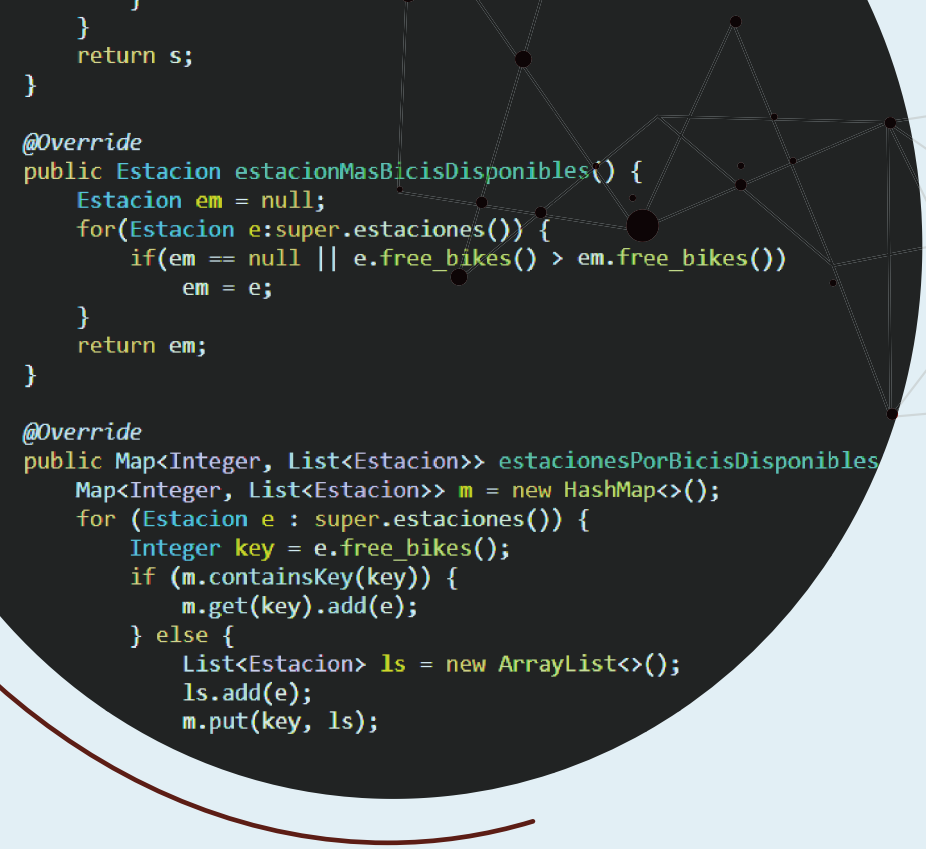


FUNDAMENTOS DE PROGRAMACIÓN: JAVA

Miguel Toro Bonilla

Editorial Universidad de Sevilla

Fundamentos de programación: JAVA

Fundamentos de programación: JAVA Miguel Toro Bonilla

Editorial Universidad de Sevilla

Colección: Manuales de Informática del
Instituto de Ingeniería Informática

Fundamentos de programación: JAVA Miguel Toro Bonilla

Miguel Toro Bonilla

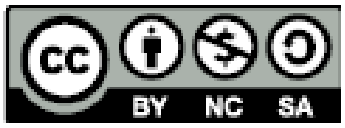
Fundamentos de programación: JAVA

Editorial Universidad de Sevilla

Sevilla 2022

Fundamentos de programación: JAVA Miguel Toro Bonilla

Colección: Manuales de Informática del
Instituto de Ingeniería Informática (I3US)
Núm.: 1



Editorial Universidad de Sevilla 2022
c/ Porvenir, 27 - 41013 Sevilla.
Tlfs.: 954 487 447; 954 487 451; Fax: 954 487 443
Correo electrónico: eus4@us.es
Web: https://editorial.us.es

DOI: https://dx.doi.org/10.12795/9788447223596

Maquetación: Miguel Toro

Diseño de cubierta y edición electrónica:
referencias.maquetacion@gmail.com

Agradecimientos

Para aprender a programar lo mejor es programar. En esta asignatura de Fundamentos de Programación vamos a aprender los conceptos básicos de la programación. Estos conceptos los vamos a concretar en dos lenguajes: Python y Java. En este volumen veremos Java.

Los lenguajes de programación tienden a ir compartiendo las mismas ideas básicas. Cada lenguaje va tomando prestadas las ideas más novedosas aportadas por otros. Aunque cada lenguaje tiene sus particularidades, podemos decir que los lenguajes de programación van convergiendo hacia elementos comunes a todos ellos. Por ello, parece sensato aprender las ideas comunes a los principales lenguajes actuales y extrapolar ideas y conceptos de unos a otros.

Vamos a abordar conceptos en Java comunes con Python que se pueden extender a otros.

En Java hay dos estilos de programación que queremos aprender: el estilo funcional usando streams y el más clásico estilo imperativo usando iterables.

El diseño de tipos ocupará un lugar central en este material. Al final se incluyen varios ejemplos que parten de un diseño de tipos.

El material en este texto procede de la experiencia de varios años de enseñanza de la asignatura de Fundamentos de Programación en la Universidad de Sevilla.

Mucho de este material está tomado de versiones anteriores de los profesores Mariano González, Fermín Cruz y Pepe Riquelme, a los que quiero agradecer sus esfuerzos y dedicación. Estas versiones anteriores han sido transformadas y actualizadas hasta alcanzar la forma actual. Sus defectos son responsabilidad única del autor.

En https://github.com/migueltoro/java_v1 puede encontrarse el código de los ejemplos. Futuras versiones aparecerán en https://github.com/migueltoro/java_v*

Miguel Toro

Sevilla, septiembre de 2021

Índice

Conceptos básicos de la P.O.O.

Los *Lenguajes Orientados a Objetos* están basados en la modelización mediante objetos. Estos objetos representan entidades del mundo real y tendrán las *propiedades* necesarias para resolver los problemas en que participen. Por ejemplo, una persona tiene un nombre, una fecha de nacimiento, unos estudios…; un vehículo tiene un dueño, una matrícula, una fecha de matrícula…; un círculo tiene un centro y un radio…

Para hacer programas sin errores es muy importante la facilidad de depuración del entorno o lenguaje con el que trabajemos. Después de hechos los programas deben ser mantenidos, lo que puede implicar la modificación o extensión de las funcionalidades de los objetos. La capacidad para *mantener* un programa está relacionada con el buen diseño de sus tipos, la encapsulación de los conceptos relevantes, la posibilidad de reutilización y la cantidad de software reutilizado y la comprensibilidad de este. Todas estas características son más fáciles de conseguir usando lenguajes orientados a objetos y en particular Java.

- *Modularidad*: es la característica por la cual un programa de ordenador está compuesto de partes separadas a las que llamamos módulos. La modularidad es una característica importante para la escalabilidad y comprensión de programas, además de ahorrar trabajo y tiempo en el desarrollo. En Java las unidades modulares son fundamentalmente las clases que se

pueden agregar, junto con las interfaces, en unidades modulares mayores que son los *paquetes*.

- *Encapsulación*: es la capacidad de ocultar los detalles de implementación. En concreto los detalles de implementación del estado de un objeto y los detalles de implementación del cuerpo de los métodos. Esta capacidad se consigue en Java mediante la separación entre interfaces y clases, y mediante la declaración de atributos privados en las clases, como veremos más adelante. Los clientes de un objeto, es decir, los otros objetos que utilizan a ese objeto interactúan con él a través del contrato ofrecido. El código de los métodos queda oculto a los clientes del objeto.
- *Reutilización*: una vez implementada una clase de objetos, puede ser usada por otros programadores ignorando detalles de implementación. Las técnicas de reutilización pueden ser de distintos tipos. Una de ellas es la herencia otra la delegación, que se verán más adelante.
- Facilidad de comprensión o legibilidad*:* es la facilidad para entender un programa. La legibilidad de un programa aumenta si está bien estructurado en módulos y se ha usado la encapsulación adecuadamente.
- *Cohesión*: se consigue cuando cada objeto tiene perfectamente definida su funcionalidad (y por tanto, sus responsabilidades dentro de la aplicación). Cuanto más precisa y atómica sea la funcionalidad de cada objeto y de cada método de cada objeto, más cohesiva es una solución. Esto redunda en mayor facilidad de depuración y extensibilidad.
- *Bajo acoplamiento*: el acoplamiento es la dependencia entre los objetos. Dos objetos están muy acoplados si hacer cambios en las propiedades o la funcionalidad de uno de ellos repercute en tener que cambiar muchos detalles del otro. Si existen muchas dependencias entre todos los tipos de objetos, se hace muy difícil realizar cualquier cambio a la aplicación. Es por tanto deseable conseguir un bajo acoplamiento, para lo cual se emplean técnicas como los patrones de diseño, que no estudiaremos en esta asignatura.

La POO (Programación Orientada a Objetos) es una forma de construir programas de ordenador donde las entidades principales son los objetos. Está basada en la forma que tenemos los humanos de concebir objetos, distinguir unos de otros mediante sus *propiedades* y asignarles *funciones* o capacidades. Estas dependerán de las propiedades que sean *relevantes* para el problema que se quiere resolver.

Los elementos básicos de la POO son:

- Objeto
- Interfaz
- Clase
    - *Atributos* (almacenan las propiedades)
    - *Métodos* (consultan o actualizan las propiedades)
- *Record*: Es un tipo específico de clase
- Paquete

Objetos

Los objetos tienen una identidad, unas propiedades, un estado y una funcionalidad asociada:

- Cada objeto tiene una *identidad* que lo hace único y lo distingue del resto de objetos del mismo tipo. Puede haber varios objetos con el mismo estado pero cada uno con su identidad propia; en ese caso se dice que los objetos son iguales, pero no idénticos.
- Las *propiedades* son las características observables de un objeto desde el exterior de este. Pueden ser de diversos tipos (números enteros, reales, textos, booleanos, etc.).
- El *estado* indica cuál es el valor de sus propiedades en un momento dado.
- La *funcionalidad* de un objeto se ofrece a través de un conjunto de *métodos*. Los métodos actúan sobre el estado del objeto (pueden consultar o modificar las propiedades) y son el mecanismo de comunicación del objeto con el exterior.

La *encapsulación* es un concepto clave en la POO y consiste en *ocultar* la forma en que se almacena la información que determina el estado del objeto. Esto conlleva la obligación de que toda la *interacción con* el objeto se haga a través de ciertos *métodos* implementados con ese propósito (se trata de ocultar información *irrelevante* para quien utiliza el objeto). Las propiedades de un objeto sólo serán accesibles para consulta o modificación a través de sus *métodos*. La encapsulación permite definir de forma estricta las responsabilidades de cada objeto, lo que facilita la depuración de las aplicaciones cuando se detectan errores.

Clases, registros e interfaces

Las *clases* son las unidades de la POO que permiten definir los detalles del *estado* interno de un objeto (mediante los *atributos*), obtener las propiedades de los objetos a partir de los atributos e implementar las funcionalidades ofrecidas por los objetos (a través de los métodos). Cada clase define un tipo nuevo con el cual podemos declarar variables de ese tipo y construir objetos.

Normalmente para implementar una clase partimos de un *diseño del tipo*. Al diseñar un tipo nuevo debemos partir de los ya existentes. Es necesario decidir a qué otros tipos *extender* o que tipos *usar*. Decimos que el nuevo tipo *usa* aquellos tipos con los que declara sus propiedades. También puede diseñarse el nuevo tipo extendiendo algunos de los disponibles. Un tipo *extiende* a otro cuando le añade nueva funcionalidad es decir nuevas propiedades y métodos.

Al diseñar un tipo estamos estableciendo un contrato entre los objetos de este tipo y sus posibles usuarios. Un tipo tiene un nombre y un conjunto de propiedades que son las características visibles. Cada propiedad tiene un nombre, un tipo, puede ser *consultada* y además modificada o sólo consultada, y puede ser una propiedad *básica* o una propiedad *derivada*. Además, las propiedades pueden ser *individuales* y *compartidas*. Las propiedades individuales son específicas de un objeto individual. Las propiedades compartidas son comunes a todos los objetos de la *población del tipo*. Las propiedades derivadas pueden ser calculadas a partir de las

otras propiedades. Las básicas no. Los tipos inmutables solo tienen propiedades modificables.

Un tipo suele tener asociado un *criterio de igualdad* entre dos de ellos y en muchos casos un *criterio de ordenación* al que llamaremos orden natural. Para un tipo también podemos definir su *representación*. Es decir la forma en la que se representará el objeto mediante una cadena de caracteres.

Un tipo, también, tiene *métodos de factoría*. Es decir mecanismos para crear objetos nuevos. Entre los métodos de factoría hay uno, que denominaremos *parse*, encargado de construir objetos a partir de una cadena de texto.

Un tipo puede ser diseñado para que sea *mutable* o *inmutable*. Un tipo inmutable no tiene operaciones para cambiar sus propiedades. Una vez creado el objeto no puede cambiarlas. Un tipo mutable sí. Nuestros tipos serán generalmente inmutables.

Un tipo nuevo en Java se implementa mediante una clase (class). Java nos ofrece una forma simple de definir tipos nuevos mediante tipo de especial de clase llamado record. En un primer momento usaremos *records* para definir tipos nuevos y clases para agregar un conjunto de funciones que definen una funcionalidad. En Java no podemos definir funciones fuera de una clase.

Las propiedades y la funcionalidad decidida en el diseño del tipo se concretan en unos atributos que especifican el estado de los objetos del tipo y en un conjunto de métodos de la clase que gestionan esos atributos. Cada método tiene una *signatura* que incluye su nombre, el tipo de los parámetros que recibe y el tipo que devuelve en su caso. Llamaremos a este tipo devuelto tipo de retorno. A las signaturas de un método las llamaremos también *cabecera* del método o *prototipo*. Además de la cabecera el método tiene un cuerpo formado por una secuencia de sentencias.

Un diseño preliminar del tipo Punto2D, un punto en el espacio de dos dimensiones es:

```
Punto2D
Propiedades
x: Double, básica
y: Double, básica
distanciaAlOrigen: Double, derivada, √(x²+y²)
distancia_a(p:Punto2D): Double, derivada, √((x−p.x)²+(−p.y)²)
cuadrante: Cuadrante, derivada
Representación
(2.3,4.5)
Orden Natural
Según su distancia al origen
Factoría
of(Double x,Double): Punto2D
parse(text:String):Punto2D
```

La implementación en Java es

```
recordPunto2D(Double x,Double y)implements
             Comparable<Punto2D>{

      public static Punto2D of(Double x, Double y) {

             return new Punto2D(x, y);
      }

      public Double distanciaAlOrigen() {
      Double dx = this.x;
             Double dy = this.y;
             return Math.sqrt(dx*dx+dy*dy);
      }
      @Override
      public String toString() {
           return String.format(
                   "(%.2f,%.2f)",this.x(),this.y());
      }
      @Override
      public int compareTo(Punto2D p) {
             return this.distanciaAlOrigen()
                        .compareTo(p.distanciaAlOrigen());
      }
}
```

Una interface es un elemento de la POO que permite, establecer propiedades y funcionalidades que debe tener un tipo dado. Una interface

contiene, fundamentalmente, *signaturas* de métodos que definen la funcionalidad pero no el cuerpo de estos. Al igual que una clase una interfaz define un tipo nuevo.

Una interface se usa para establecer funcionalidades comunes a varios tipos.

Como ejemplo veamos la interface *Comparable<E>* que define la funcionalidad de todos los tipos que tengan un orden natural.

```
interface Comparable<E> {
       int compareTo(E other);
}
```

Un tipo *E* que ofrezca esta funcionalidad asegura que si un objeto o1 de ese tipo se compara con otro o2 podremos saber si es mayor, menor o igual que este. Esto se concreta en la sentencia:

```
int r = o1.compareTo(o2);
```

La variable r tendrá un número negativo, positivo o cero según que o1 sea menor, mayor o igual a o2.

Si un tipo, implementado como *class o record*, ofrece la funcionalidad definida en una interface decimos que el tipo la implementa. Esto se concreta mediante *implements* y obliga a la *clase* o *record* a dar un cuerpo al o lo métodos contenidos en la interface.

```
record Punto2D(Double x,Double y) implements
       Comparable<Punto2D>
```

Las propiedades básicas del tipo, las vienen como parámetros de *record*, x, y en el caso del Punto2D definen el *estado del objeto*. El estado es privado e invisible fuera del objeto. Para consultar el estado disponemos de métodos. En el caso de Punto2D disponemos de los métodos *x(), y(), distanciaAlOrigen(), toString(), compareTo(Punto2D other).* Mediante el operador punto (.) podemos combinar objetos y métodos para formar expresiones bien formadas. Los métodos de factoría no se combinan con objetos sino con el nombre del tipo.

Siendo p1, p2 objetos del tipo Punto2D tenemos las expresiones siguientes. Unas bien bien formadas y otras no:

```
Punto2D p1 = Punto2D.of(3.,4.);
Double r = p1.x();
Double r2 = p1.x;   //Mal, x es invisible
p1.y = 3.4; //Mal, y es invisible
p1.y() = 5.6; //Mal, la llamada a un método no puede estar a
//     la izquierda
```

Las expresiones, cuando están bien formadas, tienen un tipo y un valor. El punto (.) es, por lo tanto, un operador que combina un objeto con sus métodos y que indica que el correspondiente método se ha invocado sobre el objeto. Como hemos dicho antes el operador punto (.) también combina el nombre de una clase o registro con un método de factoría.

Estructura de un programa en Java

Un programa en Java está formado por un conjunto de declaraciones de tipos, interfaces y clases. Un programa puede estar en dos modos distintos. En el *modo de compilación* está cuando estamos escribiendo las clases e interfaces. En este modo, a medida que vamos escribiendo, el entorno va detectado si las expresiones que escribimos están bien formadas. Si el entorno no detecta errores entonces diremos que el programa ha compilado bien y por lo tanto está listo para ser ejecutado. Se llama modo de compilación porque el encargado de detectar los errores en nuestros programas, además de preparar el programa para poder ser ejecutado, es otro programa que denominamos comúnmente *compilador*. En el *modo de ejecución* se está cuando queremos obtener los resultados de un programa que hemos escrito previamente. Decimos que ejecutamos el programa. Para poder ejecutar un programa este debe haber compilado con éxito previamente y tener un método especial denominado método principal (también denominado programa principal). En otro caso no lo podremos ejecutar.
En el modo de ejecución pueden aparecer nuevos errores. Los errores que pueden ser detectados en el modo de compilación y los que se detectan en el modo de ejecución son diferentes. Hablaremos de ellos más

adelante. Cuando queremos ejecutar un programa en Java éste empieza a funcionar en la clase concreta elegida de entre las que tengan un método de nombre *main*. Es decir, un programa Java empieza a ejecutarse por el método *main* de una clase seleccionada.
Las interfaces y clases diseñadas más arriba forman un programa. Necesitamos una que tenga un método *main*. En el ejemplo siguiente se presenta una que sirve para comprobar el buen funcionamiento del tipo Punto.

```
package test;
public class TestPunto {
	public static void main(String[] args) {
		Punto2D p = Punto2D.of(2.0, 3.0);
		System.out.println(String.format(
			"Punto: %s" + p));
	}
}
```

El programa empieza a ejecutarse en el método main de la clase seleccionada que en este caso es TestPunto. Esta ejecución sólo puede llevarse a cabo cuando han sido eliminados todos los posibles errores en tiempo de compilación y por lo tanto las expresiones están bien formadas.

En la línea

```
Punto2D p = Punto2D.of(2.0, 3.0);
```

Se declara p como un objeto de tipo *Punto2D* y se inicializa con un objeto nuevo en el estado (2.0, 3.0), mediante un método de factoría.

Las instrucciones *System.out.println(...)* permiten mostrar en pantalla las cadenas de texto que se pasan como parámetros entre los paréntesis (se verá más adelante en este tema). Mediante *String.format(formato,p1, p2,...)* se construye una cadena de caracteres con el formato dado a partir de los parámetros p1, p2, ...

Los resultados esperados en la consola son:

```
Punto: (2.0,3.0)
```

Elementos básicos del lenguaje

Identificadores

Son palabras que permiten referenciar los diversos elementos que constituyen el código. Es decir, sirven para nombrar a clases, interfaces, métodos, atributos, variables, parámetros y paquetes. Para nombrar a estos elementos, hay que seguir unas determinadas reglas para que puedan ser entendidos por el compilador. Los identificadores se construyen mediante una secuencia de letras, dígitos, o los símbolos _ y $. En cualquier caso, se debe observar que:

- No pueden coincidir con *palabras reservadas* de Java (ver más adelante)
- Deben comenzar por una letra, _ o $, aunque estos dos últimos no son aconsejables.
- Pueden tener cualquier longitud.
- Son sensibles a las mayúsculas, por ejemplo, el identificador min es distinto de MIN o de Min.

Ejemplos de identificadores válidos son los siguientes:

```
tiempo, distancia1, caso_A, PI, velocidad_de_la_luz
```

Por el contrario, los siguientes nombres no son válidos (¿por qué?)

```
1_valor, tiempo-total, dolares%, final
```

En general, es muy aconsejable elegir los nombres de los identificadores de forma que permitan conocer a simple vista qué representan, utilizando para ello tantos caracteres como sean necesarios. Esto simplifica enormemente la tarea de programación y –sobre todo– de corrección y mantenimiento de los programas. Es cierto que los nombres largos son más laboriosos de teclear, pero, en general, resulta rentable tomarse esa pequeña molestia. Unas reglas aconsejables para los identificadores son las siguientes:

- Las variables normalmente tendrán nombres de sustantivos y se escribirán con minúsculas, salvo cuando estén formadas por dos o más palabras, en cuyo caso, el primer carácter de cada palabra se escribirá en mayúscula. Por ejemplo: *salario, salarioBase, edadJubilacion.*
- Los identificadores de constantes (datos que no van a cambiar durante la ejecución del programa) se deben escribir con todos los caracteres con mayúsculas; si el identificador está compuesto por varias palabras es aconsejable separarlas con el guion bajo (_). Por ejemplo*: PI, PRIMER_VALOR, EDAD_MINIMA*.
- Los identificadores de métodos tendrán la primera letra en minúscula, y la primera letra de las siguientes palabras en mayúscula: *x, distanciaAlOrigen*, etc.
- Los identificadores de clases e interfaces se deben escribir con el primer carácter de cada palabra en mayúsculas y el resto en minúsculas: *Punto2D, Comparable*, etc.

Palabras reservadas de Java

Una palabra reservada es una palabra que tiene un significado especial para el compilador de un lenguaje, y, por lo tanto, no puede ser utilizada como identificador. Algunos ejemplos de palabras reservadas son *main, int, return, if o for*.

Literales

Son elementos del lenguaje que permiten representar valores constantes de los distintos tipos del lenguaje. Por ejemplo, 23 es un literal de tipo *Integer*, 23L es un literal de tipo Long, 3.14 es un literal de tipo *Double*, "Guadalquivir" es un literal de tipo *String* y *true* y *false* son los dos literales que corresponden al tipo *Boolean*. Para los objetos existe un literal *null* que representa un objeto sin valor.

Comentarios

Los comentarios son un tipo especial de separadores que sirven para explicar o aclarar algunas sentencias del código, por parte del programador, y ayudar a su prueba y mantenimiento. De esta forma, se intenta que el código pueda ser entendido por una persona diferente o por el propio programador algún tiempo después. Los comentarios son ignorados por el compilador.

En Java existen comentarios de línea, que se marcan con //, y bloques de comentarios, que comienzan con /* y terminan con */.

```
// Este es un comentario de una línea
/* Este es un bloque de comentario
   que ocupa varias líneas
*/
```

Tipos de datos

Tipos proporcionados por Java

Java proporciona un conjunto de tipos ya implementados. Algunos de ellos junto con sus valores son:

- *Integer*: Sus valores son enteros
- *Long*: Sus valores son enteros más largos que los anteriores
- *Boolean*: Sus valores son true y false
- *Float*: Sus valores son números reales
- *Double*: Sus valores son números reales con mayor precisión
- *Character*: Sus valores son los caracteres disponibles como 'a'.
- *Void*: No tiene ningún valor
- *String*: Sus valores son secuencias de caracteres, como por ejemplo "Hola".

Todos los tipos anteriores son inmutables, tienen un orden natural y métodos de factoría adecuados. En general a estos tipos que comienzan con mayúsculas los denominamos *tipos objeto*.

Junto a los tipos anteriores en Java también se pueden definir tipos nuevos mediante la cláusula *enum*. Un tipo enumerado puede tomar un conjunto determinado de valores, que se enumeran de forma explícita en su declaración. Por ejemplo, en Java el tipo *Color* podemos definirlo como un enumerado con seis valores: rojo, naranja, amarillo, verde, azul y violeta.

```
public enum Color {
       ROJO, NARANJA, AMARILLO, VERDE, AZUL, VIOLETA
}
```

Junto a los tipos anteriores Java también ofrece los *tipos de datos básicos, nativos o primitivos* de Java son:

- *int, long*: Sus valores son los de Integer y Long
- *float, double*: Sus valores son los de Float y Double
- *boolean*: Sus valores *true* y *false*
- *char*: Sus valores son los de Character
- *void*: No tiene valores

Los tipos primitivos se convierten automáticamente a sus correspondientes tipos objeto y viceversa.

```
int a= 34;
Integer b = a;
int d = b;
```

Los tipos primitivos tienen una funcionalidad más limitada que sus equivalentes anteriores.

Como regla general recomendamos el uso de los tipos que comienzan con mayúscula.

Junto a los tipos ofrecidos directamente por Java podemos incorporar, mediante librería, otros tipos usados en los cálculos matemáticos con *Fraction* o *Complex*. Los valores de estos tipos son objetos que se construyen mediante un constructor o un método de factoría. Ejemplos

```
Fraction f1 = Fraction.getReducedFraction(6, 8);
Fraction f2 = Fraction.getReducedFraction(1, 4);
Fraction f3 = f1.subtract(f2);
Fraction f4 = f1.multiply(f2);

Complex c = Complex.valueOf(6.2,7.1);
Double re = r.getReal();
Double md = r.abs();
```

La información sobre estos últimos tipos se pue encontrar en Fraction, Complex.

Variables y constantes

Variables

Las *variables* son elementos del lenguaje que permiten guardar y acceder a los datos que se manejan. En Java es necesario declararlas antes de usarlas en cualquier parte del código y, por convenio, se escriben en minúsculas. Mediante la declaración indicamos que la variable guardará un valor del tipo declarado. Mediante la inicialización reciben un primer valor.

Algunos ejemplos de declaraciones de variables son los siguientes:

```
Integer valor;
Double a1 = 2.25, a2 = 7.0;
Character c = 'T';
String cadena= "Curso Java";
Color relleno = Color.AZUL;
```

En la primera línea del ejemplo se declara una variable, *valor*, de tipo entero; en la segunda línea se declaran e inicializan dos variables, *a1* y *a2*, de tipo *Double*; en la tercera, se declara una variable, *c*, de tipo *Character* y se inicializa; en la cuarta, se declara e inicializa la variable *cadena* de tipo *String*. Finalmente, en la última línea se declara la variable *relleno* del tipo enumerado *Color* y se inicializa con el valor AZUL.

Cada declaración tiene un *ámbito*. Por *ámbito de una declaración* entendemos el segmento de programa donde esta declaración tiene

validez. Más adelante iremos viendo los ámbitos asociados a cada declaración. En la mayoría de los casos, en Java un ámbito está delimitado por los símbolos {…} (llaves).

Constantes

Las *constantes* son elementos del lenguaje que permiten guardar y referenciar datos que van a permanecer invariables durante la ejecución del código. La declaración de una constante comienza por la palabra reservada *final*.

Ejemplos de declaraciones de constantes:

```
final Integer DIAS_SEMANA = 7;
final Double PI = 3.1415926;
final String TITULO = "E.T.S. de Ingeniería Informática";
```

En este ejemplo se declaran tres constantes, DIAS_SEMANA, PI, y TITULO, de tipo Integer, Double y String, respectivamente. Note que, por convención, los nombres de constantes se escriben en mayúsculas.

Expresiones y operadores

Expresiones

Una *expresión* se forma con identificadores, valores constantes, operadores y llamadas a métodos. Toda *expresión bien formada* tiene asociado un valor y un tipo. Una variable dentro de una expresión decimos que está siendo *usada*. Para que una variable pueda ser usada tiene que haber sido declarada e inicializada previamente.

Por ejemplo:

```
edad >= 30
(2 + peso) / 3
Color.ROJO
```

La primera línea muestra una expresión de tipo *Boolean*. La segunda, una expresión de tipo de tipo *Double.* Y la tercera, una de tipo *Color*.

Mediante una *asignación* (representada por **=**) podemos dar nuevos valores a una variable. La asignación es un operador que da el valor de la expresión de la derecha a la variable que tiene a la izquierda. Cuando una variable está a la izquierda de una asignación decimos que ha sido *definida* y cuando está en una expresión a la derecha de la asignación ha sido *usada*. Por ejemplo:

```
Double precio = 3.0;
precio = 4.5 * peso + 34;
```

Si la variable peso, almacena el valor 2. La expresión *precio = 4.5 * peso + 34;* tiene como valor 43, y como tipo *Double*.

Una declaración puede ser combinada con una inicialización. Como por ejemplo:

```
Integer edad = 30;
Double peso = 46.5;
String s1 = "Hola ", s2 = "Anterior";
Color c = Color.VERDE;
```

Operadores y conversiones de tipos

Los operadores más habituales en Java son los siguientes:

- Operadores aritméticos: + (suma), - (resta), * (producto), / (división) y % (módulo)
- Operadores lógicos: && (and), || (or) y ! (not)
- Operadores relacionales: > (mayor que), < (menor que), >= (mayor o igual que), <= (menor o igual que), == (igual que), != (distinto de)
- Operadores de asignación: = y los abreviados: +=, -=, etc; ++ (incremento) y -- (decremento)

El operador + usado entre cadenas es concatenar.

El operador punto (.) separa un objeto (o una clase o interface) con un método

El operador [] sirva para indexar una casilla en un array.

Los tipos numéricos se pueden ordenar *Integer, Long, Float, Double*. Un valor de uno de esos tipos se convierte automáticamente en otro posterior en el orden anterior. Si se operan dos valores de tipos diferentes se convierten ambos al mayor y se operan. El tipo devuelto es el mayor.

Para convertir un tipo en otro menor hay que forzarlo con operadores de casting. Estos son: *(int)* convierte a entero, *(long)* convierte a long, *(double)* convierte a double. A su vez los tipos *Integer, Long, Float, Double*

disponen de los métodos *intValue(), longValue(), floatValue(), doubleValue()* para convertir a los tipos respectivos.

Precedencia y asociatividad de los operadores

El resultado de una expresión depende del orden en que se ejecutan las operaciones. Por ejemplo, si queremos calcular el valor de la expresión 3 + 4 * 2, podemos tener distintos resultados dependiendo de qué operación se ejecuta primero. Así, si se realiza primero la suma (3 + 4) y después el producto (7 * 2), el resultado es 14; mientras que si se realiza primero el producto (4 * 2) y luego la suma (3 + 8), el resultado es 11.

Para saber en qué orden se realizan las operaciones es necesario definir unas reglas de precedencia y asociatividad. La tabla siguiente resume las reglas de precedencia y asociatividad de los principales operadores en el lenguaje Java.

Operador	Asociatividad
. [] ()	
+ - ! ++ -- (tipo) new	derecha a izquierda
* / %	izquierda a derecha
+ –	izquierda a derecha
< <= > >=	izquierda a derecha
== !=	izquierda a derecha
&&	izquierda a derecha
||	izquierda a derecha
? :	
= += –= *= /= %=	derecha a izquierda

Los operadores que están en la misma línea tienen la misma precedencia, y las filas están en orden de precedencia decreciente. También se debe tener en cuenta que el uso de paréntesis modifica el orden de aplicación de los operadores. Si no hay paréntesis el orden de aplicación de los operadores viene definido por la precedencia y la asociatividad de estos definida arriba.

El tipo String, tipos para el manejo de fechas

Tipo String

Como hemos visto antes, el tipo String representa una secuencia o cadena de caracteres. El tamaño de un String es inmutable y, por lo tanto, no cambia una vez creado el objeto. Se representa por el método *length*.

Cada carácter de la cadena ocupa una posición, comenzando por la posición 0 y terminando por la posición *length()-1*. Para acceder al carácter que ocupa una posición dada se utiliza el método *charAt(int i)*. Por ejemplo;

```
String nombre = "Amaia";
Integer lon = nombre.length() // El valor de lon será 5
Character inicial = nombre.charAt(0) // El valor de inicial
       //será 'A'
Character ultima = nombre.charAt(lon - 1) // El valor de
       //ultima será 'a'
```

No es posible modificar un carácter de la cadena una vez que se ha creado. Tampoco se pueden añadir o eliminar caracteres. El tipo *String* es, pues, inmutable.

El tipo *String* ofrece otros métodos para, entre otras cosas, decidir si la cadena contiene una secuencia dada de caracteres, buscar la primera posición de un carácter u obtener la subcadena dada por dos posiciones. Los detalles de estos métodos pueden verse en la documentación de la clase *String* en la API de Java.

El tipo String es muy importante porque todos los datos que leemos de un fichero o de la consola son de tipos String. Tenemos que comprender bien como transformamos cadenas de texto a un tipo dado y viceversa. A la operación de transformar una cadena de texto en un tipo lo llamamos *parse*. Los tipos que nos ofrece Java ya incorporar un método de factoría para hacer este cometido. Los tipos que diseñemos nuevos es conveniente dotarlos de un método de factoría que lleve a cabo ese cometido.

```
Integer n1 = Integer.parseInt("-345");
Long n2 = Long.parseLong(("-345");
Double r = Double.parseDouble("-34.56");
Punto2D p = Punto2D.parse(3.7,-6.7);
```

El proceso inverso de parse en la transformación de un objeto de un tipo en una cadena de caracteres. Esto se consigue con el método **toString()**. Cada tipo tiene este método.

También es interesante comprender como conseguir una cadena de caracteres a partir de uno o varios objetos combinados con un texto. Esto se consigue con el método **format** del tipo String. Este método tiene la cabecera:

```
String String.format(String format, Object... args)
```

El parámetro *args* es una secuencia de parámetros de cualquier tipo. El parámetro *format* es una cadena de texto que incluye especificadores de formato. Esto especificadores tienen la estructura:

```
%[argument_index$][width][.precision]conversion
```

Donde

- *argument_index* (opcional) es la posición del parámetro en la lista de parámetros comenzando en 1. Si falta se va aplicando el especificador a cada parámetro consecutivamente
- El parámetro *width* (opcional) es la longitude mínima de la cadena de salida
- El parámetro *precision* (opcional) es un número entero que se usa normalmente para indicar el número de decimales de un número real.

- El paramero conversion indica el tipo del parámetro que va a ser formateado. Los más usuales son: c (carácter), d (entero), f (número real), s (cadena de caracteres), % (produce % en la salida). Hay especificadores para fechas que veremos más adelante.

Ejemplos

```
String.format("r = %1$5.2f, porcentaje = %2$d%%",r,30L);
String.format("(%3d,%30s,%2d,%2d,%2d,%s)",numero,name,slots,em
pty_slots,f,c);
```

Tipos para el manejo de fechas y horas

Java dispone de un conjunto de tipos para trabajar con fechas, horas, instantes temporales y duraciones, incluidos en el paquete java.time. Algunos de estos tipos son los siguientes:

- *LocalDateTime*, tipo inmutable para representar fechas y horas. Por ejemplo, 03-12-2015 10:34
- *LocalDate*, tipo inmutable para representar fechas (sin zona horaria), es decir, solo con día, mes y año. Por ejemplo, 03-12-2015
- *LocalTime*, tipo inmutable para representar horas sin fecha (ni zona horaria), solo con hora, minutos, segundos y nanosegundos (si son necesarios). Por ejemplo, 10:10:12.99.
- *Duration*, tipo inmutable que representa una cantidad temporal que se mueve en el rango de nanosegundos, segundos, minutos, horas o días.
- *Period*, tipo inmutable que representa una cantidad temporal que se mueve en el rango de años, meses o días.

Estos tipos tienen métodos de factoría adecuados para construir objetos. Veamos algunas operaciones habituales con estos tipos. Comencemos por la creación de objetos de tipo fecha y hora:

```
LocalDate fecha1 = LocalDate.of(2014, Month.MAY, 23);
LocalDate fecha2 = LocalDate.of(2014, 5, 23);
LocalTime hora1 = LocalTime.of(11, 00);
LocalTime hora2 = LocalTime.of(10, 59, 59);
```

Además del método of, se pueden crear objetos con la fecha y hora del sistema en el momento en que se invoca al método. Por ejemplo

```
LocalDate fecha3 = LocalDate.now();
LocalTime hora3 = LocalTime.now();
```

Dada una fecha o una hora, podemos acceder a una parte de ella, como por ejemplo el día de una fecha o los minutos de una hora:

```
LocalDate hoy = LocalDateTime.now();
Integer dia = hoy.getDayOfMonth();
DayOfWeek diaSemana = hoy.getDayOfWeek();
LocalDate ahora = LocalDateTime.now();
Integer minutos = ahora.getMinute();
```

También podemos sumar o restar un periodo de tiempo a una fecha u hora:

```
LocalDate f1 = LocalDate.of(2008, Month.FEBRUARY, 29);
LocalDate f2 = f1.plusYears(1);
System.out.println("Un año después..." + f2);
```

Otra operación habitual es obtener el tiempo transcurrido entre dos fechas:

```
LocalDate fechaNacimiento = LocalDate.of(2013, 12, 3);
LocalDate hoy = LocalDate.now();
Period p = fechaNacimiento.until(hoy);
Period p = Period.between (fechaNacimiento, hoy); // También
así
System.out.println ("El periodo entre las dos fechas es " +
p);
System.out.println ("Los años del periodo son " +
p.getYears());
System.out.println ("Los meses del periodo son " +
p.getMonths());
System.out.println ("Los días de periodo son " + p.getDays());
```

De manera similar para las horas, pero usando el tipo *Duration*:

```
LocalTime hora1 = LocalTime.of(15, 30);
LocalTime hora2 = LocalTime.of(15, 45);
Duration d = Duration.between(hora1, hora2);
System.out.println("Duración - " + d);
System.out.println ("Los segundos de la duración son " +
d.getSeconds());
```

Para comparar dos fechas u horas, utilizamos los métodos *isBefore, isAfter e isEqual:*

```
LocalDate f1 = LocalDate.of (2016, Month.SEPTEMBER, 19);
LocalDate f2 = LocalDate.of (2017, Month.JANUARY, 13);
System.out.println("¿Es f1 anterior a f2? " +
f1.isBefore(f2));
System.out.println("¿Es f1 posterior a f2? " +
f1.isAfter(f2));
System.out.println("¿Es f1 igual a f2? " + f1.isEqual(f2));
```

Finalmente, cuando mostramos fechas y horas en la pantalla, queremos que aparezcan en un formato concreto. Los siguientes ejemplos muestran varias formas de hacerlo:

```
LocalDate fecha = LocalDate.of(2016, 12,3);
String fechaFormateada =
fecha.format(DateTimeFormatter.ofPattern("dd-MM-yyyy"));
System.out.println("Fecha: " + fechaFormateada);
LocalTime hora = LocalTime.now();
String horaFormateada =

hora.format(DateTimeFormatter.ofPattern("HH:mm:ss:n"));
System.out.println("Hora: " + horaFormateada);
```

La salida que se produce al ejecutar el código anterior es la siguiente:

```
Fecha: 03-12-2016
Hora: 19:00:19:56000000
```

Sentencias de control selectivas

Hay dos tipos de sentencias de control: las bifurcaciones y los bucles. Las bifurcaciones permiten ejecutar un bloque de sentencias u otro, pero no ambos a la vez. En este tipo de sentencias de control se encuadran las sentencias *if* y *switch*.

Sentencia if-else

La sentencia if evalúa una expresión lógica (o condición) y, según sea cierta o falsa, ejecuta un bloque de sentencias u otro.

```
if (n % 2 == 0) {
	System.out.println("El número es par ");
} else {
	System.out.println("El número es impar");
}
```

Las sentencias *if-else* pueden encadenarse haciendo que se evalúe un conjunto de condiciones y se ejecute una sola de varias opciones.

```
Float impuesto = 0.0;
if (salario >= 5000.0) {
       impuesto = 20.0;
} else if (salario < 5000.0 && salario >= 2500.0) {
       impuesto = 15.0;
} else if (salario < 2500.0 && salario >= 1500.0) {
       impuesto = 10.0;
} else if (salario > 800.0) {
       impuesto = 5.0;
}
```

Sentencia switch

Normalmente *switch* se utiliza cuando se requiere comparar una variable con una serie de valores diferentes. Hay dos versiones de *switch*: como sentencia o como expresión. En ambos casos se indican los posibles valores que puede tomar la variable y las sentencias que se tienen que ejecutar si la variable coincide con alguno de dichos valores o las expresiones que calculan el valor resultante. Es una sentencia muy indicada para comparar una variable de un tipo enumerado con cada uno de sus posibles valores.

La expresión tiene que ser de uno de los siguientes tipos: *Character*, *Integer, String* y los tipos enumerados (*enum*).

Hay dos formas de *switch:* como elemento de una expresión:

```
Character s =
switch(r) {
   case "Lunes" -> r.charAt(0);
   case "Martes" -> r.charAt(1);
   default -> throw new IllegalArgumentException("Unexpected
       value: " + r);
};
```

Y como sentencia:

```
Character s;
switch(r) {
 case "Lunes": s = r.charAt(0); break;
 case "Martes": s = r.charAt(1); break;
 default -> throw new IllegalArgumentException("Unexpected
       value: " + r);
}
```

Existe también posibilidad de ejecutar el mismo bloque de sentencias para varios valores del resultado de expresión. Se haría de la siguiente forma:

```
switch (dia) {
       case 1: case 2: case 3: case 4: case 5: case 10: case
              12:
              System.out.println("Día laborable");break;
       case 6: case 7:
              System.out.println("Fin de semana");break;
}
```


Agregados de datos

Listas

Las listas representan colecciones de elementos de un mismo tipo en los que importa cuál es el primero, el segundo, etc. Cada elemento está referenciado mediante un índice; el índice del primer elemento es el 0. Las listas pueden contener elementos duplicados.

En Java, las listas se representan mediante el tipo *List*. Para crear una lista se invoca a un constructor del tipo *List*, indicando el tipo de los elementos que va a contener la lista. Por ejemplo, la sentencia

```
List<Double> temperaturas = new ArrayList<>();
```

crea una lista de números reales. Inicialmente la lista está vacía, y podemos añadir elementos de la siguiente forma:

```
temperaturas.add(27.5);
temperaturas.add(22.0);
temperaturas.add(25.3);
```

Al añadir elementos con el método *add*, se van colocando al final de la lista. Por tanto, en el caso anterior tendríamos la lista formada por los elementos [27.5, 22.0, 25.3]. Alternativamente podemos crear una lista con esos mismos elementos de la forma:

```
List<Double> temperaturas = Arrays.asList(27.5, 22.0, 25.3);
```

Para acceder a un elemento de la lista hemos de utilizar el método *get*, indicando la posición del elemento al que queremos acceder. Por ejemplo, para obtener la primera temperatura de la lista haríamos lo siguiente:

```
t1 = temperaturas.get(0);
```

Es importante que el índice esté dentro del rango de valores válidos, que van desde 0 hasta uno menos que el tamaño de la lista. Podemos conocer este tamaño mediante el método *size*:

```
Integer numeroElementos = temperaturas.size();
```

que en el caso del ejemplo asignaría a la variable *numeroElementos* el valor 3.

Conjuntos

El tipo *Set* de Java se corresponde con el concepto matemático de conjunto: un agregado de elementos en el que no hay orden (no se puede decir cuál es el primero, el segundo, el tercero, etc.) y donde no puede haber elementos repetidos.

La siguiente sentencia

```
Set<Character> letras = new HashSet<>();
```

crea un conjunto formado por caracteres, inicialmente vacío. A continuación podemos añadir elementos (en este caso caracteres) mediante el método *add*:

```
letras.add('A');
letras.add('V');
letras.add('E');
```

Podemos obtener el número de elementos de un conjunto con *size*, como en las listas. Alternativamente podemos crear un conjunto inmutable con esos mismos elementos de la forma:

```
Set<Character> letras = Set.of('A', 'V', 'E');
```

También podemos saber si un determinado elemento se encuentra dentro de un conjunto. La expresión

```
letras.contains('V')
```

tomará valor *true* si el conjunto letras contiene el elemento 'V', y *false* en caso contrario.

En un conjunto no existe un orden entre sus elementos. No obstante, existe un tipo especial de conjuntos, el *SortedSet*, en el cual los elementos sí están ordenados. Se construye de la siguiente forma:

```
SortedSet<Character> letrasOrdenadas = new TreeSet<>();
```

Cuando se añaden elementos a este conjunto, se colocan de forma ordenada. Todas las operaciones aplicables a *Set* son también válidas para *SortedSet*, que además incluye otras operaciones exclusivas.

El tipo Map

El tipo de dato *Map* permite modelar el concepto de aplicación: una relación entre los elementos de dos conjuntos de modo que a cada elemento del conjunto inicial le corresponde uno y solo un elemento del conjunto final. Los elementos del conjunto inicial se denominan claves (*keys*) y los del conjunto final valores (*values*).

Para crear un *Map* hay que indicar el tipo de las claves y el tipo de los valores. Por ejemplo, para construir un *Map* cuyas claves son cadenas de caracteres y cuyos valores son números reales, haríamos lo siguiente:

```
Map<String, Double> temperaturasCiudad = new HashMap<>();
```

Con este *Map* podemos representar las temperaturas de distintas ciudades.

Observa que dos ciudades pueden tener la misma temperatura, es decir, los valores del *Map* pueden repetirse, pero no así las claves, que son únicas.

Para almacenar en el *Map* la temperatura de una ciudad haríamos lo siguiente:

```
temperaturasCiudad.put("Córdoba", 19.1);
```

El método *put* crea en el *Map* una pareja con la cadena "Córdoba" como clave y el número 19.1 como valor.

Alternativamente podemos crear un Map inmutable con un conjunto de pares clave valor de la forma:

```
Map<String,Double>temperaturasCiudad = Map.of("Córdoba",
      19.1,"Sevilla",20.1);
```

Para obtener el valor asociado a una clave de un *Map*, utilizamos el método *get*. Por ejemplo, la sentencia

```
Double t = temperaturasCiudad.get("Córdoba");
```

almacena en la variable t el valor 19.1.

Los valores de un *Map* pueden ser de un tipo simple, como en este caso, o también pueden ser agregados, como una lista o un conjunto.

Para calcular el número de parejas de un *Map* disponemos del método *size*, al igual que en las listas y conjuntos. Existen otros métodos para obtener las claves, los valores y las parejas, que veremos más adelante.

Lectura y escritura de datos en pantalla y ficheros

Para escribir los datos por consola usaremos los métodos *println, print, printf* de *System.out.* El primero imprime el carácter fin de línea al final de la cadena que se pasa como argumento. El segundo no. El tercero acepta un formato y una secuencia variable de parámetros. Así, si escribimos las siguientes líneas de código:

```
System.out.print("El valor de la variable n es " + n + ".");
System.out.println("El valor de la variable n es " + n +
       ".\n");
System.out.printf("El valor de la variable n es %s .",n)
```

Para leer los datos de la consola podemos hacerlo con el método *readLine* de la clase *Scanner.*

```
Scanner in = new Scanner(System.in);
System.out.printf("Introduzca una cadena\n");
String s = in.nextLine();
```

La lectura y escritura en un fichero la haremos con las funciones siguientes cuyo código veremos más adelante. La primera no devuelve una lista con las líneas del fichero.

```
List<String> lineasFromFile(String file) {
       …
}

void write(String file, String text){
       …
}
```