades variadas previamente implementadas en paquetes específicos. Es recomendable revisar la documentación y tomar nota de aquellos paquetes de uso más frecuente y/o más prácticos:

Librería	Descripción
<i>java.io</i>	Librería de Entrada/Salida. Permite la comunicación del programa con los periféricos.
<i>java.lang</i>	Paquete con clases esenciales de Java. No hace falta utilizar la sentencia import para utilizar sus clases.
<i>java.util</i>	Librería con clases de utilidad general para el programador.
<i>java.applet</i>	Librería para desarrollar applets.
<i>java.awt</i>	Librería con componentes para el desarrollo de interfaces de usuario.
<i>java.swing</i>	Librería complementaria a .awt con componentes de desarrollo de interfaces.
<i>java.net</i>	En combinación con .io, permite la comunicación del programa con internet.
<i>java.math</i>	Librería con todo tipo de utilidades matemáticas.
<i>java.sql</i>	Librería especializada en el manejo y comunicación con bases de datos.
<i>java.security</i>	Librería que implementa mecanismos de seguridad

2. Estructuras de control

Como es lógico, cuanto más complejo es un programa, más control sobre sus funcionalidades requiere el programador. Existen tres tipos de estructuras de control en Java: selección, repetición y salto.

2.1. Estructuras de selección

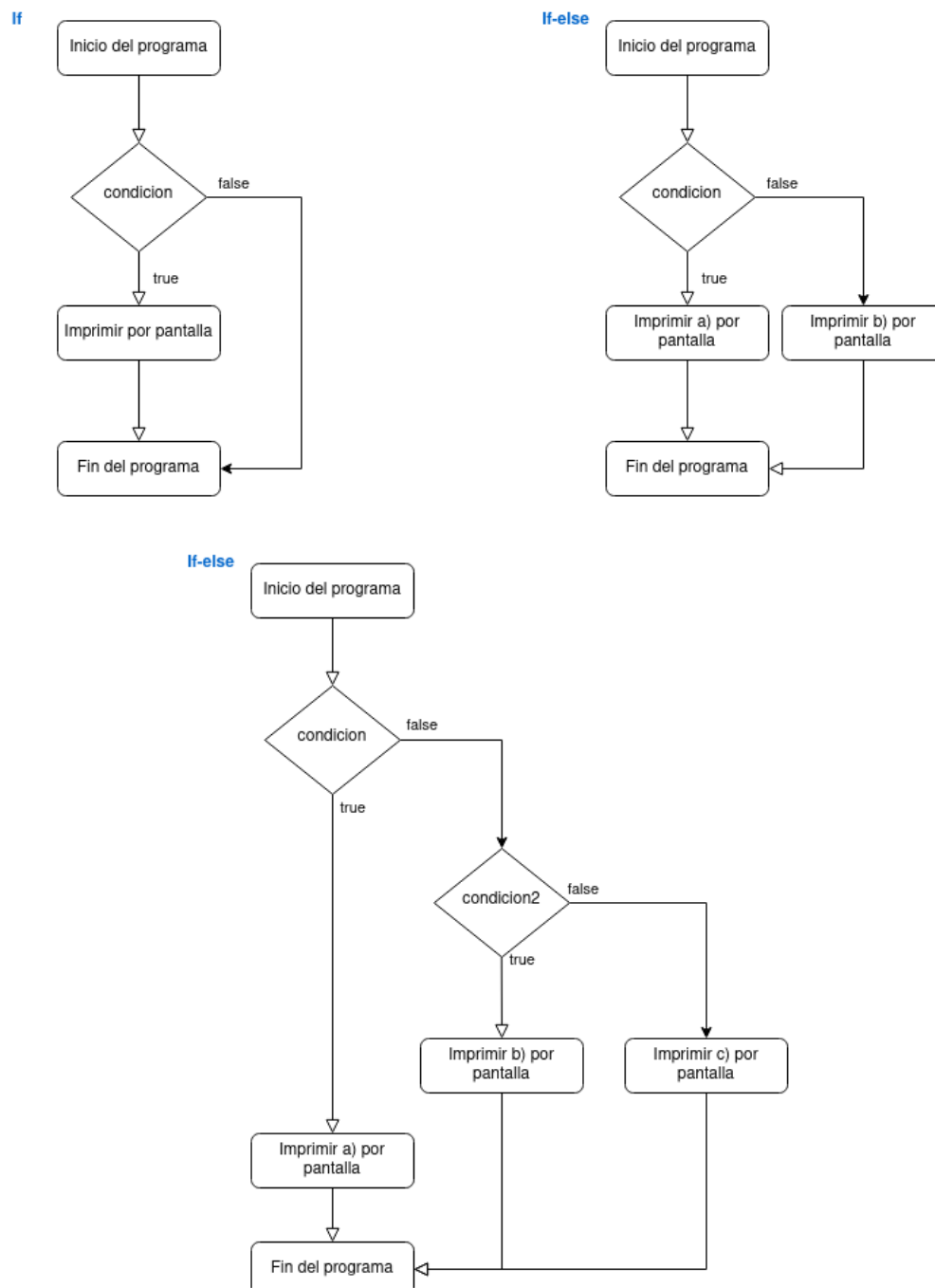
Las estructuras de selección condicionan la ejecución del fragmento de código que encapsulan al resultado de una operación lógica. En Java se utilizan dos estructuras de selección: if y sus variantes, y switch.

2.1.1. If

Existen tres variedades:

- **if simple:** se ejecuta el fragmento de código que encapsula si se cumple la condición entre paréntesis.
- **if-else:** se ejecuta el fragmento de código que encapsula si se cumple la condición entre paréntesis; en caso contrario, se ejecuta el código contenido en el bloque else.
- **if-elseif-else:** se encadena la operación if-else repetidas veces.

Se visualizan muy fácilmente a través de un diagrama de flujo:



2.1.2. Switch

La estructura switch es un “if organizado”. Nos permite ejecutar un fragmento de código según el valor de una determinada variable. Cada bloque de ejecución se divide en “cases”, separados por una sentencia break.

```
char opcion = 's';
switch(opcion){
    case 's':
        System.out.println("Opción elegida: sí");
        break;
    case 'n':
        System.out.println("Opción elegida: no");
        break;
}
```

2.2. Estructuras de repetición

Estas estructuras o bucles permiten ejecutar un bloque de código repetidas veces. La forma de ejercer la repetición y el número de veces que se ejecutará el código dependerán del tipo de bucle utilizado.

2.2.1. While

El bucle **while** se utiliza cuando se quiere ejecutar determinadas instrucciones cero o más veces. Su estructura es la siguiente:

```
int contador=0;
while(contador<10){
    System.out.println(contador);
    contador++;
}
```

2.2.2. Do-while

El bucle **do-while** se utiliza cuando se quiere ejecutar determinadas instrucciones una o más veces. Su estructura es la siguiente:

```
int contador=0;
do{
    contador++;
    System.out.println(contador);
}while(contador<10);
```

2.2.3. For

El bucle **for** se utiliza exclusivamente para cuando queremos repetir un fragmento de código un número conocido y concreto de veces. Su estructura es la siguiente:

```
int i;
for(i=1; i<=10; i++){
    System.out.println(i);
}
```

2.3. Estructuras de salto

El uso de estructuras de salto está mal considerado desde el punto de vista del paradigma Orientado a Objetos. No deben utilizarse salvo en circunstancias específicas.

2.3.1. Break y continue

La sentencia **break** interrumpe súbitamente el bucle en el que se ejecute, alterando el flujo normal de ejecución. Sólo se permite su uso para separar los distintos case de una estructura **switch**.

La sentencia **continue** se salta la iteración actual del bucle sobre el que se ejecuta. Su uso no se contempla bajo ninguna circunstancia permitida por el paradigma Orientado a Objetos.

2.3.2. Return

Esta es la única estructura de salto que está permitida dentro del paradigma de programación orientada a objetos. Esto es porque a diferencia de las anteriores, que obligan a interrumpir el flujo natural de un bucle y continuar más adelante, la sentencia **return** obliga a finalizar la ejecución del método o función.

3. Programación orientada a objetos: conceptos avanzados

3.1. Clase *Object*

Toda clase programada en Java hereda métodos y comportamientos de la clase fundamental del lenguaje: la clase *Object*. No es necesario conocer a fondo esta clase, pero sí es interesante analizar algunos de sus métodos más utilizados. De igual forma, es muy recomendable revisar la [documentación de la clase *Object*](#) para un análisis más detallado.

Es importante destacar que estos métodos funcionan exclusivamente a nivel superficial, lo que quiere decir que su efecto solo se aprecia a nivel de referencias en memoria. Para aplicarlos a nivel de contenido en memoria, será necesario sobrescribirlos.

3.1.1. Clone()

Este método toma el objeto recibido por parámetros y asocia al actual sus referencias. Devuelve las referencias al objeto actual una vez aplicada la asociación.

3.1.2. Equals()

Este método devuelve un valor booleano que comprueba si la referencia del objeto pasado como parámetro es la misma que la del objeto actual.

3.1.3. toString()

Este método devuelve un String con el nombre de la clase y la referencia hexadecimal del objeto actual. Habitualmente, se reescribe para que devuelva una descripción del objeto actual en formato String.

3.2. Miembros estáticos. Métodos de clase e instancia

Aquellos atributos o métodos con valor estáticos implementados en una clase son especiales, ya que no pueden accederse desde un objeto, sino referenciando a la propia clase.

Este comportamiento está orientado a aquellas características que deberían ser comunes y compartidas a toda la clase, independiente de los atributos individuales de cada objeto. Por ejemplo, un contador de operaciones en una clase Calculadora es un claro ejemplo de uso de esta funcionalidad.

Los métodos que poseen esta característica se denominan estáticos o de clase, y presentan siempre la palabra `static` en su definición. Todo método que no contenga dicha palabra clave es un método dinámico o de instancia.

Es posible llamar a métodos de clase y a atributos estáticos desde un método de instancia, pero a la inversa es imposible.

3.3. Constructores

Como se ha mencionado anteriormente, las clases Java suponen una abstracción de algún elemento en el mundo real. Por ello, para trabajar con estos elementos utilizamos objetos que suponen una instancia de dicha representación.

Es importante, pues, inicializar debidamente cada instancia de una clase. Para ello utilizaremos los constructores, que son bloques de código cuya única tarea es darle unos valores iniciales a los atributos de un objeto. Dependiendo de la forma que tomen esos valores, encontraremos tres tipos de constructores.

3.3.1. Por defecto

Un constructor por defecto es aquel que da valores a los atributos aún cuando el usuario o el programador no especifican ningún valor. Aunque Java incluye su propio constructor por defecto, lo ideal es implementarlo igualmente para asegurar que los valores por defecto están controlados y funcionan de acuerdo a lo que queremos.

El constructor por defecto nunca recibe parámetros de ningún tipo.

3.3.2. Con parámetros

Cuando recibimos una serie de valores concretos para inicializar el objeto, será necesario el constructor por parámetros. Los valores los recibimos en forma de parámetros, que pueden ser tantos como sea necesario, y pueden ser tanto tipos primitivos como otros objetos.

Es posible implementar tantos constructores por parámetros como sea necesario, siempre y cuando presenten listas de parámetros diferentes unos de otros.

3.3.3. De copia

Este constructor recibe un parámetro del mismo tipo del objeto que queremos inicializar. Su funcionamiento debería ser equivalente a la sobrescritura del método `clone()` de la clase `Object`.

3.4. Interfaces

A menudo, es deseable crear plantillas con una serie de comportamientos que serán comunes a diversas clases. Para ello, se utilizan las interfaces.

Se programan de forma muy similar a las clases, salvo por el hecho de que no deben llevar atributos y no deben implementarse sus métodos. Únicamente deben dejarse definidos para que cada clase realice su propia implementación.

Las interfaces son muy útiles a nivel descriptivo y de representación de la realidad. Por ejemplo, una interfaz Geometría podría contener la definición de los métodos `área()`, `perímetro()`, `volumen()`... que son comunes a todos los Polígonos que pudiéramos implementar, pero con formas de calcularse distintas.

3.5. Herencia

Hemos mencionado brevemente que existe la posibilidad de que haya clases que “hereden” de otra. En un lenguaje orientado a objetos, esto se entiende como una relación entre dos tipos de objetos A y B, en la que B es un tipo de A.

Por ejemplo, tendremos la clase Persona, con los atributos nombre y edad. Se entiende, pues, que toda Persona representable tiene un nombre y una edad. Un Empleado es un tipo de Persona, que heredando de esta clase, tiene por defecto incluidos sus atributos. Además, tendrá un atributo propio, un sueldo, que será específico de esta clase.

Para expresar una relación de herencia se utiliza la palabra reservada `extends` en la declaración de la clase:

```
class Persona{
    private String nombre;
    private int edad;
    //Constructores, getters y setters
}

class Empleado extends Persona{
    double sueldo;

    public String toString(){
        return this.getNombre()+" de "+this.getEdad()+" años de edad cobra "+sueldo;
    }
}
```

La herencia es una herramienta muy potente que permite agilizar, reducir y optimizar el código que escribimos. Además, ayuda a visualizar la representación de los objetos abstraídos del mundo real.

Cabe destacar que en Java no es posible que una clase herede de múltiples superclases, lo que se conoce en C++ y otros lenguajes como herencia múltiple.

3.6. Sobrecarga y sobrescritura de métodos

Al trabajar con clases y subclases, suele ser necesario dotar a un comportamiento de la superclase alguna opción adicional, o incluso modificar el propio comportamiento.

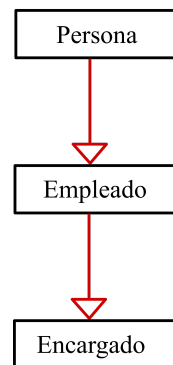
Hablamos de sobrecarga de un método cuando añadimos a la subclase una versión adicional del mismo método de la superclase pero con un número de parámetros y/o tipo de retorno distinto.

Hablamos de sobrescritura de métodos cuando la definición del método en la subclase es idéntico al de la superclase, y el comportamiento definido es distinto (en caso contrario, la redeclaración de un método resultaría redundante).

3.7. Polimorfismo

El polimorfismo es una característica inherente a Java por la existencia de la clase Object. Hablamos de que se produce polimorfismo cuando inicializamos una clase referenciando a otra con la que mantiene una relación de herencia. Veamos un ejemplo.

Imaginemos que tenemos este árbol jerárquico:



De la clase **Persona** desciende la clase Empleado. La clase Persona tendrá métodos genéricos que puedan ser utilizados por cualquier persona, como por ejemplo, setters y getters para el nombre y la edad (ya que todas las personas tienen nombre y edad).

La clase empleado tendrá otro tipo de métodos más específicos, como obtenerSueldo(), el cual devolverá el sueldo base, así como un setter para este atributo.

Los encargados son personas con más responsabilidad en la empresa, y sea cual sea su trabajo cobrarán un 10% más que un empleado normal.

La implementación de dicha jerarquía sería la siguiente:

```
public class Persona{
    private String nombre;
    public void setNombre(String nom){
        this.nombre=nom;
    }
    public String getNombre(){
        return nombre;
    }
}

public class Empleado extends Persona{
    protected int sueldoBase;
    public int getSueldo(){
        return sueldoBase;
    }
    public setSueldoBase(int s){
        this.sueldoBase = s;
    }
}

public class Encargado extends Empleado{
    public int getSueldo(){
        int bonus = sueldoBase*1.1;
        return bonus;
    }
}
```

Veamos un ejemplo de uso que demuestra cómo afecta el polimorfismo a la herencia:

```
public static void main(String [] args){
    Persona p1;
    p1 = new Empleado();
    p1.setNombre("Arthas Menethil");
    p1.setSueldoBase(100);
    Empleado e1;
    e1 = new Encargado();
    e1.setSueldoBase(500);
    e1.setPuesto("Jefe Almacén");
    System.out.println(e1.getSueldo());
}
```

Se puede ver que el objeto p1 pertenece a la clase Persona, pero se inicializa como Empleado. Esto significa que podrá hacer llamadas a los métodos de la clase Persona, pero no a los de Empleado.

Por otra parte, e1 pertenece a Empleado pero se inicializará como Encargado. Ésto modificará el resultado a la llamada a getSueldo(), ya que el método getSueldo pertenece a Empleado pero está reescrito en Encargado, de forma que Java interpreta que queremos utilizar la versión de Encargado.

4. Estructuras de datos

Es habitual querer almacenar múltiples datos en un programa, pero resultaría insostenible declarar una variable por cada dato según qué programa. Por ejemplo, si quisiéramos hacer la estadística de la temperatura de un determinado municipio a lo largo de un año, guardar la temperatura de cada día en 365 variables sería una propuesta irrisoria.

Para ello se utiliza un tipo especial de datos, que almacenan colecciones de valores simples. Conocemos al conjunto de estos tipos como estructuras de datos, y existen varias según el nivel de funcionalidad, detalle y forma de operar que necesitemos.

4.1. Array

Un **vector** o **array** se compone de una serie de posiciones consecutivas de memoria. A los arrays se accede mediante un subíndice. Si queremos acceder a la posición *n* del array *temperaturas*, tendremos que utilizar *temperaturas[n]*. Esta “*n*” puede ser una variable o un literal concreto.

Para **declarar** un array en Java, se puede hacer de dos formas distintas:

```
double [] temperaturas; //Forma nº 1
double temperaturas[]; //Forma nº 2
```

Inicialmente, los arrays se declaran sin un tamaño fijo, especificando solamente el tipo de dato que van a almacenar. Para poder utilizarlos, es necesario darles un tamaño, y para ello, se deben inicializar como cualquier otro objeto en Java.

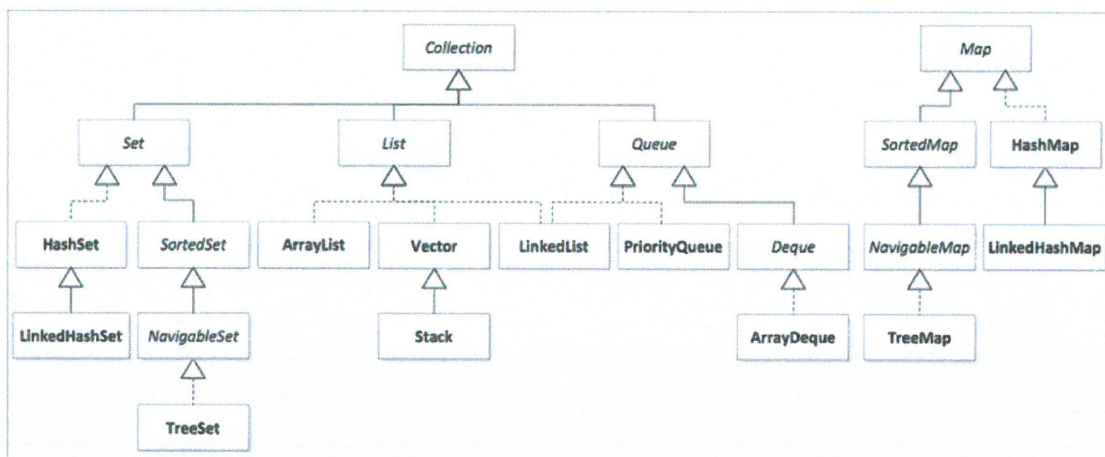
```
temperaturas = new double [100];
```

También es posible declarar un array de arrays, lo que se conoce como una matriz. Funcionan de forma idéntica al array clásico, con la salvedad de que se trabaja con dos índices de forma simultánea. Son muy útiles en la representación de cuadrículas y tableros.

Cabe destacar que actualmente, su uso en Java está bastante limitado y es por ello que se emplean otras estructuras más avanzadas, conocidas como estructuras dinámicas, para las cuales no es necesario reservar memoria y son mucho más flexibles en a la hora de insertar o borrar nuevos valores.

4.2. Colecciones

Java nos ofrece gran variedad de estructuras dinámicas agrupadas bajo la interfaz Colecciones. Esta interfaz se implementa a través del paquete `java.util`, y se usa en diversas otras clases y subinterfaces, aunando los comportamientos comunes a todas las estructuras dinámicas.



Veamos ahora algunas de las colecciones más utilizadas.

4.2.1. HashSet

El `HashSet` o conjunto equivale a la implementación Java del concepto matemático de Conjunto. Como tal, permite agrupar una serie de elementos de un tipo de dato concreto, sin incluir elementos repetidos, y sin ordenar los elementos. Además, tampoco es posible iterar de forma automática sobre los componentes del `HashSet`, es necesario utilizar un objeto de la clase `Iterator`.

4.2.2. ArrayList

Funcionalmente idéntica al array clásico, un ArrayList permite insertar y eliminar elementos de forma mucho más sencilla para el programador al coste de un incremento en el tiempo de computación.

Son la clase más extendida a la hora de operar con colecciones de valores, ya que son muy flexibles. Pueden almacenar tipos primitivos mediante sus respectivos wrappers, o incluso clases complejas diseñadas por un programador.

Aunque pueden recorrerse sobre un subíndice al igual que el array clásico, es posible utilizar un bucle for especializado que itera sobre cada elemento en la forma de un objeto concreto del mismo tipo del que almacena el ArrayList.

```
ArrayList<int> enteros = new ArrayList<>();
enteros.add(22);
enteros.add(77);
enteros.add(11);
for(int i : enteros){
    System.out.println(i);
}
```

4.2.3. HashMap

Equivalente al concepto de diccionario, un HashMap es una estructura de datos que almacena dos tipos de datos relacionados como tuplas. Si un HashMap contiene los tipos de datos A y B, A se considera la clave, y B el valor asociado a la clave.

La clase HashMap es interesante porque contiene métodos para operar tanto a través de las claves como los valores. Sin embargo, siempre que se realicen inserciones, deben incluirse las nuevas entradas en pares, y siempre que se elimina una clave, debe eliminarse su valor asociado.

5. Recursos bibliográficos y web

- *Programación para Ciclo Superior*, Juan Carlos Moreno, Editorial Ra-Ma (2011).
- *Fundamentos de Programación*, Mario Dorrego Martín, Editorial Síntesis (2019).
- *Introduction to Java*, W3School.
- *Oracle's Java Documentation*.