

```
        }
    return s;
}

@Override
public Estacion estacionMasBicisDisponibles() {
    Estacion em = null;
    for(Estacion e:super.estaciones()) {
        if(em == null || e.free_bikes() > em.free_bikes())
            em = e;
    }
    return em;
}

@Override
public Map<Integer, List<Estacion>> estacionesPorBicisDisponibles()
    Map<Integer, List<Estacion>> m = new HashMap<>();
    for (Estacion e : super.estaciones()) {
        Integer key = e.free_bikes();
        if (m.containsKey(key)) {
            m.get(key).add(e);
        } else {
            List<Estacion> ls = new ArrayList<>();
            ls.add(e);
            m.put(key, ls);
        }
    }
}
```

FUNDAMENTOS DE PROGRAMACIÓN: JAVA

Miguel Toro Bonilla

Editorial Universidad de Sevilla



Fundamentos de programación: JAVA





Editorial Universidad de Sevilla

**COLECCIÓN: MANUALES DE INFORMÁTICA DEL
INSTITUTO DE INGENIERÍA INFORMÁTICA**

DIRECTOR DE LA COLECCIÓN

Miguel Toro Bonilla. Universidad de Sevilla

CONSEJO DE REDACCIÓN

Miguel Toro Bonilla. Universidad de Sevilla

Mariano González Romano. Universidad de Sevilla

Andrés Jiménez Ramírez. Universidad de Sevilla

COMITÉ CIENTÍFICO

Antón Cívit Ballcells. Universidad de Sevilla

María José Escalona Cuaresma. Universidad de Sevilla

Francisco Herrera Triguero. Universidad de Granada

Carlos León de Mora. Universidad de Sevilla

Alejandro Linares Barranco. Universidad de Sevilla

Mario Pérez Jiménez. Universidad de Sevilla

Mario Piattini. Universidad de Castilla-La Mancha

Ernesto Pimentel. Universidad de Málaga

José Riquelme Santos. Universidad de Sevilla

Agustín Risco Núñez. Universidad de Sevilla

Nieves Rodríguez Brisaboa. Universidad de Castilla-La Mancha

Antonio Ruiz Cortés. Universidad de Sevilla

José Luis Sevillano Ramos. Universidad de Sevilla

Ernest Teniente. Universidad Politécnica de Cataluña

Francisco Tirado Fernández. Universidad Complutense de Madrid



Miguel Toro Bonilla

Fundamentos de programación: JAVA



Sevilla 2022



Colección: Manuales de Informática del
Instituto de Ingeniería Informática (I3US)

Núm.: 1

COMITÉ EDITORIAL:

Araceli López Serena
(Directora de la Editorial Universidad de Sevilla)
Elena Leal Abad
(Subdirectora)
Concepción Barrero Rodríguez
Rafael Fernández Chacón
María Gracia García Martín
Ana Ilundáin Larrañeta
María del Pópulo Pablo-Romero Gil-Delgado
Manuel Padilla Cruz
Marta Palenque Sánchez
María Eugenia Petit-Breuilh Sepúlveda
José-Leonardo Ruiz Sánchez
Antonio Tejedor Cabrera



Esta obra se distribuye con la licencia
Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional
(CC BY-NC-SA 4.0)

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

Miguel Toro 2022

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.



Agradecimientos

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

AGRADECIMIENTOS	7
ÍNDICE	9
CONCEPTOS BÁSICOS DE LA P.O.O.	14
OBJETOS	16
CLASES, REGISTROS E INTERFACES	17
ESTRUCTURA DE UN PROGRAMA EN JAVA	21
ELEMENTOS BÁSICOS DEL LENGUAJE	23
IDENTIFICADORES	23
PALABRAS RESERVADAS DE JAVA	24
LITERALES	25
COMENTARIOS	25
TIPOS DE DATOS	26
TIPOS PROPORCIONADOS POR JAVA	26
VARIABLES Y CONSTANTES	28
VARIABLES	28
CONSTANTES	29
EXPRESIONES Y OPERADORES	30
EXPRESIONES	30



OPERADORES Y CONVERSIONES DE TIPOS	31
PRECEDENCIA Y ASOCIATIVIDAD DE LOS OPERADORES	32
EL TIPO STRING, TIPOS PARA EL MANEJO DE FECHAS	34
TIPO STRING	34
TIPOS PARA EL MANEJO DE FECHAS Y HORAS	36
SENTENCIAS DE CONTROL SELECTIVAS	39
SENTENCIA IF-ELSE	39
SENTENCIA SWITCH	40
AGREGADOS DE DATOS	42
LISTAS	42
CONJUNTOS	43
EL TIPO MAP	44
LECTURA Y ESCRITURA DE DATOS EN PANTALLA Y FICHEROS	46
SENTENCIAS DE CONTROL ITERATIVAS	48
SENTENCIA WHILE	48
SENTENCIA FOR CLÁSICO	48
SENTENCIA FOR EXTENDIDO	49
SENTENCIA BREAK	50
STREAMS	51
TIPOS GENÉRICOS, REGISTROS, CLASES E INTERFACES	53
ATRIBUTOS	55
MÉTODOS	59
PASO DE PARÁMETROS	61
PARSING: MÉTODO DE FACTORÍA A PARTIR DE STRING	63
COTAS SOBRE PARÁMETROS DE TIPO	64
REUTILIZACIÓN Y HERENCIA	65
HERENCIA	65
CLASES ABSTRACTAS Y FINALES	68
REUTILIZACIÓN MEDIANTE DELEGACIÓN	68
GRAFO DE TIPOS	70



IGUALDAD, IDENTIDAD Y ORDEN NATURAL	72
IGUALDAD E IDENTIDAD	72
CONTRATO ASOCIADO A LOS MÉTODOS EQUALS, HASHCODE Y TOSTRING	75
ORDEN NATURAL, EL TIPO COMPARABLE	76
RESTRICCIONES Y EXCEPCIONES	78
EXCEPCIONES	78
LANZAMIENTO DE EXCEPCIONES	79
GESTIÓN DE EXCEPCIONES	80
AGREGADOS DE DATOS	82
LA INTERFAZ COLLECTION	82
EL TIPO LIST	84
<i>Otras operaciones sobre listas</i>	90
EL TIPO SET	92
<i>Otras operaciones sobre conjuntos</i>	93
EL TIPO SORTEDSET	95
LA CLASE DE UTILIDAD COLLECTIONS	96
EL TIPO MAP	98
<i>Definición</i>	98
<i>Métodos del tipo Map</i>	100
<i>El tipo Map.Entry</i>	102
<i>Otras operaciones sobre maps</i>	103
EL TIPO SORTEDMAP	105
EL TIPO STREAM	107
STREAMS	107
EL TIPO STREAM	111
INTERFACES FUNCIONALES Y LAMBDA EXPRESIONES	112
LA INTERFAZ FUNCIONAL PREDICATE	113
LA INTERFAZ FUNCTION Y BIFUNCTION	114
LAS INTERFACES UNARYOPERATOR Y BINARYOPERATOR	114
LA INTERFAZ CONSUMER Y BICONSUMER	115
LA INTERFACE SUPPLIER	115
LOS INTERFACES COMPARABLE Y COMPARATOR: ORDEN NATURAL Y ORDENES ALTERNATIVOS	116



MÉTODOS DE FACTORÍA DE STREAMS	118
OPERACIONES SOBRE STREAMS	120
MÉTODOS TRANSFORMADORES	121
MÉTODOS ACUMULADORES	122
MÉTODOS CONSUMIDORES	127
OTROS MÉTODOS DE STREAM	128
OPERACIONES ADICIONALES DE STREAM	128
LECTURA Y ESCRITURA DE FICHEROS Y STREAMS	129
LECTURA DE UN FICHERO	129
ESCRITURA EN UN FICHERO	130
VERSIONES IMPERATIVAS DEL CÓDIGO FUNCIONAL	131
FACTORÍA	131
FUNCIONES DE TRANSFORMACIÓN	132
ACUMULADORES	133
ACCIONES	138
DISEÑO DE RUTAS	139
DISEÑO	140
<i>Coordenadas2D</i>	140
<i>Coordenadas3D</i>	141
<i>Marca</i>	141
<i>Intervalo</i>	142
<i>Ruta</i>	142
IMPLEMENTACIÓN	143
SERVICIO DE BICICLETAS DE SEVILLA	153
DISEÑO	154
<i>Estación</i>	154
<i>Red</i>	154
IMPLEMENTACIÓN	155
CÁLCULOS SOBRE UN LIBRO	161
AEROPUERTOS, VUELOS Y COMPAÑIAS AÉREAS	171
DISEÑO	173
<i>Aeropuerto</i>	173



	<i>Índice</i>
<i>Aeropuertos</i>	174
<i>Aerolínea</i>	174
<i>Aerolíneas</i>	175
<i>Vuelo</i>	176
<i>Vuelos</i>	176
<i>OcupacionVuelo</i>	177
<i>OcupacionesVuelos</i>	178
<i>Preguntas</i>	179
IMPLEMENTACIÓN	179
OBJETOS GEOMÉTRICOS	202
TIPOS	202
<i>Vector2D</i>	202
<i>Objeto2D</i>	205
<i>Punto2D</i>	206
<i>Segmento2D</i>	209
<i>Circulo2D</i>	210
<i>Poligono2D</i>	211
<i>Agregado2D</i>	214
REPRESENTACIÓN GRÁFICA DE LOS OBJETOS GEOMÉTRICOS	216
WHATSAPP	219
TIPOS	220
EXPRESIONES REGULARES	221
MONTECARLO	225
EL MÉTODO DE MONTECARLO	227
TIPOS	228
UNIVERSO	234
TIPOS	234
BIBLIOGRAFÍA	239



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 redonda 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,  $\sqrt{x^2 + y^2}$   
distancia_a(p:Punto2D): Double, derivada,  $\sqrt{(x - p.x)^2 + (-p.y)^2}$   
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 los 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, distancial, 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
<code>. [] ()</code>	
<code>+ - ! ++ -- (tipo) new</code>	derecha a izquierda
<code>* / %</code>	izquierda a derecha
<code>+ -</code>	izquierda a derecha
<code>< <= > >=</code>	izquierda a derecha
<code>== !=</code>	izquierda a derecha
<code>&&</code>	izquierda a derecha
<code> </code>	izquierda a derecha
<code>? :</code>	
<code>= += -= *= /= %=</code>	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, tipos para el manejo de fechas

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 incorporan 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.parseDouble(3.7,-6.7);
```

El proceso inverso de parse es 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. Estos 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 longitud 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 parámetro 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:



El tipo String, tipos para el manejo de fechas

```
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:



El tipo String, tipos para el manejo de fechas

```
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.



Lectura y escritura de datos en pantalla y ficheros

```
List<String> lineasFromFile(String file) {  
    ...  
}  
  
void write(String file, String text) {  
    ...  
}
```



Sentencias de control iterativas

Los bucles son sentencias de control que ejecutan un bloque de sentencias un número determinado de veces. En Java los bucles se implementan con las sentencias *for* y *while*.

Sentencia while

La sentencia *while* ejecuta el bloque de sentencias mientras la condición evaluada sea cierta. Su sintaxis es la que se muestra a continuación.

Por ejemplo, para sumar los números comprendidos entre 1 y n, haríamos lo siguiente:

```
Integer suma = 0;
int i = 1;
while (i <= n) {
    suma = suma + i;
    i++;
}
```

Sentencia for clásico

La sentencia *for* tiene la siguiente sintaxis, donde *inicialización* y *actualización* son sentencias y *condición* es una expresión lógica.



Sintaxis:

```
for (inicialización; condición; actualización) {
    sentencia-1;
    sentencia-2;
    ...
    sentencia-n;
}
```

¿Cómo funciona el *for*? Cuando el control de un programa llega a una sentencia *for*, lo primero que se ejecuta es la *inicialización*. A continuación, se evalúa la *condición*, si es cierta (valor distinto de cero), se ejecuta el bloque de sentencias de 1 a n. Una vez terminado el bloque, se ejecuta la sentencia de actualización antes de volver a evaluar la condición, si ésta fuera cierta de nuevo se ejecutaría el bloque y la actualización, y así repetidas veces hasta que la condición fuera falsa y abandonáramos el bucle.

El bucle *while* anterior, escrito con una sentencia *for*, sería así:

```
Integer suma = 0;
for (int i = 0; i <= n; i++) {
    suma = suma + i;
}
```

Sentencia for extendido

Existe una variante del *for* que se utiliza para recorrer **iterables**. Los agregados de datos ofrecen vistas iterables para recorrer sus elementos. Las listas y los conjuntos son iterables. Supongamos por ejemplo que queremos calcular el valor medio de las temperaturas de una lista *temperaturas* de tipo *Double*. Lo podríamos hacer mediante un *for* clásico:

```
Double suma = 0.0;
for (int i = 0; i < temperaturas.size(); i++) {
    suma = suma + temperaturas.get(i);
}
Double temperaturaMedia = suma / temperaturas.size();
```

Existe una forma alternativa, usando un *for* extendido, que resulta más simple. Se haría de la forma siguiente:



```
Double suma = 0.0;
for (Double t: temperaturas) {
    suma = suma + t;
}
Double temperaturaMedia = suma / temperaturas.size();
```

La variable *t* va recorriendo las temperaturas de la lista, desde la primera hasta la última.

El *for* se puede combinar con la lectura de ficheros de la forma:

```
Integer s = 0;
List<String> lineas = lineasFromFile("datos.txt");
for(String linea: lineas){
    Integer e = Integer.parseInt(linea);
    s = s + e;
}
```

Sentencia break

La sentencia *break* dentro de un bloque de sentencias que forma parte de una estructura iterativa hace que el flujo de programa salte del bloque, dando por finalizada la ejecución de la sentencia iterativa. Imagina por ejemplo que queremos saber si existe una temperatura por debajo de cero en la lista. Si al recorrer la lista encontramos una, ya sabemos que la hay, así que no es necesario seguir recorriendo el resto de las temperaturas. En ese caso usamos un *break* para salir del bucle:

```
Boolean temperaturaNegativa = false;
for (Double t: temperaturas) {
    if (t < 0) {
        temperaturaNegativa = true;
        break;
    }
}
```

Al terminar la ejecución de este código, la variable *temperaturaNegativa* tendrá valor *true* si la lista de números contiene al menos un valor negativo, y *false* si no contiene ninguno.



Streams

Apartir de la versión 8, Java introduce una nueva y potente forma de realizar operaciones con agregados de datos mediante *streams*. Un *stream* es similar a un *iterable* pero con muchas más capacidades. Cada agregado de datos ofrecerá una o más vistas en forma de *stream*. Se obtienen *streams* a partir de una lista o conjunto mediante el método *stream()*.

Supongamos por ejemplo la lista de temperaturas vista anteriormente. La lista es iterable pero también ofrece una vista en forma de stream. Si queremos conocer el número de ellas que están por debajo de cero, podríamos hacerlo mediante el siguiente bucle:

```
long bajoCero = 0;
for (Double t: temperaturas) {
    if (t < 0) {
        bajoCero++;
    }
}
System.out.println(String.format(
    "Hay %d temperaturas bajo cero", bajocero));
```

Los *streams* nos ofrecen una alternativa a este bucle. Solo tenemos que obtener un stream a partir de la lista y aplicar las operaciones propias de los *streams*. El mismo tratamiento anterior se haría de esta otra forma:



```
long bajoCero = temperaturas.stream()
    .filter(x -> x < 0)
    .count();
System.out.println(String.format("Hay %d temperaturas bajo
cero", bajocero));
```

Veamos un segundo ejemplo. En este caso queremos saber si existe alguna temperatura por encima de los 40 grados. Sería así:

```
boolean algunaPorEncima = temperaturas.stream()
    .anyMatch(x -> x > 40);
if (algunaPorEncima) {
    System.out.println("Hay al menos una temperatura mayor
    de 40°");
}
```

El paso de usar bucles a usar *streams* supone el paso de un paradigma imperativo a un paradigma funcional, en el cual indicamos qué es lo que queremos hacer, y no cómo hacerlo; de eso se encarga ya el propio lenguaje, simplificando así nuestra tarea.

Los iterables sirven para hacer tratamientos secuenciales sobre agregados de forma imperativa. Los *streams* son adecuados para hacer tratamientos secuenciales o paralelos sobre los agregados de manera funcional.

Streams e iterables son tipos diferentes pero pueden ser convertidos uno en otro. Un iterable puede ser usado en un for extendido pero un stream no porque tiene sus propios métodos. Un stream puede ser convertido en iterable con el método iterator().



Tipos genéricos, registros, clases e interfaces

Ya hemos visto previamente que para diseñar tipos nuevos partimos de un diseño de tipo. Posteriormente implementamos el diseño mediante un *record*. Un registro (*record*) es un tipo especial de clase. Ahora veremos la relación entre clase y registro.

Un tipo puede ser genérico. Eso quiere decir que tiene uno o varios *parámetros de tipo*. Esos parámetros de tipo serán posteriormente sustituidos (*instanciados*) por tipos concretos. Veamos un ejemplo de diseño de tipo genérico y su implementación mediante un registro.

```
Par<A,B>
Propiedades
a: A, básica
b: B, básica
copy(): Par<A,B>, derivada
Representación
(a,b)
Orden Natural
No tiene
Factoría
of(A a, B b): Par<A,B>
```

La implementación como un registro es:



```
public record Par<A, B> (A a, B b) {  
  
    public static <A,B> Par<A, B> of(A a, B b) {  
        return new Par<>(a,b);  
    }  
  
    public Par<A, B> copy(){  
        return new Par<>(this.a,this.b);  
    }  
  
    @Override  
    public String toString() {  
        return String.format("(%s,%s)",this.a,this.b);  
    }  
}
```

Tal como vemos el nombre del tipo lleva unos parámetros de tipo *A*, *B* delimitados por *<>*. Junto con los tipos genéricos también tenemos métodos genéricos. Estos métodos tienen también parámetros de tipo que se declaran en la cabecera como en el caso del método de factoría *of*.

Como hemos dicho los registros son una forma particular de clase. Los registros están pensados para simplificar el código de una clase. Sirven para diseñar tipos inmutables y a partir de un estado hacen implícito (sus atributos) generan un constructor, los métodos que definen la igualdad del tipo (*equals* y *hashCode*), métodos para acceder a las propiedades básicas y su representación (el método *toString*). Las clases permiten más flexibilidad explicitando el conjunto de atributos (estado) que podemos hacer públicos o privados. Para cada clase debemos diseñar explícitamente uno o varios constructores y a partir de la igualdad definida para el tipo los métodos *equals* y *hashCode*. Así mismo debemos implementar el método *toString*.

Las *clases*, por lo tanto, son las unidades de la POO que permiten definir los detalles del *estado interno* de un objeto (mediante los *atributos*), calcular las *propiedades* de los objetos a partir de los atributos e implementar las *funcionalidades* ofrecidas por los objetos (a través de los *métodos*)

Para escribir una clase en Java tenemos que ajustarnos al siguiente patrón:



```
[Modificadores] class NombreClase [extends ...] [implements  
....] {  
    [atributos]  
    [métodos]  
}
```

El patrón lo tenemos que completar con el nombre de la clase, y, de manera opcional, una serie de modificadores, una cláusula *extends* opcional, una cláusula *implements* también opcional y una serie de atributos y métodos. Ambas cláusulas las estudiaremos más abajo.

En nuestra metodología para ir detallando la clase nos guiamos por el diseño del tipo.

Atributos

Los atributos sirven para establecer los detalles del *estado interno* de los objetos. Son un conjunto de variables, cada una de un tipo, y con determinadas restricciones sobre su visibilidad exterior. En el registro los atributos son implícitos.

Como una primera guía dentro de nuestra metodología, la clase tendrá un atributo por cada propiedad básica y será del mismo tipo que esta propiedad. Aunque más adelante aprenderemos otras posibilidades, restringiremos el acceso a los atributos para que sólo sean visibles desde dentro de la clase. Esto implica que a la hora de declarar los atributos tenemos que anteponer la palabra reservada **private** a la declaración de este. Las otras posibilidades son **public** o **protected**.

Como regla general, a los atributos les damos el mismo nombre, pero comenzando en minúscula, de la propiedad que almacenan. Solo definimos atributos para guardar los valores de las propiedades que sean básicas. Las propiedades derivadas, es decir, aquellas que se pueden calcular a partir de otras, no tienen un atributo asociado en general.

La declaración de un atributo tiene un *ámbito*, que va desde la declaración hasta el fin de la clase, incluido el cuerpo de los métodos.

La sintaxis para declarar los atributos responde al siguiente patrón:



```
[Modificadores] tipo Identificador [ = valor inicial ];
```

Como puede verse, un atributo puede tener, de forma opcional, un posible valor inicial. Los modificadores incluyen la declaración de la visibilidad y opcionalmente final. Cuando este aparece el atributo es no modificable.

Veamos el tipo Par anterior pero considerado que sus propiedades pueden ser modificadas, por lo tanto el tipo es mutable, implementado como una clase. Lo llamamos *ParM*.

```
public class ParM<A, B> {  
  
    public static <A, B> ParM<A, B> of(A a, B b) {  
        return new ParM<A, B>(a, b);  
    }  
    private A a;  
    private B b;  
  
    private ParM(A a, B b) {  
        super();  
        this.a = a;  
        this.b = b;  
    }  
}
```

Como vemos al implementar el tipo como una clase necesitamos atributos (que definen el estado del objeto). Usualmente los declaramos privados, un constructor que también lo declaramos privado y un método de factoría público.

```
public A getA() {  
    return a;  
}  
public void setA(A a) {  
    this.a = a;  
}
```

Junto a lo anterior diseñamos métodos para acceder los atributos o modificarlos. Los métodos que devuelven atributos o combinación de ellos se les llama *getters*. Son la implementación de las propiedades del tipo. Las propiedades básicas devolverán directamente el atributo correspondiente. Las propiedades derivadas devolverán el cálculo



correspondiente a partir de los atributos. Ambos métodos es costumbre que empiecen por *get* y por eso se les llama *getters*.

Si una propiedad es modificable tiene, además, asociado un método *setter*. Este es un método que devuelve *void* y modifica uno o más atributos a partir del parámetro recibido.

Por lo tanto las propiedades tienen asociado un método *get* y si son modificables otro *set*. La propiedad B, como la A anterior es modificable.

```
public B getB() {  
    return b;  
}  
public void setB(B b) {  
    this.b = b;  
}
```

El tipo puede tener otras propiedades derivadas como la siguiente que obtiene una copia del objeto.

```
public ParM<A, B> getCopy(){  
    return new ParM<>(this.a,this.b);  
}
```

Al diseñar clases debemos diseñar también el método *toString* que indica cómo convertir el objeto en una cadena de caracteres.

```
@Override  
public String toString() {  
    return String.format("(%s,%s)",a,b);  
}
```

La implementación de tipos mediante clases obliga también a implementar los métodos *equals* y *hashCode*. Ambos métodos son necesarios si los objetos del tipo van a ser usados como claves en un *Map*. El método *equals* indica cuando dos objetos del tipo son iguales. El método *hashCode* devuelve un entero asociado al objeto. Una especie de identificador del objeto necesario si este quiere usarse en claves de un *Map*, un conjunto, etc.



```
@Override  
public int hashCode() {  
    final int prime = 31;  
    int result = 1;  
    result = prime * result +  
        ((a == null) ? 0 : a.hashCode());  
    result = prime * result +  
        ((b == null) ? 0 : b.hashCode());  
    return result;  
}
```

Los métodos *equals*, *hashCode* y *toString* deben ser coherentes entre si en el sentido que si los objetos son iguales tienen el mismo *toString* y el mismo *hashCode*. Sin embargo el tener dos *hashCode* iguales no obliga a que los objetos sean iguales.

```
@Override  
public boolean equals(Object obj) {  
    if (this == obj)  
        return true;  
    if (obj == null)  
        return false;  
    if (getClass() != obj.getClass())  
        return false;  
    ParM<?,?> other = (ParM<?,?>) obj;  
    if (a == null) {  
        if (other.a != null)  
            return false;  
    } else if (!a.equals(other.a))  
        return false;  
    if (b == null) {  
        if (other.b != null)  
            return false;  
    } else if (!b.equals(other.b))  
        return false;  
    return true;  
}
```

Los esquemas para generar los métodos *equals* y *hashCode* son sistemáticos por lo que pueden ser generados automáticamente a partir



de la definición de la igualdad de los objetos del tipo. Se recomienda usar siempre al versión generada por Eclipse.

Como podemos ver implementar un tipo mediante una clase es más flexible pero mucho más verboso. Al usar un record para implementar un tipo ya tenemos disponibles el constructor, los métodos para acceder a las propiedades básicas, un `toString` básico y los métodos `equals` y `hashCode`. La igualdad se define en este caso como la igualdad de todas las propiedades básicas. En el caso de record los métodos para acceder a las propiedades no suelen comenzar por `get`. Los tipos diseñados mediante record son inmutables y finales.

Métodos

Los métodos son unidades de programa que indican la forma concreta de *consultar o modificar* las propiedades de un objeto determinado. La sintaxis para los métodos corresponde al siguiente patrón:

```
[Modificadores] TipoDeRetorno nombreMétodo ( [parámetros formales] ) {  
    Cuerpo;  
}
```

Un método tiene dos partes: *cabecera* (o *signatura* o *firma*) y *cuerpo*. La *cabecera* está formada por el nombre del método, los parámetros formales y sus tipos, y el tipo que devuelve. Cada parámetro formal supone una nueva declaración de variable cuyo ámbito es el cuerpo del método. El *cuerpo* está formado por declaraciones, expresiones y otro código necesario para indicar de forma concreta qué hace el método. Las variables que se declaran dentro de un método se denominan variables locales y su ámbito es desde la declaración hasta el final del cuerpo del método.

En una interfaz sólo aparecen las cabeceras de los métodos. Sin embargo, en una clase deben aparecer tanto la cabecera como el cuerpo. En una clase, al igual que ocurría con las interfaces, puede haber métodos con el mismo nombre pero diferente cabecera. Pero no puede haber dos métodos con el mismo nombre, los mismos parámetros formales y



distinto tipo de retorno. Esta posibilidad, que no la tienen todos los lenguajes de programación, se denomina sobrecarga de métodos.

De forma general, hay dos tipos de métodos: *observadores* y *modificadores*. Los métodos observadores no modifican el estado del objeto. Es decir, no modifican los atributos. O lo que es lo mismo, los atributos no pueden aparecer, dentro de su cuerpo, en la parte izquierda de una asignación. Cuando implementamos tipos mediante clases la costumbre, o la norma de estilo, es que métodos observadores empiecen por *get* seguido del nombre de la propiedad. Los métodos modificadores modifican el estado del objeto. Los atributos aparecen, dentro de su cuerpo, en la parte izquierda de una asignación. La costumbre es que los métodos modificadores empiezan por *set* seguido del nombre de la propiedad que modifican.

En el caso de los registros la costumbre es otra. Los atributos son siempre privados y ocultos, no existen métodos modificadores y los métodos consultores coinciden con el nombre de la propiedad sin *get*.

Ejemplos de un método modificador (*setA*) y otro consultor (*getA*) son:

```
public void setA(A a) {  
    this.a = a;  
}
```

← Cabecera
} Cuerpo

```
public A getA() {  
    return this.a;  
}
```

← Cabecera
} Cuerpo

Hay unos métodos especiales que llamaremos *métodos constructores* o simplemente *constructores*. Son métodos que tienen el mismo nombre que la correspondiente clase. Sirven para crear objetos nuevos y establecer el estado inicial de los objetos creados.



En el caso de registro el constructor está definido implícitamente. En la clase hay que definirlo explícitamente

Ejemplo de constructor de la clase *ParM* es:

```
ParM(A a, B, B) {  
    this.a = a;  
    this.b = b;  
}
```

Como vemos, los dos métodos constructores tienen el mismo nombre que la clase y puede haber varios.

La palabra *this* es una palabra reservada y representa el objeto que estamos implementando. Así, *this.x* es el atributo *x* del objeto que estamos implementando. De la misma forma, *this.getX()* es el método *getX* invocado sobre el objeto que estamos implementando y, sin embargo, *p.getX()* es la invocación del método *getX* sobre el objeto *p*.

Con la palabra *this* y el operador punto podemos formar expresiones correctas dentro de una clase.

Los métodos de factoría pueden ser genéricos. En este caso deben incluir los parámetros de tipo en su cabecera como en

```
public static <A, B> Par<A, B> of(A a, B b) {  
    return new Par<A, B>(a, b);  
}
```

Por último, un tipo genérico debe ser *instanciado* antes de ser usado. Instanciar un tipo genérico consiste en sustituir los parámetros de tipo por tipos concretos. Por ejemplo:

```
Par<Integer, Integer> p = Par.of(34, 56);
```

Paso de parámetros

Las expresiones se forman con constantes, variables, llamadas a métodos y operadores. Un método, cuando se diseña, tiene una cabecera y un cuerpo. La cabecera está compuesta del tipo de retorno, el nombre del método y los *parámetros formales*. Cada parámetro *formal* está



compuesto de un tipo y un identificador que se usa en el cuerpo del método.

La llamada a un método se compone de su nombre y varias expresiones: una por cada parámetro formal. Estas expresiones se llaman parámetros reales. En Java los parámetros son posicionales. Debe haber tantos parámetros reales como formales y cada parámetro real se sustituye por su correspondiente parámetro formal. El tipo de la expresión de un parámetro formal debe ser adecuado para ser asignado al tipo del parámetro formal correspondiente.

Ejemplo. En la llamada al método *Par.of*:

```
Par<Integer,Double> p = Par.of(3+4*5,-57.);
```

3+4*5 y -57 son parámetros reales. Como vemos el primero tiene tipo *Integer* y el segundo *Double* que son adecuados para los correspondientes tipos de los parámetros formales.

Los parámetros de un método pueden ser de dos tipos: *parámetros de entrada* y parámetros de *entrada-salida*. Los primeros son aquellos tales que aunque se modifiquen dentro del cuerpo no quedan modificados fuera de la llamada del método. Los segundos quedan modificados fuera de la llamada del método si se modifican en el cuerpo. En Java los parámetros de tipos primitivos o tipos objeto inmutables son parámetros de entrada y los de tipos objeto mutables de entrada salida.

```
Double calcula(Double s, Par<Double, Double> p) {
    s = s+7.;
    p.setA(2*s);
    p.setB(-s);
    return s+p.getA()+p.getB();
}
===
Double r = 56.7;
ParM<Double,Double> q = ParM.of(45.6,56.7);
Double s = calcula(r,q);
=====
```

En lo anterior el parámetro s es de entrada (porque es inmutable) y el p de entrada salida (porque es mutable). El valor de r tras la llamada el



método sigue siendo 56.7 pero el de q ha cambiado siendo después de la llamada (127.4,-63.7).

Parsing: método de factoría a partir de String

Con el método *toString()* convertimos un objeto en su representación como cadena de caracteres. Usualmente es adecuado dotar al tipo de un método de factoría que lleve a cabo la operación inversa, es decir, construir un objeto a partir de la información contenida en una cadena de caracteres.

Los tipos que proporciona Java ya vienen con constructores de estos métodos de factoría. Por ejemplo:

```
Integer a = Integer.parseInt("+35");
Double b = Double.parseDouble("-4.57");
```

Para diseñar los métodos anteriores necesitamos analizar la cadena y partirla en trozos adecuados. Ejemplo:

```
record Punto2D(Double x,Double y) {
    public static Punto2D parse(String text) {
        String text2 = text.substring(1,
            text.length()-1);
        String[] campos = text2.split(",");
        return new Punto2D(
            Double.parseDouble(campos[0].trim()),
            Double.parseDouble(campos[1],trim())));
    }
}
```

En primer lugar, se eliminan los posibles caracteres adicionales que no formen parte de los valores de las propiedades (en este caso los paréntesis), lo cual hacemos mediante la obtención de una *substring*. A continuación, se utiliza el método *split*, también del tipo String, para trocear la cadena según un delimitador. En este caso se parte la cadena tomando como separador el carácter coma, con lo cual se deben obtener dos trozos, correspondientes a las coordenadas x e y, si la cadena está bien formada. Estos trozos son cadenas de caracteres, y es necesario convertirlos a valores de tipo *Double* para almacenarlos en los atributos



correspondientes. Para ello usamos el método de factoría adecuado, eliminando previamente los posibles espacios en blanco al comienzo y al final de la cadena mediante el método *trim* del tipo String.

Cotas sobre parámetros de tipo

Los parámetros de un tipo genérico pueden ser acotados para restringir los tipos que pueden instanciarlos. Para expresar estas cotas usamos ?, extends, super y &. Veamos algunos ejemplos:

- *<T extends B1 & B2 & B3>*: Indica que los tipos que pueden instanciar T deben ser subtipos de B1, B2 y B3.
- *<T extends Comparable<T>>*: T debe implementar el contrato Comparable<T>.
- *List<? extends F> list*: Los elementos de la lista tienen que ser de un subtipo de F
- *List<?> list*: Los elementos de la lista pueden ser de cualquier tipo
- *List<? extends Comparable<? super T>>*: Los elementos de la lista deben ser un subtipo de un supertipo de T que implemente el contrato Comparable.



Reutilización y herencia

En general cuando diseñamos un tipo partimos de otros ya diseñados e implementados. Java ya nos ofrece muchos otros los tenemos que diseñar e implementar nosotros. Un programa en Java está compuesto de un conjunto de tipos.

Un tipo tiene propiedades que serán de otros tipos ya definidos. Decimos que el nuevo tipo **usa** esos otros tipos.

Herencia

Pero también es usual que diseñemos tipos reutilizando otros. Hay varias formas de reutilización. Una muy conocida es la *herencia*. Mediante este mecanismo podemos diseñar un nuevo tipo añadiendo propiedades y posiblemente restricciones a otro tipo previo. El nuevo tipo lo llamamos *subtipo* y al reutilizado *supertipo*. Decimos que el subtipo *extiende* al o hereda del *supertipo*.

Un tipo también puede ser diseñado para que cumpla un determinado contrato especificado en una *interface*. Esto es otro tipo de herencia. Un contrato (*interface*) se puede definir a partir de uno o varios contratos añadiendo más propiedades y métodos. Un caso concreto de este tipo contrato es *Comparable*. Una interface, como un record o una clase, también define un tipo. Un tipo que especifica un contrato.

El lenguaje Java está diseñado para que todos los tipos que diseñemos extiendan un tipo concreto que se llama *Object*.



La herencia se representa mediante la cláusula *extends* (*extiende*) cuando un tipo reutiliza a otro implementado mediante una clase y mediante *implements* (*implementa*) cuando un tipo está obligado a cumplir un contrato definido por una interface.

La *herencia* (extensión o implementación) establece relaciones entre tipos. Relaciones entre subtipos y supertipos.

En Java la herencia entre interfaces puede ser múltiple: un hijo puede tener uno o varios padres. La interfaz hija tiene todas las signaturas (métodos) declaradas en los padres. No se puede ocultar ninguna.

La sintaxis para la *interfaz* con herencia responde al siguiente patrón:

```
[Modificadores] interface NombreInterfazHija
    extends NombreInterfazPadre [, ...] {
    [signaturas de métodos]
}
```

Mediante *extends* se indica que la interfaz que estamos combinando otras interfaces padres y, posiblemente, añadiendo algunos métodos más.

Entre tipo y subtipo hay una importante propiedad. Sea una variable *b* de un tipo *B*, y otra *a* de otro tipo *A*. Si *B* es un subtipo de *A*, entonces la variable *b* puede ser asignada a la variable *a* pero no al revés.

Como ejemplo veamos el tipo Fraction2 que adapta el *Fraction* de Apache definiendo una nueva representación y añadiendo el contrato Comparable



```

class Fraction2 extends Fraction implements
    Comparable<Fraction> {
    public static Fraction2 of(int num, int den) {
        return new Fraction2(num, den);
    }
    private Fraction2(int num, int den) {
        super(num, den);
    }

    @Override
    public int compareTo(Fraction other) {
        return super.getNumerator()
            *other.getDenominator() -
        super.getDenominator()
            *other.getNumerator();
    }

    @Override
    public String toString() {
        String s = "";
        s = String.format("%d",super.getNumerator());
        if(super.getDenominator() != 1) s =
            s+ String.format("%d",
                super.getDenominator());
        return s;
    }
}

```

Aquí *super(...)* se refiere al constructor de la clase padre y *super.getNumerador()* a la correspondiente propiedad de la clase padre.

En Java hay restricciones a la herencia:

- Un record no puede extender otro record o clase ni puede ser extendido.
- Una interface puede heredar de varios interfaces
- Una clase puede extender una sola clase e implementar varios interfaces.



Clases abstractas y finales

Las clases pueden tener las etiquetas de *abstract* y *final*. Una clase abstracta se declara incluyendo *abstract* delante de *class* y significa que puede tener métodos declarados *abstract*. Es decir que solo indicamos su cabecera, pero no su cuerpo.

```
public abstract class RutaA implements Ruta {
    ...
    public abstract Double getTiempo();
}
```

Las clases abstractas sirven para diseñar código común a varias clases. Estas clases reutilizan el código mediante herencia.

Una clase final se declara incluyendo *final* delante de *class* y significa que no puede ser extendida. Las clases finales se diseñan así para que no puedan ser extendidas y por lo tanto a partir de ellas no se puede diseñar nueva funcionalidad mediante herencia.

Los records tampoco pueden ser extendidos ni pueden extender otra clase o record.

Reutilización mediante delegación

La herencia (en el sentido de una clase que extiende otra) es un mecanismo de reutilización, pero impone algunas restricciones importantes en la medida que la clase hija hereda todos los métodos de la clase padre. Es decir, es un subtipo de ella. No es posible heredar solo una parte de los métodos de la clase padre. Esto en muchos casos no es adecuado por lo que el mecanismo de herencia usado para reutilizar no es útil en muchos casos. Tampoco es posible heredar de varias clases y menos heredar una parte concreta de una clase y otra parte de otra.

En estos casos el mecanismo adecuado es la reutilización mediante *delegación*. Este mecanismo consiste esencialmente en incluir uno o varios objetos como atributos privados y delegar en ellos la implementación de algunos métodos.



Veamos como ejemplo la implementación del interface *Print*.

```
interface Print {
    void print(String s);
}
```

Diseñamos la clase *Printer* que implementará la interface *Print* y delegará en un objeto de tipo *PrintStream* que tiene, entre otras funcionalidades la posibilidad de imprimir una cadena de caracteres en un dispositivo.

Hemos definido un método de factoría para construir un objeto *Printer*. Este método recibe un objeto de tipo *PrintStream* en el que se delegará la impresión. Ejemplos de objetos que imprementan *PrintStream* son *System.out*, *System.err* o un fichero abierto para escritura.

```
class Printer implements Print {
    private PrintStream p;
    public static Printer of(PrintStream p) {
        return new Printer(p);
    }
    private Printer(PrintStream p) {
        this.p = p;
    }
    @Override
    public void print(String s) {
        p.println(s);
    }
}
```

Según el objeto que pasemos como parámetro a la factoría de la clase *Printer* podemos conseguir imprimir una cadena en la consola de salida, en la salida de errores o en un fichero adecuado.

Podemos ver los ejemplos en el código siguiente.



```

void main(String[] args) {
    Printer p1 = Printer.of(System.out);
    p1.print("Hola");
    Printer p2 = Printer.of(System.err);
    p2.print("Hola");
    Printer p3 = Printer.of("ficheros/prueba.txt");
    p3.print("Hola");
}

```

Con esta técnica podemos reutilizar funcionalidades de varios tipos lo que no era posible con la herencia.

Los conceptos de *agregación* y *composición* están relacionados con el concepto anterior.

Composición: Un objeto que consta de otros objetos que, a su vez, no pueden existir después de que el objeto sea eliminado.

Agregación: Un objeto que consta de otros objetos que pueden vivir incluso después de que el objeto sea eliminado.

Veremos ejemplos en adelante.

Grafo de tipos

Un programa Java se compone de un conjunto de tipos entre los que existe relaciones de tipo-subtipo. Como ha hemos dicho si un tipo extiende otro o lo implementa es su subtipo. El conjunto de tipos de un programa más las relaciones de tipo-subtipo forman el *grafo de tipos* representamos los tipos. El grafo de tipos es útil para visualizar el conjunto de *tipos ofrecidos por un objeto*. El conjunto de tipos ofrecidos por un objeto está formado por todos los *supertipos* del tipo asociado a la clase que creó el objeto. En este conjunto siempre se añade un tipo proporcionado por Java y que es ofrecido por todos los objetos: el tipo *Object*.

Otro uso importante del grafo de tipos es para decidir cuando una variable se puede asignar a otra. Una variable *B* se podrá asignar a otra *A* solo si *B* es un subtipo de *A*.



Junto a esta relación entre los tipos hay una más que no se suele representar en el grafo de tipos. Es la *relación de uso*. Un tipo *A* usa a otro *B* cuando declara algún parámetro formal, tipo de retorno, atributo o variable local del tipo *B*.



Igualdad, identidad y orden natural

Igualdad e Identidad

En Java los tipos que comienzan con mayúscula los llamamos tipos objetos. *String*, *Integer*, *Par*, son tipos objetos. Sin embargo *int*, *double*, *long* no lo son. Las instancias de tipos objeto son objetos. Los tipos no objeto tienen valores que no son objetos. Los objetos tienen identidad. Los valore no. Dos objetos son idénticos cuando el cambio de una propiedad de uno afecta a la propiedad del otro. Esto es importante sobre todo para los tipos mutables.

Dos objetos son *iguales* cuando los valores de sus propiedades observables son iguales. Por otro lado, dos objetos son *idénticos* cuando al modificar una propiedad observable de uno de ellos, se produce una modificación en la del otro y viceversa. De lo anterior, se deduce que *identidad implica igualdad*, pero igualdad no implica identidad. Es decir, la identidad de un objeto permanece inalterada aunque cambien sus propiedades. Dos objetos no idénticos pueden tener las mismas propiedades y entonces son iguales. Dos objetos idénticos siempre tienen las mismas propiedades.

Los valores de los tipos primitivos pueden tener igualdad pero no tienen identidad. El operador *new* crea objetos con una nueva identidad. El operador de asignación entre tipos objeto asigna la identidad del objeto de la derecha al objeto de la izquierda. Después de asignar el objeto *b* al *a* (*a = b;*) ambos objetos son idénticos y, por lo tanto, si modificamos las propiedades de *a* quedan modificadas las de *b* y al revés. Entre tipos



primitivos el operador de asignación asigna valores. Si m y n son de tipos primitivos, entonces la sentencia $m = n$; asigna el valor de n a m . Después de la asignación, m y n tienen el mismo valor, pero si modificamos el valor de m , no queda modificado el de n porque los elementos de tipos primitivos no tienen identidad, sólo tienen valor.

El operador `==` aplicado entre tipos primitivos decide si los valores de los operandos son iguales.

```
int i = 7;
int j = 4;
int k = j;
boolean a = (i == j);           // a es false
boolean b = (k == j);           // b es true
```

El valor de a es *false*. Se comparan los valores de tipos primitivos y estos valores son distintos. Después de asignar j a k sus valores son iguales luego b es *true*.

Este mismo operador aplicado entre tipos objeto decide si ambos objetos son idénticos o no. Para decidir si los objetos son iguales utilizamos el método *equals*, que se invoca mediante la expresión *o1.equals(o2)* y que devuelve un valor tipo *boolean*:

```
Punto2D p1 = Punto2D.of(1.0, 2.0);
Punto2D p2 = Punto2D.of(1.0, 2.0);
boolean c = (p1 == p2);           // c es false
boolean d = p1.equals(p2); // d es true
```

Los objetos $p1$ y $p2$ han sido creados con dos identidades distintas (cada vez que llamamos al operador `new`, o un método de factoría, se genera una nueva identidad) y no han sido asignados entre sí (no son idénticos) pero son iguales porque sus propiedades los son. Por ello c es *false* y d es *true*.

Veamos otro ejemplo:



```
Punto2D p1 = Punto2D.of(1.0, 1.0);
Punto2D p2 = Punto2D.of(3.0, 1.0);
Punto2D p3 = p1;
p1.setX(3.0);
boolean a = (p3 == p1);      // a es true
boolean b = (p3 == p2);      // b es false
Double x1 = p3.getX();       // x1 vale 3.0
```

El valor de *a* es true porque *p1* ha sido asignado a *p3* y por lo tanto tienen la misma identidad. Por ello al modificar la propiedad X en *p1* queda modificada en *p3* y por lo tanto *x1* vale 3.0.

Cuando un objeto es inmutable no pueden cambiarse sus propiedades. Cualquier operación sobre el objeto que cambie las mismas producirá como resultado un nuevo objeto (con una nueva identidad) con los valores adecuados de las propiedades.

```
Integer a = 3;
Integer b = a;
b++;
boolean e = (a == b);      // e es false
```

Al ser el tipo *Integer* inmutable, el valor de *e* es false. En la línea 3, con la expresión *b++*, se crea un objeto nuevo (con una nueva identidad) que se asigna a *b*. Si se elimina la línea 3 (*b++;*) entonces el valor de *e* será *true*.

Cuando diseñamos un tipo debemos definir la *igualdad entre objetos*. En el caso de un registro dos objetos son iguales cuando lo son sus propiedades básicas. Pero para una clase la igualdad puede ser definida de otra forma. Relacionados con la igualdad está el *hashCode* del objeto y su representación. En Java esas ideas se concretan en los métodos *equals*, *hashCode*, *toString* que ya vienen disponibles en la clase *Object*.

En Java existe una clase especial llamada *Object*. Todas las clases que definamos y las ya definidas heredan de *Object*, es decir, implícitamente *Object* es un tipo al que extienden todas las clases Java. Como cualquier tipo, *Object* proporciona una serie de métodos públicos. Aunque tiene más métodos, en este tema nos vamos a centrar solamente en tres de ellos: *equals*, *hashCode* y *toString*. Veremos sus propiedades, las restricciones entre ellos y la forma de rediseñarlos para que se ajusten a nuestras necesidades. La firma de estos métodos es:



```
boolean equals(Object o);  
int hashCode();  
String toString();
```

Como el tipo *Object* es ofrecido por todos los objetos que creemos, los métodos anteriores están disponibles en todos los objetos, es decir, todos los objetos heredan los métodos *equals*, *hashCode* y *toString* de la clase *Object*. Estos métodos tienen definido un comportamiento por defecto. Veamos para qué se utiliza cada uno de estos métodos:

- El método *equals(Object o)* se utiliza para decidir si el objeto es igual al que se le pasa como parámetro.
- El método *hashCode()* devuelve un entero, que es el código *hash* del objeto. Todo objeto tiene, por lo tanto, un código *hash* asociado.
- El método *toString()* devuelve una cadena de texto que es la representación exterior del objeto. Cuando el objeto se muestre en la consola tendrá el formato indicado por su método *toString* correspondiente.

Estos métodos son generados automáticamente en el caso de un registro (*record*).

Contrato asociado a los métodos *equals*, *hashCode* y *toString*

Todos los objetos ofrecen estos tres métodos. Por lo tanto es necesaria una buena comprensión de sus propiedades y un buen diseño de estos.

Propiedades de *equals*:

- *Reflexiva*: Un objeto es igual a sí mismo. Es decir, para cualquier objeto *x*, distinto de *null*, se debe cumplir que *x.equals(x)* es *true* y *x.equals(null)* es *false*.
- *Simétrica*: Si un objeto es igual a otro, el segundo también es igual al primero. Es decir, para dos objetos cualesquiera *x* e *y*, distintos de *null*, se debe cumplir que *x.equals(y) => y.equals(x)*. Múltiples invocaciones de *x.equals(y)* deben devolver el mismo resultado si el estado de *x* e *y* no ha cambiado.



- *Transitiva:* Si un objeto es igual a otro, y este segundo es igual a un tercero, el primero también será igual al tercero. Es decir para tres objetos *x*, *y*, *z*, distintos de *null*, se debe cumplir que *x.equals(y) && y.equals(z) => x.equals(z)*.

Propiedades de equals/toString:

- Si dos objetos son iguales, sus representaciones en forma de cadena también deben serlo. Es decir, para dos objetos cualesquiera *x e y*, distintos de *null*, se debe cumplir que *x.equals(y) => x.toString().equals(y.toString())*.

Propiedades de equals/hashCode:

- Si dos objetos son iguales, sus códigos *hash* tienen que coincidir. La inversa no tiene por qué ser cierta. Es decir, para dos objetos cualesquiera *x e y*, distintos de *null*, se debe cumplir que *x.equals(y) => x.hashCode() == y.hashCode()*. Sin embargo, no se exige que dos objetos no iguales produzcan códigos *hash* desiguales, aunque hay que ser consciente de que se puede ganar mucho en eficiencia si en la mayoría de los casos objetos distintos tienen códigos hash distintos.

Las propiedades y restricciones anteriores son muy importantes. Si no se cumplen, el programa que diseñemos puede que no funcione adecuadamente. Todos los tipos ofrecidos en la API de Java tienen un diseño adecuado de esos tres métodos.

Orden natural, el tipo Comparable

El tipo *Comparable* define un orden sobre los objetos de un tipo creado por el usuario, al que llamaremos orden natural. Java ya proporciona un orden natural para los tipos objeto como *Integer*, *Double* o *String* permitiendo que, por ejemplo, se puedan ordenar cuando forman parte de una lista.

El tipo *Comparable* está definido como:



```
public interface Comparable<T> {  
    int compareTo(T o);  
}
```

El tipo *Comparable* se compone de un único método, de nombre *compareTo*. El tipo *Comparable* usa un tipo genérico (*T*) y establece el orden natural sobre los objetos de un tipo *T* dado.

- El método *compareTo* compara dos objetos *o1* y *o2* y devuelve un entero que es:
 - Negativo si *o1* es menor que *o2*
 - Cero si *o1* es igual a *o2*
 - Positivo si *o1* es mayor que *o2*

El orden natural definido debe ser coherente con la definición de igualdad. Si *equals* devuelve *true*, *compareTo* debe devolver cero. Aquí también incluimos, tal como se recomienda en la documentación de Java, la inversa. Es decir, que si *compareTo* devuelve cero, entonces *equals* devuelve *true*. Esto lo podemos enunciar diciendo que la expresión siguiente es verdadera para cualquier par de objetos *x* e *y*: $(x.compareTo(y) == 0) \Leftrightarrow (x.equals(y))$.

En general, para implementar el método *compareTo* usaremos los métodos *compareTo* de las propiedades involucradas en la igualdad o algunas otras derivadas. Un orden natural adecuado puede ser comparar en primer lugar por una propiedad elegida arbitrariamente, si resulta cero comparar por la segunda propiedad, etc.



Restricciones y excepciones

Cuando diseñamos un tipo podemos establecer *restricciones* a los valores que puede tomar una propiedad. Por ejemplo, el radio de un círculo no puede ser negativo, la fecha de llegada de un vuelo no puede ser anterior a la fecha de salida, o el nombre de una persona no puede estar vacío.

Cuando se crea un objeto, hay que comprobar que se cumplen todas las restricciones que existan sobre sus propiedades. Por tanto, los lugares de la clase donde hay que chequear las restricciones son los métodos de factoría y los modificadores si hay.

Si se incumple una restricción, el programa debe detener su ejecución, informando al usuario de la restricción incumplida. Para ello contamos en Java con un mecanismo: las excepciones.

Excepciones

Las excepciones son, junto con las sentencias de control vistas anteriormente, otro mecanismo de control. Es decir, son un instrumento para romper y gestionar el orden en que se evalúan las sentencias de un programa.

Se denomina *excepción* a un evento que ocurre durante la ejecución de un programa, y que indica una situación normal o anormal que hay que gestionar. Por ejemplo, una división por cero o el acceso a un fichero no disponible en el disco. Estos eventos pueden ser de dos grandes tipos:



eventos generados por el propio sistema y eventos generados por el programador. En ambos casos hay que gestionar el evento. Es decir, decidir qué sentencias se ejecutan cuando el evento ocurre.

Cuando se produce un evento como los comentados más arriba, decimos que se ha *disparado una excepción*. Cuando tomamos decisiones después de haberse producido un evento decimos que *gestionamos la excepción*.

Cuando se dispara (o se lanza, o se eleva) una excepción tras la ocurrencia de un evento, se crean unos objetos que transportan la información del evento. A estos objetos también los llamamos excepciones, y Java dispone de un conjunto predefinido de ellos.

Lanzamiento de excepciones

Cuando se lanza una excepción, se interrumpe el flujo del programa. El lanzamiento de una excepción suele tener la siguiente estructura:

```
if (condicion_de_lanzamiento) {
    throw new tipo_excepcion("Texto informativo");
}
```

Este código lo encapsulamos en la clase *Preconditions*. Como vemos, hay una sentencia *if* que evalúa una condición y, si es cierta, se lanza la excepción. Llamaremos *condición de lanzamiento* a la condición que controla el lanzamiento de la excepción. La condición de lanzamiento es una expresión lógica que depende de los parámetros del método y de las propiedades del objeto. La cláusula *throw* crea un objeto del tipo adecuado y dispara la excepción.

Veamos un ejemplo con el tipo *Circulo*. Una restricción de este tipo consiste en que el radio debe ser mayor que cero. Por tanto, si intentamos crear un círculo con un radio negativo, se incumplirá la restricción y deberá lanzarse una excepción. La restricción hay que chequearla en los métodos de factoría y en los métodos modificadores si el tipo es mutable.



```
record Circulo2D(Punto2D centro,Double radio) {  
  
    public static Circulo2D of(Punto2D centro,  
                               Double radio) {  
        Preconditions.checkNotNull(radio>=0,  
                                   String.format(  
                                       "El radio debe ser mayor o igual a  
                                       cero y es %.2f",radio));  
        return new Circulo2D(centro, radio);  
    }  
}
```

Los métodos pueden disparar excepciones de diversos tipos y en algunos casos documentar este hecho en su cabecera mediante *throws*.

```
public static void dividir() throws ArithmeticException {  
    System.out.println(dividendo / divisor);  
}
```

Gestión de excepciones

En el ejemplo anterior, el programa finaliza lanzando una excepción y mostrando el correspondiente mensaje de error en la consola. Hay otras excepciones que se pueden disparar ligadas a problemas con los ficheros, con las operaciones aritméticas, etc. Podemos gestionar la excepción de forma que el programa finalice de una forma más controlada. Para ello, en lugar de lanzar la excepción, la podemos capturar, es decir, detectar que se ha producido, y tomar las acciones necesarias para que el programa finalice de la forma más adecuada posible. Para ello usamos la cláusula *try/catch*. Veamos cómo sería en el ejemplo anterior.

```
try {  
    ...  
} catch (Exception e) {  
    ...  
}
```

Dentro del bloque *try* se intenta a las sentencias que pueden lanzar una excepción. El bloque *catch* *captura* esta excepción antes de que se aborte



Restricciones y excepciones

el programa ya realiza la corrección adecuada. El programa continúa su ejecución con la instrucción que sigue al bloque.

La llamada a un método que incluye *throws* en su cabecera debe hacerse dentro de una cláusula *try/catch*.



Agregados de datos

Anteriormente presentamos el concepto de agregados de datos y vimos el manejo básico de algunos de ellos, concretamente los tipos *List*, *Set* y *Map*. La adecuada utilización de tipos de datos agregados predefinidos en la API de Java, como los citados, simplifica mucho el desarrollo de programas correctos en Java y tiene muchas otras ventajas:

- Reduce el esfuerzo de programación puesto que proporciona estructuras de datos y algoritmos útiles.
- Incrementa la velocidad y la calidad de los programas, puesto que Java ofrece implementaciones optimizadas y libres de errores.
- Simplifica la interoperabilidad y la reemplazabilidad entre aplicaciones, puesto que facilita estructuras que se pueden intercambiar entre distintos componentes.
- Reduce esfuerzos de aprendizaje y diseño.

En este capítulo vamos a ver las colecciones, modeladas mediante la interfaz *Collection* y dos de las interfaces que heredan de ella: *List*, que modela las listas, y *Set*, que modela los conjuntos. También veremos el tipo *Map*, que modela el concepto matemático de Aplicación.

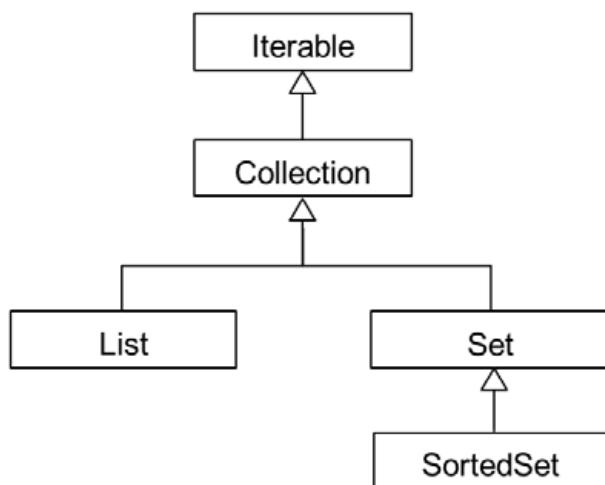
La interfaz Collection

La interfaz *Collection*, que define un tipo que podemos denominar Colección, está definida en el paquete *java.util*. Una colección es un tipo



muy general, que agrupa objetos (elementos) de un mismo tipo; su comportamiento específico viene determinado por sus subinterfaces, que pueden admitir elementos duplicados o no, y cuyos elementos pueden estar ordenados o no según determinado criterio. No existe una implementación específica de la interfaz *Collection*; sí la tienen sus subinterfaces. El tipo *Collection* se utiliza para declarar variables o parámetros donde se quiere la máxima generalidad posible, esto es, que puedan ser utilizados tanto por listas como por conjuntos (*Collection* tiene otros subtipos que se tratarán más adelante).

La interfaz *Collection* es genérica, por lo que hablaremos de *Collection<E>*. Hereda de *Iterable<E>*, lo que implica que las colecciones, así como las listas y los conjuntos, subtipos suyos, son iterables y se puede utilizar con ellas el *for* extendido. La siguiente figura representa la jerarquía de interfaces de las colecciones.



Las operaciones del tipo *Collection* se especifican a continuación. En general, el valor devuelto por los métodos *add*, *addAll*, *remove*, *removeAll* y *retainAll* será *true* si la colección queda modificada por la aplicación del método y *false* en caso contrario.

boolean	<code>add(E e)</code>
---------	-----------------------

Añade un elemento a la colección, devuelve *false* si no se añade.



boolean	addAll(Collection<? extends E> c)
	Añade todos los elementos de c a la colección que invoca. Es el operador unión. Devuelve <i>true</i> si la colección original se modifica.
void	clear()
	Borra todos los elementos de la colección.
boolean	contains(Object o)
	Devuelve <i>true</i> si o está en la colección invocante.
boolean	containsAll(Collection<?> c)
	Devuelve <i>true</i> si la colección que invoca contiene todos los elementos de c.
boolean	isEmpty()
	Devuelve <i>true</i> si la colección no tiene elementos.
boolean	remove(Object o)
	Borra el objeto o de la colección que invoca; si no estuviera se devuelve <i>false</i> .
boolean	removeAll(Collection<?> c)
	Borra todos los objetos de la colección que invoca que estén en c. Devuelve <i>true</i> si la colección original se modifica.
boolean	retainAll(Collection<?> c)
	En la colección que invoca sólo se quedarán aquellos objetos que están en c. Por tanto, es la intersección entre ambas colecciones. Devuelve <i>true</i> si la colección original se modifica.
int	size()
	Devuelve el número de elementos.

El tipo List

Conceptualmente, las listas representan colecciones de elementos 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.



Desde el punto de vista de la API de Java, las listas son un subtipo de *Collection*: la interfaz *List* hereda de la interfaz *Collection*. Inicializar listas con varios elementos puede hacerse de las formas:

```
List<String> ls0 = Arrays.asList("A", "B", "C", "B", "B");
List<String> ls1 = List.of("A", "B", "C", "B", "B");
```

La lista *ls0* es mutable y la *ls1* inmutable.

Por otra parte las listas tienen todas las operaciones vistas para las colecciones, aunque debemos prestar atención a la semántica de algunas operaciones: La operación *addAll* añade los elementos de la lista que se pasa al final de la lista sobre la que se invoca. Si en la lista sobre la que se invoca hay elementos duplicados, la operación *removeAll* elimina de esta todas las instancias de los elementos que aparecen en la lista que se pasa. De manera análoga se comporta *retainAll*: si en la lista sobre la que se invoca un elemento aparece *n* veces, y este aparece en la lista que se pasa (independientemente del número de veces que aparezca), en la lista resultado permanecerán las *n* apariciones del elemento.

Veamos esto con un ejemplo. Si se ejecuta el siguiente código,



```

List<String> l1 = new ArrayList<String>();
List<String> l2 = new ArrayList<String>();
l1.add("A");
l1.add("B");
l1.add("C");
l2.add("B");
l2.add("B");
l1.removeAll(l2);
System.out.println("l1 después de l1.removeAll(l2): " + l1);
l1.clear();
l2.clear();
l1.add("A");
l1.add("B");
l1.add("C");
l2.add("B");
l2.add("B");
l2.removeAll(l1);
System.out.println("l2 después de l2.removeAll(l1): " + l2);
l1.clear();
l2.clear();
l1.add("A");
l1.add("B");
l1.add("C");
l2.add("B");
l2.add("B");
l2.retainAll(l1);
System.out.println("l2 después de l2.retainAll(l1): " + l2);

```

Se obtiene como salida lo siguiente:

```

l1 después de l1.removeAll(l2): [A, C]
l2 después de l2.removeAll(l1): []
l2 después de l2.retainAll(l1): [B, B]

```

La interfaz *List* aporta varias operaciones específicas, que no aparecen en *Collection*; en las operaciones que tienen índice, si no se cumple la restricción sobre este se eleva la excepción *IndexOutOfBoundsException*.

void	add(int index, E element)
------	---------------------------

Inserta el elemento especificado en la posición especificada. El que estaba previamente en la posición `index` pasará a la posición `index + 1`, el que estaba en la posición `index + 1` pasará a la



	posición <code>index + 2</code> , etc. Los valores lícitos para <code>index</code> son $0 \leq index \leq size()$.
<code>boolean</code>	<code>addAll(int index, Collection<? extends E> c)</code> Inserta todos los elementos de <code>c</code> en la posición especificada, desplazando el que estaba previamente en la posición <code>index</code> a la posición <code>index + c.size()</code> , etc. Los valores lícitos para <code>index</code> son $0 \leq index \leq size()$.
<code>E</code>	<code>get(int index)</code> Devuelve el elemento de la lista en la posición especificada. Los valores lícitos para <code>index</code> son $0 \leq index < size()$.
<code>int</code>	<code>indexOf(Object o)</code> Devuelve el índice donde se encuentra por primera vez el elemento <code>o</code> (si no está devuelve <code>-1</code>).
<code>int</code>	<code>lastIndexOf(Object o)</code> Devuelve el índice donde se encuentra por última vez el elemento <code>o</code> (si no estuviera devuelve <code>-1</code>).
<code>E</code>	<code>remove(int index)</code> Borra el elemento de la posición especificada. Los valores lícitos para <code>index</code> son $0 \leq index < size()$.
<code>E</code>	<code>set(int index, E element)</code> Reemplaza el elemento de la posición indicada por el que se da como argumento. Los valores lícitos para <code>index</code> son $0 \leq index < size()$.
<code>List<E></code>	<code>subList(int fromIndex, int toIndex)</code> Devuelve una vista de la porción de la lista entre <code>fromIndex</code> , inclusive, and <code>toIndex</code> , sin incluir. Los valores lícitos de <code>fromIndex</code> y <code>toIndex</code> son $0 \leq fromIndex \leq toIndex \leq size()$.

La operación `subList` devuelve una vista de la lista original. Esto quiere decir que las operaciones que se realicen sobre la sublista se verán reflejadas en la lista original y viceversa. Hay que prestar atención al hecho de que si se produce un cambio estructural que afecte al tamaño de



la lista original, pueden ocurrir comportamientos extraños, en particular la elevación de una excepción cuando se utiliza la sublist.

Veamos el comportamiento de *subList* con un ejemplo. Supongamos que se ejecuta el código:

```
List<String> ls = new LinkedList<String>();
ls.add("A");
ls.add("B");
ls.add("C");
ls.add("D");
ls.add("E");
List<String> subLs = ls.subList(1, 4);
System.out.println("Sublista: " + subLs);
ls.set(2, "F");
System.out.println(
"Sublista después de modificar el elemento 2 de la lista: "+
subLs);
subLs.remove(1);
System.out.println("Sublista después de eliminar el elemento
1: " + subLs);
System.out.println(
"Lista original después de modificar la sublist: " + ls);
subLs.add("X");
subLs.add("Y"); // añade "X" e "Y" al final de la sublist
System.out.println("Sublista después de añadir X e Y:
" + subLs);
System.out.println("Lista después de la modificación de
la sublist: " + ls);
ls.remove(0);
System.out.println("Lista después de eliminar el primer
elemento: " + ls);
```

El resultado que se obtiene como salida será el siguiente:



```
Sublista: [B, C, D]
Sublista después de modificar el elemento 2 de la lista:
[B, F, D]
Sublista después de eliminar el elemento 1: [B, D]
Lista original después de modificar la sublista: [A, B, D, E]
Sublista después de añadirle X e Y: [B, D, X, Y]
Lista después de la modificación de la sublista:
[A, B, D, X, Y, E]
Lista después de eliminar el primer elemento: [B, D, X, Y, E]
```

En este otro ejemplo, si se ejecuta el código:

```
List<String> ls = new LinkedList<String>();
ls.add("A");
ls.add("B");
ls.add("C");
ls.add("D");
ls.add("E");
List<String> subLs = ls.subList(1, 4);
System.out.println("Sublista: " + subLs);
ls.remove(2);
System.out.println("Sublista después de modificar la lista: "+
    subLs);
```

Se obtiene como salida:

```
Sublista: [B, C, D]
Exception in thread "main"
java.util.ConcurrentModificationException
```

Las implementaciones de las listas son *ArrayList* y *LinkedList*¹. La elección de una u otra implementación dependerá del tipo de operaciones que realicemos sobre ellas. Si se va a acceder preferentemente a los elementos mediante índice o a realizar búsquedas, debe usarse *ArrayList*. Si preferentemente se van a realizar operaciones de inserción o borrado al principio o al final debe usarse *LinkedList*. Las dos clases tienen dos constructores cada una: uno sin argumentos, que construye una lista vacía y otro que recibe una *Collection* y construye una lista con los



elementos de la colección, en el orden en el que un *for* extendido devolvería sus elementos.

Otras operaciones sobre listas

Como hemos visto anteriormente las listas en Java son un tipo mutable. Esto hace que tengan operaciones de modificación que devuelven algún tipo de datos distinto a la propia lista. En muchos casos, como hemos comentado, dejan la lista actualizada y devuelven un *Boolean*. Veamos otro conjunto de funciones sobre listas disponibles en el repositorio que devuelven una nueva lista cuando hacemos una operación de actualización. Esán disponible en la clase List2 del repositorio.

```
static <E> E first(List<E> ls) {
    Preconditions.checkNotNull(!ls.isEmpty(),
        "La lista no puede estar vacía");
    return ls.get(0);
}

static <E> E last(List<E> ls) {
    Preconditions.checkNotNull(!ls.isEmpty(),
        "La lista no puede estar vacía");
    int n = ls.size();
    return ls.get(n-1);
}

static <E> E removeLast(List<E> ls) {
    int last = ls.size()-1;
    E e = ls.remove(last);
    return e;
}

static <E> void addFirst(List<E> ls, E e) {
    ls.add(0,e);
}

static <E> List<E> of(E... elements) {
    List<E> r = new ArrayList<E>();
    r.addAll(Arrays.stream(elements).collect(Collectors.toList()));
    return r;
}
```



```
static <E> List<E> concat(List<E> ls1, List<E> ls2) {
    List<E> r = new ArrayList<>(ls1);
    r.addAll(ls2);
    return r;
}

static <E> List<E> difference(List<E> s1, List<E> s2) {
    List<E> s = new ArrayList<>(s1);
    s.removeAll(s2);
    return s;
}

static <E> List<E> union(List<E> s1, List<E> s2) {
    List<E> s = new ArrayList<>(s1);
    s.addAll(s2);
    return s;
}

static <E> List<E> intersection(List<E> s1, List<E> s2) {
    List<E> s = new ArrayList<>(s1);
    s.retainAll(s2);
    return s;
}

static <E> Boolean contains(List<E> s1, List<E> s2) {
    return s1.containsAll(s2);
}

static <E> String toString(List<E> c) {
    return c.stream()
        .map(e->e.toString())
        .collect(Collectors.joining("\n"));
}

static <E> String toString(List<E> c, String sep,
    String prefix, String suffix) {
    return c.stream()
        .map(e->e.toString())
        .collect(Collectors.joining(sep,prefix,suffix));
}
```

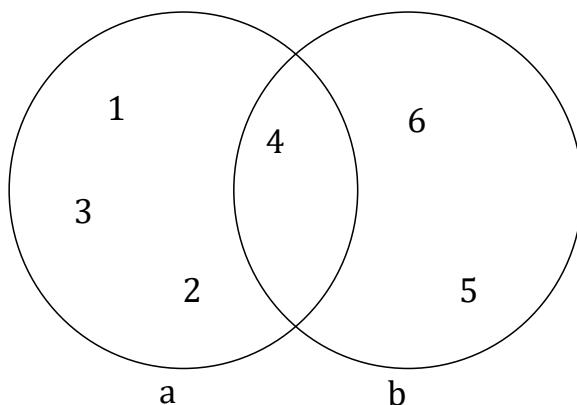


El tipo Set

El tipo Set 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.

Dados dos conjuntos (de Integer) a y b:

```
a = {1, 2, 3, 4}      b = {4, 5, 6}
```



recordemos que:

```
a ∪ b = {1, 2, 3, 4, 5, 6}
a - b = {1, 2, 3}
a ∩ b = {4}
```

La interfaz Set no aporta ningún método extra a los que ya tiene *Collection*; por tanto, sus métodos son los de *Collection* y solo estos. Las operaciones *addAll*, *retainAll* y *removeAll* se pueden identificar con la unión, intersección y diferencia de conjuntos, respectivamente; la operación *contains* equivale a la pertenencia en conjuntos (\in); la operación *containsAll* se corresponde con la de subconjunto (\subseteq).



La implementación más habitual del tipo *Set* es la clase *HashSet*². Tiene dos constructores: uno vacío, que construye un conjunto vacío, y otro que recibe una colección y construye un conjunto con los elementos de la colección (sin duplicados).

Inicializar conjuntos con varios elementos puede hacerse de las formas:

```
Set<String> s0 =  
    new HashSet<String>(Arrays.asList("A", "B", "C", "B", "B")) ;  
Set<String> s1 = Set.of("A", "B", "C", "B", "B") ;
```

El conjunto *s0* es mutable y el *s1* inmutable.

Como el conjunto es iterable el *for extendido* aplicado a un conjunto devuelve todos sus elementos en un orden que no es predecible. Igual sucede si se muestra el conjunto de una vez con *println*. Por ejemplo, si se ejecuta el código

```
Set<Character> s = new HashSet<Character>();  
s.add('A'); s.add('B'); s.add('P'); s.add('Q');  
System.out.println(s);
```

Se puede obtener una salida como la siguiente, aunque también podría ser cualquier otra combinación de los cuatro elementos:

```
[P, A, Q, B]
```

Otras operaciones sobre conjuntos

Como hemos visto anteriormente los conjuntos en Java son un tipo mutable. Esto hace que tengan operaciones de modificación que devuelven algún tipo de datos distinto a la propia lista. Veamos otro conjunto de funciones sobre conjuntos disponibles en el repositorio que devuelven una nuevo conjunto cuando hacemos una operación de actualización. Esán disponible en la clase *Set2* del repositorio.



```
static <E> Set<E> of(E... e) {
    return Arrays.stream(e).collect(Collectors.toSet());
}

static <E> Set<E> difference(Set<E> s1, Set<E> s2) {
    Set<E> s = new HashSet<E>(s1);
    s.removeAll(s2);
    return s;
}

static <E> Set<E> union(Set<E> s1, Set<E> s2) {
    Set<E> s = new HashSet<E>(s1);
    s.addAll(s2);
    return s;
}

static <E> Set<E> intersection(Set<E> s1, Set<E> s2) {
    Set<E> s = new HashSet<E>(s1);
    s.retainAll(s2);
    return s;
}

static <E> Boolean contains(Set<E> s1, Set<E> s2) {
    return s1.containsAll(s2);
}

static <E> String toString(Set<E> c) {
    return c.stream()
        .map(e->e.toString())
        .collect(Collectors.joining("\n"));
}

static <E> String toString(Set<E> c, String sep,
    String prefix, String suffix) {
    return c.stream()
        .map(e->e.toString())
        .collect(Collectors.joining(sep,prefix,suffix));
}
```



El tipo SortedSet

El tipo *SortedSet* es un subtipo de los conjuntos (por tanto los elementos no están indexados y no puede haber elementos repetidos), en el que existe una relación de orden entre los elementos que permite decir cuál va antes y cuál después. Para simplificar hablaremos de menores o mayores, entendiendo que significa “va antes o después”, según el criterio de ordenación.

La implementación de los conjuntos ordenados es la clase *TreeSet*. Esta clase tiene varios constructores:

- Un constructor vacío, que crea un conjunto ordenado vacío donde los elementos se ordenarán según su orden natural (los elementos tienen que implementar *Comparable*).
- Un constructor con un argumento de tipo *Comparator*, que crea un conjunto ordenado vacío cuyos elementos se ordenarán según el orden inducido por el comparador.
- Un constructor con un argumento de tipo *Collection*, que crea un conjunto ordenado con los elementos de la colección ordenados según su orden natural (los elementos de la colección tienen que implementar *Comparable*).
- Un constructor que recibe un *SortedSet*, que crea un conjunto ordenado con los mismos elementos que el que recibe como argumento y usando su mismo orden.

Inicializar conjuntos ordenados con varios elementos puede hacerse de las formas:

```
SortedSet<String> ss0 =
    new TreeSet<String>(Arrays.asList("A", "B", "C", "B", "B"));
```

El conjunto *ss0* es mutable.

El *for extendido* sobre los elementos de un *SortedSet* los devuelve en el orden que tienen inducido. Por ejemplo, dado el conjunto ordenado anterior, el código



```
SortedSet<Character> ss = new TreeSet<Character>();
ss.add('X');
ss.add('C');
ss.add('F');
ss.add('P');
ss.add('R');
ss.add('Q')
for (Character ch: ss) {
    System.out.println(ch);
}
```

Produce la siguiente salida:

```
C
F
P
Q
R
X
```

La clase de utilidad Collections

El paquete `java.util` contiene la clase de utilidad *Collections*. Esta clase está formada por métodos estáticos que permiten realizar operaciones sofisticadas sobre las colecciones, como invertir una lista, barajarla, ordenarla, buscar una sublistas dentro de una lista, encontrar el máximo o el mínimo de los elementos de una colección, contar las veces en las que aparece un elemento, etc.

Algunos de sus métodos (los más usados) son:

<code>static <T> boolean addAll(Collection<? super T> c, T... elements)</code>	Añade a la colección los elementos indicados en <code>elements</code> .
<code>static void fill(List<? super T> l, T o)</code>	Reemplaza todos los elementos de la lista <code>l</code> por <code>o</code> .



<pre>static <T extends Object & Comparable<? super T>> T</pre>	max (Collection<? extends T> coll) Devuelve el elemento máximo de la colección coll según el orden natural de sus elementos.
<pre>static <T extends Object & Comparable<? super T>> T</pre>	min (Collection<? extends T> coll) Devuelve el elemento mínimo de la colección coll según el orden natural de sus elementos.
static void	reverse (List<?> list) Invierte los elementos de la lista list.
static void	shuffle (List<?> list) Mezcla aleatoriamente los elementos de la lista list.
<pre>static <T extends Comparable<? super T>> void</pre>	sort (List<T> list) Ordena la lista según el orden natural del tipo.

Veamos un ejemplo de uso de la clase Collections. Si se ejecuta el código

```
List<String> l = new LinkedList<String>();  
l.add("R");  
l.add("T");  
l.add("B");  
l.add("A");  
l.add("M");  
System.out.println(l);  
Collections.reverse(l);  
System.out.println(l);  
Collections.sort(l);  
System.out.println(l);  
Collections.fill(l, "X");  
System.out.println(l);
```

La salida será:



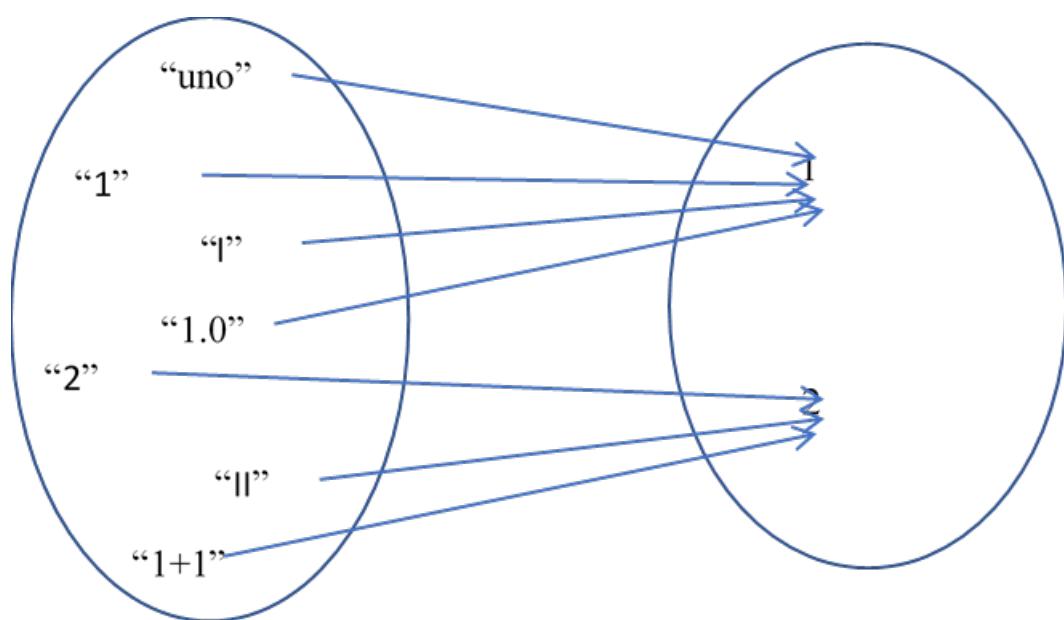
```
[R, T, B, A, M]
[M, A, B, T, R]
[A, B, M, R, T]
[X, X, X, X, X]
```

El tipo Map

Definición

El tipo de dato *Map*, incluido en el paquete `java.util`, 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*).

En la figura puede verse un ejemplo de *Map*. En este caso las claves son cadenas de caracteres, cada una de las cuales representa un valor numérico en diversas representaciones textuales, y los valores son enteros. Cada cadena tiene asociada el valor numérico correspondiente.



Este mismo ejemplo se podría representar como una tabla con dos columnas:

Clave	Valor
“uno”	1
“1”	1
“I”	1
“1.0”	1
“2”	2
“II”	2
“1 + 1”	2

Vemos que la tabla refleja la misma información que el diagrama anterior.

Como entre las claves no puede haber elementos duplicados (una clave no puede estar asociada con más de un valor), las claves forman un conjunto (*Set*). Sin embargo sí que puede haber valores duplicados, por los que estos están en una *Collection*.

Por tanto, un *Map* está definido por un conjunto de claves, una colección de valores y un conjunto de pares clave-valor (también llamadas *entradas*), que son realmente los elementos de los que está compuesto.

Dentro de la jerarquía de tipos de Java, *Map* no extiende a *Collection* ni a *Iterable*. Por tanto, no se puede aplicar un *for* extendido sobre sus elementos, es decir, sobre sus pares, salvo que accedamos explícitamente a ellos. Los *Map* se utilizan en muchas situaciones. Por ejemplo, los listines telefónicos (clave: nombre del contacto, valor: número de teléfono), listas de clase (clave: nombre del alumno, valor: calificaciones), ventas de un producto (clave: código de producto, valor: total de euros de recaudación), índices de palabras por páginas en un libro (clave: palabra, valor: lista de números de página donde aparece), la red de un Metro (clave: nombre de la estación, valor: conjunto de líneas que pasan por esa estación, o al revés, clave: número de línea, valor: lista de estaciones de



esa línea), etc. Tiene también muchas otras aplicaciones en informática: representar diccionarios o propiedades, almacenar en memoria tablas de bases de datos, cachés, etc.

Centrándonos en Java, la interfaz *Map* tiene dos tipos genéricos, que suelen denominarse K (de Key) y V (de Value): *Map<K, V>*. El *Map* del ejemplo anterior sería *Map<String, Integer>*.

La clase que implementa la interfaz *Map* es *HashMap* (aunque también se puede utilizar *TreeMap*, que se verá más adelante). La clase *HashMap* obliga a definir de forma correcta el método *hashCode* del tipo de las claves para evitar comportamientos incorrectos, como claves repetidas. Asimismo, al igual que pasa en el tipo *Set*, los objetos que se introducen en un *Map* son “vistas” o referencias a objetos y, por tanto, si estos cambian, el *Map* puede dejar de ser coherente. En general, deben usarse claves que sean tipos inmutables.

Un *Map* cuyas claves sean *String* y cuyos valores sean *Integer* se definiría e inicializaría de la siguiente forma:

```
Map<String, Integer> m = new HashMap<String, Integer>();
```

Inicializar maps con varios elementos puede hacerse de las formas:

```
Map<String, Integer> m0 =
    New HashSet<String>(
        Map.of("A", 24, "B", 22, "C", 33, "F", 55));
Map<String, Integer> m1 = Map.of("A", 24, "B", 22, "C", 33, "F", 55);
```

El *m0* es mutable y el *m1* inmutable.

Métodos del tipo Map

void	<code>clear()</code> Elimina todos los elementos (pares o entradas) del Map.
boolean	<code>containsKey(Object key)</code> Devuelve <code>true</code> si el Map contiene la clave especificada.



boolean	containsValue(Object value) Devuelve <code>true</code> si una o más claves del Map tienen asociadas el valor especificado.
V	get(Object key) Devuelve el valor asociado con la clave especificada o <code>null</code> si esa clave no está asociada con ningún valor (la clave no está en el conjunto de claves).
boolean	isEmpty() Devuelve <code>true</code> si el Map no contiene ningún par.
Set<K>	keySet() Devuelve un Set que es una vista de las claves que contiene el Map.
V	put(K key, V value) Asocia el valor con la clave especificada. Devuelve el valor previamente asociado con la clave si esta ya estaba en el Map o <code>null</code> , en caso contrario.
V	remove(Object key) Elimina el par que tiene como clave el parámetro especificado. Devuelve el valor previamente asociado con la clave, o <code>null</code> si la clave no existía.
int	size() Devuelve el número de pares del Map.
Collection<V>	values() Devuelve una Collection que es una vista de los valores del Map.
Set<Map.Entry<K,V>>	entrySet() Devuelve una vista del conjunto de todos los pares del Map (el tipo Map.Entry se verá más adelante).



void	putAll(Map<? extends K, ? extends V> m) Añade al Map todos los pares contenidos en m. Tiene el mismo efecto que hacer put de todos los elementos de m.
------	---

Los métodos del tipo Map se relacionan a continuación. La información completa sobre este tipo se puede encontrar en la API de Java.

Tanto los conjuntos devueltos por *keySet* y *entrySet* como la colección devuelta por *values* son vistas del Map original, por lo que las modificaciones que se realicen sobre estos repercuten en los pares almacenados en el Map original (con las posibles repercusiones negativas) y viceversa. Hay que destacar que esos tres objetos son iterables. El orden en el que el iterador recorre los elementos de los conjuntos (*keySet* y *entrySet*) o la colección (*values*) es impredecible.

La clase HashMap tiene dos constructores: el constructor sin argumentos, que construye un Map vacío, y el constructor con un argumento de tipo Map (constructor copia), que construye un Map con los mismos pares que el Map que se le pasa como argumento (equivale a crear un Map vacío y aplicar *putAll*). Existe una clase LinkedHashMap, que se comporta igual que HashMap excepto en que los iteradores sobre las claves, los valores o los pares, los devuelven en el orden en que se insertaron.

La operación *equals* entre dos maps devuelve true si y solo si los conjuntos devueltos por *entrySet* en ambos conjuntos (sus pares) son iguales.

El tipo Map.Entry

Los pares de elementos (también llamados entradas) de los que está compuesto un Map<K, V> son de un tipo que viene implementado por la interfaz Map.Entry<K, V>. Esta interfaz, aparte de la operación equals que permite comparar pares para comprobar su igualdad (y el correspondiente hashCode), tiene tres operaciones:



K	<code>getKey()</code> Devuelve la clave del par.
V	<code>getValue()</code> Devuelve el valor del par.
V	<code>setValue(V value)</code> Modifica el valor del par, dándole como nuevo valor <code>value</code> . Devuelve el valor (segunda componente) previo del par.

Por ejemplo, si tenemos una entrada (par) p que asocie "II" con 2, `p.getKey()` devuelve la cadena "II" y `p.getValue()` devuelve el entero 2; si se escribe `p.setValue(3)`, devuelve el entero 2 y modifica el valor asociado con la clave dándole el valor 3. El `toString` de las entradas es de la forma clave=valor, de modo que la representación como cadena del par resultante sería II=3.

Otras operaciones sobre maps

Como hemos visto los objetos de tipo `Map<K,V>` en Java son un tipo mutable. Como en el caso de listas y conjuntos algunas operaciones de modificación devuelven algún tipo de datos distinto al propio map. Veamos otro conjunto de funciones sobre maps disponibles en el repositorio que devuelven un nuevo map cuando hacemos una operación de actualización. Están disponible en la clase `Map2` del repositorio.



```
static <V,E> String toString(Map<E,V> m) {  
    return m.keySet().stream()  
        .map(key->String.format(  
            " (%s,%s)",key,m.get(key)))  
        .collect(Collectors.joining("\n"));  
  
}  
  
static <V,E> String toString(Map<E,V> m, String sep,  
    String prefix, String suffix) {  
    return m.keySet().stream()  
        .map(key->String.format(  
            " (%s,%s)",key,m.get(key)))  
        .collect(Collectors.joining(sep,prefix,suffix));  
  
}  
static <K,V> Map<V,List<K>> invert(Map<K,V> m) {  
    return m.keySet().stream()  
        .collect(Collectors  
            .toMap(x->m.get(x),x->List.of(x),List2::union));  
}  
  
static <K, V> Map<K, V> of(Entry<? extends K,  
    ? extends V>... entries) {  
    Map<K, V> result = new HashMap<>(entries.length);  
    for (Entry<? extends K, ? extends V> entry : entries)  
        if (entry.getValue() != null)  
            result.put(entry.getKey(), entry.getValue());  
    return result;  
}
```



```
static <K,V> Map<K,V> of(K key,V value) {
    Map<K,V> m = new HashMap<>();
    m.put(key,value);
    return m;
}

static <K,V> Map<K,V> of(K key1,V value1,K key2,V value2) {
    Map<K,V> m = new HashMap<>();
    m.put(key1,value1);
    m.put(key2,value2);
    return m;
}

static <K,V> Map<K,V> of(K key1,V value1,K key2,V value2,K key3,V value3) {
    Map<K,V> m = new HashMap<>();
    m.put(key1,value1);
    m.put(key2,value2);
    m.put(key3,value3);
    return m;
}

static <K, V> Map<K, V> merge(Map<K,V> m1, Map<K,V> m2) {
    Map<K, V> r = new HashMap<>(m1);
    r.putAll(m2);
    return r;
}

static <K, V> Entry<K,V> entry(K key, V value) {
    return new AbstractMap.SimpleEntry<>(key,value);
}
```

El tipo SortedMap

El tipo **SortedMap** es un subtipo de **Map** en el que el conjunto de las claves está ordenado. La clase que implementa **SortedMap** es **TreeMap**. Es necesario que el tipo de las claves tenga un orden natural (es decir, que implemente **Comparable**), o bien que se indique el orden mediante un comparador.



Por ejemplo, una inicialización de SortedMap como la siguiente:

```
SortedMap<String, List<Integer>> indice =new  
TreeMap<String, List<Integer>>();
```

crea un *SortedMap* donde las claves son de tipo String y están ordenadas por su orden natural.

Inicializar *sortedmaps* con varios elementos puede hacerse de las formas:

```
Comparator<String> cmp = ...  
SortedMap<String, Integer> sm2= new HashTree<String>(cmp);  
sm2.putAll(Map.of("A",24,"B",22,"C",33,"F",55));
```

El *sm2* es mutable. La clase *TreeMap* tiene varios constructores:

- Sin argumentos: construye un *SortedMap* vacío que utilizará el orden natural de las claves.
- Con un argumento de tipo *Comparator*: construye un *SortedMap* vacío que utilizará el orden inducido por el comparador.
- Con un argumento de tipo *Map*: construye un *SortedMap* con los mismos pares que el *Map* que recibe como argumento, pero donde las claves estarán ordenadas según el orden natural de estas.

Con un argumento de tipo *SortedMap*: construye un *SortedMap* con los mismos elementos que el que recibe como argumento y ordenado según el mismo criterio (sea el natural o uno inducido). Es el constructor copia.



El tipo Stream

En este apartado se explica la forma de realizar tratamientos secuenciales en Java 8 y posteriores. Se introduce el tipo Stream, sus métodos y las interfaces funcionales que se utilizan en la llamada a los mismos.

Streams

Java 8 introduce una nueva y potente forma de realizar operaciones con agregados de datos. Un concepto fundamental es el de *Stream*. Se podría traducir como *flujo de datos*. Es un concepto similar al iterable que ya hemos visto anteriormente pero con algunas diferencias importantes.

Tanto el iterable como el stream son vistas de un agregado de datos. Este puede tener varias vistas en forma de iterable o de stream. Ambos son mecanismos para recorrer uno tras otro, el algún orden, los elementos del agregado sin estar obligados a tener todo el agregado en memoria.

El iterable permite llevar a cabo una programación secuencial imperativa. Es decir combinando el *for* extendido, *if*, *while* y sentencias de asignación.

Un stream permite hacer programación secuencial o paralela con un estilo de programación funcional. Es decir usando un conjunto de métodos, funciones, para obtener el resultado sin usar el *for*, *if*, *while*, *asignación* imperativos.

Un iterable puede ser convertido en un stream y viceversa.



Un primer ejemplo

Vamos a recordar cómo hemos realizado los tratamientos secuenciales con iterables hasta ahora, y vamos a ver cómo se hace con el uso de streams. Para ello tomemos un ejemplo.

Supongamos que tenemos una lista de vuelos y queremos obtener el número de ellos que parten en una fecha determinada. Se trata de un tratamiento secuencial de tipo ‘contador’. El problema se resolvería así:

```
public Integer numeroVuelosSalidaFecha(LocalDate fecha) {
    Integer num = 0;
    for (Vuelo v : getVuelos()) {
        if (v.getFechaSalida().equals(fecha)) {
            num++;
        }
    }
    return num;
}
```

Ahora veamos cómo se hace mediante streams:

```
public Long numeroVuelosSalidaFecha(LocalDate fecha) {
    return getVuelos().stream()
        .filter(x -> x.getFechaSalida().equals(fecha))
        .count();
}
```

Analicemos los elementos que vemos aquí.

En primer lugar tenemos una llamada al método *stream*, que convierte la lista en un objeto de tipo *Stream*. Este es un método de factoría que tiene el agregado de tipo lista para ofrecer un stream. Cada agregado de datos tendrá un método similar. Este será siempre el primer paso obtener un stream a partir de un agregado de datos.

```
getVuelos().stream()
```

En segundo lugar se aplica una llamada al método *filter*, que filtra los elementos de la lista que queremos contar. Produce un stream a partir de otro filtrando los elementos adecuados y toma como parámetro un predicado expresado como una lambda expresión. Un predicado es una función que produce como resultado un valor de tipo *Boolean*. El tipo de



esta función se describe en Java mediante una interface funcional. Este se llama *Predicate<E>*.

```
.filter(x -> x.getFechaSalida().equals(fecha))
```

En este caso son los vuelos que tienen una fecha de salida determinada. Para indicar esto, utilizamos una expresión lambda.

```
x -> x.getFechaSalida().equals(fecha)
```

Finalmente, se hace una llamada al método *count*, que cuenta el número de elementos de la lista filtrada. Este método acumula los valores proporcionados por el stream en un valor.

```
.count();
```

Como vemos, en primer lugar se convierte el agregado en un stream, y a partir de ahí se trata de ir realizando operaciones una tras otra en secuencia sobre el stream. Cada operación se realiza sobre el resultado de la anterior. En el ejemplo, primero se filtran los vuelos con una fecha de salida determinada y luego se cuentan. Este estilo, cuando se aplican métodos al resultado de otros, se denomina *fluente*.

Han aparecido varios conceptos que ahora explicaremos con más detalle: estilo fluente, lambda expresiones, stream, factorías de streams, interfaces funcionales.

Un segundo ejemplo

Veamos un nuevo ejemplo. En este caso, queremos obtener el número de vuelos que tienen un destino concreto. Nuevamente se trata de un tratamiento secuencial de tipo ‘contador’. Según el esquema visto para este tratamiento, sería así:



```
public Integer numeroVuelosDestino(String destino) {  
    Integer num = 0;  
    for (Vuelo v : getVuelos()) {  
        if (v.getDestino().equals(destino)) {  
            num++;  
        }  
    }  
    return num;  
}
```

Ahora veamos cómo sería mediante streams

```
public Long numeroVuelosDestino(String destino) {  
    return getVuelos().stream()  
        .filter(x -> x.getDestino().equals(destino))  
        .count();  
}
```

Se podrían hacer más operaciones en cascada sobre el stream (de manera fluente). Por ejemplo, filtrar los vuelos que salen en una fecha, a continuación quedarse con los que tienen un destino determinado, y por último contarlos. Así podríamos responder por ejemplo a la pregunta ¿Cuántos vuelos salen hoy hacia París?

```
public Long numeroVuelosSalidaFechaYDestino(LocalDate fecha,  
    String destino) {  
    return getVuelos().stream()  
        .filter(x->x.getFechaSalida().equals(fecha))  
        .filter(x->x.getDestino().equals(destino))  
        .count();  
}
```

Ejercicio: responder a la pregunta ¿Cuántos vuelos con plazas libres salen hoy hacia París? Solución: filtrar los vuelos que salen en una fecha, a continuación quedarse con los que tienen un destino determinado, luego descartar los que están completos, y por último contarlos.

El esquema es pues siempre el mismo. Dependiendo de la operación que queramos hacer utilizaremos los métodos correspondientes, que iremos viendo a lo largo del tema.



El tipo Stream

Un agregado de elementos es virtual cuando está definido implícitamente. Por ejemplo el rango de elementos de 10 a 100 de 2 en 2. Este conjunto de elementos está bien definido pero no tiene porqué tener todos sus elementos en memoria: es un agregado virtual.

Según hemos visto, un stream es una vista de un agregado de elementos, que puede ser real o virtual, y sirve para recorrer uno tras otro los elementos de este. Es decir un stream es un flujo de datos, es un orden adecuado, procedente de un agregado.

A partir Java 8 existe el tipo *Stream<T>* para representar un flujo de datos. Este tipo, como todos los tipos, tiene métodos de factoría, operaciones que transforman un stream en otro (como *filter*), y otras que acumulan los valores del stream como *count*. Las que transforman un stream en otro se pueden ser aplicadas de manera fluente. Las que acumulan un resultado se aplican al final. Ambas pueden llevar a cabo tratamientos secuenciales o paralelos.

Veremos ahora los diferentes métodos disponibles para el tipo *Stream<T>*.



Interfaces funcionales y lambda expresiones

Para trabajar con *streams* se usan un estilo de programación funcional y fluente. En este estilo es muy usual pasar funciones como parámetros. Estas funciones, como otros parámetros, deben tener un tipo. En Java se usan *interfaces funcionales* para especificar el tipo de una función. Una interface funcional es una interface con un solo método individual aunque puede tener varios métodos estáticos o default. La firma de ese método individual especifica el tipo de una función: es decir un *tipo funcional*. Al igual que los tipos numéricos se inicializan con valores los tipos funcionales se inicializan con *lambda expresiones*.

Una lambda expresión es la forma de codificar un método anónimo. Cada lambda expresión tiene un tipo funcional y puede ser asignada a una variable declarada de una interface funcional compatible. Una expresión lambda se escribe de la forma de una de las formas:

```
x->x.exp;  
(x,y)-> {exp1; exp2; return exp3;};
```

Como se ve tienen uno o varios parámetros (o ninguno) y un cuerpo donde calculan un resultado a partir de los parámetros. Parámetros y resultado están separados por `->`. iremos viendo ejemplos en lo que sigue. Es importante saber encontrar en tipo funcional de una lambda expresión.

Sea la interface funcional:



Interfaces funcionales y lambda expresiones

```
public interface MiInterface {
    public Integer calcula(Integer a1, Integer a2);
}
```

Con el podemos declarar e inicializar la variable:

```
MiInterface v = (a, b) -> 2*a+b;
```

Tanto la interface funcional como la lambda expresión tienen tipo funcional $\text{int} \times \text{int} \rightarrow \text{int}$ por lo que la asignación es posible. Podemos hacer el cálculo:

```
Integer r = v.calcula(23, 45);
```

Java introduce el *operador llamada a método* (`::`) para simplificar la escritura de lambda expresiones. Siendo T un tipo, me un método estático del mismo, mi un método individual y obj un objeto tenemos la equivalencia de este operador con en expresiones lambda:

```
T::me   ≡ (x, y) -> T.me(x, y)
obj::mi ≡ (x, y) -> x.mi(y)
```

Los interfaces funcionales tienen nombres que vamos a ir aprendiendo. Veamos las interfaces funcionales más conocidas.

La interfaz funcional Predicate

Es una interface que define el tipo funcional $E \rightarrow \text{Boolean}$ y es de la forma:

```
public interface Predicate<T> {
    boolean test(T o);
    default Predicate<T> and(Predicate<? super T> p);
    default Predicate<T> negate();
    default Predicate<T> or(Predicate<? super T> p);
}
```

Vemos que incluye algunos métodos estáticos para combinar predicados como *and*, *or* y *not*. Ejemplo:



```
Predicate<Fraction> p =
    f->f.getNumerador() > f.getDenominador();
```

La interfaz Function y Bifunction

El tipo funcional definido por *Function<E,R>* es $E \rightarrow R$ y su interface:

```
public interface Function<E, R> {
    R apply(E o);
    default <V> Function<E, V> andThen(
        Function<? super R, ? super V> after);
    default <V> Function<V, R> compose(
        Function<? super V, ? super E> before);
}
```

Ejemplo:

```
Function<String, Punto2D> f = Punto2D::parse;
```

Los métodos *compose* y *andThen* de la interfaz *Function* permiten componer dos funciones una tras otra. Así, *f1.andThen(f2)* construye una nueva función que aplica sucesivamente *f1* y luego *f2* a los resultados obtenidos. Por su parte, *f1.compose(f2)* construye una nueva función que aplica sucesivamente *f2* y luego *f1* a los resultados obtenidos.

El interface funcional *Bifunction<E,U,R>* tiene el tipo funcional $E \times U \rightarrow R$

```
public interface BiFunction<E, U, R> {
    R apply(E e, U u);
}
```

Las interfaces Unaryoperator y Binaryoperator

UnaryOperator<E> y *BinaryOperator<E>* son casos particulares de *Function* y *Bifunction*. Tienen los tipos funcionales $E \rightarrow E$ y $E \times E \rightarrow E$ respectivamente y los interfaces funcionales



```
public interface UnaryOperator<E> {
    E apply(E e);
    static <T> UnaryOperator<T> identity();
}

public interface BinaryOperator<E> {
    E apply(E e1, E e2);
    static <E> BinaryOperator<T> maxBy(
        Comparator<? super T> comparator);
    static <E> BinaryOperator<T> minBy(
        Comparator<? super T> comparator);
}
```

Ejemplos:

```
UnaryOperator<Integer> uo = e->e+1;
BinaryOperator<Integer> bo = (a,b)-> BinaryOperator
    .minBy(Comparator.naturalOrder());
```

La interfaz Consumer y BiConsumer

La interfaz funcional *Consumer* tiene asociado el tipo funcional $E \rightarrow void$ y *BiConsumer* $E \times U \rightarrow void$. Su objetivo es realizar una acción sobre el objeto, o par de objetos, sin devolver ningún valor.

```
public interface Consumer<E> {
    void accept(E e);
}

public interface BiConsumer<E, U> {
    void accept(E e, U u);
}
```

Ejemplo:

```
Consumer<Punto2D> c = c->System.out.println(c);
Consumer<Punto2D> c = System.out::println;
```

La interfaz Supplier

La interfaz funcional *Supplier* tiene asociado el tipo funcional $() \rightarrow E$. Es el tipo funcional adecuado para representar los métodos de factoría de un tipo.



```
public interface Supplier<E> {  
    E get();  
}
```

Ejemplo:

```
Supplier<Integer> s = Random::nextInt;
```

Los interfaces Comparable y Comparator: orden natural y ordenes alternativos

Hemos visto cómo definir un orden natural en un tipo. La interfaz del tipo T debe extender a la interfaz $Comparable<T>$, y en la clase hay que implementar el método $compareTo$. Este método es un método individual:

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

La interface funcional $Comparator<T>$ representa el tipo funcional $T \times T \rightarrow int$. $Comparable$ y $Comparator$ están pensados para representar órdenes: el primero se concreta mediante un método individual y representa el orden natural del tipo si lo tiene, el segundo representa un orden alternativo y se concreta mediante lambda expresiones normalmente o métodos ofrecidos por la propia interface. La interface $Comparator$ incluye los métodos estáticos siguientes:



```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
    static <T,U extends Comparable<? super U>> Comparator<T>  
        comparing(Function<? super T,>? extends U>  
            keyExtractor);  
    static <T extends Comparable<? super T>> Comparator<T>  
        naturalOrder();  
    default Comparator<T> reversed();  
    static <T extends Comparable<? super T>> Comparator<T>  
        reverseOrder();  
    default Comparator<T>  
        thenComparing(Comparator<? super T> other);  
    . . .  
}
```

Métodos de Comparator<T>:

- *comparing(Function<? super T,>? extends U> keyExtractor):* Genera un orden definido por los valores de la función *keyExtractor* aplicada a los objetos de tipo T.
- *naturalOrder():* El orden natural del tipo que debe tenerlo para usar este método.
- *reverseOrder():* El inverso del orden natural del tipo que debe tenerlo para usar este método.
- *thenComparing(Comparator<? super T> other):* Combina dos órdenes aplicando primero uno y luego otro.

Ejemplos:

```
Comparator<Vuelo> comparadorVueloDestino =  
    Comparator.comparing(Vuelo::getDestino);  
Comparator<Vuelo> comparadorVueloOcupacion =  
    Comparator.comparing(  
        x -> 100.  
            * x.getNumPasajeros() / x.getNumPlazas());
```



Métodos de factoría de streams

Los streams, como el resto de los objetos se crean con factorías. Las más relevantes son:

- `IntStream IntStream.range(a,b)`: Stream de int en un rango.
- `LongStream LongStream.range(a,b)`: Stream de long en un rango.
- `IntStream string.chars()`: Stream con los valores enteros de los caracteres del string
- `Stream<E> collection.stream()`: Stream con los objetos de la colección.
- `Stream<E> Arrays.stream(E[] elements)`: Stream con los objetos del array
- `Stream<E> Stream.iterate(E initialValue, Predicate<E> hasNext, UnaryOperator<E> next)`: Stream cuyos elementos se forman a partir de uno primero al que se aplica un operador binario mientras que se cumpla el predicado.
- `Stream<String> Files.lines(Path file)`: Un Stream con las líneas de un fichero



```
class FileTools {  
    ...  
    public static Stream<String> fromFile(String file) {  
        Stream<String> r = null;  
        try {  
            r = Files.lines(Paths.get(file),  
                           Charset.defaultCharset());  
        } catch (IOException e) {  
            throw new IllegalArgumentException(  
                "No se ha encontrado el fichero " + file);  
        }  
        return r;  
    }  
}
```

IntStream, *LongStream* y *DoubleStream* tienen disponibles los métodos aritméticos *sum* y *average* que no están disponibles es Stream.

El método *boxed()* de *IntStream*, *LongStream* y *DoubleStream* las transforma en *Stream<Integer>*, *Stream<Long>* y *Stream<Double>* respectivamente.



Operaciones sobre streams

Las operaciones sobre *streams* pueden ser de dos tipos: operaciones intermedias (o transformadoras y terminales (o acumuladoras). Las operaciones intermedias producen un stream a partir de otro. Es el caso, por ejemplo, de la operación *filter*. Las operaciones terminales producen un objeto que no es de tipo stream. Por ejemplo, el número de elementos del stream devuelto por el método *count*.

Hay muchas operaciones aplicables sobre *streams*. En este tema veremos las siguientes:

- Operaciones intermedias:
 - `distinct`
 - `filter`
 - `flatMap`
 - `map`
 - `sorted`
- Operaciones terminales:
 - `allMatch`
 - `anyMatch`
 - `average`
 - `collect`
 - `count`
 - `forEach`
 - `max`
 - `min`
 - `sum`



Cada uno de estos métodos puede tener como parámetro un objeto de un tipo funcional.

Métodos transformadores

distinct():

Obtiene un nuevo stream sin elementos repetidos

sorted(Comparator<T>)

El método *sorted* permite, como su nombre indica, ordenar un stream. Para indicar el orden le pasamos como parámetro un comparador. Por ejemplo, supongamos que queremos ordenar los vuelos según su fecha de salida. Lo haríamos de la siguiente forma:

```
getVuelos().stream()
    .sorted(Comparator.comparing(Vuelo::getFechaSalida));
```

map(Function<T,R>)

El método *map* obtiene un stream de elementos de un tipo a partir de un stream de elementos de otro tipo. Para ello recibe como parámetro la función que transforma los objetos de un tipo en otro.

En realidad existe una familia de métodos *map*, según el tipo de los objetos del stream de salida. Así, existen los métodos *mapToInt*, *mapToLong*, *mapToDouble*. Estos producen como resultado streams de tipos específicos *IntStream*, *LongStream*, *DoubleStream* que tienen métodos específicos.

Veamos como ejemplo un método que obtiene el número total de pasajeros de los vuelos que salen en una fecha determinada.

```
public IntStream numPasajerosVuelosFecha(LocalDate fecha) {
    return getVuelos().stream()
        .filter(x -> x.getFechaSalida().equals(fecha))
        .mapToInt(x -> x.getNumPasajeros());
}
```

flatMap(Function<T,Stream<R>)



El método *flatMap* transforma cada elemento de un stream en otro stream y luego concatena los resultados en un nuevo stream.

Veámoslo con un ejemplo: queremos obtener una lista con todos los pasajeros de los vuelos que salen en una fecha dada. El código sería el siguiente:

```
Stream<Pasajero> pasajerosVuelosFecha(LocalDate fecha) {
    return getVuelos().stream()
        .filter(x -> x.getFechaSalida().equals(fecha))
        .flatMap(x -> x.getPasajeros().stream());
}
```

Métodos acumuladores

Aritmeticos: sum, average, count

Los métodos *sum*, *average* y *count* calculan la suma, la media y el número de elementos. Los dos primeros están disponibles para *IntStream*, *LongStream* y *DoubleStream*, el tercero para todo tipo de stream.

Lógicos: *allMatch(Predicate)*, *anyMatch(Predicate)*, *noneMatch(Predicate)*

El método *allMatch* obtiene true si todos los elementos del stream cumplen el predicado, *anyMatch* obtiene true si algún elemento del stream cumplen el predicado y *noneMatch* obtiene true si ninguno de los elementos del stream cumplen el predicado,

Con este método podríamos responder, por ejemplo, a la pregunta ¿Todos los vuelos que salen en una fecha dada están completos? Lo haríamos así:

```
public Boolean todosVuelosFechaCompletos(LocalDate fecha) {
    return getVuelos().stream()
        .filter(x -> x.getFechaSalida().equals(fecha))
        .allMatch(x->x.getNumPlazas()
            .equals(x.getNumPasajeros()));
}
```

min(Comparator), *max(Comparator)*



El método *min*, *max* devuelven respectivamente el elemento mínimo y el máximo de los elementos de un stream de acuerdo con el orden definido por un comparador, que recibe como parámetro. Por ejemplo, para obtener el vuelo con un destino dado que sale más pronto, haríamos lo siguiente:

```
public Vuelo primerVueloDestino(String destino) {
    return getVuelos().stream()
        .filter(x->x.getDestino().equals(destino))
        .min(Comparator.comparing(Vuelo::getFechaSalida))
        .get();
}
```

Hay que tener en cuenta que podría no existir un mínimo, algo que sucedería en el caso de que el stream estuviese vacío, como si en el ejemplo no hubiese ningún vuelo con el destino indicado. Por eso el método *min*, al igual que *max* y algún otro método, devuelve un objeto de un tipo *Optional<T>*. A este objeto le aplicamos el método *get*, que devuelve el valor del objeto si este existe, y en caso contrario lanza la excepción *NoSuchElementException*.

Si queremos evitar el lanzamiento de la excepción, utilizamos el método *orElse*. Este método tiene como parámetro el valor que queremos devolver en el caso de que no se encuentre un mínimo. Sería de esta manera:

```
public Vuelo primerVueloDestino(String destino) {
    return getVuelos().stream()
        .filter(x->x.getDestino().equals(destino))
        .min(Comparator.comparing(Vuelo::getFechaSalida))
        .orElse(null);
}
```

Reducción: *reduce(BinaryOperator)*, ...

Los métodos de reducción permiten acumular los elementos de un stream mediante un operador binario. Hay tres variantes:

- *Optional<T> reduce(BinaryOperator<T> b)*: Acumula el primer elemento con los sucesivos mediante un operador binario. El resultado puede ser Optional si el stream está vacío.



- $T \text{ reduce}(T a0, \text{BinaryOperator} < T > b)$: Acumula un valor inicial $a0$ con los sucesivos mediante un operador binario.
- $A \text{ reduce}(A a0, \text{Function} < A, T > f, \text{BinaryOperator} < A > b)$: Acumula un valor inicial $a0$ mediante un operador binario con los sucesivos elementos transformados por la función f .

Por ejemplo la suma de los cuadrados de un stream es:

```
stream.reduce(0, e->e*e, (a, a)->a+a);
```

El método collect y la interfaz Collector

El método *collect* proporciona un mecanismo para acumular un stream mediante un acumulador. Un *acumulador* es un objeto de tipo **Collector** de Java.

Casos particulares de acumuladores son *toList* y *toSet*:

```
List<Integer> numPasajerosVuelosFecha(LocalDate fecha) {
    return getVuelos().stream()
        .filter(x -> x.getFechaSalida().equals(fecha))
        .map(x -> x.getNumPasajeros())
        .collect(Collectors.toList());
}
```

La clase *Collectors* es, en definitiva, una factoría de acumuladores. Los detalles de como se implementa un acumulador (*Collector*) se verán en cursos posteriores.

Acumuladores conocidos aportados por la factoría *Collectors* están *summing*, *averaging*, *joining*, *reducing*, etc.

Esquemas de agrupación: *groupingBy(Function)*,
partitioningBy(Predicate)

Hay un conjunto de esquemas para definir grupos con los elementos de un stream. Los grupos se definen mediante un *Map* donde para grupo de elementos está asociado a una clave. Estos esquemas se basan en el acumulador *groupingBy*. Este acumulador tiene como parámetro una función (*clave*) que define los grupos. Dos objetos estarán en el mismo grupo si la clave devuelve el mismo valor aplicada a ellos. Si en vez de una función aportamos un predicado entonces se definen dos grupos asociados a los dos valores booleanos y el correspondiente acumulador se llama *partitioningBy*.



Basado en el acumulador *groupingBy* podemos definir los siguientes esquemas de agrupación:

```
Map<K, List<E>> groupingList(Stream<E> st, Function<E, K> key) {
    return st.collect(
        Collectors.groupingBy(Function<E, K> key));
}

Map<K, Set<E>> groupingSet(Stream<E> st, Function<E, K> key) {
    return st.collect(
        Collectors.groupingBy(
            Function<E, K> key, Collectors.toSet()));
}

SortedMap<K, List<E>> groupingListSorted(Stream<E> st,
    Function<E, K> key,
    Comparator<E> cmp) {
    return st.collect(
        Collectors.groupingBy(
            Function<E, K> key,
            () -> new TreeMap<>(cmp),
            Collectors.toList()));
}

Map<K, Integer> groupsSize(Stream<E> st, Function<E, K> key) {
    return st.collect(
        Collectors.groupingBy(Function<E, K> key,
            Collectors.collectingAndThen(
                Collectors.counting(),
                Long::intValue)));
}

Map<K, R> groupingListAndThen(Stream<E> st, Function<E, K> key,
    Function<E, T> value, Function<List<T>, R> f) {
    return st.collect(
        Collectors.groupingBy(key,
            Collectors.mapping(value,
                Collectors.collectingAndThen(
                    Collectors.toList(), f)))));
}

Map<K, E> groupingReduce(Stream<E> st, Function<E, K> key,
    BinaryOperator<E> op) {
    return st.collect(Collectors.groupingBy(key,
        Collectors.collectingAndThen(
            Collectors.reducing(op), e -> e.get())));
}
```



Veamos cada uno de ellos:

- `Map<K,List<E>> groupingList(Stream<E> st, Function<E,K> key):`
Forma grupos mediante una clave. Cada grupo es una lista.
- `Map<K,Set<E>> groupingSet(Stream<E> st, Function<E,K> key):`
Forma grupos mediante una clave. Cada grupo es un conjunto.
- `SortedMap<K,List<E>> groupingListSorted(Stream<E> st, Function<E,K> key, Comparator<E> cmp):`: Forma grupos mediante una clave. Cada grupo es una lista y las claves se mantienen ordenadas.
- `Map<K,Integer> groupsSize(Stream<E> st, Function<E,K> key):`: Cuenta el número de elementos de cada uno de los grupos definidos por la clave. Si la clave es la identidad calcula las frecuencias de los elementos
- `Map<K,R> groupingListAndThen(Stream<E> st, Function<E,K> key, Function<E,T> value, Function<List<T>,R> f):`: Forma grupos mediante una clave. Los grupos que son listas son transformados en otras listas mediante la función `value` y a cada una de las listas resultantes se le aplica la función `f`.
- `Map<K,E> groupingReduce(Stream<E> st, Function<E,K> key, BinaryOperator<E> op):`: Forma grupos mediante una clave. Los grupos son acumulados mediante un operador binario.

Ejemplos:

Devuelve un `Map` que haga corresponder a cada ciudad destino el vuelo de más barato

```
Map<String, Vuelo> masBarato(Stream<Vuelo> st) {
    return
    st.collect(Collectors.groupingBy(Vuelo::ciudadDestino,
        Collectors.collectingAndThen(
            Collectors.reducing(
                BinaryOperator.minBy(
                    Comparator.comparing(Vuelo::precio))),
                e -> e.get())));
}
```



Devuelve para los vuelos completos un *Map* que haga corresponder a cada ciudad destino la media de los precios de los vuelos a ese destino.

```
Double preM(List<OcupacionVuelo> ls) {
    return ls.stream()
        .mapToDouble(ocp->ocp.vuelo().precio())
        .average()
        .getAsDouble();
}

Map<String, Double> precioMedio() {
    Stream<OcupacionVuelo> st = OcupacionesVuelos.datos()
        .ocupaciones().stream()
        .filter(ocp -> ocp.numPasajeros()
            .equals(ocp.vuelo().numPlazas())));
    return st.collect(Collectors.groupingBy(
        ocp -> ocp.vuelo().ciudadDestino(),
        Collectors.collectingAndThen(
            Collectors.toList(),
            g->preM(g))));
}
```

Los tipos *Vuelo*, *OcupacionVuelo* y otros pueden verse en el [repositorio](#).

Métodos consumidores

`forEach(Consumer)`

El método *forEach* ejecuta una acción sobre los elementos de un stream. La acción se define mediante un objeto de tipo *Consumer*. Por ejemplo, supongamos que un temporal de nieve obliga a desviar todos los vuelos que se dirigen a un determinado aeropuerto hacia un aeropuerto alternativo. Lo haríamos de la siguiente forma:

```
public void desviaVuelosDestino(String destino, String
nuevoDestino) {
    getVuelos().stream()
        .filter(x -> x.getDestino().equals(destino))
        .forEach(x -> x.setDestino(nuevoDestino));
}
```

La expresión



```
x -> x.setDestino(nuevoDestino) ;
```

representa la operación a realizar con los objetos de tipo vuelo, consistente en cambiar su destino al nuevo destino. Como vemos, el método no devuelve nada, sino que modifica los objetos del stream, en este caso los vuelos con el destino afectado por el temporal.

Otros métodos de Stream

Además de los métodos que hemos visto, existen otros métodos que se pueden aplicar sobre streams, como los siguientes:

- *findAny()*. Método acumulador. Devuelve un objeto Optional que representa un elemento cualquiera del stream.
- *findFirst()*. Método acumulador. Devuelve un objeto Optional que representa el primer elemento del stream.
- *limit(Long)*. Método intermedio. Devuelve un stream con el tamaño máximo indicado.
- *peek(Consumer)*. Método intermedio. Aplica una acción a todos los elementos del stream, devolviendo el stream original. Nótese la diferencia respecto a *forEach*, que es un método terminal y aplica la acción al stream sin devolver nada.

Operaciones adicionales de Stream

- Similar al zip de Python <L,R,T> Stream<T> zip(Stream<L> left, Stream<R> right, BiFunction<L, R, T> combiner);
- Similar al enumerate de Python <E> Stream<Enumerate<E>> enumerate(Stream<E> stream, Integer start); <E> Stream<Enumerate<E>> enumerate(Stream<E> stream);
- Tipo Enumerate<E>:

Propiedades

- Counter: Integer
- Value: E



Lectura y escritura de ficheros y streams

Java dispone de una clase *Files* con un conjunto de métodos estáticos para operar con ficheros y directorios. Veamos cómo realizar las operaciones básicas de lectura y escritura utilizando esta clase.

Lectura de un fichero

Para leer un fichero se utiliza el método `lines`:

```
static Stream<String> lines(Path path);
```

Este método puede disparar excepciones por lo cual lo encapsulamos, como habíamos comentado arriba, en el método:

```
public static Stream<String> fromFile(String file) {
    Stream<String> r = null;
    try {
        r = Files.lines(Paths.get(file),
                      Charset.defaultCharset());
    } catch (IOException e) {
        throw new IllegalArgumentException(
            "No se ha encontrado el fichero " + file);
    }
    return r;
}
```

Una posible llamada a este método sería:



```
Stream<String> s = fromFile("res/vuelos.txt");
```

Escritura en un fichero

Para la escritura en ficheros utilizaremos el método write de la clase Files. Este método escribe los elementos de un iterable de String en un fichero de texto, a razón de uno por línea. Como este método puede disparar excepciones lo encapsulamos en:

```
void writeStream(Stream<String> s, String file) {  
    Iterable<String> it = ()->s.iterator();  
    try {  
        Files.write(Paths.get(file),it);  
    } catch (IOException e) {  
        System.out.println(e.toString());  
    }  
}
```



Versiones imperativas del código funcional

Factoría

```
IntStream IntStream.range(a,b);
```

```
for(Integer i=a, i<b; i++) {  
    ...  
}
```

```
IntStream string.chars();
```

```
for(Integer i=a, i< string.length();i++) {  
    Character c = string.charAt(i);  
    ...  
}
```

```
Stream<E> collection.stream();
```

```
for(E e: collection) {  
    ...  
}
```

```
Stream<E> Arrays.stream(E[] elements);
```

```
for(Integer i=a; i< elements.length; i++) {  
    E e = elements[i];  
    ...  
}
```

```
Stream<E> Stream.iterate(E initialValue, Predicate<E> hasNext,  
UnaryOperator<E> next);
```



```
for(E e=initialValue; hasNext(e);e = next(e)) {  
    ...  
}
```

Stream<String> Files.lines(Path file);

```
List<String> lineas(String file){  
    List<String> lineas = null;  
    try {  
        BufferedReader bufferedReader =  
            new BufferedReader(new FileReader(file));  
        lineas = bufferedReader.lines()  
            .collect(Collectors.toList());  
        bufferedReader.close();  
    } catch (IOException e) {  
        System.out.println(e.toString());  
    }  
    return lineas;  
}
```

Funciones de transformación

Stream<R> map(Function<E,R> f);

```
for(E e: collection){  
    R r = f(e);  
    ...  
}
```

Stream<R> flatMap(Function<E,Stream<R>> f);

```
for(E e: collection){  
    for(R r: f(e)){  
        ...  
    }  
}
```

Stream<E> filter(Predicate<E> p);



```
for(E e: collection) {  
    if(p(e)) {  
        ...  
    }  
}
```

Acumuladores

Long count()

```
Integer a = 0;  
for(E e: collection) {  
    a = a+1;  
}
```

Double/Integer/Long sum();

```
Integer a = 0;  
for(E e: collection) {  
    a = a+e;  
}
```

Double average();

```
Double a = 0;  
Integer n = 0;  
for(E e: collection) {  
    a = a+e;  
    n = n+1;  
}  
return a/n;
```

Boolean allMatch(Predicate<E> p);

```
Boolean a = true;  
for(E e: collection) {  
    a = p(e);  
    if(!a) break;  
}  
return a;
```

Boolean anyMatch(Predicate<E> p);



```
Boolean a = false;
for(E e: collection) {
    a = p(e);
    if(a) break;
}
return a;
```

Optional<E> filter(p).findFirst();

```
E a = null;
for(E e: collection) {
    if(p(e)) {
        a = e;
        break;
    }
}
return Optional.ofNullable(a);
```

Optional<E> min(Comparator<E>); Optional<E> max(Comparator<E>);

```
E a = null;
for(E e: collection) {
    if(a == null || e < a) { // con el comparador
        a = e;
    }
}
return Optional.ofNullable(a);
```

Optional<E> max(Comparator<E>); Optional<E>
max(Comparator<E>);

```
E a = null;
for(E e: collection) {
    if(a == null || e > a) { // con el comparador
        a = e;
    }
}
return Optional.ofNullable(a);
```

Optional<E> reduce(BinaryOperator<E> bo);



```
E a = null;
for(E e: collection) {
    if(a == null) {
        a = e;
    } else {
        a = bo(a, e);
    }
}
return Optional.ofNullable(a);
```

List<E> collect(Collectors.toList());

```
List<E> a = new ArrayList<>();
for(E e: collection) {
    a.add(e);
}
return a;
```

Set<E> collect(Collectors.toSet());

```
Set<E> a = new HashSet<>();
for(E e: collection) {
    a.add(e);
}
return a;
```

String collect(Collectors.joining(sp,pf,sf)); // aplicado a Stream<String>

```
String a = pf;
Boolean primero = true;
for(E e: collection) {
    if(primero) {
        a = a+e;
        primero = false;
    } else {
        a = a +sp+e;
    }
}
return a+sf;
```

Map<K,List<E>> collect(Collectors.groupingBy(Function<E,K> fkey));



```
Map<K, List<E>> a = new HashMap<>();
for(E e: collection) {
    K key = fkey(e);
    if(!a.keySet().contains(key)) {
        a.put(key, new ArrayList<>());
    }
    m.get(key).add(e);
}
return a;
```

Map<K, Set<E>> collect(Collectors.groupingBy(Function<E,K> key,
Collectors.toSet()));

```
Map<K, Set<E>> a = new HashMap<>();
for(E e: collection) {
    K key = fkey(e);
    if(!a.keySet().contains(key)) {
        a.put(key, new HashSet<>());
    }
    a.get(key).add(e);
}
return a;
```

SortedMap<K, Set<E>> collect(Collectors.groupingBy(
Function<E,K> key,
()>new TreeMap<>(Comparator.reverseOrder()),
Collectors.toSet()));

```
SortedMap<K, Set<E>> a = new TreeMap<>( Comparator.reverseOrder());
for(E e: collection) {
    K key = fkey(e);
    if(!a.keySet().contains(key)) {
        a.put(key, new HashSet<>());
    }
    a.get(key).add(e);
}
return a;
```

Map<K, Long> collect(Collectors.groupingBy(
Function<E,K> key,
Collectors.counting()));



```
Map<K, Long> a = new HashMap<<();  
for(E e: collection){  
    K key = fkey(e);  
    Long r = 1L+ a.getOrDefault(key, 0L);  
    a.put(key, r);  
}  
return a;
```

```
Map<K, Integer> collect(Collectors.groupingBy(Function<E,K> key,  
                           Collectors.collectingAndThen(  
                               Collectors.counting(),  
                               Long::intValue)));
```

```
Map<K, Integer> a = new HashMap<<();  
for(E e: collection){  
    K key = fkey(e);  
    Integer r = 1+ a.getOrDefault(key, 0);  
    a.put(key, r);  
}  
return a;
```

```
Map<K, Double > collect(Collectors.groupingBy(Function<E,K> fkey,  
                           Collectors.summingDouble(  
                               Function<E,Double> fsum))));
```

```
Map<K, Double> a = new HashMap<<();  
for(E e: collection){  
    K key = fkey(e);  
    Integer r = fsum(e)+a.getOrDefault(key, 0);  
    a.put(key, r);  
}  
return a;
```

```
Map<K, List<T>> collect(Collectors.groupingBy(Function<E,K> fkey,  
                           Collectors.mapping(Function<E,T> fm,  
                               Collectors.toList())));
```



```
Map<K, List<R>> a = new HashMap<<();  
for(E e: collection) {  
    K key = fkey(e);  
    R r = fm(e);  
    if(!a.keySet().contains(key)) {  
        a.put(key, new ArrayList<>());  
    }  
    a.get(key).add(r);  
}  
return a;
```

Acciones

```
void forEach(Consumer<E> c);
```

```
for(E e: collection) {  
    c.accept(e);  
}
```



Diseño de rutas

Veamos varios ejemplos del problemas más completos. En cada problema de una determinada complejidad la primera tarea es abordar el diseño de tipos que necesitaremos. Esta tarea es básica para una programación de calidad. También lo es el diseño de funciones reutilizables. Un primer ejemplo es en del calculos sobre rutas de las que conocemos la secuencia de sus coordenadas.

Las rutas GPS (también llamadas tracks) contienen información sobre los puntos de un determinado trayecto. Casi cualquier dispositivo que tenga GPS (móviles, relojes, pulseras fitbit...) permite registrar esta información. Trabajaremos con un formato simple: CSV. Los datos que acompañan el ejercicio se corresponden con una ruta real.

Queremos hacer una serie de cálculos sobre los datos de entrada. Todos esos cálculos los acumularemos en tipo Ruta descrito más abajo.

El formato de entrada con el que trabajaremos contempla una línea por cada punto del trayecto que incluye cuatro informaciones:

- tiempo en el que fue tomada la medición en horas, minutos y segundos
- latitud del punto en el que fue tomada la medición en grados
- longitud del punto en el que fue tomada la medición en grados
- altitud del punto en el que fue tomada la medición en metros

He aquí un fragmento de dicho fichero con las cinco primeras líneas:



```
00:00:00,36.74991256557405,-5.147951105609536,712.2000122070312
00:00:30,36.75008556805551,-5.148005923256278,712.7999877929688
00:01:30,36.75017642788589,-5.148165263235569,714.0
00:02:04,36.750248931348324,-5.148243047297001,714.5999755859375
00:02:19,36.750430315732956,-5.148255117237568,715.0
```

Distancia a lo largo de la superficie de la tierra según la fórmula de Harvesine:

$$a = \sin^2(\Delta\varphi/2) + \cos \varphi_1 * \cos \varphi_2 * \sin^2(\Delta\lambda/2)$$

$$d = 2 R \operatorname{atan2}(a, \sqrt{1-a})$$

Donde φ_1, φ_2 son las latitudes de los puntos, $\Delta\varphi$ la diferencia de latitudes y $\Delta\lambda$ la diferencia de longitudes.

Diseño

Los tipos adecuados para resolver el problema son:

Coordenadas2D

Propiedades

- latitud: Double, básica, en grados
- longitud: Double, básica, en grados
- copy: Coordenada2D, derivada
- distance(other: Coordenada2D): Double, derivada, *formula de harvesine*
- es_cercana(other: Coordenada2D, d:Double): Boolean, derivada, distance(other) < d

Representación

- (longitude,latitude)

Factoría

- parse(text:str)-> Coordenadas2D,en grados
- of(latitude:float,longitude:float)-> Coordenadas2D
- center(coordenadas: List<Coordenada2D>) -> Coordenada2D



Coordenadas3D

Propiedades

- latitud: Double, básica, en grados
- longitud: Double, básica, en grados
- altitud: Double, básica, en kms
- coordenada2d: Coordenada2D, derivada
- copy: Coordenada3D, derivada
- distance(other: Coordenada3D): float, derivada, $\text{sqrt}\{\text{coordenada2d.distance}(\text{other.coordenada2d})^2 - (\text{altitud} - \text{other.altitud})^2\}$
- es_cercana(other:Coordenada3D, d:Double): Boolean, derivada, $\text{distance}(\text{other}) < d$

Representación

- (longitude,latitude,altitude)

Factoría

- parse(text:str)-> Coordenadas3D, en grados, ms
- of_radianes(latitud:float,longitud:float,altitud:Double) -> Coordenadas3D, en radianes y km
- of_grados(latitud:float,longitud:float,altitud:Double) -> Coordenadas3D, en grados y m
- of(coordenadas:Coordenadas2D,altitude:Double) -> Coordenadas3D, en km

Marca

Propiedades

- tiempo: LocalTime, básica
- latitud: Double, básica, en grados
- longitud: Double, básica, en grados
- altitud: Double, básica, en Km

Representación:

- (00:00:30,2.3,0.5,7.9)



Factoría

- parse(linea>List<String>)->Marca
- of(tiempo: LocalTime,latitud: Double,longitude: Double,altitud:Double))->Marca

Intervalo

Propiedades

- principio: Marca, básica
- fin: Marca, básica
- desnivel: Double, derivada, km
- velocidad: Double, derivada, km/hora
- longitud: Double, derivada, km
- tiempo: Double, derivada, km

Representación:

- ((00:00:30,2.3,0.5,7.9), (00:00:35,2.4,0.6,8.1))

Factoría

- of(principio: Marca, fin: Marca)->Marca, fin.tiempo >= principio.tiempo

Ruta

Propiedades

- marcas: List<Marca>, básica
- tiempo: , derivada
- longitud: Double, derivada
- velocidad: Double,
- velocidad_en_intervalo(i:Integer): Double
- desnivel_en_intervalo(i:int):Double
- desnivel_en_intervalo(i:int):Double
- desnivel_acumulado: Pair<Double,Double>

Representación:

- (00:00:30,2.3,0.5,7.9) ..., (00:00:35,2.4,0.6,8.1)}



Factoría

- leer_de_fichero(fichero:String)->Ruta

Implementación

Por cuestiones didácticas queremos hacer una implementación funcional y otra imperativa del tipo Ruta.

Veamos los aspectos más destacables de los tipos anteriores que implementaremos mediante un *record*. No se incluye las cláusulas import que pueden verse en el repositorio.

```
record Coordenadas2D(Double latitud, Double longitud) {

    public static Coordenadas2D of(Double latitud,
        Double longitud) {
        return new Coordenadas2D(latitud, longitud);
    }

    public Coordenadas2D toRadians() {
        Double latitud = Math.toRadians(this.latitud);
        Double longitud = Math.toRadians(this.longitud);
        return Coordenadas2D.of(latitud, longitud);
    }

    public Double distancia(Coordenadas2D c) {
        return distancia(this, c);
    }
}
```



```

public static Double distancia(Coordenadas2D c1,
                               Coordenadas2D c2) {
    Double radio_tierra = 6373.0;
    Coordenadas2D c1R = c1.toRadians();
    Coordenadas2D c2R = c2.toRadians();
    Double incLat = c2R.latitud-c1R.latitud;
    Double incLong = c2R.longitud-c1R.longitud;
    Double a = Math.pow(Math.sin(incLat/2),2) +
               Math.cos(c1R.latitud)
               *Math.cos(c2R.latitud)*
               Math.pow(Math.sin(incLong/2),2);
    Double c = 2 * Math.atan2(Math.sqrt(a),
                              Math.sqrt(1 - a));
    return radio_tierra*c;
}

```

```

public static Coordenadas2D center(
    List<Coordenadas2D> coordenadas) {
    Double averageLat = coordenadas.stream()
        .mapToDouble(x->x.latitud).average()
        .getAsDouble();
    Double averageLng = coordenadas.stream()
        .mapToDouble(x->x.longitud)
        .average().getAsDouble();
    return Coordenadas2D.of(averageLat,averageLng);
}
@Override
public String toString() {
    return String.format("(%.2f,%.2f)",
                         this.latitud,this.longitud);
}
}

```



```

record Coordenadas3D(Double latitud,Double longitud,
    Double altitud) {
    public static Coordenadas3D of(Double latitud,
        Double longitud, Double altitud) {
        return new Coordenadas3D(latitud, longitud,
            altitud);
    }
    public Double distancia(Coordenadas3D c) {
        return distancia(this,c);
    }
    public Coordenadas2D to2D() {
        return Coordenadas2D.of(latitud, longitud);
    }
    public static Double distancia(Coordenadas3D c1,
        Coordenadas3D c2) {
        Coordenadas2D c12D = Coordenadas2D.of(c1.latitud,
            c1.longitud);
        Coordenadas2D c22D = Coordenadas2D.of(c2.latitud,
            c2.longitud);
        return Math.sqrt(Math.pow(
            Coordenadas2D.distancia(c12D,c22D),2)
            +Math.pow(c1.altitud-c1.altitud,2));
    }
}

```

```

@Override
public String toString() {
    return String.format("(%f,%f,%f)",
        latitud,longitud,altitud);
}
}

```

Destacamos los detalles del método *toString()* para conseguir el formato adecuado.



```

record Marca(LocalTime time, Coordenadas3D coordenadas) {

    public static Marca parse(String text) {
        String[] campos = text.split(",");
        LocalTime time = LocalTime.parse(campos[0],
            DateTimeFormatter.ofPattern("HH:mm:ss"));
        Coordenadas3D coordenadas =
            Coordenadas3D.of(Double.parseDouble(campos[1]),
                Double.parseDouble(campos[2]),
                Double.parseDouble(campos[3])/1000));
        return Marca.of(time, coordenadas);
    }
    public static Marca of(LocalTime time,
        Coordenadas3D coordenadas) {
        return new Marca(time, coordenadas);
    }
    @Override
    public String toString() {
        return String.format("(%s,%2f,%2f,%2f)",
            time,coordenadas.latitud(),
            coordenadas.longitud(),coordenadas.altitud());
    }
}

```

Destacamos los detalles de la función parse y el uso del parsing de los datos del tiempo. Igualmente el formato de salida para estos datos.

```

record Intervalo(Marca principio, Marca fin) {

    public static Intervalo of(Marca principio, Marca fin) {
        return new Intervalo(principio, fin);
    }
    @Override
    public String toString() {
        return String.format("(%,%),",
            this.principio.toString(),this.fin.toString());
    }
    public Double desnivel() {
        return this.fin.coordenadas().altitud()-
            this.principio().coordenadas().altitud();
    }
}

```



```

public Double longitud() {
    return this.principio().coordenadas()
        .distancia(this.fin().coordenadas());
}
public Double tiempo() {
    return this.principio().time()
        .until(this.fin().time(),
            ChronoUnit.SECONDS) / 3600.;
}
public Double velocidad() {
    return this.longitud() / this.tiempo();
}
}

```

Del tipo Ruta queremos hacer dos implementaciones: una funcional y otra imperativa. Para concretar los detalles diseñaremos un interface Ruta con los métodos relevantes del tipo y dos clases *RutaF* y *RutaI* con las implementaciones funcional e imperativa respectivamente.

Las dos clases *RutaF* y *RutaI* pueden tener aspectos comunes que sería interesante programar una sola vez. Para conseguirlo diseñamos la clase abstracta *RutaA* de la que heredarán las dos anteriores. Por esa razón deben ser clases y no records. Esta clase implementará *Ruta* y los métodos cuyo código puede ser compartido entre *RutaF* y *RutaI*.

Veamos en primer lugar el interface *Ruta* y la clase abstracta *RutaA*.



```

public interface Ruta {
    public static Ruta of(List<Marca> marcas) {
        return switch(RutaA.tipo) {
            case Funcional->RutaF.of(marcas);
            case Imperativa->RutaI.of(marcas);
        };
    }
    public static Ruta leeDeFichero(String fichero) {
        return switch(RutaA.tipo) {
            case Funcional->RutaF.leeDeFichero(fichero);
            case Imperativa->RutaI.leeDeFichero(fichero);
        };
    }
    Double getTiempo();
    Double getLongitud();
    Double getVelocidadMedia();
    Intervalo getIntervalo(Integer i);
    Double getDesnivelCrecienteAcumulado();
    Double getDesnivelDecrecienteAcumulado();
}

```

```

public abstract class RutaA implements Ruta {
    public static enum
        TipoImplementacion{Imperativa,Funcional}
    public static TipoImplementacion tipo =
        TipoImplementacion.Funcional;
    protected List<Marca> marcas;
    protected RutaA(List<Marca> marcas) {
        this.marcas = marcas;
    }
    @Override
    public Double getVelocidadMedia() {
        return this.getLongitud()/this.getTiempo();
    }
    @Override
    public Intervalo getIntervalo(Integer i) {
        return Intervalo.of(this.marcas.get(i),
            this.marcas.get(i+1));
    }
}

```



En la clase abstracta *RutaA* se incluyen algunos métodos comunes a las dos implementaciones.

```
public class RutaF extends RutaA implements Ruta {
    public static Ruta of(List<Marca> marcas) {
        return new RutaF(marcas);
    }
    private RutaF(List<Marca> marcas) {
        super(marcas);
    }
    public static Ruta leeDeFichero(String fichero) {
        List<Marca> marcas =
            FileTools.streamFromFile("ficheros/ruta.csv")
                .map(x->Marca.parse(x))
                .collect(Collectors.toList());
        return of(marcas);
    }
}
```

Veamos ahora la implementación del resto de propiedades.

```
public Double getTiempo() {
    Integer n = this.marcas.size();
    return this.marcas.get(0).time()
        .until(this.marcas.get(n-1).time(),
               ChronoUnit.SECONDS)/3600.;
}
```

```
public Double getLongitud() {
    Integer n = this.marcas.size();
    return IntStream.range(0,n-1).boxed()
        .map(i->this.intervalo(i))
        .mapToDouble(x->x.longitud())
        .sum();
}
```

```
@Override
public String toString() {
    return marcas.stream().map(m->m.toString())
        .collect(Collectors.joining("\n"));
}
```



```
public Double desnivelCrecienteAcumulado() {
    Integer n = this.marcas.size();
    return IntStream.range(0, n-1)
        .boxed()
        .map(i->this.intervalo(i))
        .filter(e->e.desnivel()>0)
        .mapToDouble(e->e.longitud())
        .sum();
}
```

```
public Double desnivelDecrecienteAcumulado() {
    Integer n = this.marcas.size();
    return IntStream.range(0, n-1)
        .boxed()
        .map(i->this.intervalo(i))
        .filter(e->e.desnivel()<0)
        .mapToDouble(e->e.longitud())
        .sum();
}
```

La implementación imperativa sigue el mismo esquema que la funcional.
Diseñamos un constructor y dos métodos de factoría.

```
public class RutaI extends RutaA implements Ruta {
    public static Ruta of(List<Marca> marcas) {
        return new RutaI(marcas);
    }
    public static Ruta leeDeFichero(String fichero) {
        List<Marca> marcas = new ArrayList<>();
        for (String linea:FileTools.lineasFromFile(
                "ficheros/ruta.csv")) {
            Marca m = Marca.parse(linea);
            marcas.add(m);
        }
        return of(marcas);
    }
    private RutaI(List<Marca> marcas) {
        super(marcas);
    }
}
```

Las propiedades se implementan ahora siguiendo el patrón imperativo.



```

@Override
public Double getLongitud() {
    Integer n = this.marcas.size();
    Double a = 0.;
    for(Integer i=0; i< n-1; i++) {
        Intervalo it = this.getIntervalo(i);
        Double ln = it.longitud();
        a = a + ln;
    }
    return a;
}

```

```

@Override
public Double getTiempo() {
    Integer n = this.marcas.size();
    Double a = 0.;
    for(Integer i=0; i< n-1; i++) {
        Intervalo it = this.getIntervalo(i);
        Double ln = it.tiempo();
        a = a + ln;
    }
    return a;
}

```

```

@Override
public Double getDesnivelCrecienteAcumulado() {
    Integer n = this.marcas.size();
    Double a = 0.;
    for(Integer i=0; i< n-1; i++) {
        Intervalo it = this.getIntervalo(i);
        if (it.desnivel()>0) {
            Double ln = it.longitud();
            a = a + ln;
        }
    }
    return a;
}

```



```
@Override  
public Double getDesnivelDecrecienteAcumulado() {  
    Integer n = this.marcas.size();  
    Double a = 0.;  
    for(Integer i=0; i< n-1; i++) {  
        Intervalo it = this.getIntervalo(i);  
        if (it.desnivel()<0) {  
            Double ln = it.longitud();  
            a = a + ln;  
        }  
    }  
    return a;  
}
```

```
@Override  
public String toString() {  
    String r = marcas.get(0).toString();  
    for(int i = 1;i<marcas.size();i++) {  
        r += "\n"+marcas.get(i).toString();  
    }  
    return r;  
}
```



Servicio de bicicletas de sevilla

Se dispone de los datos de las estaciones de la red Servicio de bicicletas de sevilla (Sevici). Los datos se encuentran en un fichero CSV. Cada línea del fichero contiene seis datos:

- Nombre de la estación
- Número total de bornetas de la estación
- Número de bornetas vacías
- Número de bicicletas disponibles
- Latitud
- Longitud

Los datos dependen del instante en que se obtiene el fichero, ya que el número de bicicletas y bornetas libres varía de forma continua. Están serían, por ejemplo, las primeras líneas del fichero en un momento dado:

```
name,slots,empty_slots,free_bikes,latitude,longitude
149_CALLE ARROYO,20,11,9,37.397829929383,-5.97567172039552
257_TORRES ALBARRACIN,20,16,4,37.38376948792722,-5.908921914235877
243_GLORIETA DEL PRIMERO DE MAYO,15,6,9,37.380439481169994,-
5.953481197462
109_AVENIDA SAN FRANCISCO JAVIER,15,1,14,37.37988413609134,-
5.974382770011
073_PLAZA SAN AGUSTIN,15,10,4,37.38951386231434,-5.984362789545622
```

Los principales aspectos que tendremos que resolver a la hora de procesar estos datos de entrada serán saltar la línea de encabezado del fichero, separar adecuadamente los campos mediante las comas e



interpretar el formato de cada uno de los campos, que puede ser de tipo cadena, entero o real.

Diseño

Estación

Propiedades:

- Numero: int;
- Name: str
- Slots: int;
- Empty_Slots: int;
- Free_Bykes; Integer,
- Ubicacion: Coordenadas2D;
- Nombre_completo: String, derivada

Representación: A definir

Métodos de factoría

- Parse: A definir
- Of: A definir

Red

Propiedades:

- Estaciones: List<Estacion> //básica
- EstacionesCercanasA(Coordenadas2D c, float distance):
- Set[Estacion] //derivada
- Numero: int //derivada
- PorName(String s): Estacion //derivada
- PorNumero(Integer e): Estacion //derivada
- EstacionesConBicisDisponibles: Set<Estacion>/derivada
- EstacionesConBicisDisponibles(Integer n): Set<Estacion> //derivada
- Ubicaciones: Set<Coordenadas2D>//



- UbicacionEstacionesDisponibles(int k):
- EstacionMasBicisDisponibles: Estacion //derivada
- EstacionesPorBicisDisponbles: Map[Integer,List<Estacion>> //derivada
- NumeroDeEstacionesPorBicisDisponibles: Map<Integer,Integer> //derivada

Representacion: A definir

Métodos de factoría

- Lee_de_fichero: Red,lectura de un fichero

Implementación

Solo mostramos el código de algunos métodos el resto está en el repositorio. Diseñamos Estación como un record y RedF como una clase. Aquí veremos solo la versión functional y la case abstracta. La versión imperativa (RedI) y el interface Red (siguiendo un diseño similar al caso del problema Ruta anterior) pueden verse en el repositorio.

Veamos primero el método de parsing del tipo Estación:



```

public static Estacion parse(String linea) {
    String[] partes = linea.split(",");
    String[] sp = partes[0].split("_");
    Integer numero = Integer.parseInt(sp[0]);
    String name = sp[1];
    Integer slots= Integer.parseInt(partes[1]);
    Integer empty_slots = Integer.parseInt(partes[2]);
    Integer free_bikes = Integer.parseInt(partes[3]);
    Preconditions.checkArgument(slots >=0,
        String.format("Slots %d en %s",slots,linea));
    Preconditions.checkArgument(empty_slots >=0,
        String.format("Empty_Slots %d en
            %s",empty_slots,linea));
    Preconditions.checkArgument(free_bikes >=0,
        String.format("Free_Bikes %d en
            %s",free_bikes,linea));
    Coordenadas2D coordenadas =
        Coordenadas2D.of(Double.parseDouble(partes[4]),
    Double.parseDouble(partes[5]));
    return new Estacion(numero,name,slots,
        empty_slots,free_bikes,coordenadas);
}

```

El tipo Estacion es:

```

public record Estacion(Integer numero,
    String name,
    Integer slots,
    Integer empty_slots,
    Integer free_bikes,
    Coordenadas2D coordenadas) {
    ...
}

```

La clase abstracta *RedA* acumula los métodos comunes a las dos implementaciones: funcional e imperativa. El tipo *RedF* con su método de factoría para leer el fichero. La estación tiene como propiedad básica estaciones, pero añadimos una propiedad derivada de mucho uso, índices.

El diseño que vamos a usar es inicializar índices a *null* y calcular su valor cuando sea necesario en el método correspondiente. Esta idea se puede usar siempre en el diseño de propiedades derivadas cuyo uso sea muy frecuente.



```
public abstract class RedA implements Red {  
    public static enum TipoImplementacion{Imperativa,Funcional}  
    public static TipoImplementacion tipo =  
        TipoImplementacion.Funcional;  
  
    protected List<Estacion> estaciones;  
    protected Map<Integer, Estacion> indices;  
    protected RedA(List<Estacion> estaciones) {  
        this.estaciones = estaciones;  
        this.indices = null;  
    }  
  
    @Override  
    public void add(Estacion e) {  
        Preconditions.checkNotNull(  
            !this.indices().containsKey(e.numero()),  
            "La estacion esta ya incluida");  
        this.estaciones.add(e);  
        this.indices = null;  
    }  
  
    @Override  
    public void remove(Estacion e) {  
        this.estaciones.remove(e);  
        this.indices = null;  
    }  
  
    @Override  
    public Integer numero() {  
        return this.estaciones.size();  
    }  
  
    @Override  
    public List<Estacion> estaciones() {  
        return estaciones;  
    }  
}
```



```
public class RedF extends RedA implements Red {
    public static Red parse(String fichero) {
        List<Estacion> estaciones =
            FileTools.streamFromFile(
                "ficheros/estaciones.csv").skip(1)
                .map(linea -> Estacion.parse(linea))
                .collect(Collectors.toList());
        return of(estaciones);
    }
    public static RedF of(List<Estacion> estaciones) {
        return new RedF(estaciones);
    }
    private RedF(List<Estacion> estaciones) {
        super(estaciones);
    }
}
```

Hay propiedades derivadas de mucho uso. Es decir su cálculo se hace de forma repetida. Para no repetir el cálculo podemos seguir el esquema siguiente. La idea es tener un atributo privado inicializado a null. Se pregunta si el atributo es *null* y si lo es se hace el cálculo y se guarda en el atributo. Sea o no *null* el atributo se devuelve este al final.

```
public Map<Integer, Estacion> indices() {
    if(super.indices == null)
        super.indices= estaciones.stream()
            .collect(Collectors.toMap(e->e.numero(), e->e));
    return super.indices;
}
```

Buscar la estación con un número dado

```
public Estacion porNumero(Integer numero) {
    Estacion e = this.indices().getOrDefault(numero,null);
    Preconditions.checkNotNull(e);
    return e;
}
```

Buscar las estaciones con un nombre dado.



```
public Set<Estacion> porNombre(String nombre) {  
    return this.estaciones.stream()  
        .filter(e -> e.name().equals(nombre))  
        .collect(Collectors.toSet());  
}
```

Las estaciones con bicicletas disponibles.

```
public Set<Estacion> estacionesConBicisDisponibles() {  
    return this.estaciones.stream()  
        .filter(e -> e.free_bikes() > 0)  
        .collect(Collectors.toSet());  
}
```

Las estaciones con n bicicletas disponibles

```
public Set<Estacion> estacionesConBicisDisponibles(Integer n) {  
    return this.estaciones.stream()  
        .filter(e -> e.free_bikes() >= n)  
        .collect(Collectors.toSet());  
}
```

Las coordenadas de las estaciones

```
public List<Coordenadas2D> ubicaciones() {  
    return this.estaciones  
        .stream()  
        .map(e -> e.coordenadas())  
        .collect(Collectors.toList());  
}
```

Las coordenadas de las estaciones con k o más bicicletas disponibles

```
public List<Coordenadas2D>  
    ubicacionEstacionesDisponibles(Integer k) {  
    return this.estaciones.stream()  
        .filter(e -> e.free_bikes() >= k)  
        .map(e -> e.coordenadas())  
        .collect(Collectors.toList());  
}
```

La estación con más bicicletas disponibles



```
public Estacion estacionMasBicisDisponibles() {  
    return this.estaciones.stream()  
        .max(Comparator.comparing(e -> e.free_bikes())).get();  
}
```

Los grupos de estaciones según su número de bicicletas disponibles.

```
public Map<Integer, List<Estacion>>  
    estacionesPorBicisDisponibles() {  
    return this.estaciones.stream().collect(  
        Collectors.groupingBy(e -> e.free_bikes()));  
}
```

El tamaño de los grupos de estaciones según su número de bicicletas disponibles

```
public Map<Integer, Integer>  
    numeroDeEstacionesPorBicisDisponibles() {  
    return this.estaciones.stream().collect(  
        Collectors.groupingBy(e -> e.free_bikes(),  
            Collectors.collectingAndThen(  
                Collectors.toList(), ls -> ls.size())));  
}
```

La representación de la red.

```
@Override  
public String toString() {  
    return estaciones.stream()  
        .map(e -> e.toString())  
        .collect(Collectors.joining("\n"));  
}
```

Un método para escribir los datos de n estaciones en un fichero

```
public void escribe(Integer n, String file) {  
    Stream<String> s = this.estaciones.subList(0, n)  
        .stream()  
        .map(e -> e.toString());  
    FileTools.writeStream(s, file);  
}
```



Cálculos sobre un libro

El objetivo es leer un libro de un fichero y hacer cálculos sobre las palabras que contiene, en qué líneas aparecen, ordenarlas por frecuencias, etc. Se usará el fichero *quijote.txt* para hacer los cálculos que está disponible en el repositorio.

Se supone que las palabras están separadas una de otras por separadores que podemos definir. Los separadores que usaremos son "[,:\\n\\?\\!\\;\\:\\"]" aunque podemos añadir o eliminar separadores.

Queremos implementar las siguientes funciones:

- *numeroDeLineas(file:String):int*
- *numeroDePalabrasDistintasNoHuecas(file:String):int*
- *palabrasDistintasNoHuecas(file:String):Set<String>*
- *longitudMediaDeLineas(file:str):float*
- *numeroDeLineasVacias(file:str):int*
- *lineaMasLarga(file:str):String*
- *primeraLineaConPalabra(file:str,palabra:str):int*
- *lineaNumero(file:str,n_int):String*
- *frecuenciasDePalabras(file:String):Map<String,Integer>,*
Frecuencia de cada palabra. Ordenadas por palabras
- *palabrasPorFrecuencias(file:String:Map<Integer,Set<String>>),*
palabras agrupadas por sus frecuencias de aparición. Ordenadas por frecuencias
- *lineasDePalabra(file:String):Map<String,Set<Integer>>, grupos*
de líneas donde aparece cada palabra.



Ahora vamos a resolver el problema como un conjunto de funciones agrupadas en una clase. Algunos detalles se incluyen aquí y el resto se pueden ver en el repositorio. Para el diseño de estas funciones reutilizamos varias funciones para leer ficheros. Algunas funciones las diseñamos en una forma funcional y otra imperativa.

La primera el número de líneas el fichero en su manera funcional.

```
static Integer numeroDeLineas(String file) {  
    return (int) FileTools.streamFromFile(file).count();  
}
```

El número de líneas el fichero en su forma imperativa.

```
static Integer numeroDeLineas2(String file) {  
    List<String> ls = FileTools.lineasFromFile(file);  
    return ls.size();  
}
```

El conjunto de palabras huecas en su manera funcional.

```
static Set<String> palabrasHuecas(String file) {  
    return FileTools.streamFromFile(file)  
        .collect(Collectors.toSet());  
}
```

El conjunto de palabras huecas en su manera imperativa.

```
static Set<String> palabrasHuecas2(String file) {  
    Set<String> s = new HashSet<>();  
    for(String p: FileTools.lineasFromFile(file)) {  
        s.add(p);  
    }  
    return s;  
}
```

Palabras no huecas en su manera funcional.



```
static Set<String> palabrasDistintasNoHuecas(String file) {  
    Set<String> palabrasHuecas =  
        Libro.palabrasHuecas("ficheros/palabras_huecas.txt");  
    return FileTools.streamFromFile(file)  
        .filter(ln->!ln.isEmpty())  
        .flatMap(ln->Arrays.stream(ln.split(Libro.separadores)))  
        .filter(p->!palabrasHuecas.contains(p))  
        .distinct()  
        .collect(Collectors.toSet());  
}
```

Palabras no huecas en su manera imperativa.

```
static Set<String> palabrasDistintasNoHuecas2(String file) {  
    Set<String> palabrasHuecas =  
        Libro.palabrasHuecas("ficheros/palabras_huecas.txt");  
    List<String> lineas = FileTools.lineasFromFile(file);  
    Set<String> palabrasDistintas = new HashSet<>();  
    for(String linea: lineas) {  
        if(!linea.isEmpty()) {  
            for(String p: linea.split(separadores)) {  
                if(!palabrasHuecas.contains(p)) {  
                    palabrasDistintas.add(p);  
                }  
            }  
        }  
    }  
    return palabrasDistintas;  
}
```

Número de palabras no huecas en su manera funcional.

```
static Integer numeroDePalabrasDistintasNoHuecas(String file){  
    Set<String> palabrasHuecas = Libro.palabrasHuecas(  
        "ficheros/palabras_huecas.txt");  
    return (int) FileTools.streamFromFile(file)  
        .filter(ln->!ln.isEmpty())  
        .flatMap(ln->Arrays.stream(  
            ln.split(Libro.separadores)))  
        .filter(p->!palabrasHuecas.contains(p))  
        .distinct()  
        .count();  
}
```



Número de palabras no huecas en su manera imperativa.

```
static Integer numeroDePalabrasDistintasNoHuecas2(String file) {  
    return palabrasDistintasNoHuecas2(file).size();  
}
```

Longitud media de las líneas en su manera funcional

```
static Double longitudMediaDeLineas(String file) {  
    return FileTools.streamFromFile(file)  
        .mapToInt(ln->ln.length())  
        .average()  
        .getAsDouble();  
}
```

Longitud media de las líneas en su manera imperativa

```
static Double longitudMediaDeLineas2(String file) {  
    Integer numLineas = 0;  
    Integer sumTamLineas = 0;  
    for(String linea: FileTools.lineasFromFile(file)) {  
        Integer ln = linea.length();  
        numLineas++;  
        sumTamLineas += ln;  
    }  
    return ((double)sumTamLineas)/numLineas;  
}
```

Número de líneas vacías en su manera funcional

```
static Integer numeroDeLineasVacias(String file) {  
    return (int) FileTools.streamFromFile(file)  
        .filter(ln->ln.isEmpty())  
        .count();  
}
```

Líneas más larga en su manera funcional



```
static String lineaMasLarga(String file) {  
    return FileTools.streamFromFile(file)  
        .max(Comparator.comparing(  
            (String ln)->ln.length()))  
        .get();  
}
```

Líneas más larga en su manera funcional

```
static String lineaMasLarga2(String file) {  
    String lineaMaslarga = null;  
    Integer lnlineaMaslarga = null;  
    for(String linea: FileTools.lineasFromFile(file)) {  
        Integer nl = linea.length();  
        if(lnlineaMaslarga == null ||  
            nl > lnlineaMaslarga) {  
            lineaMaslarga = linea;  
            lnlineaMaslarga = nl;  
        }  
    }  
    return lineaMaslarga;  
}
```

Primera línea con una palabra dada en su forma funcional.

```
static Integer primeraLineaConPalabra(String file,  
    String palabra) {  
    return StreamTools.enumerate(  
        FileTools.streamFromFile(file))  
            .filter(p->p.value().contains(palabra))  
            .findFirst()  
            .get()  
            .counter();  
}
```

Primera línea con una palabra dada en su forma imperativa



```
static Integer primeraLineaConPalabra2(String file, String palabra) {
    Integer n = 0;
    Integer r = -1;
    for(String linea: FileTools.lineasFromFile(file)) {
        if(linea.contains(palabra)) {
            r = n;
            break;
        }
        n++;
    }
    return r;
}
```

Frecuencias de palabras ordenadas por palabras en su forma funcional:

```
static SortedMap<String, Integer> frecuenciasDePalabras(
    String file){
    return FileTools.streamFromFile(file)
        .filter(ln->!ln.isEmpty())
        .flatMap(ln->Arrays.stream(ln.split(separadores)))
        .collect(Collectors.groupingBy(p->p, ()->new TreeMap<>(),
            Collectors.collectingAndThen(
                Collectors.toList(), ls->ls.size())));
}
```

Frecuencias de palabras ordenadas por palabras en su forma imperativa:



```
static SortedMap<String, Integer> frecuenciasDePalabras2(String file) {
    Set<String> palabrasHuecas =
        Libro.palabrasHuecas("ficheros/palabras_huecas.txt");
    List<String> lineas = FileTools.lineasFromFile(file);
    SortedMap<String, Integer> m = new TreeMap<>();
    for(String linea: lineas) {
        if(!linea.isEmpty()) {
            for(String p: linea.split(separadores)) {
                if(!palabrasHuecas.contains(p)) {
                    if(!m.containsKey(p)) {
                        m.put(p,1);
                    } else {
                        Integer f = m.get(p);
                        m.put(p, f+1);
                    }
                }
            }
        }
    }
    return m;
}
```

Palabras agrupadas por frecuencias en su versión funcional

```
static SortedMap<Integer, Set<String>> palabrasPorFrecuencias(
    String file){
    SortedMap<String, Integer> fq =
        Libro.frecuenciasDePalabras(file);
    return fq.keySet().stream()
        .collect(Collectors.groupingBy(f->fq.get(f),
            () ->new TreeMap<>(),
            Collectors.toSet()));
}
```

Palabras agrupadas por frecuencias en su versión imperativa



```

static SortedMap<Integer, Set<String>>
    palabrasPorFrecuencias2(String file) {
    SortedMap<String, Integer> fq =
        Libro.frecuenciasDePalabras(file);
    SortedMap<Integer, Set<String>> r = new TreeMap<>();
    for(String p: fq.keySet()) {
        Integer f = fq.get(p);
        if(!r.containsKey(f)) {
            Set<String> s = new HashSet<>();
            s.add(p);
            r.put(f, s);
        } else {
            r.get(f).add(p);
        }
    }
    return r;
}

```

Para usar enumerate necesitamos el tipo Enumerate que es una tupla entero, valor. Este tipo tiene el método expand que convierte una tupla en un stream de otras tuplas con el mismo entero y valores obtenidos del valor original.

```

record Enumerate<E>(Integer counter, E value) {

    public static <E> Enumerate<E> of(Integer num, E value) {
        return new Enumerate<E>(num, value);
    }

    public <R> Stream<Enumerate<R>> expand(
        Function<E, Stream<R>> f) {
        return f.apply(this.value())
            .map(e->Enumerate.of(this.counter(),e));
    }
}

```

Con ese tipo podemos escribir un método para calcular el conjunto de líneas donde se encuentra cada palabra en su versión funcional.



```
static SortedMap<String, Set<Integer>> lineasDePalabra(
    String file){
    Set<String> palabrasHuecas =
        Libro.palabrasHuecas(
            "ficheros/palabras_huecas.txt");
    return StreamTools.enumerate(
        FileTools.streamFromFile(file))
        .filter(pp->!pp.value().isEmpty())
        .flatMap(pp->pp.expand(
            ln->Arrays.stream(ln.split(separadores))))
        .filter(pp->!palabrasHuecas.contains(pp.value()))
        .collect(Collectors.groupingBy(
            pp->pp.value(),
            ()->new TreeMap<>(),
            Collectors.mapping(pp->pp.counter(),
                Collectors.toSet())));
}
```

Conjunto de líneas donde se encuentra cada palabra en su versión imperativa.



```
static SortedMap<String, Set<Integer>> lineasDePalabra2(
    String file){
    Set<String> palabrasHuecas =
        Libro.palabrasHuecas("ficheros/palabras_huecas.txt");
    List<String> lineas = FileTools.lineasFromFile(file);
    SortedMap<String, Set<Integer>> r = new TreeMap<>();
    Integer nl = 0;
    for(String linea: lineas) {
        if(!linea.isEmpty()) {
            for(String p: linea.split(separadores)) {
                if(!palabrasHuecas.contains(p)) {
                    if(!r.containsKey(p)) {
                        Set<Integer> s =
                            new HashSet<>();
                        s.add(nl);
                        r.put(p, s);
                    } else {
                        r.get(p).add(nl);
                    }
                }
            }
            nl++;
        }
    }
    return r;
}
```



Aeropuertos, vuelos y compañías aéreas

Se dispone de los datos de aeropuertos en el fichero *Aeropuertos.csv*. Cada línea contiene, *nombre, pais, código, ciudad*. Las líneas son de la forma

```
Tirana Airport,Albania,TIA,Tirana
Berlin Brandeburgo Airport,Alemania,BER,Berlin
Bremen Airport,Alemania,BRE,Bremen
Colonia Bonn Airport,Alemania,CGN,Colonia
```

De aerolíneas disponemos del fichero *Aerolíneas.csv*. Cada línea contiene el *código, nombre* en la forma

```
AA,American Airlines
CO,Continental Airlines, Inc.
DL,Delta Airlines Inc.
N7,National Airlines Inc.
```

El fichero *Vuelos.csv* contiene en cada línea el *codigoAerolinea, numero, codigoDestino, codigoOrigen, precio, numPlazas, duracion, hora, diaSemana*. Las líneas son de la forma

```
TP,0705,BER,KTW,294,21,170,287,14:50,FRIDAY
FS,0596,TZX,AAR,761,64,49,45,07:54,THURSDAY
FJ,0612,BHD,TPS,113,98,128,180,16:41,MONDAY
CX,0930,LJU,NCL,741,17,11,159,02:40,WEDNESDAY
```

El fichero *OcupacionesVuelos.csv* contiene en cada línea *codigoVuelo, fecha, InumPasajeros* de la forma



```
NH0818,2020-04-13 16:43:00,7  
PE0174,2020-11-17 09:03:00,89  
5Z0373,2020-05-16 01:46:00,64  
7F0434,2020-10-01 03:24:00,94
```

Queremos diseñar un programa que pueda llevar a cabo entre otros posibles los *cálculos* sobre los vuelos como los siguientes:

1. Si existe un vuelo en una fecha dada Dado a un conjunto de destinos dado
2. Encontrar los destinos en una fecha dada
3. Encontrar el número total de pasajeros de los destinos que tienen un prefijo dado
4. Encontrar una relación ordenada de destinos con su número de pasajeros de llegada en un año dado.
5. Dado un destino encontrar el código del primer vuelo con plazas libres a ese destino.
6. Encontrar el destino con mayor número de vuelos de entrada y salida
7. Encontrar una relación que asocie a cada fecha la lista de los destinos de los n vuelos de la mayor duración
8. Obtener un conjunto ordenado con las duraciones de todos los vuelos cuya duración es mayor que un número de minutos dado
9. Dada una fecha f encontrar el precio medio de los vuelos con salida posterior a f. Si no hubiera vuelos devuelve 0.0
10. Obtener un conjunto con los n destinos de los vuelos con mayor duración.
11. Obtener una relación de los destinos junto a la media de los precios de los vuelos a ese destino.
12. Obtener una relación de destinos junto con las fechas de los vuelos a ese destino.
13. Obtener los n destinos que con más vuelos
14. Obtener los destinos que tienen más de n vuelos
15. Obtener una relación de destinos junto con el porcentaje de vuelos que van a ese destino.
16. Obtener una relación de destinos junto con el vuelo más barato a ese destino.



17. Obtener una relación de destinos junto a las fechas posibles a ese destino.

18.

El diseño de los tipos debe tener en cuenta que los objetos de tipo *Aeropuerto*, *Vuelo*, *Aerolínea*, *OcupacionVuelo* necesitan tener una propiedad, su *identificador*, que los distinga de manera única y los diseñamos como tipos inmutables. También necesitamos tipos que contengan los objetos disponibles para poder buscarlos. Es el objeto de los tipos *Aeropuertos*, *Vuelos*, *Aerolíneas*, *OcupacionVuelos*. Estos son tipos los diseñamos como tipos mutables y los dotamos de un objeto único que podemos obtener mediante el método *datos()*. Los objetos de tipos *Aeropuertos*, *Vuelos*, *Aerolíneas*, *OcupacionVuelos* disponen, además, de alguna propiedad que permite buscar rápidamente un objeto dado su identificador y otras propiedades para búsquedas de objetos que cumplan estas propiedades.

Diseño

Los tipos que vamos a necesitar son:

Aeropuerto

Inmutable

Propiedades:

- Código: String, básica, se compone de tres caracteres, Identifica el aeropuerto de manera única
- Ciudad: String, básica
- País: String, básica
- Nombre: String, básica

Representación:

Código, Ciudad, País, Nombre

Igualdad: Si tienen el mismo código. Aunque con la restricción impuesta al código todas las propiedades básicas serán iguales.



Métodos de Factoría:

- `parse(text:String): Aeropuerto`
- `of(Codigo: String, Ciudad: String, País: String, Nombre: String): Aeropuerto`

Aeropuertos

Propiedades:

- `aeropuerto(codigo:String):Aeropuerto`, derivada,
- `ciudadDeAeropuerto(codigo:String): String`, derivada,
- `aeropuertosEnCiudad(ciudad:String): Set<Aeropuerto>`, derivada,
- `size: Integer`, derivada, número de aeropuertos
- `get(i): Aeropuerto`, el aeropuerto i.
- `stream:Stream<Aeropuerto>`, derivada

Invariante

Los códigos de los aeropuertos deben ser diferentes

Operaciones:

- `addAeropuerto(v: Aeropuerto):void`, Añade un aeropuerto
- `removeAeropuerto(v: Aeropuerto):void`, Elimina un aeropuerto

Representación:

Lista de aeropuertos uno por línea

Métodos de Factoría:

- `leeAeropuertos(fichero:String)`, void, lee de un fichero las propiedades básicas

Este tipo los diseñaremos como una clase con métodos estáticos.

Aerolínea

Inmutable

Propiedades:



- Código: String, básica, se compone de dos caracteres, Identifica el aeropuerto de manera única
- Nombre: String

Representación:

- Código,NOMBRE

Igualdad: Si tienen el mismo código. Aunque con la restricción impuesta al código todas las propiedades básicas serán iguales

Métodos de Factoría:

- parse(text:String): Aerolinea
- of(Código: String,NOMBRE: String): Aerolinea

Aerolíneas

Propiedades:

- aerolinea(código:String): Aerolinea, derivada,
- size: Integer, derivada
- get(i): Aerolinea, la aerolinea i.
- stream: Stream<Aerolinea>, derivada

Invariante

Los códigos de las aerolíneas deben ser diferentes

Operaciones:

- addAerolinea(v: Aerolinea):void, Añade una aerolinea
- removeAerolinea(v: Aerolinea):void, Elimina una aerolinea

Representación:

Lista de aerolíneas una por línea

Métodos de Factoría:

- leeAerolineas(fichero:String), void, lee de un fichero las propiedades básicas

Este tipo los diseñaremos como una clase con métodos estáticos.



Vuelo

Inmutable

Propiedades:

- CódigoAerolinea: String, básica
- Numero: String, básica, cuatro caracteres que representan un número
- CodigoDestino: String, básica, código del aeropuerto destino
- CodigoOrigen: String, básica, código del aeropuerto origen
- Precio: Integer, básica, debe ser mayor que cero
- NumPlazas: Integer, básica, debe ser mayor que cero
- Duración: Duration, básica, duración del vuelo
- Hora: LocalTime, básica,
- DiaSemana, DayOfWeek, básica
- CiudadDestino, String, derivada
- CiudadOrigen, String, derivada
- Codigo, String, derivada, identifica de manera única un vuelo y se compone de la concatenación del CódigoAerolinea y el Numero.

Representación:

- Codigo,Nombre

Igualdad: Si tienen el mismo código. Aunque con la restricción impuesta al código todas las propiedades básicas serán iguales

Métodos de Factoría:

- parse(text:String): Vuelo
- of(Codigo: String,Nombre: String): Aerolínea
- random(): Vuelo, un vuelo construido aleatoriamente con los aeropuertos y las aerolíneas disponibles.

Vuelos

Propiedades:

- vuelo(codigo:String):Vuelo, derivada
- size(): Integer, derivada



- `get(i): Vuelo`, el vuelo i.
- `stream: Stream<Vuelo>`, derivada

Incluir aquí como propiedades algunos de los cálculos enunciados arriba

Invariante

Los códigos de los vuelos deben ser diferentes

Operaciones:

- `addAerolinea(v: Aerolinea): void`, Añade una aerolinea
- `removeAerolinea(v: Aerolinea): void`, Elimina una aerolinea

Representación:

- Lista de aerolíneas una por línea

Métodos de Factoría:

- `leeAerolineas(fichero:String)`, void, lee de un fichero las propiedades básicas
- `random(num: Integer)`: void, construye numVuelos

Este tipo los diseñaremos como una clase con métodos estáticos.

OcupacionVuelo

Inmutable

Propiedades:

- `CódigoVuelo: String`, básica
- `Fecha: LocalDateTime`, básica, fecha-hora de salida
- `NumPasajeros: Integer`, básica, debe ser mayor o igual a cero
- `Vuelo: Vuelo`, derivada
- `Llegada: LocalDateTime`, fecha-hora de llegada
- `FechaSalida: LocalDate`, fecha de salida
- `HoraSalida: LocalTime`, hora de salida

Representación:

- `CódigoVuelo, FechaSalida, HoraSalida`



Igualdad: Si tienen el mismo código. Aunque con la restricción impuesta al código todas las propiedades básicas serán iguales

Métodos de Factoría:

- `parse(text:String): Vuelo`
- `of(Codigo: String, Nombre: String): Aerolínea`
- `random(): Vuelo`, un vuelo construido aleatoriamente con los aeropuertos y las aerolíneas disponibles.

OcupacionesVuelos

Propiedades:

- `size: Integer`, Derivada
- `get(i): OcupacionVuelo`, la ocupación i.
- `stream: Stream<OcupacionVuelo>`, derivada

Incluir aquí como propiedades algunos de los cálculos enunciados arriba

Operaciones:

- `addOcupacionVuelo(oc: OcupacionVuelo): void`, Añade una ocupación de un vuelo
- `removeOcupacionVuelo(oc: OcupacionVuelo): void`, Elimina una ocupación de un vuelo

Representación:

- Lista de ocupaciones una por línea

Métodos de Factoría:

- `leeOcupaciones(fichero:String)`, void, lee de un fichero las propiedades básicas
- `random(num: Integer, año: Integer): void`, construye num ocupaciones en el año dado a partir de los vuelos disponibles

Este tipo los diseñaremos como una clase con métodos estáticos.



Preguntas

Una clase con funciones estáticas para responder las preguntas que nos hemos planteado.

Implementación

Empecemos por el primer tipo: *Aeropuerto*

```
record Aeropuerto(String codigo, String ciudad, String pais,
String nombre) {
    public static Aeropuerto parse(String text) {
        String[] campos = text.split(",");
        String codigo = campos[2];
        String ciudad = campos[3];
        String pais = campos[1];
        String nombre = campos[0];
        return Aeropuerto.of(codigo,ciudad,pais,nombre);
    }
    public static Aeropuerto of(String codigo,
                                String ciudad,
                                String pais,
                                String nombre) {
        return new Aeropuerto(codigo,ciudad,pais,nombre);
    }
}
```

El tipo *Aeropuertos* es un tipo mutable que mantiene la población de aeropuertos junto con los mecanismos para buscar un aeropuerto dado su código y las posibilidades de añadir o eliminar aeropuertos. Solo vamos a necesitar un objeto que mantenga la población de aeropuertos por eso diseñamos la clase *Aeropuertos* con un atributo estático privado *aeropuertos* y un conjunto de métodos estáticos para gestionar ese atributo.



```
public class Aeropuertos {  
    private static List<Aeropuerto> aeropuertos;  
  
    public static String string() {  
        return String.format("Aeropuertos\n\t%s",  
            Aeropuertos.aeropuertos.stream()  
                .map(a->a.toString())  
                .collect(Collectors.joining("\n\t")));  
    }  
}
```

La lectura de los datos desde un fichero es de la forma:

```
public static void leeAeropuertos(String fichero) {  
    List<Aeropuerto> aeropuertos =  
        FileTools.streamFromFile(fichero)  
            .map(x -> Aeropuerto.parse(x))  
            .collect(Collectors.toList());  
    Aeropuertos.aeropuertos = aeropuertos;  
}
```

La diferentes propiedades derivadas de la lista de aeropuertos: obtener un aeropuerto dado su código

```
private static Map<String,Aeropuerto> codigosAeropuertos= null;  
public static Aeropuerto aeropuerto(String codigo) {  
    if(codigosAeropuertos == null)  
        codigosAeropuertos = Aeropuertos.aeropuertos  
            .stream().collect(  
                Collectors.toMap(a->a.codigo(),a->a));  
    return codigosAeropuertos.get(codigo);  
}
```

La implementación la hacemos asumiendo que el cálculo se va a repetir muchas veces y no queremos repetirlo. Para ello diseñamos una variable privada que mantiene el resultado. Este resultado se devuelve salvo que sea null en cuyo caso se hace el cálculo y se guarda en la variable.

Obtener la ciudad donde está un aeropuerto dado su código. Usamos la misma técnica anterior.



```
private static Map<String, String> ciudadDeAeropuerto = null;
public static String ciudadDeAeropuerto(String codigo) {
    if(ciudadDeAeropuerto == null)
        ciudadDeAeropuerto =
            Aeropuertos.aeropuertos.stream()
                .collect(Collectors.toMap(a->a.codigo(),
                                          a->a.ciudad()));
    return ciudadDeAeropuerto.get(codigo);
}
```

Conjunto de aeropuertos en una ciudad.

```
private static Map<String, Set<Aeropuerto>>
    aeropuertosEnCiudad= null;
public static Set<Aeropuerto> aeropuertosEnCiudad(
    String ciudad) {
    if(aeropuertosEnCiudad == null)
        aeropuertosEnCiudad = Aeropuertos.aeropuertos.stream()
            .collect(Collectors.groupingBy(
                a->a.ciudad(),Collectors.toSet()));
    return aeropuertosEnCiudad.get(ciudad);
}
```

Los métodos para añadir o eliminar un aeropuerto deben tener en cuenta el diseño de los métodos anteriores.

```
public static void addAeropuerto(Aeropuerto a) {
    Aeropuertos.aeropuertosEnCiudad = null;
    Aeropuertos.ciudadDeAeropuerto = null;
    Aeropuertos.codigosAeropuertos = null;
    Aeropuertos.aeropuertos.add(a);
}
```

```
public static void removeAeropuerto(Aeropuerto a) {
    Aeropuertos.aeropuertosEnCiudad = null;
    Aeropuertos.ciudadDeAeropuerto = null;
    Aeropuertos.codigosAeropuertos = null;
    Aeropuertos.aeropuertos.remove(a);
}
```

Otras funciones que ofrece el tipo son:



```
public static Integer size() {  
    return Aeropuertos.aeropuertos.size();  
}
```

```
public static Stream<Aeropuerto> stream() {  
    return aeropuertos.stream();  
}
```

```
public static Aeropuerto get(Integer i) {  
    return aeropuertos.get(i);  
}
```

El tipo *Aerolinea* sigue los detalles del diseño:

```
public record Aerolinea(String codigo, String nombre) {  
    public static Aerolinea parse(String text) {  
        String[] campos = text.split(",");  
        String codigo = campos[0];  
        String nombre = campos[1].trim();  
        return new Aerolinea(codigo,nombre);  
    }  
    public static Aerolinea of(String codigo,  
        String nombre) {  
        return new Aerolinea(codigo,nombre);  
    }  
}
```

El tipo *Aerolineas* es un tipo mutable que mantiene la población de aerolineas junto con los mecanismos para buscar una aerolinea dado su código y las posibilidades de añadir o eliminar aeropuertos. Vamos a necesitar un solo objeto que mantenga la población de aeropuertos por eso diseñamos la clase *Aerolineas* con un atributo estático privado *aeropuertos* y un conjunto de métodos estáticos para gestionar ese atributo.



```
public class Aerolineas {  
    private static List<Aerolinea> aeroLineas;  
    public static String string() {  
        return String.format("Aerolineas\n\t%s",  
            Aerolineas.aeroLineas.stream()  
                .map(a->a.toString())  
                .collect(Collectors.joining("\n\t")));  
    }  
}
```

La lectura de los datos de un fichero:

```
public static void leeAerolineas(String fichero) {  
    List<Aerolinea> datos =  
        FileTools.streamFromFile(fichero)  
            .map(x -> Aerolinea.parse(x))  
            .toList();  
    Aerolineas.aeroLineas = datos;  
}
```

Otras propiedades del tipo:

```
public static Integer size() {  
    return Aerolineas.aeroLineas.size();  
}
```

```
public static Stream<Aerolinea> stream() {  
    return Aerolineas.aeroLineas.stream();  
}
```

```
public static Aerolinea get(Integer i) {  
    return Aerolineas.aeroLineas.get(i);  
}
```

El tipo Vuelo siguiendo el diseño:



```
public record Vuelo(String codigoAerolinea, String numero,
    String codigoDestino, String codigoOrigen,
    Double precio, Integer numPlazas, Duration duracion,
    LocalTime hora, DayOfWeek diaSemana
) {
...
}
```

El método de factoría asociado

```
public static Vuelo of(String codigo, String numero,
    String codeDestino, String codeOrigen, Double precio,
    Integer numPlazas, Duration duracion, LocalTime hora,
    DayOfWeek diaSemana) {
    return new Vuelo(codigo, numero, codeDestino, codeOrigen,
        precio, numPlazas, duracion, hora, diaSemana);
}
```

Su método de parsing asumiendo que una línea del fichero vuelos es de la forma:

```
TP,0705,BER,KTW,294,170,287,14:50,FRIDAY
```

```
public static Vuelo parse(String text) {
    String[] campos = text.split(",");
    String codigo = campos[0];
    String numero = campos[1];
    String codeDestino = campos[2];
    String codeOrigen = campos[3];
    Double precio = Double.parseDouble(campos[4]);
    Integer numPlazas = Integer.parseInt(campos[5]);
    Duration duracion = Duration.of(
        Integer.parseInt(campos[6]), ChronoUnit.MINUTES);
    LocalTime hora = LocalTime.parse(campos[7],
        DateTimeFormatter.ofPattern("HH:mm"));
    DayOfWeek diaSemana =
        DayOfWeek.valueOf(campos[8]);
    return Vuelo.of(codigo, numero, codeDestino, codeOrigen,
        precio, numPlazas, duracion, hora, diaSemana);
}
```

La representación:



```
@Override  
public String toString() {  
    DateTimeFormatter formatter =  
        DateTimeFormatter.ofPattern("HH:mm");  
    return String.format(  
        new Locale("en", "US"), "%s,%s,%s,%s,.2f,%d,%d,%s,%s",  
        codigoAerolinea, numero, codigoDestino, codigoOrigen,  
        precio, numPlazas, duracion.toMinutes(),  
        hora.format(formatter), diaSemana.toString());  
}
```

Un método estático para crear un vuelo aleatorio dado

```
private static Random rnd = new Random(System.nanoTime());  
public static Vuelo random() {  
    Integer e = rnd.nextInt(Aerolineas.size());  
    String codigo = Aerolineas.get(e).codigo();  
    String numero = String.format("%04d", rnd.nextInt(1000));  
    Integer ad = rnd.nextInt(Aeropuertos.size());  
    String codeDestino = Aeropuertos.get(ad).codigo();  
    Integer ao;  
    do {  
        ao = rnd.nextInt(Aeropuertos.size());  
    } while (ao == ad);  
    String codeOrigen = Aeropuertos.get(ao).codigo();  
    Double precio = 1000 * rnd.nextDouble();  
    Integer numPlazas = rnd.nextInt(300);  
    Duration duracion = Duration.of(rnd.nextInt(360),  
        ChronoUnit.MINUTES);  
    LocalTime hora =  
        LocalTime.of(rnd.nextInt(24), rnd.nextInt(60));  
    DayOfWeek diaSemana = DayOfWeek.of(1 + rnd.nextInt(7));  
    return new Vuelo(codigo, numero, codeDestino, codeOrigen,  
        precio, numPlazas, duracion, hora, diaSemana);  
}
```

Para conseguir un vuelo aleatorio elegimos en primer lugar una aerolínea de forma aleatoria. Esto lo conseguimos sorteando un entero aleatorio entre 0 y n siendo n el número de aerolíneas. Con el método *get* de Aerolíneas tenemos la aerolínea buscada. Posteriormente elegimos un número de vuelo entre los vuelos de la compañía. Este es un entero



aleatorio entre 0 y 1000. Ese número lo guardamos en una cadena de texto de cuatro caracteres.

Siguiendo las mismas ideas escogemos aleatoriamente dos aeropuertos, el de origen y el destino, asegurándonos que son diferentes lo que conseguimos con un bucle do. Repetimos el segundo intento mientras que sean iguales.

Escogemos aleatoriamente el precio, el número de plazas y la duración en minutos dentro de rangos adecuados. Por último elegimos aleatoriamente un día de la semana.

Algunas propiedades derivadas

```
public String ciudadDestino() {  
    return Aeropuertos  
        .ciudadDeAeropuerto(this.codigoDestino);  
}
```

```
public String ciudadOrigen() {  
    return Aeropuertos  
        .ciudadDeAeropuerto(this.codigoOrigen);  
}
```

```
public String codigo() {  
    return this.codigoAerolinea+this.numero;  
}
```

La clase Vuelos gestiona la población de vuelos. Como en los tipos anteriores creamos una sola clase con una lista privada de vuelos.

```
public class Vuelos {  
    private static List<Vuelo> vuelos;  
    ...
```

Algunas propiedades derivadas



```
public static Stream<Vuelo> stream() {  
    return vuelos.stream();  
}
```

```
public static Vuelo get(Integer index) {  
    return vuelos.get(index);  
}
```

```
private static Integer numVuelos;  
public static Integer size() {  
    return numVuelos;  
}
```

```
private static Map<String, Vuelo> codigosVuelos;  
public static Vuelo vuelo(String codigo) {  
    if(codigosVuelos == null)  
        Vuelos.codigosVuelos = Vuelos.stream()  
            .collect(Collectors.toMap(Vuelo::codigo,x->x));  
    return codigosVuelos.get(codigo);  
}
```

Métodos de factoría para crear vuelos aleatorios y leerlos de un fichero

```
public static void random(Integer numVuelos) {  
    List<Vuelo> vuelos = toList(IntStream.range(0,numVuelos)  
        .boxed().map(e->Vuelo.random()));  
    Vuelos.vuelos = vuelos;  
    Vuelos.numVuelos = Vuelos.vuelos.size();  
}
```

```
public static void leeFicheroVuelos(String fichero) {  
    List<Vuelo> vuelos = FileTools.streamFromFile(fichero)  
        .map(x -> Vuelo.parse(x))  
        .toList();  
    Vuelos.vuelos = vuelos;  
    Vuelos.numVuelos = Vuelos.vuelos.size();  
}
```

Y métodos para añadir o eliminar un vuelo



```
public void addVuelo(Vuelo v) {  
    Vuelos.codigosVuelos = null;  
    Vuelos.vuelos.add(v);  
    Vuelos.numVuelos +=1;  
}
```

```
public void removeVuelo(Vuelo v) {  
    Vuelos.codigosVuelos = null;  
    Vuelos.vuelos.remove(v);  
    Vuelos.numVuelos -=1;  
}
```

El tipo *OcupacionVuelo* representa una ocupación concreta de un vuelo.
Seguimos el diseño para implementarlo

```
public record OcupacionVuelo(String codigoVuelo,  
    LocalDateTime fecha, Integer numPasajeros) {  
  
    public static OcupacionVuelo of(String codeVuelo,  
        LocalDateTime fecha, Integer numPasajeros) {  
        return new OcupacionVuelo(codeVuelo, fecha, numPasajeros);  
    }  
}
```

Con un método parse

```
public static OcupacionVuelo parse(String text) {  
    String[] campos = text.split(",");  
    String codeVuelo = campos[0];  
    LocalTime t = Vuelos.get(codeVuelo).hora();  
    LocalDate d = LocalDate.parse(campos[1],  
        DateTimeFormatter.ofPattern("dd/MM/yyyy"));  
    LocalDateTime fecha = LocalDateTime.of(d, t);  
    Integer numPasajeros = Integer.parseInt(campos[2]);  
    return OcupacionVuelo.of(codeVuelo, fecha, numPasajeros);  
}
```

Algunas propiedades derivadas

```
public Vuelo vuelo() {  
    return Vuelos.get(this.codigoVuelo);  
}
```



```
public LocalDateTime llegada() {  
    Vuelo vuelo = Vuelos.get(this.codigoVuelo);  
    return LocalDateTime.of(fecha.toLocalDate(),  
                           vuelo.hora()).plus(vuelo.duracion());  
}
```

```
public LocalDate fechaSalida() {  
    return this.fecha().toLocalDate();  
}
```

```
public LocalTime horaSalida() {  
    return this.fecha().toLocalTime();  
}
```

```
@Override  
public String toString() {  
    DateTimeFormatter formatter =  
        DateTimeFormatter.ofPattern(  
            "yyyy-MM-dd HH:mm:ss");  
    return String.format("%s,%s,%d", codigoVuelo,  
                        fecha.format(formatter), numPasajeros);  
}
```

Un método para crear una ocupación de vuelo aleatoria. Esta ocupación se crea para un vuelo en un año dado.



```
private static Random rnd = new Random(System.nanoTime());
public static OcupacionVuelo random(Vuelo v, Integer anyo) {
    String codeVuelo = v.codigo();
    Integer np = v.numPlazas();
    LocalTime t = v.hora();
    DayOfWeek dw = v.diaSemana();
    LocalDate d = Stream.iterate(LocalDate.of(anyo, 1, 1),
        dt -> dt.plus(1, ChronoUnit.DAYS))
        .filter(dt -> dt.getDayOfWeek().equals(dw))
        .findFirst().get();
    d = d.plus(7 * rnd.nextInt(53), ChronoUnit.DAYS);
        //53 semanas en un año
    LocalDateTime fecha = LocalDateTime.of(d, t);
    Integer numPasajeros = np > 0 ? rnd.nextInt(np) : 0;
    return new OcupacionVuelo(codeVuelo, fecha, numPasajeros);
}
```

La gestión de la población de ocupaciones se hace en la clase *OcupacionesVuelos* con las mismas ideas que los tipos anteriores.

```
public class OcupacionesVuelos {
    private static List<OcupacionVuelo> ocupaciones;
    public static Stream<OcupacionVuelo> stream() {
        return ocupaciones.stream();
    }
    public static OcupacionVuelo get(Integer i) {
        return ocupaciones.get(i);
    }
    public static Integer size() {
        return ocupaciones.size();
    }
    ...
}
```

Esta clase tiene un método para leer los datos de un fichero



```
public static void leeFicheroOcupaciones(String fichero) {  
    List<OcupacionVuelo> r =  
        FileTools.streamFromFile(fichero)  
        .map(x -> OcupacionVuelo.parse(x))  
        .collect(Collectors.toList());  
    OcupacionesVuelos.ocupaciones = r;  
}
```

Otro método para generar un número de ocupaciones aleatorias en un año dado.

```
public static void random(Integer numOcupaciones,  
    Integer anyo) {  
    Integer n = Vuelos.size();  
    List<OcupacionVuelo> r = toList(  
        IntStream.range(0, numOcupaciones).boxed()  
        .map(e -> OcupacionVuelo.random(  
            Vuelos.get(rnd.nextInt(n)), anyo)));  
    OcupacionesVuelos.ocupaciones = r;  
}
```

Con estos tipos diseñados podemos empezar a implementar métodos para responder a distintas preguntas. Todos esos métodos los incluimos en la clase Preguntas.

-
1. *Dada una cadena de caracteres s encontrar el número total de pasajeros a ciudades destino que tienen como prefijo s (esto es, comienzan por s)*
-

La versión funcional

```
public static Integer numeroDepasajeros(String prefix) {  
    IntStream st = OcupacionesVuelos.stream()  
        .filter(ocp -> ocp.vuelo().ciudadDestino()  
            .startsWith(prefix))  
        .mapToInt(v -> v.numPasajeros());  
    return st.sum();  
}
```

La versión imperativa



```
public static Integer numeroDepasajeros2(String prefix) {  
    List<OcupacionVuelo> ls = OcupacionesVuelos.stream()  
        .toList();  
    Integer sum = 0;  
    for(OcupacionVuelo ocp:ls) {  
        if(ocp.vuelo().ciudadDestino()  
            .startsWith(prefix)) {  
            Integer numPasajeros = ocp.numPasajeros();  
            sum = sum + numPasajeros;  
        }  
    }  
    return sum;  
}
```

-
2. Dado un conjunto de ciudades destino s y una fecha f el método devuelve cierto si existe un vuelo en la fecha f con destino en s
-

La versión imperativa y la funcional

```
public static Boolean hayDestino2(Set<String> destinos,  
LocalDate f) {  
    List<OcupacionVuelo> ls = OcupacionesVuelos.stream()  
        .toList();  
    Boolean a = false;  
    for(OcupacionVuelo ocp:ls) {  
        if(ocp.fecha().toLocalDate().equals(f)) {  
            if(destinos.contains(  
                ocp.vuelo().ciudadDestino())) {  
                a = true;  
                break;  
            }  
        }  
    }  
    return a;  
}
```

La versión funcional



```
public static Boolean hayDestino(Set<String> destinos,
    LocalDate f) {
    Stream<OcupacionVuelo> st = OcupacionesVuelos.stream()
        .filter(ocp -> ocp.fecha().toLocalDate().equals(f));
    return st.anyMatch(ocp -> destinos.contains(
        ocp.vuelo().ciudadDestino())));
}
```

-
3. Dada una fecha f encontrar el conjunto de ciudades destino diferentes de todos los vuelos de fecha f
-

La versión funcional

```
public static Set<String> destinosDiferentes(LocalDate f) {
    Stream<String> st = OcupacionesVuelos.stream()
        .filter(ocp -> ocp.fecha().toLocalDate().equals(f))
        .map(ocp -> ocp.vuelo().ciudadDestino());
    return st.collect(Collectors.toSet());
}
```

La versión imperativa

```
public static Set<String> destinosDiferentes2(LocalDate f) {
    List<OcupacionVuelo> ls = OcupacionesVuelos.stream()
        .toList();
    Set<String> a = new HashSet<>();
    for(OcupacionVuelo ocp:ls) {
        if(ocp.fecha().toLocalDate().equals(f)) {
            String ciudadDestino =
                ocp.vuelo().ciudadDestino();
            a.add(ciudadDestino);
        }
    }
    return a;
}
```

-
4. Dado un año obtener un SortedMap que relacione cada destino, ordenado de mayor a menor, con el total de pasajeros a ese destino en ese año
-



```
public static SortedMap<String, Integer>
    totalPasajerosADestino(Integer a) {
    Stream<OcupacionVuelo> st = OcupacionesVuelos.stream()
        .filter(ocp -> ocp.fecha().getYear() == a);

    return st.collect(Collectors.groupingBy(
        ocp -> ocp.vuelo().ciudadDestino(),
        () -> new TreeMap<String, Integer>(
            Comparator.reverseOrder()),
        Collectors.summingInt(
            ocp -> ocp.numPasajeros()))));
}
```

```
public static SortedMap<String, Integer>
    totalPasajerosADestino2(Integer any) {
    List<OcupacionVuelo> ls =
        OcupacionesVuelos.stream().toList();
    SortedMap<String, Integer> a = new TreeMap<String,
    Integer>(
        Comparator.reverseOrder());
    for(OcupacionVuelo ocp:ls) {
        if(ocp.fecha().getYear() == any) {
            String key = ocp.vuelo().ciudadDestino();
            if(a.containsKey(key)) {
                Integer numPasajeros =
                    a.get(key)+ocp.numPasajeros();
                a.put(key,numPasajeros);
            } else {
                a.put(key,ocp.numPasajeros());
            }
        }
    }
    return a;
}
```

-
5. Dado un destino encontrar el código de la aerolinea primer vuelo con plazas libres a ese destino
-



La versión funcional

```
public static String primerVuelo(String destino) {  
    Stream<OcupacionVuelo> st = OcupacionesVuelos.stream()  
        .filter(ocp -> ocp.vuelo().ciudadDestino()  
            .equals(destino))  
        .filter(ocp->ocp.vuelo()  
            .numPlazas() > ocp.numPasajeros())  
        .filter(ocp -> ocp.fecha()  
            .isAfter(LocalDateTime.now()));  
  
    return st.min(  
        Comparator.comparing(OcupacionVuelo::fecha))  
            .get()  
            .vuelo()  
            .codigoAerolinea();  
}
```

La versión imperativa

```
public static String primerVuelo2(String destino) {  
    List<OcupacionVuelo> ls =  
        OcupacionesVuelos.stream().toList();  
    OcupacionVuelo a = null;  
    for(OcupacionVuelo ocp:ls) {  
        if(ocp.vuelo().ciudadDestino().equals(destino) &&  
            ocp.vuelo().numPlazas() > ocp.numPasajeros() &&  
            ocp.fecha().isAfter(LocalDateTime.now())) {  
            if(a==null || ocp.fecha().isBefore(a.fecha())) {  
                a = ocp;  
            }  
        }  
    }  
    if(a==null) throw new IllegalArgumentException(  
        "La lista está vacía");  
    return a.vuelo().codigoAerolinea();  
}
```

-
6. Encontrar para los vuelos completos un Map que haga corresponder a cada ciudad destino la media de los precios de los vuelos a ese destino
-



```

private static Double preM(List<OcupacionVuelo> ls) {
    return ls.stream().mapToDouble(
        ocp->ocp.vuelo().precio()).average().getAsDouble();
}

public static Map<String, Double> precioMedio() {
    Stream<OcupacionVuelo> st = OcupacionesVuelos.stream()
        .filter(ocp -> ocp.numPasajeros()
            .equals(ocp.vuelo().numPlazas())));
    return st.collect(Collectors.groupingBy(
        ocp -> ocp.vuelo().ciudadDestino(),
        Collectors.collectingAndThen(Collectors.toList(),
            g->preM(g))));
}

```

Para simplificar el código hemos diseñado un método privado que calcula a partir de una lista de ocupaciones de vuelos el precio medio de esas ocupaciones. La versión imperativa en la que hemos diseñado un método privado con el mismo objetivo.

```

public static Map<String, Double> precioMedio2() {
    List<OcupacionVuelo> ls =
        OcupacionesVuelos.stream().toList();
    Map<String, List<OcupacionVuelo>> a = new HashMap<>();
    for(OcupacionVuelo ocp:ls) {
        if(ocp.numPasajeros()
            .equals(ocp.vuelo().numPlazas())))
            String key = ocp.vuelo().ciudadDestino();
            if(a.containsKey(key)) {
                a.get(key).add(ocp);
            } else {
                List<OcupacionVuelo> lsn = new ArrayList<>();
                lsn.add(ocp);
                a.put(key, lsn);
            }
        }
    }
    Map<String, Double> r = new HashMap<>();
    for(String key:a.keySet()) {
        r.put(key,Preguntas.preM2(a.get(key)));
    }
    return r;
}

```



```
private static Double preM2(List<OcupacionVuelo> ls) {
    Double sum = 0.;
    Integer n = 0;
    for(OcupacionVuelo ocp:ls) {
        sum = sum + ocp.vuelo().precio();
        n = n +1;
    }
    if(n==0) throw new
        IllegalArgumentException("El grupo está vacío");
    return sum/n;
}
```

-
- 7 Encontrar un Map tal que dado un entero n haga corresponder a cada fecha la lista de los n destinos con los vuelos de mayor duración
-

Para claridad de la versión funcional diseñamos primero un orden sobre ocupaciones de vuelos y posteriormente una función que nos da las ciudades destino de los n vuelos de mayor duración entre todas las ocupaciones incluidas en una lista.

```
private static Comparator<OcupacionVuelo> cmp =
    Comparator.comparing((OcupacionVuelo ocp) -> ocp.vuelo()
        .duracion().getSeconds()).reversed();

private static List<String> mayorDuracion(
    List<OcupacionVuelo> ls, Integer n) {
    Stream<String> st = ls.stream()
        .sorted(cmp).limit(n)
        .map(ocp -> ocp.vuelo().ciudadDestino());
    return st.toList();
}
```

Con esos elementos ya podemos implementar la versión funcional.



```
public static Map<LocalDate, List<String>>
    destinosConMayorDuracion(Integer n) {
    Stream<OcupacionVuelo> st = OcupacionesVuelos.stream();
    return st.collect(Collectors.groupingBy(
        oc -> oc.fecha().toLocalDate(),
        Collectors.collectingAndThen(
            Collectors.toList(),
            ls->mayorDuracion(ls, n))));
```

La versión imperativa queda como ejercicio

-
8. Dada una fecha *f* obtener el precio medio de los vuelos con salida posterior a *f*. Si no hubiera vuelos devuelve 0.0
-

```
public static Double precioMedio(LocalDateTime f) {
    DoubleStream st = OcupacionesVuelos.stream()
        .filter(ocp -> ocp.fecha().isAfter(f))
        .mapToDouble(ocp -> ocp.vuelo().precio());
    return st.average().orElse(0.0);
}
```

-
9. Obtener un Map que haga corresponder a cada destino un conjunto con las fechas de los vuelos a ese destino
-

```
public static Map<String, Set<LocalDate>> fechasADestino() {
    Stream<OcupacionVuelo> st = OcupacionesVuelos.stream();
    return st.collect(Collectors.groupingBy(
        ocp -> ocp.vuelo().ciudadDestino(),
        Collectors.mapping(
            OcupacionVuelo::fechaSalida,
            Collectors.toSet())));
}
```

-
10. El destino con mayor número de vuelos
-



```
public static String destinoConMasVuelos() {  
    Map<String, Integer> numVuelosDeDestino = Vuelos.stream()  
        .collect(Collectors.groupingBy(  
            Vuelo::codigoDestino,  
            Collectors.collectingAndThen(  
                Collectors.counting(), Long::intValue)));  
    return numVuelosDeDestino.keySet().stream()  
        .max(Comparator.comparing(d->numVuelosDeDestino.get(d)))  
        .get();  
}
```

-
11. Dado un entero m encontrar un conjunto ordenado con las duraciones de todos los vuelos cuya duración es mayor que m minutos
-

```
public static SortedSet<Duration> duraciones(Integer m) {  
    Stream<Duration> st = Vuelos.stream()  
        .map(Vuelo::duracion)  
        .filter(d->d.getSeconds() / 60. > m);  
    return st.collect(Collectors.toCollection(  
        () ->new TreeSet<>()));  
}
```

-
12. Dado un número n encontrar un conjunto con los destinos de los vuelos que están entre los n que más duración tienen
-

```
public static Set<String> destinosMayorDuracion(Integer n) {  
    Stream<String> st = Vuelos.stream()  
        .sorted(Comparator.comparing(Vuelo::duracion).reversed())  
        .limit(n)  
        .map(Vuelo::codigoDestino);  
    return st.collect(Collectors.toSet());  
}
```

-
13. Dado un número n devuelve un conjunto con los n destinos con más vuelos
-



```
public static Set<String> entreLosMasVuelos(Integer n) {  
    Map<String, Long> vuelosADestino = Vuelos.stream()  
        .collect(Collectors.groupingBy(  
            Vuelo::codigoDestino, Collectors.counting()));  
    return vuelosADestino.keySet().stream()  
        .sorted(Comparator.comparing(  
            d->vuelosADestino.get(d)).reversed())  
        .limit(n)  
        .collect(Collectors.toSet());  
}
```

-
14. Dado un número entero n devuelve una lista con los destinos que tienen más de n vuelos
-

```
public static List<String> masDeNVuelos(Integer n) {  
    Map<String, Long> vuelosADestino = Vuelos.stream()  
        .collect(Collectors.groupingBy(  
            Vuelo::codigoDestino, Collectors.counting()));  
    return vuelosADestino.keySet().stream()  
        .filter(d->vuelosADestino.get(d) > n)  
        .collect(Collectors.toList());  
}
```

-
15. Obtener un Map que relacione cada destino con el porcentaje de los vuelos del total que van a ese destino
-

```
public static Map<String, Double>  
porcentajeADestinoOcupacionesVuelos() {  
    Integer n = OcupacionesVuelos.size();  
    return OcupacionesVuelos.stream()  
        .map(ocp->ocp.vuelo())  
        .collect(Collectors.groupingBy(  
            Vuelo::codigoDestino,  
            Collectors.collectingAndThen(  
                Collectors.toList(), g->(1.0*g.size()) / n)));  
}
```

-
16. Devuelve un Map que haga corresponder a cada ciudad destino el vuelo
-



de más barato de cualquier procedencia

```
public static Map<String, Vuelo> masBarato() {  
    return Vuelos.stream().collect(  
        Collectors.groupingBy(  
            Vuelo::ciudadDestino,  
            Collectors.collectingAndThen(  
                Collectors.reducing(  
                    BinaryOperator.minBy(Comparator.comparing(Vuelo::precio)),  
                    e->e.get())));  
}
```

17. Encontrar un Map que haga corresponder a cada destino el número de fechas distintas en las que hay vuelos a ese destino

```
public static Map<String, Integer> fechasDistintas() {  
    Stream<OcupacionVuelo> st = OcupacionesVuelos.stream();  
    return st.collect(Collectors.groupingBy(  
        ocp -> ocp.vuelo().ciudadDestino(),  
        Collectors.mapping(OcupacionVuelo::fecha,  
            Collectors.collectingAndThen(  
                Collectors.toSet(), s->s.size()))));  
}
```



Objetos geométricos

En este ejercicio nos proponemos implementar un conjunto de objetos geométricos: *Punto2D*, *Segmento2D*, *Circulo2D*, *Poligono2D*. Todos ellos comparten propiedades que estarán representadas en objetos del tipo *Objeto2D*. A su vez un *Agregado2D* será un agregado de los objetos geométricos anteriores. Todos ellos podrán ser sometidos a algunas operaciones geométricas y representados gráficamente.

Junto a los anteriores vamos a diseñar otros tipos de objetos que aparecen en la geometría del plano como *Vector2D* y *Recta2D*.

Tipos

Vector2D

Empecemos creando el tipo inmutable *Vector2D* mediante un record.

```
record Vector2D(Double x,Double y) {  
    ...  
}
```

El diseño del tipo *Vector2D* lo suponemos conocido. En la implementación hemos decidido escoger las propiedades básicas *x*, *y*. Las propiedades modulo y angulo las consideramos derivadas.

Y diversos métodos de factoría



```
static Vector2D of(Double x, Double y) {
    return new Vector2D(x, y);
}
```

```
static Vector2D ofGrados(Double modulo, Double angulo){
    Preconditions.checkNotNull(modulo > 0, String.format(
        "El módulo debe ser mayor o igual a cero y es %.2f",
        modulo));
    return ofRadianes(modulo, Math.toRadians(angulo));
}
```

```
Vector2D ofRadianes(Double modulo, Double angulo) {
    Preconditions.checkNotNull(modulo >= 0, String.format(
        "El módulo debe ser mayor o igual a cero y es %.2f",
        modulo));
    return of(modulo*Math.cos(angulo),
              modulo*Math.sin(angulo));
}
```

Ahora diseñamos las propiedades derivadas

```
public Double modulo() {
    return Math.abs(Math.hypot(x, y));
}
public Double angulo() {
    return Math.atan2(y, x);
}
public Double anguloEnGrados() {
    return Math.toDegrees(this.angulo());
}
public Vector2D ortogonal() {
    return Vector2D.of(-this.y, this.x);
}
public Vector2D unitario() {
    return Vector2D.ofRadianes(1., this.angulo());
}
public Vector2D opuesto() {
    return Vector2D.of(-x, -y);
}
```

La representación podría ser limitando los decimales a dos:



```
@Override  
public String toString() {  
    return String.format("(%.2f,%.2f)",this.x, this.y);  
}
```

Ahora diseñamos la operaciones sumar, restar, rotar y producto por escalar con los métodos correspondientes.

```
public Vector2D add(Vector2D v) {  
    return Vector2D.of(this.x+v.x,this.y+v.y);  
}  
public Vector2D minus(Vector2D v) {  
    return Vector2D.of(this.x-v.x,this.y-v.y);  
}  
public Vector2D rota(Double angulo) {  
    return Vector2D.ofRadianes(this.modulo(),  
        this.angulo()+angulo);  
}  
public Vector2D multiply(Double factor) {  
    return Vector2D.of(this.x*factor,this.y*factor);  
}
```

Y por último el producto escalar, el vectorial, el ángulo con otro vector y la proyección sobre otro vector:

```
public Double multiplicaVectorial(Vector2D v) {  
    return this.x()*v.y()-this.y()*v.x();  
}  
public Double multiplicaEscalar(Vector2D v) {  
    return this.x()*v.x()+this.y()*v.y();  
}  
public Double angulo(Vector2D v) {  
    return Math.asin(this.multiplicaVectorial(v) /  
        (this.modulo()*v.modulo()));  
}  
public Vector2D projetaSobre(Vector2D v){  
    Vector2D u = v.unitario();  
    return u.multiply(this.multiplicaEscalar(u));  
}
```



Objeto2D

Los siguientes elementos que queremos diseñar comparten un conjunto de métodos y propiedades comunes. Estos son:

- Rotar un ángulo con respecto a un punto
- Trasladar según un vector
- Hacer una homotecia según un punto y una razón
- Proyectar sobre una recta
- Obtener el Shape de un objeto geométrico. Donde Sahpe es un tipo adecuado para la representación gráfica del objeto

Todos estos métodos comunes los incluimos en el interfaz Objeto2D. Este interfaz especifica un contrato con métodos que serán concretados en los tipos que lo implementen.

Definida este tipo ObjetoGeometrico2D podemos ya tener, como más adelante veremos, agregados de objetos geométricos cada uno de los cuales implementará de una forma concreta cada uno de los métodos anteriores. El tipo ObjetoGeometrico2D será de la forma.

```
public interface ObjetoGeometrico2D {  
    ObjetoGeometrico2D rota(Punto2D p, Double angulo);  
    ObjetoGeometrico2D traslada(Vector2D v);  
    ObjetoGeometrico2D homotecia(Punto2D p, Double factor);  
    ObjetoGeometrico2D proyectaSobre(Recta2D r);  
    ObjetoGeometrico2D simetrico(Recta2D r);  
    ObjetoGeometrico2D transform(Ventana v);  
    void show(Ventana v);  
}
```

Recta2D

Este no es un tipo que hereda de ObjetoGeometrico2D pero vamos a diseñarlo primero para poder concretar el resto de los tipos.

Una recta viene especificada por un punto, el tipo Punto2D que diseñaremos más tarde, y un vector de tipo Vector2D.



```

public record Recta2D(Punto2D punto, Vector2D vector) {
    public static Recta2D of(Punto2D punto,
                            Vector2D vector) {
        return new Recta2D(punto, vector);
    }
    public static Recta2D of(Punto2D p1, Punto2D p2) {
        return new Recta2D(p1, p2.minus(p1));
    }
    @Override
    public String toString() {
        return String.format("(%.2f,%.2f)",
                            this.punto, this.vector).toString();
    }
}

```

Como vemos se han diseñado dos factorías: mediante un punto y un vector o por dos puntos.

Punto2D

Veamos la implementación de un punto en el plano con dos propiedades básicas: x,y . Hemos de tener en cuenta que es un objeto geométrico que implementará el contrato definido por ObjetoGeometrico2D.

```

record Punto2D(Double x,Double y) implements
    ObjetoGeometrico2D, Comparable<Punto2D>, ShapeDeObjeto {
    private static Punto2D cero = Punto2D.of(0.,0.);
    public static Punto2D origen(){
        return Punto2D.cero;
    }
    public static Punto2D of(Double x, Double y) {
        return new Punto2D(x, y);
    }
    ...
}

```

Se ha diseñado el método de factoría `of`. También otro método que devuelva el origen de coordenadas. Los hemos diseñado con el patrón Singleton. Con este patrón hacemos posible que cada vez que pidamos el origen nos devuelva el mismo punto y no uno nuevo cada vez.

Diseñamos también un método de factoría para hacer el parsing de un punto. Asumimos que el formato de un punto es “(3.45,5.67)”.



```
public static Punto2D parse(String text) {
    String text2 = text.substring(1, text.length()-1);
    String[] campos = text2.split(",");
    return new Punto2D(Double.parseDouble(campos[0]),
                      Double.parseDouble(campos[1]));
}
```

La representación para conseguir un formato similar al usado en el parsing.

```
public String toString() {
    return String.format("(%.2f,%.2f)",this.x(),this.y());
}
```

Algunas propiedades: conversión a vector, suma y resta con un vector.

```
public Punto2D add(Vector2D v) {
    return Punto2D.of(this.x+v.x(),this.y+v.y());
}
public Punto2D minus(Vector2D v) {
    return Punto2D.of(this.x-v.x(),this.y-v.y());
}
public Vector2D minus(Punto2D p) {
    return Vector2D.of(this.x-p.x(),this.y-p.y());
}
public Vector2D vector() {
    return Vector2D.of(this.x, this.y);
}
```

La distancia a otro punto se determina por el teorema de pitágoras:

```
public Double distanciaA(Punto2D p) {
    Double dx = this.x()-p.x();
    Double dy = this.y()-p.y();
    return Math.sqrt(dx*dx+dy*dy);
}
```

Para el cálculo del cuadrante usamos la sentencia if_then_else



```
public Cuadrante cuadrante() {
    Cuadrante c;
    if(this.x() >=0 && this.y() >=0)
        c = Cuadrante.PRIMER_CUADRANTE;
    else if(this.x() <=0 && this.y() >=0)
        c = Cuadrante.SEGUNDO_CUADRANTE;
    else if(this.x() <=0 && this.y() <=0)
        c = Cuadrante.TERCER_CUADRANTE;
    else c = Cuadrante.CUARTO_CUADRANTE;
    return c;
}
```

Ahora no falta diseñar los métodos heredados de ObjetoGeometrico2D para darles la funcionalidad concreta para este tipo. En primer lugar *traslada*.

```
public Punto2D translada(Vector2D v) {
    return this.add(v);
}
```

Como vemos simplemente hemos sumado un vector al punto. La operación rotar un ángulo con respecto a otro punto se implementa fácilmente usando los correspondientes método de los vectores.

```
public Punto2D rota(Punto2D p, Double angulo) {
    Vector2D v = this.minus(p).rota(angulo);
    return p.add(v);
}
```

La homotecia de un factor dado con respecto a un punto la podemos implementar multiplicando el vector que se forma por el factor de escala.

```
public Punto2D homotecia(Punto2D p, Double factor) {
    return p.add(Vector2D.of(p, this).multiply(factor));
}
```

La proyección de un punto sobre una recta lo podemos conseguir escogiendo un punto p en la recta ($r.punto()$) y sumándole la proyección sobre el vector de la recta del vector formado por el punto y p .



```
public Punto2D proyectaSobre(Recta2D r) {  
    return r.punto().add(this.minus(r.punto())  
        .proyectaSobre(r.vector()));  
}
```

El simétrico con respecto a una recta se consigue de forma similar.

```
public Punto2D simetrico(Recta2D r) {  
    Punto2D p = this.proyectaSobre(r);  
    return p.vector().multiply(2.)  
        .minus(this.vector()).punto();  
}
```

Segmento2D

Un segmento es un objeto geométrico, hereda por tanto de ObjetoGeometrico2D, definido por dos puntos.

```
public record Segmento2D(Punto2D p1, Punto2D p2)  
    implements ObjetoGeometrico2D, ShapeDeObjeto {  
    public static Segmento2D of(Punto2D p1, Punto2D p2) {  
        return new Segmento2D(p1, p2);  
    }  
    public Vector2D vector() {  
        return Vector2D.of(this.p1, this.p2);  
    }  
    public Double longitud() {  
        return p1.distanciaA(p2);  
    }  
    ...  
}
```

Los métodos heredados de Objeto2D se implementan reutilizando los correspondientes de Punto2D.

```
public Segmento2D rota(Punto2D p, Double angulo) {  
    return Segmento2D.of(this.p1.rota(p, angulo),  
        this.p2.rota(p, angulo));  
}
```

```
public Segmento2D traslada(Vector2D v) {  
    return Segmento2D.of(this.p1.traslada(v),  
        this.p2.traslada(v));  
}
```



```
public Segmento2D homotecia(Punto2D p, Double factor) {  
    return Segmento2D.of(this.p1.homotecia(p, factor),  
        this.p2.homotecia(p, factor));  
}
```

```
public Segmento2D proyectaSobre(Recta2D r) {  
    return Segmento2D.of(this.p1.proyectaSobre(r),  
        this.p2.proyectaSobre(r));  
}
```

```
public Segmento2D simetrico(Recta2D r) {  
    return Segmento2D.of(this.p1.simetrico(r),  
        this.p2.simetrico(r));  
}
```

Círculo2D

El círculo es un objeto geométrico definido por un punto y un radio. No incluimos.

```
public record Circulo2D(Punto2D centro, Double radio)  
    implements ObjetoGeometrico2D, ShapeDeObjeto{  
  
    public static Circulo2D of(Punto2D centro, Double radio){  
        Preconditions.checkArgument(radio>=0, String.format(  
            "El radio debe ser mayor o igual a cero  
            y es %.2f", radio));  
        return new Circulo2D(centro, radio);  
    }  
}
```

El área y la longitud de la circunferencia se pueden calcular con las fórmulas conocidas.

```
public Double area() {  
    return Math.PI*this.radio*this.radio;  
}  
public Double perimetro() {  
    return 2*Math.PI*this.radio;  
}  
...
```

En el método de factoría no aseguramos que el radio sea mayor o igual a cero. Algunas propiedades del tipo son:



Los métodos heredados de Objeto2D se implementan reutilizando los métodos de Punto2D.

```
public Circulo2D rota(Punto2D p, Double angulo) {  
    return Circulo2D.of(this.centro.rota(p,angulo),  
        this.radio);  
}  
public Circulo2D translada(Vector2D v) {  
    return Circulo2D.of(this.centro.translada(v),  
        this.radio);  
}
```

```
public Circulo2D homotecia(Punto2D p, Double factor) {  
    return Circulo2D.of(this.centro.homotecia(p,factor),  
        this.radio*factor);  
}  
public Circulo2D simetrico(Recta2D r) {  
    return Circulo2D.of(this.centro.simetrico(r),  
        this.radio);  
}
```

La proyección dará lugar a un segmento cuyo centro estará en la proyección del centro del circulo y cuyo módulo es el dobel del radio.

```
@Override  
public Segmento2D proyectaSobre(Recta2D r) {  
    Punto2D pc = this.centro.proyectaSobre(r);  
    Vector2D u = r.vector().unitario();  
    return Segmento2D.of(pc.add(u.multiply(this.radio)),  
        pc.add(u.multiply(-this.radio)));  
}
```

Polígono2D

El polígono es un objeto geométrico definido por una lista de puntos.

Podemos definir métodos de factoría para construir polígonos particulares: triángulos, triángulos equiláteros, cuadrados, rectángulos, etc.



```
public record Poligono2D(List<Punto2D> vertices) implements
    ObjetoGeometrico2D {

    public static Poligono2D empty() {
        return new Poligono2D(new ArrayList<>());
    }
    public static Poligono2D ofPuntos(Punto2D... lp) {
        return new Poligono2D(Arrays.asList(lp));
    }
    public static Poligono2D ofPuntos(List<Punto2D> lp) {
        return new Poligono2D(lp);
    }
    public static Poligono2D triangulo(Punto2D p1,
        Punto2D p2, Punto2D p3) {
        return new Poligono2D(List.of(p1, p2, p3));
    }
    ...
}
```

```
public static Poligono2D trianguloEquilatero(Punto2D p1,
    Vector2D lado) {
    var p2 = p1.add(lado);
    var p3 = p1.add(lado.rota(Math.PI / 3));
    return new Poligono2D(List.of(p1,p2,p3));
}
```

```
public static Poligono2D cuadrado(Punto2D p, Vector2D lado) {
    var p2 = p.add(lado);
    var p3 = p.add(lado).add(lado.ortogonal());
    var p4 = p.add(lado.ortogonal());
    return new Poligono2D(List.of(p,p2,p3,p4));
}
```

```
public static Poligono2D rectangulo(Punto2D p, Vector2D base,
    Double altura) {
    var p2 = p.add(base);
    var p3 = p.add(base).add(base.ortogonal()
        .multiply(altura));
    var p4 = p.add(base.ortogonal()).multiply(altura));
    return new Poligono2D(List.of(p,p2,p3,p4));
}
```

Veamos algunas de sus propiedades: area, perímetro, número de vértices, vertice(i), lado(i), diagonal(i). El vertice(i) es un Punto2D, el lado(i) y la diagonal(i) ambos Vector2D.



```

public Punto2D vertice(Integer i) {
    Integer n = this.numeroDeVertices();
    Preconditions.checkElementIndex(i, n);
    return vertices.get(i);
}
public Vector2D lado(Integer i) {
    Integer n = this.numeroDeVertices();
    Preconditions.checkElementIndex(i, n);
    return Vector2D.of(this.vertices.get(i),
        this.vertices.get((i + 1) % n));
}
public Vector2D diagonal(Integer i, Integer j) {
    Integer n = this.numeroDeVertices();
    Preconditions.checkElementIndex(i, n);
    Preconditions.checkElementIndex(j, n);
    return Vector2D.of(this.vertices.get(i),
        this.vertices.get(j));
}

```

Ahora podemos calcular el perímetro como la suma de los módulos de los lados y el área como la suma del área de los triángulos definidos por dos diagonales consecutivas.

```

public Double area() {
    Integer n = this.numeroDeVertices();
    Double area = IntStream.range(1, n - 1)
        .mapToDouble(i -> this.diagonal(0, i)
            .multiplicaVectorial(this.lado(i))).sum();
    return area / 2;
}

```

```

public Double perimetro() {
    Integer n = this.numeroDeVertices();
    Double ln = IntStream.range(0, n)
        .mapToDouble(i -> this.lado(i).modulo()).sum();
    return ln;
}

```

Los métodos heredados de Objeto2D pueden ser implementados basados en los de Punto2D.



```
public Poligono2D rota(Punto2D p, Double angulo) {  
    return Poligono2D.ofPuntos(this.vertices.stream()  
        .map(x -> x.rota(p, angulo))  
        .collect(Collectors.toList()));  
}
```

```
public Poligono2D traslada(Vector2D v) {  
    return Poligono2D.ofPuntos(this.vertices.stream()  
        .map(p -> p.traslada(v))  
        .collect(Collectors.toList()));  
}
```

```
public Poligono2D homotecia(Punto2D p, Double factor) {  
    return Poligono2D.ofPuntos(this.vertices.stream()  
        .map(x -> x.homotecia(p, factor))  
        .collect(Collectors.toList()));  
}
```

```
public Poligono2D proyectaSobre(Recta2D r) {  
    return Poligono2D.ofPuntos(this.vertices.stream()  
        .map(x -> x.proyectaSobre(r))  
        .collect(Collectors.toList()));  
}
```

```
public Poligono2D simetrico(Recta2D r) {  
    return Poligono2D.ofPuntos(this.vertices.stream()  
        .map(x -> x.simetrico(r))  
        .collect(Collectors.toList()));  
}
```

Agregado2D

Un agregado será una colección de objetos geométricos de tipo ObjetoGeometrico2D. Esto implica que un agregado puede contener objetos simples u otros agregados. Este tipo implementa los métodos heredados de ObjetoGeometrico2D.

El tipo Agregado2D es mutable por ello lo implementamos como una clase. Podremos añadir objetos geométricos al agregado.



```
public class AgregadoGeometrico2D implements  
    ObjetoGeometrico2D {  
    public static AgregadoGeometrico2D empty() {  
        return new AgregadoGeometrico2D();  
    }  
    public static AgregadoGeometrico2D of(  
        ObjetoGeometrico2D... objetos) {  
        return new AgregadoGeometrico2D(  
            Arrays.asList(objetos));  
    }  
    private List<ObjetoGeometrico2D> objetos;  
  
    private AgregadoGeometrico2D() {  
        this.objetos = new ArrayList<>();  
    }
```

```
private AgregadoGeometrico2D(  
    List<ObjetoGeometrico2D> objetos) {  
    this.objetos = objetos;  
}
```

```
public int size() { return objetos.size(); }
```

```
public boolean isEmpty() {  
    return objetos.isEmpty();  
}  
public boolean contains(Object o) {  
    return objetos.contains(o);  
}  
public boolean add(ObjetoGeometrico2D e) {  
    return objetos.add(e);  
}  
public boolean remove(Object o) {  
    return objetos.remove(o);  
}  
public boolean addAll(  
    Collection<? extends ObjetoGeometrico2D> c) {  
    return objetos.addAll(c);  
}
```

Diseñamos los métodos heredados de `ObjetoGeometrico2D` invocando a los respectivos métodos de los objetos en el agregado. Estos métodos tienen una implementación específica para cada tipo de objetos.



```
public AgregadoGeometrico2D rota(Punto2D p, Double angulo) {  
    return new AgregadoGeometrico2D(this.objetos.stream()  
        .map(x->x.rota(p,angulo))  
        .collect(Collectors.toList()));  
}
```

```
public AgregadoGeometrico2D traslada(Vector2D v) {  
    return new AgregadoGeometrico2D(objetos.stream()  
        .map(x->x.traslada(v))  
        .collect(Collectors.toList()));  
}
```

```
public AgregadoGeometrico2D homotecia(Punto2D p,  
    Double factor) {  
    return AgregadoGeometrico2D.of(this.objetos.stream()  
        .map(x->x.homotecia(p, factor))  
        .collect(Collectors.toList()));  
}
```

```
public AgregadoGeometrico2D proyectaSobre(Recta2D r) {  
    return AgregadoGeometrico2D.of(this.objetos.stream()  
        .map(x->x.proyectaSobre(r))  
        .collect(Collectors.toList()));  
}
```

```
public AgregadoGeometrico2D simetrico(Recta2D r) {  
    return AgregadoGeometrico2D.of(this.objetos.stream()  
        .map(x->x.simetrico(r))  
        .collect(Collectors.toList()));  
}
```

Todos estos tipos pueden usarse en la forma que se encuentra en el repositorio.

Representación gráfica de los objetos geométricos

La representación gráfica de estos objetos podemos hacerla usando el tipo Shape de la librería de Java y un ventana para representar los objetos



gráficos. La idea es asociar a cada objeto geométrico un Shape y posteriormente mostrar todos los de un agregado.

Para obtener el tipo Shape adecuado debemos transformar las coordenadas del objeto la sistema de coordenadas de la ventana. Veamos con algo de detalle el caso del Cirulo2D. Este este caso usamos el tipo *Ellipse2D.Double*, que implementa Shape. Una elipse viene definida por un rectángulo conocidas las cordenadas del límite superior izquierdo, *x,y*, la anchura *w* y la altura *h*.

```
public Ellipse2D.Double(double x, double y, double w, double h);
```

Para ubicar el Circulo2D en la ventana debemos transformar las coordenadas y tamaño del circulo.

```
public Circulo2D transform(Ventana v) {
    Punto2D c = this.centro.transform(v);
    return Circulo2D.of(c, this.radio() * v.escala);
}
```

El Shape asociado al circulo lo podemos obtener transformando el circulo según el sistema de coordenadas y la escala de la ventana y obteniendo el punto superior izquierdo del cuadrado donde se inserta el círculo.

```
public Shape shape(Ventana v) {
    Circulo2D ct = (Circulo2D) this.transform(v);
    Punto2D sc = ct.centro().add(Vector2D.of(-1., -1.))
        .multiply(ct.radio()));
    return new Ellipse2D.Double(sc.x(), sc.y(),
        2 * ct.radio(), 2 * ct.radio());
}
```

Finalmente mostramos el shape obtenido en la ventana.

```
public void show(Ventana v) {
    v.draw(this.shape(v));
}
```

La ventana es un tipo implementado mediante un clase que tiene las siguientes propiedades.



```
public class Ventana {  
    public Integer xMax;  
    public Integer yMax;  
    public Integer xCentro;  
    public Integer yCentro;  
    public Double escala;  
    public Vector2D centro;  
    public Function<Double,Double> xt;  
    public Function<Double,Double> yt;  
    public void draw(Shape sp) {...}  
    public void fill(Shape sp) {...}  
    public DoublePair transform(Double x, Double y) {...}  
    public void axes() { ... }  
}
```



WhatsApp

En este ejercicio vamos a analizar los mensajes de un grupo de Whatsapp. En concreto, podremos obtener toda esta información:

- Número de mensajes a lo largo del tiempo
- Número de mensajes según el día de la semana, o la hora del día
- Número de mensajes escritos por cada participante en el grupo
- Palabras más utilizadas en el grupo
- Palabras más características de cada participante en el grupo

Para llevar a cabo la tarea, necesitamos un archivo de log de un grupo de Whatsapp. Un fichero de log ficticio que podemos utilizar para desarrollar los ejercicios (generado a partir de los diálogos de la primera temporada de The Big Bang Theory).

El formato del fichero que genera Whatsapp varía según se trate de la versión Android o iOS. Veamos un ejemplo de cada caso:

- *Android:*

```
26/02/16, 09:16 - Leonard: De acuerdo, ¿cuál es tu punto?  
26/02/16, 16:16 - Sheldon: No tiene sentido, solo creo que es  
una buena idea para una camiseta.
```

- *iOS:*



[26/2/16 09:16:25] Leonard: De acuerdo, ¿cuál es tu punto?
 [26/2/16 16:16:54] Sheldon: No tiene sentido, solo creo que
 es una buena idea para una camiseta.

Como se puede observar, cada mensaje del chat viene precedido por la fecha, la hora, y el nombre del usuario. El formato no es CSV, como en otros ejercicios, sino que es, digamos, más libre. Así, la fecha ocupa el principio de una línea, que puede venir seguida de una coma. Tras un espacio en blanco, viene la hora (que puede contener o no los segundos), seguida de un espacio, un guión y otro espacio, o bien de dos puntos y un espacio, según la versión de Whatsapp. Por último, tenemos el nombre del usuario, acabando en dos puntos, y tras otro espacio, aparece el texto del mensaje.

Tipos

PalabraUsuario:

Propiedades

- Palabra: String, básica
- Usuario: String, básica

Mensaje:

Propiedades:

- Fecha: LocalDate, básica
- Hora: LocalTime, básica
- Usuario: String, básica
- Texto: String, básica

Representación: en el formato Android

Factoría:

- Parse dado un texto con todas las partes del mensaje

Conversacion:

Propiedades:

- Mensajes: List<Mensaje>, básica
- PalabrasHuecas: Set<String> , básica



- MensajesPorPropiedad(Function<Mensaje,P> f): Map<P,List<Mensaje>>, derivada
- NumeroMensajesPorPropiedad(Function<Mensaje,P> f): dict[P,Integer], derivada
- NumeroPalabrasUsuario: Map<String,Integer>, derivada, NPU
- FrecuenciaPalabrasUsuario: Map<PalabraUsuario,Integer>, derivada, FPU
- NumeroPalabrasResto: Map<String,Integer>, derivada, NPR
- FrecuenciaPalabrasResto: Map<PalabraUsuario,Integer>, derivada, FPR
- ImportanciaPalabra(usuario:String, umbral:Integer): Double. IP
- PalabrasCaracteristicasUsuario(usuario:String, umbral:Integer): dict[String,Double], PCU

Notas

$$IP(u, p) = \frac{FPU(u, p) * NPR(u)}{NPU(u) * FPR(u, p)}$$

Asumimos que $FPR(u, p)$ tiene un valor mínimo, caso de que no use p, de un valor dado por ejemplo 0.00001

PalabrasCaracteristicasUsuario(usuario,umbral): Un diccionario con la importancia de las palabras para el usuario que tengan frecuencia superior a umbral

Expresiones regulares

De cada línea del fichero habrá que extraer los distintos campos de información (fecha, hora, usuario y texto de cada mensaje). La mejor manera de extraer esta información es mediante el uso de *expresiones regulares*. Estas expresiones nos permiten definir patrones en los que pueden encajar distintos trozos de texto, y extraer información a partir



de distintos trozos de la cadena que ha sido reconocida. Cada lenguaje de programación tiene pequeñas diferencias para tratar las expresiones regulares. En este el caso el patrón que describe la estructura de los mensajes es:

```
String RE =
" (?<fecha>\\d\\d?/\\d\\d?/\\d\\d?) , ? (?<hora>\\d\\d?:\\d\\d)
- (?<usuario>[^:]*) : (?<texto>.+) ";
```

- Cada uno de los tramos encerrados entre paréntesis, y con ? seguido de un nombre, se corresponde con un dato que vamos a extraer. Hay cuatro tramos.
- El primer tramo de la cadena, *(?<fecha>\\d\\d?/\\d\\d?/\\d\\d?)*, es un patrón que encaja con las fechas que aparecen en los mensajes. Los \\d indican dígitos, y las interrogaciones indican partes que pueden aparecer o no.
- El segundo tramo, *(?<hora>\\d\\d?:\\d\\d)*, encaja con las horas.
- El tercer tramo, *(?<usuario>[^:]*)*, reconoce los nombres de usuario. La expresión *[^:]** significa "cualquier secuencia de caracteres hasta dos puntos excluidos" (es decir, que reconoce todo el trozo de texto que viene antes de los dos puntos).
- Y el último tramo, *(?<texto>.+)*, captura el texto de los mensajes.

La forma de trabajar tras declarar el patrón es:

```
String RE = patrón
m = re.match(RE, text) // si ha encontrado matching o no
Integer n = m. groupCount(); //número de grupos. Deberían ser 4
String fecha = m.group("fecha");
String hora = m.group("hora");
String usuario = m.group("usuario");
String texto = m.group("texto");
...
```

Veamos la implementación de los tipos. En primer lugar la tupla PalabraUsuario.



```

public record PalabraUsuario(String palabra, String usuario) {
    public static PalabraUsuario of(String palabra,
                                    String usuario) {
        return new PalabraUsuario(palabra, usuario);
    }
    @Override
    public String toString() {
        return String.format("(%s,%s)", palabra, usuario);
    }
}

```

En segundo lugar Mensaje

```

public record Mensaje(LocalDate fecha, LocalTime hora,
                      String usuario, String texto) {

    public static Mensaje of(LocalDate fecha,
                           LocalTime hora, String usuario, String texto) {
        return new Mensaje(fecha, hora, usuario, texto);
    }
    ...
    @Override
    public String toString() {
        return String.format("%s,%s,%10s,%s",
                            fecha.toString(), hora.toString(), usuario, texto);
    }
}

```

Y el método de parsing para el tipo Mensaje



```

private static String RE = "(?<fecha>\\d\\\\d?/\\d\\\\d?/\\d\\\\d?)"
    "(?<hora>\\d\\\\d?:\\d\\\\d) - (?<usuario>[^:]*) : (?<texto>.+)";
private static Pattern pattern = Pattern.compile(RE);

public static Mensaje parse(String mensaje) {
    Matcher m = Mensaje.pattern.matcher(mensaje);
    Preconditions.checkArgument(m.find() &&
        m.groupCount() == 4,
        String.format("Formato incorrecto en grupos = %d",
            mensaje = %s", m.groupCount(), mensaje));
    String[] pf = m.group("fecha").split("/");
    LocalDate fecha = LocalDate.of(
        Integer.parseInt("20"+pf[2]),
        Integer.parseInt(pf[1]),
        Integer.parseInt(pf[0]));
    String[] ph = m.group("hora").split(":");
    LocalTime hora = LocalTime.of(
        Integer.parseInt(ph[0]), Integer.parseInt(ph[1]));
    String usuario = m.group("usuario");
    String texto = m.group("texto");
    return new Mensaje(fecha, hora, usuario, texto);
}

```

Por último el tipo Conversacion

```

public class Conversacion {
    private List<Mensaje> mensajes;
    private Set<String> palabrasHuecas;
    private Conversacion(String file) {
        this.mensajes = FileTools.lineasFromFile(file)
            .stream().filter(x->x.length()>0)
            .map(m->Mensaje.parse(m))
            .collect(Collectors.toList());
        this.palabrasHuecas = FileTools.lineasFromFile(
            "resources/palabras_huecas.txt").stream()
            .filter(x->x.length()>0)
            .collect(Collectors.toSet());
    }
    public static Conversacion ofFile(String file) {
        return new Conversacion(file);
    }
}

```

El resto de los métodos queda como ejercicio aunque pueden verse resueltos en el repositorio.



Montecarlo

Usaremos ahora el método de Montecarlo para estimar la probabilidad de victoria de una determinada mano de póker. El póker es un juego de cartas de apuestas, en el que el jugador que tiene la mejor combinación de cartas gana toda la cantidad apostada en una mano. Tiene elementos de muchos juegos antiguos pero su expansión, con reglas parecidas a las que se usan hoy en día, tuvo lugar desde Nueva Orleans en el siglo XIX. Se juega con la denominada baraja inglesa que tiene trece cartas de cuatro palos distintos. Las trece cartas ordenadas de menor a mayor según su valor son:

['2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K', 'A']

No hay diferencia de valor entre los cuatro palos: *Tréboles*, *Corazones*, *Picas*, *Diamantes*.

El póker se basa en un *ranking* bastante heterogéneo de jugadas. En la siguiente tabla se muestran las jugadas con las que trabajaremos en este ejercicio, junto con sus descripciones y la probabilidad de cada combinación. Consideraremos una baraja de 52 cartas sin comodines, por lo que la jugada del *repóker* no será posible:



Jugada	Descripción	Probabilidad
Escalera real ER	Cinco cartas seguidas del mismo palo del 10 al as	4 de 2.598.960 ($1,539 \cdot 10^{-4} \%$)
Escalera de color EC	Cinco cartas consecutivas del mismo palo	36 de 2.598.960 ($1,385 \cdot 10^{-3} \%$)
Póker P	Cuatro cartas iguales en su valor	624 de 2.598.960 ($2,4 \cdot 10^{-2} \%$)
Full F	Tres cartas iguales en su valor (tercia), más otras dos iguales en su valor (pareja)	3.744 de 2.598.960 ($0,144 \%$)
Color C	Cinco cartas del mismo palo, sin ser necesariamente consecutivas	5.108 de 2.598.960 ($0,196 \%$)
Escalera E	Cinco cartas consecutivas sin importar el palo	10.200 de 2.598.960 ($0,392 \%$)
Trío T	Tres cartas iguales de valor	54.912 de 2.598.960 ($2,111 \%$)
Doble pareja D	Dos pares de cartas del mismo valor (par y par)	123.552 de 2.598.960 ($4,759 \%$)
Pareja P	Dos cartas del mismo valor (y tres diferentes)	1.098.240 de 2.598.960 ($42,257 \%$)
Carta alta C	Gana quien tiene la carta más alta	1.302.540 de 2.598.960 ($50,117 \%$)

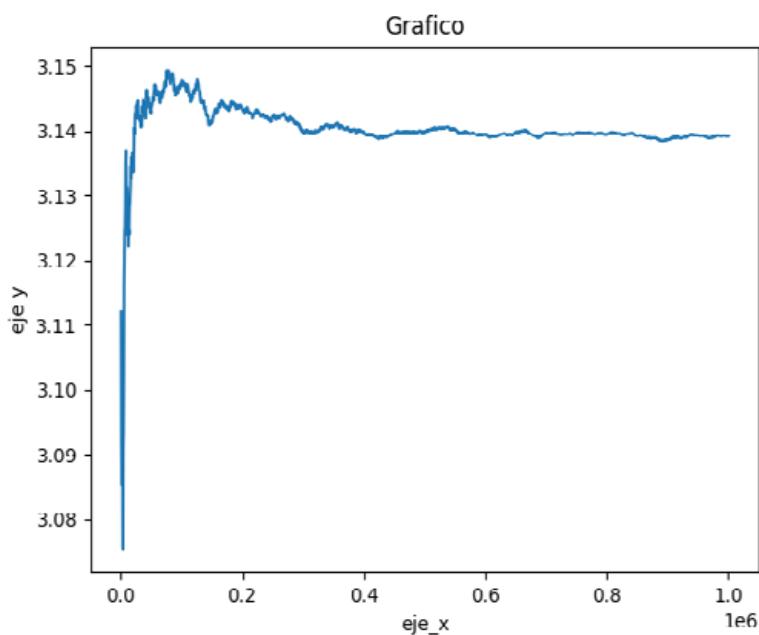


El método de Montecarlo

El objetivo del problema estimar en que proporción una mano gana a otra escogida aleatoriamente. Para ello usaremos el método de Montecarlo.

Este funcionamiento de este método podemos comprenderlo estimando el área de un círculo por este método. Para ello sorteamos dos números reales en los intervalos (-1,1.). Todos los puntos caen en un cuadrado de área 1. El número de puntos que están a una distancia del $(0,0) < 1$ forman un círculo cuyo área queremos estimar. Sea f la proporción de puntos que están a una distancia del centro igual o menor de 1. El área del cuadrado es 4 luego la estimación del área del círculo es $4f$.

Como el radio del círculo es 1 su área es π por lo que el método nos puede servir también para hacer una estimación de π . Si repetimos el cálculo para distintos valores de n cada vez más grandes vemos como el resultado se va aproximando a π .



El código para una aproximación es:



```

public static Double montecarlo(Integer n) {
    List<Punto2D> st = IntStream.range(0,n)
        .mapToObj(e-> Punto2D.of(
            Math.random()*2-1,Math.random()*2-1))
        .toList();

    Integer pt = 0;
    Integer pi = 0;
    for(Punto2D p:st) {
        pt = pt + 1;
        if(p.distanciaAlOrigen() <= 1)
            pi = pi + 1;
    }
    return 4.*pi/pt;
}

```

Una aproximación con n=1000000 es 3.143496.

Tipos

Valor: enum [2,3,4,5,6,7,8,9,10,J,Q,K,A]

Palo: enum [T,C,P,D]

Jugada: enum [C,P,D,T,E,C,F,P,EC,ER]

Carta:

Propiedades:

- *Valor:* Integer en [0..13), básica, ordenadas de mayor a menor valor
- *Palo:* Integer en [0..4), básica
- *Id:* Integer en [0..53), derivada, $id = palo * 4 + valor$;
- *NombresDeValores:* List<String>, compartida *NombresDePalos:* List<Character>, compartida,

Representación: "VP".

Factoría:

- Dado el id
- Dado valor y palo



- Dada su representación textual

Orden natural: por valor

Mano:

Propiedades:

- *Cartas: List<Carta>, tamaño numeroDeCartas, ordenadas por valor*
- *NumeroDeCartas: Integer, compartida, 5*
- *Son5ValoresConsecutivos: Boolean, derivada*
- *FrecuenciasDeValores: Map<Integer,Integer>, derivada*
- *ValoresOrdenadosPorFrecuencias(): List<Integer>, de mayor a menor*
- *ValorMasFrecuente: Integer, derivada, el valor que más se repite*
- *FrecuenciasDePalos: Map<Integer,Integer>,*
- *PalosOrdenadosPorFrecuencias: List<Integer>, de mayor a menor*
- *EsColor: Boolean, derivada*
- *EsEscalera: Boolean, derivada*
- *EsPoker: Boolean, derivada*
- *EsEscaleraDeColor: Boolean, derivada*
- *EsFull: Boolean, derivada*
- *EsTrio, Boolean, derivada*
- *EsDoblePareja: Boolean, derivada*
- *EsPareja: Boolean, derivada*
- *EsEscaleraReal: Boolean, derivada*
- *EsMano: Boolean, derivada*
- *Jugada: Jugada, derivada*
- *Fuerza: Double, derivada, la probabilidad de ganar a un mano aleatoria*

Representación: Representación de las cartas entre paréntesis y separadas por comas

Factoría:

- *Random, selecciona 5 cartas sin repetición al azar*
- *Dada un Set<Carta> de tamaño 5*



- Dada la representación textual de una mano

Orden Natural: Por tipo de jugada y si es la misma por el valor que más se repite

Veamos el código de los tipos y algunos métodos. El resto se puede encontrar en el repositorio.

```
public record Card(Integer id, Integer palo, Integer valor)
    implements Comparable<Card> {

    public static List<String> nombreValores =
        Arrays.asList("2", "3", "4", "5", "6",
                      "7", "8", "9", "10", "J", "Q", "K", "A");
    public static List<Character> symbolsPalos =
        Arrays.asList('C', 'H', 'S', 'D');
    public static List<String> nombrePalos =
        Arrays.asList("clubs", "hearts", "spades",
                     "diamonds");

    @Override
    public String toString() {
        return Card.nombreValores.get(valor) +
            Card.symbolsPalos.get(palo);
    }
    @Override
    public int compareTo(Card card) {
        return this.valor.compareTo(card.valor);
    }
    ...
}
```

Los métodos de factoría

```
public static Card of(String text) {
    Character pc = text.charAt(text.length()-1);
    String ind = text.substring(0, text.length()-1);
    Integer palo = Card.symbolsPalos.indexOf(pc);
    Integer valor = Card.nombreValores.indexOf(ind);
    return Card.of(valor, palo);
}
```



```
public static Card of(Integer id) {
    Preconditions.checkArgument(id >= 0 &&
        id < 52, "No es posible");
    Integer palo = id % 4;
    Integer valor = id % 13;
    return new Card(id,palo,valor);
}
```

```
public static Card of(Integer valor, Integer palo) {
    return new Card(palo*4+valor,valor, palo);
}
```

El tipo Mano

```
public class Mano implements Comparable<Mano>{

    public static Integer numeroDeCartas = 5;

    public enum Jugada {EscaleraReal,EscaleraDeColor,Poker,
        Full,Color,Escalera,Trio,DoblePareja,
        Pareja,CartaAlta}

    private static Random rnd =
        new Random(System.nanoTime());

    private List<Card> cartas;
    public static Mano of(List<Card> cartas) {
        return new Mano(cartas);
    }
    ...
}
```

```
@Override
public int compareTo(Mano mano) {
    Integer r = -this.getJugada()
        .compareTo(mano.getJugada());
    if (r == 0) r = this.valoresOrdenadosPorFrecuencias()
        .get(0).compareTo(valoresOrdenadosPorFrecuencias()
        .get(0));
    return r;
}
```



```

@Override
public String toString() {
    return this.cartas.stream()
        .map(c->c.toString())
        .collect(Collectors.joining(", ", " (", " )"));
}

```

Y los métodos de factoría

```

public static Mano parse(String txt) {
    txt = txt.substring(1,txt.length()-1);
    String[] partes = txt.split(",");
    List<Card> cartas = Arrays.stream(partes)
        .map(x->Card.parse(x)).collect(Collectors.toList());
    return Mano.of(cartas);
}

```

```

public static Mano random() {
    List<Card> cartas = new ArrayList<>();
    for(int i = 0;i<numeroDeCartas;i++) {
        Integer n = Mano.rnd.nextInt(52);
        Card card = Card.of(n);
        cartas.add(card);
    }
    return new Mano(cartas);
}

```

Algunos métodos. Diseñamos un conjunto de predicados para decidir el tipo de jugada y formamos una lista con ellos. La implementación de los predicados puede verse en el repositorio.

```

private static List<Predicate<Mano>> jugadas = Arrays.asList(
    Mano::esEscaleraReal, Mano::esEscaleraDeColor,
    Mano::esPoker, Mano::esFull, Mano::esColor,
    Mano::esEscalera, Mano::esTrio, Mano::esDoblePareja,
    Mano::esPareja,
    Mano::esCartaMasAlta);

```

Buscamos el primer predicado que se satisface en esa lista y este nos dará el tipo de jugada. Es una forma compacta de hacer un *if-then-else*.



```

public Jugada getJugada() {
    if(this.jugada==null) {
        Integer r = IntStream.range(0, jugadas.size())
            .filter(i->jugadas.get(i).test(this))
            .findFirst()
            .getAsInt();
        this.jugada = Jugada.values()[r];
    }
    return this.jugada;
}

```

Simulamos por Montecarlo para obtener la probabilidad de ganar a una mano aleatoria.

```

public static Double fuerza(Mano mano, Integer n) {
    Integer gana = 0;
    Integer pierde = 0;
    Integer empata = 0;
    for(int i=0;i<n;i++) {
        Mano mr = Mano.random();
        if(mano.compareTo(mr)<0) pierde++;
        else if(mano.compareTo(mr)>0) gana++;
        else empata++;
    }
    return ((double)gana) / (gana+pierde+empata);
}

public Double fuerza(Integer n) {
    return fuerza(this,n);
}

```



Universo

En este ejemplo abordamos la simulación de objetos celestes. Usaremos los conceptos OO: herencia y clases abstractas.

Vamos a considerar las siguientes entidades: universo, objetos celestes, estrellas, planetas con una órbita dada, cometas, cometas acelerados y cometas erráticos. Todos ellos dentro de un universo. Cada una de estas entidades dará lugar a un tipo. Unos tipos serán más generales que otros. Los tipos más específicos heredarán de otros más generales y posiblemente añadan algunas propiedades o funcionalidad.

Nos interesa, además, explorar la reutilización de funcionalidad.

Tipos

Veamos en primer lugar los tipos.

Orbita2D

Es una elipse en el plano 2D. Esta elipse puede estar inclinada con respecto al eje horizontal

Propiedades

- a: Double, semieje mayor, $a>0$
- b: Double, semieje menor, $b>0$
- c: Double, semidistancia focal
- α : Double, angulo del eje mayor con el eje X, $0 \leq \alpha \leq \pi/2$



- e: Double, excentricidad, $0 \leq e < 1$
- d: Double,
- d1: Double, mínima distancia al foco
- d2: Double, máxima distancia al foco
- T: Double, período, $T > 0$
- r(θ :Double):Vector2D, vector desde un foco a un punto de la elipse. El vector resultante forma un angulo de θ con el eje mayor. Donde θ es el ángulo del radio vector con el eje mayor
- $\omega(\theta$:Double):Double, velocidad angular

Invariante

$$a^2 + b^2 = c^2$$

$$e = c/a$$

$$d = a e^2$$

$$d1 = a - c$$

$$d2 = a + c$$

$$|r(\theta)| = d/(1+e \cos \theta)$$

$$\omega(\theta) = \frac{2\pi a}{T |r(\theta)|} \sqrt{\frac{2a}{|r(\theta)|} - 1}$$

CuerpoCeleste:

Es un tipo muy general del cual heredarán otros. Alguna de su funcionalidad quedará sin concretar por lo que decimos que es un tipo abstracto. Las propiedades u operaciones que no concretarán su funcionalidad las etiquetamos como abstractas.

Los cuerpos celestes se mueven dentro de un universo que tendrá un tamaño. Los cuerpos serán visibles si están dentro del universo e invisibles si están fuera.

Propiedades

- nombre: String,
- diametro: Integer
- color: Color
- universo: Universo2D, el universo donde está este cuerpo celeste
- location: Location, la ubicación en el gráfico



- coordenadas: Punto2D, abstracta
- esVisible:Boolean, abstracta
- distanciaA(c:CuerpoCeleste):Double,abstracta
- location: Location

Operaciones

- unPaso:void, abstracta, da un paso de simulación
- cambiaPropiedades:void, abstracta, cambia las propiedades
- mover: void, mueve
- mostrarCuerpoCeleste: void, muestra el cuerpo celeste
- ocultarCuerpoCeleste: void, oculta el cuerpo celeste

Location es un tipo enumerado

```
public static enum Location{Inside,Left,Right,Up,Down,OutSide}
```

Estrella extiende CuerpoCeleste

Es un cuerpo celeste, por lo tanto hereda de CuerpoCeleste, que tiene una posición fija en el universo

Una estrella concretará todas las propiedades y operaciones abstractas de CuerpoCeleste

Planeta extiende CuerpoCeleste

Es un cuerpo celeste, por lo tanto hereda de CuerpoCeleste, que gira alrededor de una estrella, en cuyo caso será un planeta, o de un planeta en cuyo caso será un satélite.

Propiedades adicionales

- centroOrbita: CuerpoCeleste, el cuerpo alrededor del cual gira el planeta
- angulo: Double, el ángulo del radio vector con respecto al eje mayor de la órbita
- orbita: Orbita2D, la órbita del planeta

Métodos de factoría

- of(String nombre, Estrella estrella):Planeta



- satelite(String nombre, CuerpoCeleste planeta):Planeta

Cometa extiende CuerpoCeleste

Es un cuerpo celeste, por lo tanto hereda de CuerpoCeleste, que se mueve en el universo siguiendo una recta.

Propiedades adicionales

- direccion: Vector2D, dirección del movimiento
- velocidad: Double, velocidad del movimiento

Los cometas rebotan al llegar a los límites del universo y por lo tanto cambian de dirección.

CometaAcelerado extiende Cometa

Es un cometa que va aumentando su velocidad según la aceleración especificada

Propiedades adicionales

- aceleracion: Double

CometaErratico extiende Cometa

Es un cometa que va cambiando la dirección de manera aleatoria

Universo2D

Es un objeto que contiene los cuerpos celestes

Propiedades

- xMax: Integer, anchura de la ventana
- yMax: Integer, altura de la ventana
- tiempo: Double, el tiempo de la simulación
- tiempoMax: Double, el tiempo máximo
- incrTiempo: Double, incremento de tiempo en cada paso de simulación
- ventana: Canvas, la ventana para mostrar los objetos celestes
- cuerposCelestes: List<CuerpoCeleste>, los cuerpos celestes
- distanciaMinima: Double, la distancia mínima entre dos cuerpos celestes



Operaciones

- simular: Void, va haciendo pasos de tiempo y moviendo cada objeto en la ventana. Termina la simulación cuando hay un choque: dos cuerpos celestes están a una distancia inferior a un umbral.

Se detiene la simulación cuando se detecta la primera colisión.

El código puede verse en el repositorio



Bibliografía

Dos enlaces a material de Java y un libro. El primero el API. El segundo un tutorial de Oracle.

1. API de Java

<https://docs.oracle.com/en/java/javase/18/docs/api/index.html>

2. Tutorial de Java:

<https://docs.oracle.com/javase/tutorial/>

3. Programación orientada a objetos con Java usando Blue J. David J. Barnes and Michael Kölking. Prentice-Hall, 2013. ISBN: 978-8483227916.





Miguel Toro es doctor en Ingeniería Industrial por la Universidad de Sevilla, institución en la que desarrolla su labor docente e investigadora como catedrático del Departamento de Lenguajes y Sistemas Informáticos y en la que ejerce también como director del Instituto de Ingeniería Informática.

Ha ocupado diversos cargos de responsabilidad, entre los que cabe señalar los siguientes: director de la Oficina de Transferencia de Resultados de la Investigación (OTRI), director general de Investigación, Tecnología y Empresa de la Junta de Andalucía, presidente de Sistedes (Sociedad Nacional de Ingeniería del Software y Tecnologías de Desarrollo de Software) y presidente de la Sociedad Científica Informática de España (SCIE), que engloba a los informáticos de las universidades españolas.

Colabora asiduamente con varias agencias nacionales de evaluación universitaria; en este sentido, ha tenido un papel activo en la Agencia Andaluza de Evaluación de la Calidad y Acreditación Universitaria (AGAE) y en el consejo asesor de la Agencia Nacional de Evaluación de la Calidad y Acreditación (ANECA).

En su extensa trayectoria profesional cuenta con varios reconocimientos: Premio Fama de la Universidad de Sevilla, Premio Sistedes de la Sociedad Nacional de Ingeniería del Software y Tecnologías de Desarrollo de Software (en reconocimiento a su labor de promoción y consolidación de la Informática en España) y Premio Nacional de Informática José García Santesmases a la trayectoria profesional, otorgado por la Sociedad Científica Informática de España.



Este volumen está dirigido a los alumnos de primero de los grados en Ingeniería Informática, especialmente a aquellos que cursan la asignatura Fundamentos de Programación en la que se imparte Java, dado que se exponen aquí los conceptos básicos de esta materia. Los contenidos que se abordan son los siguientes:

1. Introducción al lenguaje Java
2. Expresiones y tipos básicos
3. Sentencias de control de flujo y abstracción funcional
4. Programación orientada a objetos. Diseño de tipos de datos.
Tipos de agregados de datos
5. Esquemas secuenciales: estilo imperativo y estilo funcional

