Lenguaje de Programación JAVA

Contenidos

INTRODUCCIÓN

Características

Sun lo describe como:

- "simple, orientado a objetos, distribuido, interpretado, robusto, seguro, de arquitectura neutra, portable, de altas prestaciones, multitarea y dinámico".

Toma su sintaxis del C++.

Recoge las tendencias actuales en desarrollo de software.

Soporta:

- Encapsulado
- Herencia simple
- Polimorfismo
- Sobrecarga de métodos
- Tratamiento de excepciones

Historia

Historia:

- Lenguaje desarrollado por ingenieros de Sun Microsystems en 1991 destinado a electrodomésticos.
- Netscape incorpora en 1995 a su navegador la versión de Java 1.0 (12 packages).

Versiones de JAVA

- Java JDK 1.0 (1996), java 1.1 (1997)
- Java J2SE 1.2 (1999) (Java 2 Standart Edition)
- Java J2SE 1.3 (2000)
- Java J2SE 1.4 (2002)
- Java 1.5 J2SE (2004)
- Java SE 6 (2006) (Se cambia el nombre y se elimina 1.X)
- Java SE 7 (2011)
- Java SE 8 (2014)
- Java SE 9 (2017), Java SE 10 (2017), Java SE 11 (2017),…[semestral+LTS]

Preview Features

Una función de vista previa es una nueva función cuyo diseño, especificación e implementación están completos, pero que no es permanente, lo que significa que la función puede existir en una forma diferente o no existir en las futuras versiones de JDK.

Esto permite que la mayor audiencia de desarrolladores posible pruebe la función en el mundo real y proporcione comentarios.

Para activarlas es nercesario:

- javac --enable-preview --release 12 MyApp.java

Las funciones de vista previa no deben utilizarse en contextos de aplicaciones para explotación.

Licencia de código binario

Binary Code License for Java SE (“BCL”)?
- Puedes usarlo, pero no puedes modificarlo
- Aceptas no demandar a Oracle si algo sale mal
- Puedes redistribuirlo / publicarlo (para que pueda vender productos con Java integrado), pero si lo hace, acepta indemnizar a Oracle; así que si alguien te demanda, no puedes arrastrar a Oracle contigo.

Oracle JDK (licencias de Oracle)
- BCL Public
- BCL Non-comercial
- Comercial License

Oracle OpenJDK (carcateristicas)
- GPL: Sin soporte, Actualizaciones exclusivamente para la versión actual (incluido LTS)

Alternativas a JRE (binarios no disponibles a partir de la versión 11)
- Producciones propias: jlink (OpenJDK)
- Proveedores OpenJDK

OpenJDK Providers

- Oracle JDK - https://www.oracle.com/technetwork/java/javase/downloads
- Oracle OpenJDK - http://jdk.java.net
- AdoptOpenJDK - https://adoptopenjdk.net
- Amazon – Corretto - https://aws.amazon.com/corretto
- Azul Zulu - https://www.azul.com/downloads/zulu/
- BellSoft Liberica - https://bell-sw.com/java.html
- IBM - https://www.ibm.com/developerworks/java/jdk
- jClarity - https://www.jclarity.com/adoptopenjdk-support/
- OpenJDK Upstream - https://adoptopenjdk.net/upstream.html
- ojdkbuild - https://github.com/ojdkbuild/ojdkbuild
- RedHat - https://developers.redhat.com/products/openjdk/overview
- SapMachine - https://sap.github.io/SapMachine

Ediciones

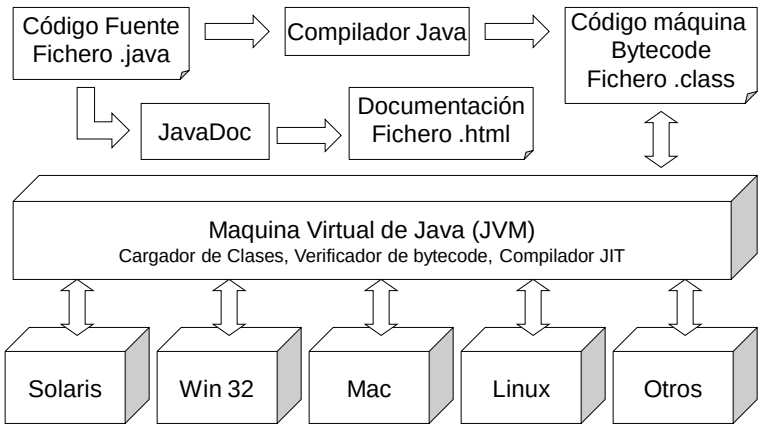
Ediciones:

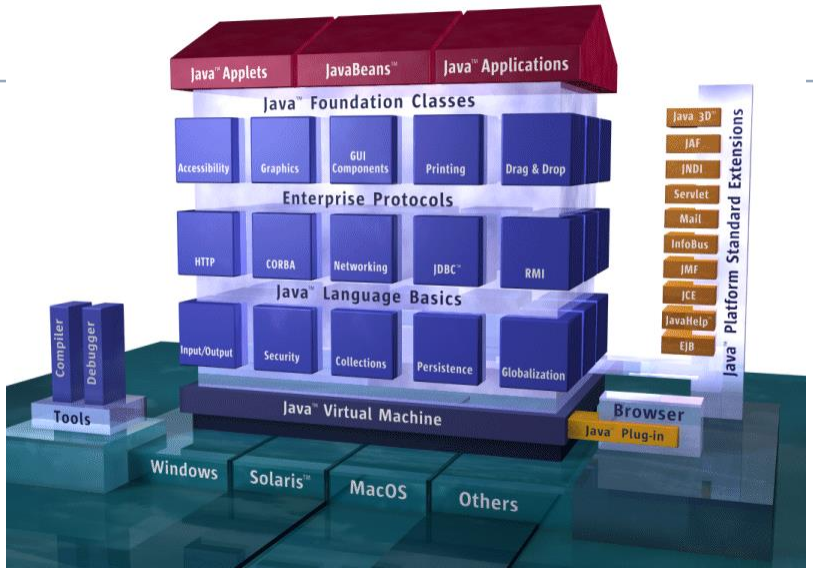
- J2SE: Standard Edition
- J2EE: Enterprise Edition
- J2ME: Micro Edition
- JAVA EMBEDDED
  - Java SE Embedded
  - Java ME Embedded
- JavaFX (rich client platform), Java DB (Apache Derby), Java Card (smart cards), Java TV (Java ME to smart TV).

Referencia:

http://java.sun.com/ → http://www.oracle.com

Arquitectura Java





Desarrollos en Java

El principal desarrollo de Java se produce en Internet pero NO se limita a ello.

Principalmente se pueden desarrollar varios tipos de aplicaciones:

- Aplicaciones Independientes (Standalone)
- Applets: ~~Aplicaciones que se ejecutan en un navegador.~~
- Servlets: Aplicaciones que se ejecutan en un servidor.

También es posible desarrollar componentes software, reutilizables con herramientas de desarrollo. Son los denominados JavaBeans.

Entorno de desarrollo

JRE (Java Runtime Environment)

JDK (Java Development Kit)

> *A partir de JDK 9 también se eliminó la distinción entre Java Development Kit (JDK) y Java Runtime Environment (JRE).*

Comandos:

- java, javac, jar, jdeps, jshell, javadoc, …

IDEs (Integrated Development Environment):

- Eclipse, NetBeans, JBuilder, ...

Variables de entorno:

- CLASSPATH
- PATH

Estructura de un programa

El Java es un lenguaje que permite la creación de nuevos tipos: clases, interfaces y enumeraciones.

Un fichero .java solo puede contener un único tipo (no anidado)

tipo.java → tipo.class

Los ficheros se almacenan en una jerarquía de directorios y subdirectorios. Cada uno de ellos se denomina paquete.

Un directorio, individualmente o con sus subdirectorios, se puede empaquetar en un fichero comprimido .jar.

dir → dir.jar

Conjuntos de paquetes se agrupan en módulos. Cada módulo establece que paquetes se pueden utilizar externamente, así como de que otros módulos depende.

Estructura del código

```
[package <nombre_paquete>;]
[<Importación de paquetes>]

public class <NombreAplicación> [<herencia>]
[<interface>] {
…
    public static void main(String arg[]) {
    …
    }
…
}
```


Conceptos O-O

Objetos: Clases e instancias
Miembros: Campos (atributos) y métodos
Constructores y destructores
Encapsulación, reutilización y herencia
Polimorfismo e interfaces
Sobrecarga, reemplazo y sombreado
Clases abstractas
Miembros de clase e instancia

JSHELL (v9)

Introducción

La herramienta Java Shell (JShell) es una herramienta interactiva para aprender el lenguaje de programación Java y crear prototipos de código Java. JShell es un ciclo de lectura-evaluación-impresión (REPL), que evalúa declaraciones, instrucciones y expresiones a medida que se introducen y muestra inmediatamente los resultados. La herramienta se ejecuta desde la línea de comandos.

El desarrollo de programas Java normalmente implica el siguiente proceso:

- Escribir el programa completo.
- Compilarlo y corregir los errores.
- Ejecutar el programa.
- Averiguar qué tiene de malo.
- Editarlo.
- Repetir el proceso.

JShell le ayuda a probar código y explorar opciones fácilmente a medida que desarrolla su programa. Se pueden probar declaraciones individuales, diferentes variaciones de un método y experimentar con API desconocidas dentro de la sesión de JShell. JShell no reemplaza un IDE.

Fragmentos y comandos

JShell acepta declaraciones de Java, definiciones de variables, métodos y clases, importaciones y expresiones, denominados fragmentos.

Los fragmentos de código Java se introducen en JShell y se evalúan inmediatamente: muestra resultados, comentarios sobre los resultados, las acciones realizadas y los errores que ocurrieron.

```
jshell> String saluda(String nombre) {
   ...>    return "Hola " + nombre;
   ...> }
|  created method saluda(String)

jshell> saluda("Mundo");
$2 ==> "Hola Mundo"
```

Los comandos se distinguen de los fragmentos por una barra inclinada (/) y permite obtener información acerca de las variables actuales (/vars), métodos (/methods), tipos (/types), fragmentos (/list), ficheros (/open /save), …

Para la ayuda (/help /?), reiniciar el entorno (/reset) y salir (/exit)

EL DOCUMENTADOR: JAVADOC

Documentar clases

El Java permite introducir comentarios que documenten las clases y aporta herramientas que convierten dichos comentarios en páginas HTML con la documentación de la clase.

Deben documentarse los siguientes elementos: las clases, los interfaces, los atributos y los métodos.

Se documentan los elementos precediendolos por un comentario de tipo:

```
/**
*       Descripción del elemento
*       @etiquetas
*/
```

Las descripciones de los elementos y de las etiquetas pueden ser formateadas con etiquetas HTML.

Etiquetas de javadoc

@author Nombres de los autores
- Solo clases.

@exception NombreExcepción Explicación
- Solo métodos. Indica cada excepción que genera y se explica por qué se produce y como tratarla.

@param NombreParámetro Descripción
- Solo métodos. Indica cada parámetro y para que se emplea.

@see NombreClase#Miembro
- Aparece bajo la etiqueta “Véase también” (See Also) los enlaces relacionados. El #Miembro es opcional.

@since Descripción
- Solo clases. Indica a partir de que versión del Java es operativa.

@return Descripción
- Solo métodos. Describe el tipo y valor de retorno del método.

@version Número
- Solo clases. Número de la versión de la clase.

LENGUAJE

Sintaxis

Sensible a mayúsculas y minúsculas.

Sentencias separadas por punto y coma (;).

Formato libre en las sentencias, una sentencia puede utilizar varias líneas y una línea puede incluir varias sentencias.

Los espacios en blanco se compactan a uno solo.

Bloques marcados por llaves: { <sentencias> }

Comentarios:

- // hasta el final de la línea
- /* … */ una o varias líneas
- /** … */ una o varias líneas de documentación

59 palabras reservadas (todas en minúsculas).

Separadores

( )	para contener listas de parámetros en los métodos, expresiones de control de flujo, tipos de conversión y establecer precedencias en la expresiones.
{ }	para definir bloques de código y agrupar elementos de tablas inicializadas automáticamente.
[ ]	para declarar y referencias tablas.
;	separar instrucciones.
,	separar identificadores en su declaración y concatenar sentencias dentro del for.
.	Separar paquetes y clases, y clases/instancias de atributos/métodos.

Identificadores

Compuesto de letras, números, _, $.

Juego de caracteres UNICODE.

NO pueden comenzar con números.

NO pueden ser palabras reservadas ni el nombre de un valor booleano (true / false).

NO debe tener el mismo nombre que otro elemento del mismo ámbito.

Sin limite en el número de caracteres.

El identificador _ no esta permitido. (v9)

Notación recomendada

Utilizar nombres descriptivos: sustantivos para variables y objetos, y verbos para los métodos.

Uso de mayúsculas y minúsculas:

- Métodos, variables e instancia (Camel):
  - nombre, nombreCompuesto
- Clases e Interfaces (Pascal):
  - Nombre, NombreCompuesto
- Constantes:
  - NOMBRE, NOMBRE_COMPUESTO

Uso de mnemotécnicos:

- btnBoton, txtTexto, frmFormulario, ...

Valores constantes y Literales

Enteros
- Base 10 dddd...
- Base 8 0ddd
- Base 16 0xddd
- Binario 0b1001 (v7)

Punto Flotante
- estándar 3.14159
- científica 6.022E23

(v7)
- Para facilitar la lectura de grandes numero, el _ como separador de dígitos (siempre entre dos dígitos)

Booleanos
- true / false

Caracteres
- Unicode - 16 bits
- 'a', '\n', '\\' ...
- octal: \ddd
- hexadecimal: \udddd

Cadenas
- entrecomillas: "...."
- Bloques de texto: """
  .... """ (v15)

Sin referencia:
- null

Variables

Lenguaje fuertemente tipado.

Dos tipos según contenido:

- Primitivos: Contienen un único valor de tipo base.
- Referencias: Contienen las referencias a objetos o tablas.

Dos tipos según su ámbito:

- Miembros: Atributos de una clase.
- Locales: Definidos dentro de un bloque.

Tipos Primitivos

- Enteros:
  - byte 8 bits -128 .. 127
  - short 16 bits -32.768 .. 32.765
  - int 32 bits -2.147.483.648 .. 2.147.483.647
  - long 64 bits -10^{19} .. 10^{19} (sufijo L para las constantes)
- Reales:
  - float 32 bits entre 6 y 7 decimales equivalentes
  - double 64 bits unos 15 decimales equivalentes
- boolean 1 byte true o false
  - tipo devuelto en operaciones de comparación
  - requerido en operaciones de control
- char 2 byte 1 carácter UNICODE
  - pueden operarse como enteros
- void
  - indica ausencia de tipo

Declaración de variables

Se declaran como:

[final] <tipo> <nombre> [= <valor inicial>];

Pueden declararse varias a la vez del mismo tipo separando los nombres por comas.

Una variable final solo puede asignarse una vez, habitualmente en la inicialización.

A una variable local se le debe dar explícitamente un valor antes de ser utilizada, ya sea por inicialización o asignación.

Se pueden declarar en cualquier punto, pero siempre antes de utilizarlas. Se recomienda definirlas al principio del bloque.

Las definidas en bloques interiores solapan a las de bloques superiores con el mismo nombre.

Alcance:

- desde donde se definen hasta el final del bloque.

Referencias

Permiten direccionar instancias de los objetos.

Al declarar una variable de tipo referencia, NO se crea una instancia del objeto.

Contiene null hasta que se le asigna un objeto.

Se declaran:

- <Nombre Clase> <nombre> [= <valor inicial>];
- <Nombre Clase> <nombre> [= new <Nombre Clase> (<lista de parámetros>)];

Inferencia de tipos (v.10)

La inferencia de tipos permite al compilador obtener un tipo estático basándose un valor inicial sin que sea necesario expresarlo de forma explícita.

La inferencia de tipos para variables locales se declaran como:

var <nombre> = <valor inicial>;

Cuando se infiere de un tipo de retorno hace innecesaria la importación explicita del tipos.

El uso de var no está permitido en retornos, parámetros, propiedades, variables sin inicializar o inicializadas a null.

Se permite en los parámetros de una expresión lambda asignada a un interfaz funcional. (v.11)

Casting

El Java es un lenguaje fuertemente tipado, por lo que hace comprobación de tipos antes de realizar cualquier operación.

La conversión de un tipo de menor precisión a uno de mayor es automática.

Se denomina casting a la operación que fuerza la conversión de un tipo a otro.

Se realiza:

- (<nuevo tipo>) <expresión>

La conversión de tipos de mayor precisión a uno de menor implica la pérdida de las posiciones altas.

Tablas (I)

La tablas son un tipo especial de objetos en Java.

Los elementos pueden ser de cualquier tipo, pero todos los elementos de una tabla deben ser del mismo tipo.

Se declaran (se crea una referencia):

- <Tipo> <nombreTabla> [] [= <Inicialización>];
- <Tipo> [] <nombreTabla> [= <Inicialización>];

Se dimensionan:

- <nombreTabla> = new <Tipo> [<Dimensión>];
- <nombreTabla> = {<lista de elementos>};
  - La dimensión puede ser una expresión.
  - Los elementos de la lista separados por comas. El número de los mismos fijan la dimensión.

Tablas (II)

Los elementos se inicializan al valor por defecto de su tipo.

Se referencian:

- <nombreTabla> [<Indice>]
- Rango de los índices: 0 .. <Dimensión> - 1
- No se puede acceder a elementos fuera del rango.

La dimensión de la tabla se obtiene dinámicamente con la propiedad:

- <nombreTabla>.length

La asignación de una tabla a otra implica un cambio de referencia no una copia.

Tablas Multidimensionales

Tablas bidemensionales:

- Se pueden crear directamente:
  - <Tipo> <nombreTabla> [] [] = new <Tipo> [<Dimensión 1>] [<Dimensión 2>];
- o por dimensión:
  - <Tipo> <nombreTabla> [] [] = new <Tipo> [<Dimensión 1>] [];
  - <nombreTabla> [<Indice>] = new <Tipo> [<Dimensión 2>]
  - Las segunda dimensión puede no coincidir en los diferentes elementos.

Una tabla bidemensional se considera como una tabla de referencias a otras tablas, por lo que se pueden implementar tablas multidimensionales.

Se recomienda no superar las cuatro dimensiones por la dificultad de su manejo conceptual.

Operadores (I)

Asignación

- Simple: <Destino> = <Expresión>

Aritméticos

- Suma (+)
  - si el primer operando es una cadena se concatena.
- Resta (-)
- Producto (*)
- División (/)
  - La división de enteros es un entero.
- Resto (%)
- Acumulativa:
  - += , -= , *= , /= , %=

Operadores relacionales

- > : mayor
- >= : mayor o igual
- < : menor
- <= : menor o igual
- == : igual
- != : distinto

Operadores lógicos (los operandos son booleanos)

- && : AND Optima
- || : OR Optima
- ! : NOT
- & : AND Completo
- | : OR Completo

Operadores (II)

Binarios
- >> : Desplazamiento Derecha
- << : Desplazamiento Izquierda
- >>> : Desplazamiento Derecha con signo
- & : AND binario
- | : OR binario
- ^ : XOR binario
- ~ : complemento
- Acumulativos:
  - >>=
  - <<=
  - >>>=
  - &=
  - |=
  - ^=
  - ~=

Incremento/decremento
- ++<Variable>
  - Se incrementa en 1
  - Se consulta.
- --< Variable >
  - Se decrementa en 1
  - Se consulta.
- < Variable >++
  - Se consulta
  - Se incrementa en 1.
- < Variable >--
  - Se consulta
  - Se decrementa en 1.

Operadores (III)

Operador condicional:
- <Condición> ? <Expresión true> : <Expresión false>

Operador instanceof:
- informa si el operando pertenece a una determinada clase.
- Coincidencia de patrones para if y case (v15):
  - referencia instanceof Clase nueva_referencia

Operador new:
- crea una instancia de la clase
  - … new <NombreClase>([<Lista de Parámetros>])

Prioridad de Operadores

1.	postfijos	[] . (params) expr++ expr--
2.	operadores unarios	++expr --expr +expr -expr ~ !
3.	creación o cast	new (type)expr
4.	multiplicativo	* / %
5.	aditivo	+ -
6.	desplazamiento	<< >> >>>
7.	relacional	< > <= >= instanceof
8.	igualdad	== !=
9.	bitwise	&
10.	bitwise	^
11.	bitwise	|
12.	lógico	&&
13.	lógico	||
14.	condicional	? :
15.	asignación	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Bifurcación simple

if(<Condición>)

<Instrucción>; o { <Bloque de instrucciones> }

[

else

<Instrucción>; o { <Bloque de instrucciones> }

]

- Sin limite en los anidamientos.
- La parte else es opcional. Para evitar confusión es correcto crear bloques cuando existan anidamientos.

Bifurcación Múltiple

```
switch(<Expresión>) {
case <Valor 1> :
                    <Bloque de instrucciones>
                    [break;]
case <Valor 2> :
                    <Bloque de instrucciones>
                    [break;]
        …
[default:
                    <Bloque de instrucciones>]
}
```

<Expresión> debe ser char, byte, short, int, Character, Byte, Short o Integer (a partir de la v5 permite enumerados y de la v7 permite String).

Cada case solo puede expresar una única constante (a partir de la v12 permite una lista separada por comas).

Expresión switch (v.12)

Las expresiones switch se evalúan con un solo valor, siempre deben devolver valor y se pueden usar en declaraciones. El valor de retorno se devuelve con la instrucción yield.

```
var rstl = switch(<Expresión>) {
case <Valor 1> [, … ] :
                    <Bloque de instrucciones>
                    yield <Valor de la expresión>;
case <Valor 2> :
                    <Bloque de instrucciones>
                    yield <Valor de la expresión>;
        …
[default:
                    <Bloque de instrucciones>]
                    yield <Valor por defec>;
}
```

Si el calculo del valor de retorno no necesita un bloque porque es el resultado de una expresión, se puede contraer a:

```
case MONDAY, FRIDAY, SUNDAY -> 6; // case MONDAY, FRIDAY, SUNDAY : yield 6;
```

Coincidencia de patrones (v17)

La coincidencia de patrones implica probar si un objeto tiene una estructura particular y luego extraer a variables datos de ese objeto si hay una coincidencia.

```
if (shape instanceof Rectangle r) {
```

La coincidencia de patrones permite los objetos en las expresiones de evaluación de los switch:

```
switch(obj) {
    case null -> // equivale a case Object obj ->
    case String s ->
        <Bloque de instrucciones>
```

Un patrón protegido permite refinar un patrón con una expresión booleana. Un valor coincide con un patrón protegido si coincide con el patrón y la expresión booleana se evalúa como verdadera.

```
case String s && (s.length() == 1):
```


Bucles

Bucle Mientras

while(<Condición>)

; o <Instrucción>; o {<Bloque de instrucciones>}

- Se ejecuta de 0 a n veces, mientras se cumpla la condición.

Bucle Repetir

```
do {
<Bloque de instrucciones>
} while(<Condición>);
```

- Se ejecuta de 1 a n veces, mientras se cumpla la condición.

Bucle Para

for(<Inicio>; <Final>; <Iteración>)
; o <Instrucción>; o {<Bloque de instrucciones>}

Apartados:

- <Inicio> : Se ejecutan antes de comenzar.
  - Lista de expresiones separadas por comas.
- <Final> : Expresión condicional de finalización, se ejecuta mientras se cumpla la condición.
- <Iteración> : Se ejecutan en cada iteración.
  - Lista de expresiones separadas por comas.

Bucle Para en iteradores

for(<Tipo> varDest : <Colección>)
<Instrucción>; o {<Bloque de instrucciones>}

Equivale a:

```
for (Iterator i = <Colección>.iterator(); i.hasNext(); ) {
   <Tipo> varDest = (<Tipo>) i.next();
     …
}
```

El tipo de <Colección> debe ser un subtipo de Iterable o un tipo de matriz

Control de Flujo

<NombreEtiqueta> :
- Define un punto de reentrada.

break [<NombreEtiqueta>];
- Sale de la instrucción actual.
- En caso de aparecer la etiqueta, salta a ella.

continue [<NombreEtiqueta>];
- Fuerza la evaluación de la condición.
- En caso de aparecer la etiqueta, salta a ella.

return [<Expresión>];
- Termina un método y, opcionalmente, devuelve un valor.

yield <Expresión>; (v.12)
- Termina una expresión switch y devuelve un valor.

Condicional de Error

```
try {
        <Bloque de instrucciones vigiladas>
}
catch(<TipoExcepción> <Referencia>) {
        <Instrucciones de tratamiento de la excepción>
}
… [<Otros tratamientos de excepción>]
finally {
        <Sentencias que se ejecutan siempre>
}
```

- Los catch y el finally son opcionales pero debe aparecer al menos un tratamiento de error aunque no haga nada.

throw <Excepción>

synchronized

synchronized(<Expresión>)
  <Instrucción>; o {<Bloque de instrucciones>}

Una declaración synchronized adquiere un bloqueo de exclusión mutua en nombre del subproceso en ejecución, ejecuta un bloqueo y libera el bloqueo al terminar el bloque.

No prosigue hasta que la acción de bloqueo se haya completado con éxito (espera). Ningún otro subproceso puede adquirir el bloqueo hasta que sea liberado, debe esperar.

Solo un hilo a la vez puede mantener un bloqueo en un monitor. Cualquier otro hilo que intente bloquear ese monitor se bloquea hasta que pueda obtener un bloqueo en ese monitor.

La sincronización es el mecanismo básico para la seguridad de subprocesos.

TIPOS

Enumerados (v5)

Tipo que restringe los valores permitidos a un conjunto de constantes predefinidas con nombres. Por convención, los nombres de valores deber ir en mayúsculas.
Los enumerados son un tipo especial de clase en Java, extienden implícitamente a java.lang.Enum, y pueden declarar constructores, métodos y atributos.
Los nombres de valores pueden tener asociados valores primitivos.

```
public enum Genero { DESCONOCIDO, MASCULINO, FEMENINO, NEUTRO }
public enum Day {
   SUNDAY(1), MONDAY(2), TUESDAY(3), WEDNESDAY(4),
   THURSDAY(5), FRIDAY(6), SATURDAY(7);
   private int numericValue;
   Day(int value) { numericValue = value; }
   public int getValue() { return numericValue; }
   public static Day getEnum(int value) {
            // switch value to enum
   }
}
```


Definición de Clases

```
[<modificador>] class <Nombre> [<herencia>] [<interface>] {
            <Cuerpo de la clase>
}
```

Modificadores:

- abstract : clase abstracta no instanciable.
- public: clase publica instanciable e importable, debe coincidir con el nombre del fichero (el fichero solo puede tener una clase publica a primer nivel).
- final: clase sellada que no admite herencia.
- strictfp: tratamiento estricto de coma flotante para double y float (compatibilidad heredada con IEEE-754 para la precisión)
- Sin modificador es una clase interna del paquete.

Para clases internas:

- static: clase interna no miembro.
- protected: clase miembro accesible por sus herederos.
- private: clase miembro no accesible fuera de la clase.

Miembros

Atributos

- Se declaran como las variables
- Si no se asigna un valor inicial automáticamente se inicializan a 0 (cero) las numéricas, false las lógicas, '' o '\0' los caracteres y a null las referencias.

[<acceso>] [<modificador>] <tipo> <nombre> [= <valor inicial>];

Métodos

[<acceso>] [<modificador>] <tipo> <nombre> ([<ListaDeParámetros>]) [<excepciones>] {

    <Cuerpo del Método>

}

Se invocan (mensaje):

- <nombreInstancia>.<nombre>
- <nombreInstancia>.<nombre> ([<Lista de Argumentos >])

Paso de argumentos

Las lista de parámetros se define:

- <nombreMetodo>(<Tipo> <Nombre>[, <Tipo> <Nombre>, ...])
- No es obligatorio definir parámetros, en cuyo caso aparecerán los paréntesis vacíos.

La argumentos se separan por comas, y deben ser compatibles con los tipos definidos por los parámetros. Si el método no recibe parámetros aparecerán los paréntesis vacíos.

Todos los argumentos se pasan por valor:

- Los tipos primitivos no permiten modificar su valor.
- Las referencias no pueden cambiar la instancia que referencian.
- Se pueden utilizar métodos que cambien el contenido de las instancias.

Varargs (v5)

Lista de argumentos variables:

- Solo un parámetro.
- Deben ser el último de la lista.
- Puede no recibir ningún argumento.

```
void Tradicional(int fijo1, int fijo2, int[] lista) {
    //…
}
void Nueva(int fijo1, int fijo2, int… lista) {
    //…
    for(int i : lista)
    //…
}
Tadicional(1, 2, new int[] { 3, 4, 5 });
Nueva(1, 2, 3, 4, 5);
```


Sobrecarga

Se denomina sobrecarga a la existencia de varias versiones del mismo método dentro de la misma clase.

Las versiones del método se diferencian por el número y/o tipo de parámetros. El valor de retorno no influye.

El sistema determina qué método se está invocando por la lista de argumentos y sus tipos:

1. Coinciden exactamente
2. Promoviendo los tipos actuales a tipos superiores

En muchos casos es necesario forzar la conversión de tipos mediante un casting.

Control de acceso

private
- solo accesible desde la clase donde está definido.

protected
- accesible desde la clase donde está definido, **el resto de las clases del paquete** y por sus herederos en otros paquetes.

public
- accesible por todos.

sin definir
- accesible desde el resto del paquete.

Modificador: static

Indica que el atributo o el método es de clase.

Una única copia del atributo para todas las instancias de la clase. Se crean cuando se carga la clase.
- <NombreClase>.<atributo>
  - (desde fuera de la clase)

Los métodos son comunes para la clase, solo pueden acceder a otros miembros de clase. Se invocan:
- <NombreClase>.<nombreMetodo> (<Param>)

Modificador: final

Indica que el atributo es una constante y no se puede modificar.

- [<acceso>] final <tipo> <NOMBRE> = <valor>;
- El valor puede ser el resultado de evaluar una expresión.
- Se pueden inicializar en el constructor.
- Pueden ser de clase
- Se invocan:
  - <NombreClase>.<CONSTANTE>
  - <NombreInstancia>.<CONSTANTE>

Los métodos finales no pueden ser redefinidos por las subclases.

Modificador: transient

[<acceso>] transient <tipo> <NOMBRE> [= <valor>];

Se aplica solamente a atributos.

Especifica que un atributo es transitorio y no interviene en los procesos de serialización.

Por defecto todos los atributos son persistentes.

Modificador: volatile

[<acceso>] volatile <tipo> <NOMBRE> [= <valor>];

Se aplica solamente a atributos.

Especifica que un atributo es modificado de forma asíncrona (normalmente por procesos concurrentes).

Obliga a la máquina a consultar su valor inmediatamente antes de ser usado.

Modificador: synchronized

<modificador> synchronized <tipo> <nombre> ([<Lista de Parámetros>]) {
<Cuerpo del Método>
}

Se aplica solamente a métodos y en programación multihilo.
Impide que más de un hilo pueda ejecutar el método a la vez.

Evita los problemas de sobreescritura de atributos.

Modificador: native

<modificador> native <tipo> <nombre> ([<Lista de Parámetros>]) ;

Se aplica solamente a métodos.

Indica que la implemetenación del método se ha realizado en código nativo y de forma externa.

Los métodos nativos necesitan ser cargados en el iniacializador de la clase. Para cargar un método se utiliza:

- System.load("…");
- System.loadLibrary("…");

Solo se utilizarán métodos nativos cuando sea estrictamente necesario y la seguridad lo permita.

Herencia

<Modificador> class <Nombre> extends <SuperClase> {

    <Cuerpo de la clase>

}

La clase (subclase) hereda todos los atributos y métodos de su superclase y, opcionalmente, redefinirlos.

No acepta herencia múltiple.

No se puede heredar de una clase final.

En caso de no definir herencia explícita, se asume que la clase hereda de la metaclase Object.

Clases Abstractas

<Modificador> abstract class <Nombre> [<herencia>] [<interface>] {
    <Cuerpo de la clase>
}

Se definen para sintetizar conocimiento.

Se utilizan en la herencia.

No pueden ser instanciadas.

Pueden definir métodos abstractos pero no son obligatorios.

- <modificador> abstract <tipo> <nombre> ([<Lista de Parámetros>]) ;
- No implementan el método.
- Solo pueden aparecer en clases abstractas.
- Obliga a sus subclases (no abstractas) a implementar el método.
- Permiten definir el interfaz de sus herederos.

Sobreescritura de Métodos

Una subclase puede redefinir los métodos heredados.

El tipo de retorno, el nombre y la lista de parámetros debe ser iguales a los del método heredado. En caso contrario se denomina sobrecarga, y debe seguir las limitaciones de la misma.

No puede lanzar excepciones no definidas para el método de la superclase, aunque puede restringir dicha lista.

Los modificadores no pueden restringir el acceso definido en la superclase, pero si pueden ampliarlo.

No se pueden sobrrescribir métodos finales.

Los métodos sobreescritos solapan a los métodos originales. Para acceder a los métodos originales es necesario hacer referencia a la superclase.

Referencias especiales

this

- Referencia a la instancia con la que se está trabajando.
- Solo es necesaria cuando en la implementación de los métodos de la clase es necesario hacer una referencia a la instancia del propio objeto.
- Pueden usarse como pseudo nombre de los constructores de la clase (solo dentro del constructor)

super

- Cualificador de ámbito a la superclase.
- Se emplea para invocar a los métodos originales redefinidos por la subclases y los atributos suplantados en las subclases.
- Pueden usarse como pseudo nombre de los constructores de la superclase (solo dentro del constructor)

Constructores

Dirigen la creación e inicialización de las instancias de la clase.

Son métodos que se denominan de la misma forma que la clase y no devuelven ningún tipo.

Aceptan sobrecarga, pueden existir varias versiones con diferentes parámetros.

El compilador proporciona automáticamente un constructor predeterminado sin argumentos para cualquier clase sin constructores. Este constructor predeterminado se pierde al definir el primer constructor explicito y, si se desea, es necesario definir explícitamente el constructor sin parámetros.

Un constructor puede invocar otro constructor de la misma clase o de su superclase: obligatoriamente como primera instrucción.

Los constructores no se heredan como tales.

Si se hereda de una clase, se invocara implícitamente a su constructor sin parámetros si no se invoca explícitamente, pero si no dispone del constructor sin parámetros es obligatorio tener constructores que empiecen invocando uno de los constructores de la superclase.

Aceptan modificadores de acceso. Si se define como private, solo podrá ser invocado por otro constructor o desde un método de clase.

Se invocan con el operador new en el proceso de instanciación.

Inicializadores

Dirigen la creación e inicialización de la clase.

Inicializan los atributos de clase, cargan los métodos nativos, permiten gestionar excepciones.

Se invocan una sola vez para toda la clase.

Se definen como métodos de clase sin nombre y sin tipo de retorno:

```
static {
        <Cuerpo del inicializador>
}
```


Finalizadores

Dirigen la destrucción de las instancias.

Son invocados automáticamente por el recolector de la basura cuando va a destruir la instancia.

Cierra, detiene y libera los recurso que utiliza.

No es necesario que libere memoria, salvo la reservada por métodos nativos.

Es un método de objeto, sin tipo de retorno, sin argumentos y se llama siempre finalize:

```
finalize( ) {
        <Cuerpo del finalizador>
        [super.finalize()]
}
```

Vida de un objeto

Al crear la primera instancia de la clase o al utilizar el primer método o variable static se localiza la clase y se carga en memoria.

Se ejecutan los inicializadores static (sólo una vez).

Cada vez que se quiere crear una nueva instancia:

- se comienza reservando la memoria necesaria
- se da valor por defecto a las variables miembro de los tipos primitivos
- se ejecutan los constructores

Destrucción de un objeto

- El sistema destruirá todas las instancias que no tengan referencias.
- Para quitar una referencia a una instancia se asigna a la referencia:
  - otra instancia
  - el valor null
- El objeto no se destruye inmediatamente, periódicamente el recolector de basura revisara la memoria para eliminar dichos objetos.

Administrador de Memoria

El sistema de forma automática ejecuta el recolector de basura (garbage collector) que elimina los objetos que no tienen referencias.

Se puede sugerir que se ejecute:

- System.gc( );

o que ejecute todos los finalizadores pendientes:

- System.runFinalization( );

Clases Internas

El lenguaje de programación Java permite definir una clase dentro de otra clase o de un método. Esta clase se llama clase anidada o interna.

Las clases anidadas permiten:

- Agrupar lógicamente las clases que solo se usan en un lugar: si una clase es útil solo para otra clase, entonces es lógico incrustarla en esa clase y mantener las dos juntas. Anidar tales "clases auxiliares" hace que su paquete sea más ágil.
- Aumentar la encapsulación: las clases anidadas pueden acceder a los atributos y métodos de su clase contenedora o variables locales del método contenedor.
- Producir un código más legible y fácil de mantener: Anidar clases pequeñas dentro de clases de nivel superior coloca el código más cerca de donde se usa y evita la proliferación de múltiples ficheros.

Clases Internas

Clases Internas static

- Se declaran dentro de otra clase o interface a máximo nivel.
- Pueden utilizar los miembros de clases de la clase contenedora.
- Son accesibles como clases del paquete: contenedor.clase.

Clases Internas miembro

- Se declaran dentro de otra clase a máximo nivel.
- Pueden utilizar los miembros de la clase contenedora.
- No pueden definir ningún miembro estático por sí mismas.
- Solo son accesibles a través de la propia clase contenedora.

Clases Internas locales

- Se declaran dentro de un bloque.
- No pueden definir ningún miembro estático por sí mismas.
- Solo son accesibles el propio bloque donde están definidas.

Clases anónimas

Clases anónimas

Una clase anónima es una expresión que define e instancia una clase: un solo uso.

Su sintaxis es como la invocación de un constructor:

- El operador new
- El nombre de una interfaz para implementar o una clase para extender.
- Paréntesis que contienen los argumentos del constructor de la clase que extiende o vacíos si implementa una interfaz o no se requieren argumentos.
- El cuerpo de la clase como en el resto de las clases salvo que no pueden tener miembros de clase (salvo constantes) ni constructores.

Interfaces

```
[public] interface <Nombre> [extends <ListaDeSuperInterfaces>] {
        [<Declaración de constantes>]
        [<Tipos anidados>]
        <Métodos del Interfaz>
        <Métodos default> (v8)
        <Métodos static> (v8)
        <Métodos private> (v9)
        <Métodos private static> (v9)
}
```

Aceptan herencia múltiple, la lista debe ir separada por comas.

Por defecto el acceso al interface es de paquete, con el modificador public permite acceso a todos, en cuyo caso el nombre del interface debe coincidir con el nombre del fichero.

Se pueden crear referencias del tipo del interfaz pero no instancias.

Los métodos del interface son siempre public y abstract de modo implícito.

Los métodos NO se implementan en el cuerpo del interface, solo se declaran.

Implementación de Interfaces

[<modificador>] class <Nombre> implements <ListaDeInterfaces> {
    <Cuerpo de la clase>
}

Una clase puede implementar varios interfaces, en cuyo caso los nombres irán separados por comas.

Las subclase heredan los interfaces de las superclases.

Las clases están obligadas a implementar todos los métodos de sus interfaces, con excepción de las clases abstractas que podrán implemertarlo parcialmente, siendo responsabilidad de sus subclases completarlos.

Métodos por defecto y estáticos en interfaces (v8)

Ahora se puede añadir una implementación por defecto a los métodos de los interfaces:

```
public interface Persona {
   String getNombre();
   String getApellidos();
   default String toStringDatos() { return getNombre() + " " + getApellidos(); }
}
```

Definiendo métodos estáticos en las interfaces se evitar tener que crear clases de utilidad.

```
public interface Calc {
   int add(int a, int b);
   static int multiply(int a, int b) { return a * b;  }
}
```

Se pueden crear métodos privados para refactorizar el código. (v9)

Tipos Genéricos o parametrizados

Java Generics es un mecanismo introducido en Java 1.5 para "especializar" las clases "genéricas", similares a las plantillas en lenguaje C++

Una clase es genérica declara una o más variables de tipos que se conocen como parámetros de tipo de la clase. Gracias a los genéricos podemos postergar la resolución de tipos variables al momento de la instanciación.

Los genéricos se compilan a tipos internos específicos, en tiempo de ejecución la información de tipo especificada en la creación de instancias no está disponible (borrado de tipos).

Los parámetros de tipo pueden sustituir a los tipos reales en la declaración de atributos, parámetros, tipos de retorno y variables. No se pueden emplear en los miembros de clase.

Los parámetro de tipo se declaran con paréntesis angulares a continuación del nombre de la clase.

Tipos Genéricos o parametrizados

Por convención, los nombres de los parámetros de tipo son letras mayúsculas únicas:

- T – Type
- E – Elemento (Usado extensivamente en las colecciones en java)
- K – Key
- V – Value
- N – Number
- S, U, V etc. – 2nd, 3rd, 4th types

Los parámetros de tipo pueden ser resueltos a cualquier tipo referencia, no a tipos valor (pero si sus tipos envolventes). Se puede limitar la resolución de los parámetros de tipo a tipos que hereden o implemente determinadas clases o interfaces, por lo que se podrá acceder a sus miembros:

- <T extends MySuper>
- <T extends MyInterface>
- <T extends MySuper & MyInterface & …>

Tipos Genéricos o parametrizados

```
Modificador class Nombre<T, …> … {
 // T stands for "Type"
   private T t;

   public void procedure(T t) {
      …
   }
   public T fuction() {
     …
     return t;
   }
          …
}
```


Tipos Genéricos o parametrizados

Los parámetros de tipo se resuelven por los tipos reales cuando se declaran atributos, parámetros, tipos de retorno y variables.

```
MiGenerico<String> var1 = new MiGenerico("uno");
List<MiGenerico<Integer>> lista;
```

Los argumentos de los parámetros de tipo pueden ser comodines que son útiles en situaciones en las que solo se requiere un conocimiento parcial del parámetro de tipo.

El comodín se representa por el símbolo ? y pueden tener límites explícitos:

- <? extends MySuper> debe ser del tipo o uno de sus herederos
- <? super MyClass> debe ser del tipo o uno de sus antecesores

```
List<? extends Entity> lista;
```

Métodos Genéricos

public <U> void method(U u) { ... }
- El tipo U se infiere del parámetro pasado.

Si no puede inferir el tipo del parámetro hay que pasarlo especificamente:

```
public <U> U method(Class tipo) { ... }
```

Los contructores tambien aceptan genéricos:

```
public <T> MyClass(T t) {
  // ...
 }
```

Se puede limitar con:

```
public <U extends MySuper> void method(U u) { ... }
```


Borrado de tipos

Para implementar genéricos, el compilador de Java aplica el borrado de tipos:
- Borra todos los parámetros de tipo y reemplaza cada uno con su primer límite si el parámetro de tipo está delimitado o por Object si el parámetro de tipo es ilimitado.
- También borra los parámetros de tipo en los argumentos de métodos genéricos.
- Puede crea un método sintético, llamado método puente, como parte del proceso de borrado de tipos para solucionar problemas de sobrecarga y preservar el polimorfismo en tipos genéricos extendidos.
- Inserta type casts si es necesario para preservar la seguridad del tipo.

El borrado de tipos asegura que no se creen nuevas clases para tipos parametrizados; en consecuencia, los genéricos no incurren en gastos generales de ejecución. El bytes code producido, por lo tanto, solo contiene clases, interfaces y métodos ordinarios.

Restricciones sobre genéricos

- No se pueden crear instancias de tipos genéricos con tipos primitivos.
- No se pueden crear matrices de tipos parametrizados.
- No se pueden consultar el tipo de una referencia con instanceof con tipos parametrizados concretos, solo con comodines.
- No se puede utilizar el operador new en parámetros de tipo para crear una instancia de su tipo.
- No se pueden utilizar las parámetros de tipo en los miembros de clase.
- Los tipos parametrizados no pueden ser Throwable, o herederos, por lo que no se pueden usar para crear, atrapar o lanzar excepciones.
- No permiten resolver la sobrecarga de métodos.
- Presentan múltiples limitaciones con la reflexión.

Inferencia de tipos genéricos (v7)

Inferencia de tipos para la creación de instancia genérico:

- Se puede sustituir los argumentos de tipo necesarios para invocar el constructor de una clase genérica con los paréntesis angulados de tipo vacíos (<>), informalmente llamados operador diamante, porque el compilador puede inferir los argumentos de tipo del contexto.

La expresión:

```
Map<String, List<String>> myMap = new HashMap<String,
List<String>>();
```

Equivale a:

```
Map<String, List<String>> myMap = new HashMap<>();
```

Interfaces funcionales (v8)

- Cada interfaz funcional tiene un único método abstracto (SAM), llamado método funcional, al cual se hace coincidir o adaptar el los parámetros de la expresión lambda y el tipo de retorno.
- Puede contener métodos adicionales si son default, static o private (v9).
- Para definir una interfaz funcional se puede usar la anotación @FunctionalInterface y pueden representarse con una expresión lambda.
- Algunos ejemplos de interfaces funcionales son Runnable, ActionListener, Comparator y Callable.

```
@FunctionalInterface
public interface ICalculadoraLambda {
   public int operacion (int x,int y);
}
```


Expresiones lambda (v8)

Importadas de la programación dinámica del paradigma funcional.
Facilitan la implementación del patrón Delegate. Es el compilador el encargado de la creación de las clases anónimas.

```
public interface ICompararLambda {
      public boolean compara(String a, String b);
}
String extrae(String[] lista, ICompararLambda operar) { … }

String[] lst = {"hola", "adios"};
String s = extrae(lst, new ICompararLambda() {
         @Override
         public boolean compara(String a, String b) {
            return a.compareTo(b) > 0;
         }
      });
s = extrae(lst, (a, b) -> a.compareTo(b) > 0);
```

Expresiones lambda (v8)

Sintaxis:

p -> exp

() -> exp

p -> { …; return rslt; }

(p1, p2) -> { … }

- Los nombres de los parámetros son los que se utilizaran en la expresión o implementación del método.
- Si solo tiene un parámetro los paréntesis son opcionales.
- Si no tiene parámetros los paréntesis vacíos son obligatorios.
- El cuerpo puede ser una expresión, en cuyo caso la {} son opcionales y, en caso de ser una función, tiene un return implícito del valor.
- Si el cuerpo esta compuesto de instrucciones deben ir entre {} como bloque, terminadas en ; y, en caso de ser una función, tiene que hacerse un return explicito del resultado.

Referencias de método (v8)

Se pueden utilizar métodos existentes como expresiones lambda.

Requieren el selector :: para evitar la invocación directa del método:

- Referencia a un método estático
  ContainingClass::staticMethodName
- Referencia a un constructor
  ClassName::new
- Referencia a un método de instancia de un objeto determinado
  containingObject::instanceMethodName
- Referencia a un método de instancia de un objeto arbitrario de un tipo particular (la instancia de la clase se pasa como primer argumento cuando se llama al método).
  ContainingType::methodName

java.util.function (v8)

Consumer<T>	Representa una operación que acepta un único argumento de entrada y no devuelve ningún resultado.
Supplier<T>	Representa un proveedor de resultados (sin argumentos).
Function<T,R>	Representa una función que acepta un argumento y produce un resultado.
Predicate<T>	Representa un predicado (función con valor booleano) de un argumento.
UnaryOperator<T>	Representa una operación en un solo operando que produce un resultado del mismo tipo que su operando.
BinaryOperator<T>	Representa una operación sobre dos operandos (argumentos) del mismo tipo, produciendo un resultado del mismo tipo que los operandos.
BiConsumer<T,U>	Representa una operación que acepta dos argumentos de entrada y no devuelve ningún resultado.
BiFunction<T,U,R>	Representa una función que acepta dos argumentos y produce un resultado.
BiPredicate<T,U>	Representa un predicado (función con valor booleano) de dos argumentos.

Especializaciones:

BooleanSupplier, DoubleBinaryOperator, DoubleConsumer, DoubleFunction<R>, DoublePredicate, DoubleSupplier, DoubleToIntFunction, DoubleToLongFunction, DoubleUnaryOperator, IntBinaryOperator, IntConsumer, IntFunction<R>, IntPredicate, IntSupplier, IntToDoubleFunction, IntToLongFunction, IntUnaryOperator, LongBinaryOperator, LongConsumer, LongFunction<R>, LongPredicate, LongSupplier, LongToDoubleFunction, LongToIntFunction, LongUnaryOperator, ObjDoubleConsumer<T>, ObjIntConsumer<T>, ObjLongConsumer<T>, ToDoubleBiFunction<T,U>, ToDoubleFunction<T>, ToIntBiFunction<T,U>, ToIntFunction<T>, ToLongBiFunction<T,U>, ToLongFunction<T>

Clases registro (v16)

Las clases de registro, son un tipo especial de clase, que ayudan a modelar agregados de datos simples e inmutables con una sintaxis más concisa que las clases normales.

Una declaración de registro especifica en un encabezado una descripción de su contenido; el constructor y los métodos descriptores de acceso, equals, hashCode, y toString se crean automáticamente. Los campos de un registro son inmutables porque la clase está destinada a servir como un simple "soporte de datos".

```
record Rectangle(double length, double width) { }
```

Clases registro (v16)

Una clase de registro declara automáticamente los siguientes miembros :

- Para cada componente del encabezado, los siguientes dos miembros:
  - Un campo/atributo private final con el mismo nombre y tipo declarado que el componente de registro. Este campo a veces se denomina campo de componente.
  - Un método de acceso public con el mismo nombre y tipo del campo componente (getter).
- Un constructor canónico cuya firma es la misma que la del encabezado. Este constructor asigna cada argumento de la expresión new, que instancia la clase de registro, al campo de componente correspondiente.
- Implementaciones de los métodos equals y hashCode, que especifican que dos clases de registros son iguales si son del mismo tipo y sus valores de campos de componentes son iguales.
- Una implementación del método toString que incluye la representación en cadena de todos los componentes de la clase de registro, con sus nombres.

Clases selladas (v17)

Las clases e interfaces selladas restringen qué otras clases o interfaces pueden extenderlas o implementarlas.

```
public sealed class Shape
    permits Circle, Square, Rectangle {
}
```

Las subclases permitidas tienen las siguientes restricciones:

- Deben extender directamente la clase sellada.
- Deben tener exactamente uno de los siguientes modificadores para describir cómo continúa el sellado iniciado por su superclase:
  - final: No se puede ampliar más
  - sealed: Solo se puede extender por sus subclases permitidas
  - non-sealed: Puede ampliarse con subclases desconocidas; una clase sellada no puede evitar que sus subclases permitidas hagan esto.
- Deben estar en el mismo módulo que la clase sellada o en el mismo paquete. La clase sellada debe poder acceder a ellas en el momento de la compilación.

TRATAMIENTO DE ERRORES Y LOGS

Excepciones

Situaciones anómalas que impiden la correcta ejecución del código y, en caso de no tratarlas, provocan la detención del programa.

Dos tipos de tratamiento:

- Antes de que se puedan producir se investigan todas las posibles causas de error.
- Después de producidas se trata el error.

Las excepciones derivan de la clase Throwable:

- Error: Errores irrecuperables de compilación, del sistema o de la JVM.
- Exception: Excepciones recuperables:
  - Implícitas: RuntimeException: muy frecuentes, normalmente por errores de programación.
  - Explícitas El resto de las excepciones, el Java obliga a tratarlas.

Métodos de las Excepciones

La causa de la excepción se identifica por la clase de la instancia generada en la situación anómala.

Todas las excepciones derivan de la clase Throwable, por lo tanto cuentan con una serie de métodos comunes:

- String getMessage()
  - Extrae el mensaje asociado con la excepción.
- String toString()
  - Devuelve una cadena que describe la excepción.
- void printStackTrace([PrintStream stream])
  - Imprime la traza de métodos donde se produjo la excepción.

Todas las excepciones cuentan con un constructor que acepta una cadena con la descripción de la excepción.

Tratamiento de las Excepciones

try

- Bloque de instrucciones vigiladas.

catch(<TipoExcepción> <Instancia de la Excepción>)

- Captura un tipo de excepción y realiza el tratamiento de la misma. En algunos casos pueden no tener tratamiento.
- Pueden existir múltiples cláusulas catch aunque son opcionales.

finally

- Sentencias que se ejecutan siempre, aunque no se produzca una excepción.
- Es opcional y única.

throw <Instancia de la excepción>;

- Lanza una excepción.

  Throw new <TipoExcepción>("…Mensaje de la Excepción…");

Captura de múltiples tipos de excepción (v7)

Para tratamientos similares de diferentes tipos de excepciones:

```
    catch (IOException ex) {
        logger.log(ex);
        throw ex;
    } catch (SQLException ex) {
        logger.log(ex);
        throw ex;
    }
```

Se puede:

```
    catch (IOException|SQLException ex) {
       logger.log(ex);
       throw ex;
    }
```


Try-con-recursos (v7)

```
try {
  BufferedReader br = new BufferedReader(new FileReader(path));
  try {
    // …
  } catch (IOException e) { … }
  } finally { br.close(); }
  }
} catch (Exception e) { … }
}
```

Para cualquier objeto que implemente java.lang.AutoCloseable:

```
try (BufferedReader br = new BufferedReader(new FileReader(path)))  {
  // …
} catch (IOException e) { … }
```

Notificar Excepciones

Es obligatorio tratar las excepciones implícitas.

No es necesario tratarlas en el momento de producirse, es posible relanzarlas para que sean tratadas a un nivel superior.

```
[...] <tipo> <nombreMétodo> ([<ListaDeParámetros>]) throws
<ListaDeExcepciones> {
        <Cuerpo del Método>
}
```

- La lista de excepciones separada por comas.
- Las excepciones que no aparezcan en la lista debes ser tratadas dentro del método.
- Las excepciones relanzadas deben ser tratadas por el método superior o, a su vez, ser relanzadas.

Excepciones propias

El tratamiento de excepciones requiere la creación de excepciones propias.

Existen varias formas de clasificar los errores: por origen (componente, módulo, API, …), por tipo (dispositivos, red, código, …), …

La recomendación es definirlas en función a como se capturan en los bloques catch para dar el tratamiento. La mayorías de los sistemas disponen de una jerarquía de clases para derivar la nuevas excepciones y utilizan el sufijo Exception en su nombre.

Al usar APIs de terceros siempre es conveniente envolver sus excepciones en excepciones propias (patrón facade) y definir un controlador por encima para procesar las cancelaciones.

Crear nuevas Excepciones

Se pueden crear excepciones propias.
Se crean heredando de la clase Exception o una de sus subclases.
Por convenio se utiliza el sufijo Exception en el nombre.
Es conveniente crearlas con dos constructores:

- Un constructor sin argumentos.
- Un constructor que recibe la cadena con la descripción de la excepción.

```
Class <NombreExcepción> extends Exception {
        public <NombreExcepción>() {
                super();
        }
        public <NombreExcepción>(String descripcion) {
                super(descripcion);
        }
}
```


Null

Null representa la ausencia de valor o valor vacío. Los nulos son una fuente inagotable de errores y excepciones (billion dollar mistake).

Si una función devuelve un valor, siempre debe devolverlo o, en caso de no ser posible, generar una excepción. Nunca debería devolver null, es lo mismo que no devolver valor pero obligando a comprobar el valor o incurrir en una NullPointerException sin contexto.

Siguiendo la misma línea, un argumento nunca debería ser nulo: si el parámetro es opcional debería ser tratado como tal (sobrecarga, valores por defecto, …). Para no abusar de la NullPointerException, en caso de recibir un null no esperado, se debería generar una excepción especifica (como InvalidArgumentException) o utilizar aserciones.

Optional<T> (v8)

Contenedor de objetos que pueden ser nulos (alternativa más segura).

- .isPresent(): indica si la referencia es distinta de null.
- .isEmpty(): indica si la referencia es null. (v11)
- .get(): devolver el valor o una Exception no hay valor.
- .orElse(): devolver un valor predeterminado si el valor no está presente.
- .ifPresent(): ejecutar un bloque de código si el valor está presente.

Se crean con los métodos de clase:

- .of(): Devuelve un Optional con el valor no nulo especificado.
- .empty(): Devuelve un Optional vacío.
- .ofNullable(): Devuelve una Optional si del valor especificado no es null o un Optional vacío si es null.

Aserciones

Las aserciones, como invariantes, permiten comprobar en cualquier parte del código de un método que se cumplen los invariantes establecidos.

Una aserción tiene la siguiente sintaxis:

```
assert condicion;
assert condición : mensaje;
```

donde la condición es una expresión lógica que debe cumplirse en el momento en que se evalúa la aserción.

Si la condición no se cumple, el programa termina con un AsertionError.

Tras el operador dos puntos puede ponerse un String con el mensaje que se devuelve como error.

Las aserciones pueden ser desactivadas una vez terminada la fase de depuración y pruebas.

Run Configurations > VM arguments: -ea

Aserciones como precondiciones

Como convenio, las precondiciones de un método público en Java es preferible que se comprueben mediante una condición y lancen la excepción IllegalArgumentException, IndexOutOfBoundsException, NullPointerException o la excepción apropiada de acuerdo con el error encontrado.

Sin embargo, en un método no público, sí se puede sustituir esa comprobación por una aserción que asegure que el método se utiliza de forma consistente en la clase.

Aserciones como postcondiciones

En este caso sí se recomienda el uso de aserciones para la comprobación de las postcondiciones de un método, pues se supone que el método se encuentra bien implementado.

En este caso, el uso de postcondiciones permite asegurar que se cumplen en todas las ejecuciones que se hacen del método.

Registros

El registro es el proceso de escribir en un lugar central mensajes con sucesos, errores o eventos durante la ejecución de un programa. Este registro permite notificar y conservar mensajes de error y advertencia, así como mensajes de información (por ejemplo, estadísticas de tiempo de ejecución) para que se puedan recuperar y analizar posteriormente.

El objeto que realiza el registro en las aplicaciones generalmente se llama Logger.

Java define la API de Registro de Java. Esta API de registro permite configurar qué tipos de mensajes se escriben. Las clases individuales pueden usar este registrador para escribir mensajes en los archivos de registro configurados.

El paquete java.util.logging proporciona las capacidades de registro a través de la clase Logger.

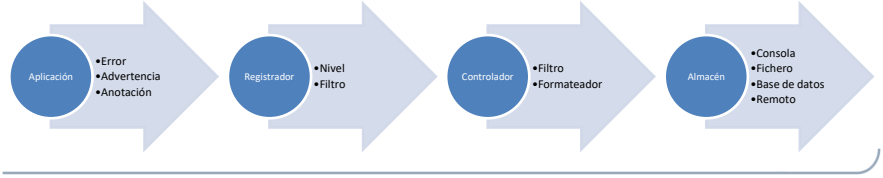
Registro

Las aplicaciones realizan llamadas de registro en objetos Logger (registradores). Estos objetos Logger asignan objetos LogRecord que se pasan a los objetos Handler (controladores) para su publicación.

Tanto los registradores como los controladores pueden usar niveles de registro y (opcionalmente) filtros para decidir si están interesados en un registro de registro en particular .

Cuando es necesario publicar un LogRecord externamente, un controlador puede (opcionalmente) usar un formateador para localizar y formatear el mensaje antes de publicarlo en un flujo de E / S.

Las API está estructuradas de modo que las llamadas al Logger tengan un coste mínimo cuando el registro está desactivado.



Niveles de registro

Cada mensaje de registro tiene un nivel de registro asociado. El nivel ofrece una guía aproximada de la importancia y urgencia de un mensaje de registro. Los objetos de nivel de registro encapsulan un valor entero, y los valores más altos indican prioridades más altas.

La clase Level define siete niveles de registro estándar:

- SEVERE (más alta)
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST (más leve)

Controladores y Formateadores

Java SE proporciona los siguientes controladores:

- ConsoleHandler: un controlador simple para escribir registros formateados en System.err
- StreamHandler: un controlador simple para escribir registros formateados en un OutputStream.
- FileHandler: un controlador que escribe registros de registro formateados en un solo archivo o en un conjunto de archivos de registro rotativos.
- SocketHandler: un controlador que escribe registros de registro formateados en puertos TCP remotos.
- MemoryHandler: un controlador que almacena los registros de registro en la memoria.

Java SE también incluye dos formateadores estándar:

- SimpleFormatter : escribe breves resúmenes "legibles por humanos" de los registros.
- XMLFormatter : escribe información detallada estructurada en XML.

Se pueden desarrollar nuevos controladores y formateadores que requieran una funcionalidad específica con las abstracciones e interfaces del API.

Configuración

La configuración de registro se puede inicializar mediante un archivo de configuración de registro que se leerá al inicio (tiene el formato estándar java.util.Properties). Alternativamente, la configuración de registro se puede inicializar especificando una clase que se puede utilizar para leer las propiedades de inicialización de fuentes arbitrarias, como LDAP, JDBC, etc.

- java.util.logging.MemoryHandler.size=100

La configuración inicial puede especificar niveles para registradores particulares. Estos niveles se aplican al registrador nombrado, o cualquier registrador debajo de él en la jerarquía de nombres, y se aplican en el orden en que están definidos en el archivo de configuración.

La configuración inicial puede contener propiedades arbitrarias para que las utilicen los controladores o los subsistemas que realizan el registro.

La configuración predeterminada que se envía con JRE pero los ISV, los administradores del sistema y los usuarios finales pueden anularla.

La configuración predeterminada hace un uso limitado del espacio en disco, no inunda al usuario con información, pero se asegura de capturar siempre la información clave de fallas. Establece un solo controlador en el registrador raíz para enviar la salida a la consola.

Realizar notificaciones

Para crear un registrador asociado a una clase:

```
private final static Logger LOGGER = Logger.getLogger(
        MyClass.class.getName());
```

Para establecer el nivel de registro:

```
LOGGER.setLevel(Level.INFO);
```

Para escribir en el registro:

```
LOGGER.severe("Es un error");
LOGGER.warning("Es un aviso");
LOGGER.info("Solo notifica");
LOGGER.finest("Carece de importancia");
```

Para configurar el registro:

```
Logger.getLogger("").addHandler(new FileHandler("%t/my.log"));
FileHandler fileTxt = new FileHandler("Logging.txt");
fileTxt.setFormatter(new SimpleFormatter());
Logger.getLogger("").addHandler(fileTxt);
```

REFLEXIÓN

Introducción

La reflexión se basa en el concepto de que las clases (y el resto de tipos) son objetos y se puede obtener una instancia de la definición de tipo para interactuar con ella.

La reflexión es comúnmente utilizada por programas que requieren la capacidad de examinar o modificar el comportamiento en tiempo de ejecución de las aplicaciones que se ejecutan en la máquina virtual Java. La reflexión es una técnica muy potente que permite realizar operaciones que de otro modo serían imposibles. La reflexión es una característica relativamente avanzada y solo debe ser utilizada cuando se tenga un conocimiento sólido de los fundamentos del lenguaje.

La reflexión permite:

- Codificar en tiempo de ejecución: guardar nombre en cadenas y utilizarlas para cargar dinámicamente clases por su nombre, instanciar las clases cargadas, invocar sus métodos por el nombre, …
- Explorar las clases para poder enumerar los miembros de las clases, extraer anotaciones, decidir que utilizar, …
- Poder examinar miembros privados de las clases para crear framewoks avanzados de pruebas, validaciones, serialización, mapeos, …

Inconvenientes

La reflexión es poderosa, pero no debe usarse de forma indiscriminada. Es preferible evitar su uso si es posible realizar una operación sin usar la reflexión.

Sobrecarga de rendimiento

Debido a que la reflexión involucra tipos que se resuelven dinámicamente, no se pueden realizar ciertas optimizaciones de máquinas virtuales Java. En consecuencia, las operaciones reflexivas tienen un rendimiento más lento que las alternativas no reflexivas y deben evitarse en secciones de código que se llaman con frecuencia en aplicaciones sensibles al rendimiento.

Restricciones de seguridad

Reflection requiere un permiso en tiempo de ejecución que puede no estar presente cuando se ejecuta en contextos de seguridad restringido (Ej. Applet).

Exposición de internos

Dado que la reflexión permite que el código realice operaciones que serían ilegales sin ella, como el acceso a miembros private, el uso de la reflexión puede provocar efectos secundarios inesperados y puede destruir la portabilidad. La reflexión rompe las abstracciones y, por lo tanto, puede cambiar el comportamiento con las actualizaciones de la plataforma.

Objetos de clase

Las instancias de tipo java.lang.Class encapsulan la definición de los tipos. No se pueden instanciar, es necesario utilizar uno de los siguientes mecanismos:

- Tipo.class
- Class.forName(Cadena con el nombre calificado)
- instancia.getClass()
- envolvente.TYPE

Las instancias de Class disponen de métodos que devuelven clases:

.getSuperclass(): Devuelve la superclase para la clase dada.

.getClasses(): Devuelve todas las clases, interfaces y enumeraciones públicas que son miembros de la clase, incluidos los miembros heredados.

.getDeclaredClasses(): Devuelve todas las clases, interfaces y enumeraciones que se declaran explícitamente en la clase incluidos las privados.

Obtención de Miembros

Están disponibles métodos que devuelven todos los miembros de un tipo o que devuelven uno concreto.

Así mismo, la versión por defecto solo obtiene los miembros públicos incluidos los heredados, la versión Declared solo obtiene los miembros declarado en la clase incluidos privados y protegidos.

- getFields(), getDeclaredFields(), getField(), getDeclaredField()
- getMethods(), getDeclaredMethods(), getMethod(), getDeclaredMethod()
- getConstructors(), getConstructor()
- getAnnotations(), getDeclaredAnnotations(), getAnnotation(), getDeclaredAnnotation()

Acceder a Miembros

Obtener y establecer valores de atributos:

```
Field cmd = instancia.getClass().getDeclaredField("atributo");
cmd.setAccessible(true);
String cad = (String)(cmd.get(instancia));
cmd.set(instancia, "nuevo valor");
```

Obtener e invocar un método:

```
Method m = instancia.getClass().getDeclaredMethod("metodo");
// m.setAccessible(true);
Object o = m.invoke(instancia);
m.invoke(instancia, "parámetro");
```

Crear nuevas instancias:

```
Class.newInstance() //solo puede invocar el constructor sin argumentos
Constructor ctor = Console.class.getDeclaredConstructor()[0];
ctor.newInstance()
```

Restringir el accesos a miembros

La invocación de setAccessible(true) indica que el objeto debe suprimir las comprobaciones del control de acceso del lenguaje Java cuando se utiliza la reflexión. Acceder a miembros privados fuera de la clase puede ser una amenaza para la seguridad de una aplicación Java.

Se puede restringir usando Security Manager, aunque no se instala de forma predeterminada, por lo que todos los permisos están disponibles. Algunos lanzadores (como el entorno de subprogramas) instalarán un SecurityManager, que a su vez utiliza una política basada en archivos de políticas de forma predeterminada. Se puede cambiar en el archivo "java.security". Otros entornos, como un contenedor de servlets o un contenedor Java EE, también pueden instalar un SecurityManager.

Se puede habilitar manualmente:

```
System.setSecurityManager(new SecurityManager());
```

Una vez habilitado el SecurityManager, de forma predeterminada el código ya no puede llamar a "setAccessible".

La seguridad se define en ${JRE_HOME}/conf/security/java.policy. Para volver a habilitarlo:

```
grant {
   permission java.lang.reflect.ReflectPermission "suppressAccessChecks";
};
```


Limitar la accesibilidad de los paquetes

Los contenedores pueden ocultar el código de implementación agregando a la propiedad de seguridad “package.Access”. Esta propiedad evita que las clases que no son de confianza de otros cargadores de clases se vinculen y utilicen la reflexión sobre la jerarquía de paquetes especificada.

```
private static final String PACKAGE_ACCESS_KEY = "package.access";
static {
   String packageAccess = java.security.Security.getProperty(
      PACKAGE_ACCESS_KEY
   );
   java.security.Security.setProperty(PACKAGE_ACCESS_KEY, (
      (packageAccess == null || packageAccess.trim().isEmpty()) ? "" :
      (packageAccess + ",")) + "com.example.paquete.protegido"
   );
}
```

ANOTACIONES

Metadatos

Las anotaciones son metadatos (o decoradores), con información adicional sobre el código que está escrito, destinado a las herramientas que manejan el código (que pueden utilizar esta información adicional si así lo desean) y sin significado en el propio código.

Aparecen en la versión 5.

Comienzan por @

Preceden inmediatamente al elemento que complementan (clase, interfaz, método, atributo, …)

Pueden llevar parámetros

Pueden ser usadas por el propio código a través de la reflexión (reflection).

Anotaciones predefinidas

java.lang

@Deprecated : Indica a los compiladores que adviertan cuando se utilizan que el elemento está en desuso o es obsoleto.

@Override : Indica que el método está sobreescribiendo un método de la superclase y que debe existir en la misma.

@SuppressWarnings : Indica a los compiladores que no muestren determinados avisos.

@SafeVarargs : Indica a los compiladores que supriman las advertencias de comprobación de tipos en los argumentos variables con genéricos. (v9)

@FunctionalInterface : Indica que debe cumplir con la especificación de interfaz funcional. (v8)

Anotaciones predefinidas

java.lang.annotation

@Documented : Indica las anotaciones cuyo uso deben ser documentados por javadoc de forma predeterminada.

@Retention : Indica cuánto tiempo se deben conservar las anotaciones con el tipo anotado: hasta que se compile (SOURCE), en el bytecode de clase (CLASS, por defecto) o en tiempo de ejecución (RUNTIME, para reflection)

@Target : Indica el tipo de elemento en el que es aplicable la anotación: PACKAGE, TYPE, ANNOTATION_TYPE, CONSTRUCTOR, METHOD, PARAMETER, FIELD y LOCAL_VARIABLE. [v8: TYPE_PARAMETER (genéricos) y TYPE_USE (casting)].

@Inherited : Indica que también se aplica a los herederos la anotación.

@Repeable : Indica que se pueden adjuntar múltiples instancias de la anotación al destino de la anotación. (v8)

Crear nuevas anotaciones

Los tipos de anotación se definen con @interface. Los parámetros se definen de manera similar a los métodos de una interfaz regular:

- Pueden ser valores únicos o matrices, de tipo String, Class, enumeraciones, tipos primitivos y tipos de anotaciones (anotaciones compuestas).
- Son opcionales si tienen un valor por defecto.
- Se asignan por su nombre (nombre="valor"), como valores únicos o lista de valores (nombre={1,2,3}). El primer parámetro se puede asignar sin nombre.

```
@Target({ElementType.FIELD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
public @interface MiAnotacion { int id(); String valor() default "";  }
```

Para la comprobación de las anotaciones en tiempo de ejecución a través de la reflexión:

```
Annotation[] anotaciones = MyClass.class.getAnnotations();
int id = this.getClass().getAnnotation(MiAnotacion.class).id();
```


PAQUETES Y MÓDULOS

Paquetes

Un paquete es una colección de clases e interfaces.
- Agrupan una serie de clases relacionadas.
- Resuelven los conflictos de nombre (nombre.paquete.NombreClase).
- Controlan la accesibilidad.

Definición de paquete:
- Implícita: Todas la clases de un directorio.
- Explicita: Utilizando en la primera línea no comentada del fichero de clase la cláusula:
  - package <nombre_paquete>;

Se recomienda:
- usar minúsculas en el nombre.
- Comenzar con el dominio propio invertido: com.miempresa.clasifica

Paquetes

Importación de paquetes para calificar la referencia a tipos de otros paquetes:
- import <paquete>.<clase>;
- import <paquete>.*;
  - El * hace referencia a todas las clases del paquete.

```
import java.lang.Math;
double r = Math.cos(Math.PI * theta);
```

Importaciones estáticas: Permiten el acceso sin cualificar a miembros estáticos de otras clases como si fueran propios:
- import static <paquete>.<clase>.<Método estático>;
- import static <paquete>.<clase>.*;
  - El * hace referencia a todos los métodos de la clase.

```
import static java.lang.Math.PI;
import static java.lang.Math.*;
double r = cos(PI * theta);
```

Administrar archivos

Muchas implementaciones de la plataforma Java dependen de sistemas de archivos jerárquicos para administrar archivos fuente y de clase, aunque la especificación del lenguaje Java no lo requiere.

El nombre calificado del miembro del paquete y el nombre de la ruta al archivo son paralelos (sustituyendo el punto por la barra apropiada). Un conjunto de directorios de puede comprimir en un fichero ZIP con extensión ".jar".

La ruta de clases se establece en la variable de sistema CLASSPATH y puede incluir varias rutas, separadas por un punto y coma (Windows) o dos puntos (UNIX). De forma predeterminada, el compilador y la JVM buscan las clases en el directorio actual, incluidos los contenidos de los ficheros JAR, y en la ruta de clases.

Paquetes básicos

java.lang
- Objetos fundamentales del lenguaje, es el único que no necesita importación.

java.util
- Clases auxiliares que permiten el manejo de estructuras de datos.

java.io
- Clases de entrada/salida a ficheros

java.applet
- Clase base de los applets.

java.awt, javax.swing
- Contiene la mayoría de las clases para implementar el interface gráfico de usuario.

Paquetes Avanzados

java.sql
- Proporciona las clases e interfaces para acceso a bases de datos con JDBC (Java Database Connectivity)

java.beans
- Soporte para el desarrollo de componentes

java.text
- Clases relacionadas con el formato de textos internacionales.

java.math
- Soporte avanzado de operaciones matemáticas en forma de métodos estadísticos.

Paquetes de Red

java.net
- Soporte de red: TCPI/IP, HTTP y soporte Internet y WWW

java.rmi
- Soporte para objetos distribuidos (Invocación Remota de Métodos).

java.security
- Clases que permiten el establecimiento de comunicaciones seguras.

java.servlet
- Soporte para desarrollo de aplicaciones servidor.

java.util.mime
- Soporte para las extensiones de correo Internet multiproposito.

Otras Ingenierías

Java Fundation Classes (JFC) y Java 2D
- Desarrollo de interfaces gráficos de usuarios (GUI), sustituye al AWT por componentes swing, permitiendo elegir el estilo, ampliando la funcionalidad y realizar operaciones de arrastrar y soltar. Incluye mejoras en el tratamiento de gráficos y textos en 2D.

Java Media Frameworks (JMF)
- Ingeniería que permite trabajar con elementos multimedia. Incorpora el Java 3D, que permite la manipulación de geometrías complejas en 3D.

Java IDL
- Permite la conectividad entre objetos distribuidos utilizando el estándar CORBA.

Java Native Interface (JNI)
- Es el interface de programación Java para ejecutar código nativo.

Framework

Es un conjunto de clases que definen un diseño abstracto para solucionar un conjunto de problemas relacionados.

¿Qué contiene un Framework?
- Un conjunto de clases e interfaces
- Modelo de uso de esas clases e interfaces
- Modelo de funcionamiento del framework.

Módulos (v9)

Si un conjunto de paquetes es lo suficientemente cohesivo, entonces dichos paquetes pueden agruparse en un módulo.

Un módulo categoriza algunos o todos sus paquetes como exportados, lo que significa que se puede acceder a sus tipos desde un código fuera del módulo.

Un paquete no es exportado por un módulo, solo es accesible por el código dentro de su módulo.

Si el código de un módulo desea acceder a los paquetes de otros módulos, entonces el módulo debe depender explícitamente de los otros módulo.

Por lo tanto, un módulo controla cómo sus paquetes usan otros módulos (dependencias) y controla cómo otros módulos usan sus paquetes (exportaciones).

Módulos (v9)

Hay cuatro tipos de módulos en Java Platform Module System (JPMS):

- Módulos del sistema: Estos son los módulos enumerados cuando ejecutamos el comando: java --list-modules. Incluyen los módulos Java SE (java.) y JDK (jdk.).
- Módulos de aplicación: Estos módulos son los que normalmente queremos construir. Se nombran y definen en el archivo module-info.java, que se compila e incluye en el archivo JAR.
- Módulos automáticos: El sistema de módulos crea por defecto un módulo automático para cada JAR sin descriptor de módulo. Dado que no contiene información sobre la modularidad, exporta todos los paquetes como abiertos y que lean todos los demás módulos.
- Módulo sin nombre: Cuando se carga una clase o JAR en la ruta de clases, pero no en la ruta del módulo, se agrega automáticamente al módulo sin nombre. Es un único módulo general para mantener la compatibilidad con versiones anteriores sin módulos. Todas las clases dentro del módulo sin nombre pueden leer todos los demás módulos (con o sin nombre) sin ninguna declaración explícita de ningún tipo. Los paquetes exportados por un módulo sin nombre solo pueden ser leídos por otro módulo sin nombre. No es posible que un módulo con nombre pueda leer (requiera) el módulo sin nombre.

Módulos (v9)

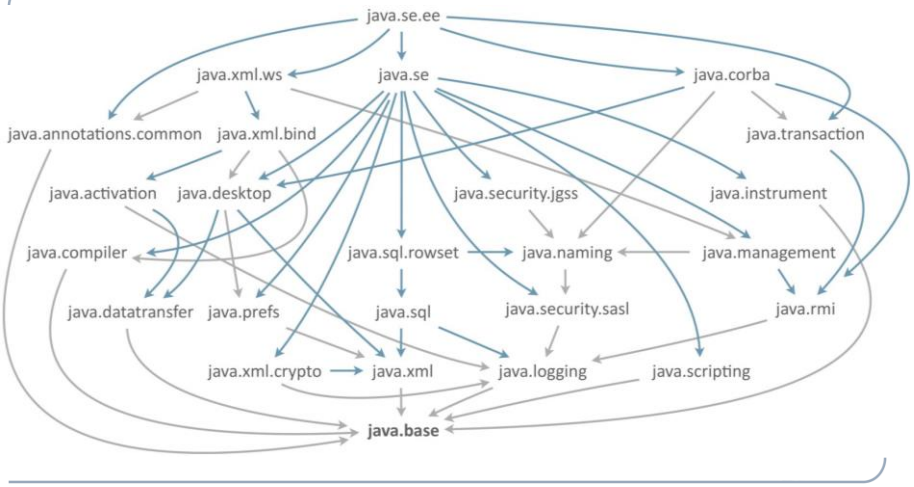
Los objetivos principales de la modularidad de Java son: hacer que las implementaciones de la Plataforma sean más fácilmente escalables a dispositivos pequeños, mejorar la seguridad y el mantenimiento, permitir un mejor rendimiento de las aplicaciones y proporcionar a los desarrolladores mejores herramientas para la programación en general.

La estructura modular del JDK a quedado como:

- El módulo “java.base” contiene las API fundamentales de Java SE Platform, todos los módulos tienen una dependencia implícita y obligatoria a este módulo.
- Los módulos estándar, cuyas especificaciones se rigen por el JCP, tienen nombres que comienzan con la cadena "java.".
- Todos los demás módulos son simplemente parte del JDK y tienen nombres que comienzan con la cadena "jdk.".

La estructura modular del JDK se puede visualizar como un grafo: cada módulo es un nodo y las flechas indican de quien dependen.

Módulos (v9)



Módulos (v9)

Un módulo de la plataforma Java consta de:

- Una colección de paquetes
- Opcionalmente, archivos de recursos y otros archivos como bibliotecas nativas
- Una lista de los paquetes accesibles en el módulo.
- Una lista de todos los módulos de los que depende este módulo, en caso ser necesario.
- Una lista de servicios que provee o consume, en caso ser necesario.

Al crear un módulo de aplicación, es importante que su nombre sea único. Se espera que los nombres de los módulos sigan la convención del "nombre de dominio inverso", al igual que los nombres de los paquetes.

Módulos (v9)

El archivo especial module-info.java es el descriptor del modulo y debe estar en el directorio raíz de los paquetes de clases de Java.

Todos los módulos estándar de Java SE tienen una dependencia implícita y obligatoria de java.base. No es necesario definir la dependencia java.base explícitamente.

No se permite tener dependencias circulares entre módulos. Si el módulo A requiere al módulo B, entonces el módulo B no puede requerir directa o indirectamente al módulo A.

Hay dos tipos de módulos:

- Abierto, con el modificador open, otorga acceso en tiempo de compilación a los tipos de los paquetes que se exportan explícitamente, pero otorga acceso mediante reflexión a los tipos de todos sus paquetes en tiempo de ejecución, como si todos los paquetes hubieran sido exportados.
- Normal, sin el modificador open, otorga acceso en tiempo de compilación y tiempo de ejecución a los tipos en los paquetes que se exportan explícitamente.

Módulos (v9)

Las directivas de una declaración de módulo especifican las dependencias del módulo en otros módulos, los paquetes que pone a disposición de otros, los servicios que proporciona o que consume:

- requires: indica un módulo del que depende. Modificadores:
  - transitive: propaga las dependencias transitivas del modulo requerido como dependencias implícitas del módulo actúa.
  - static: es obligatoria en tiempo de compilación pero opcional en tiempo de ejecución.
- exports: otorga acceso externo a los tipos public y protected de un paquete en tiempo de compilación y de ejecución. Con el modificador to se limitan los módulos (calificados) a los que se otorga acceso (amigos).
- opens: otorga acceso externo mediante reflexión a los tipos public y protected de un paquete en tiempo de ejecución, pero no en tiempo de compilación.
- provides: especifica el tipo del servicio provisto y con la cláusula with especifica uno o más proveedores de servicios.
- uses: especifica el tipo del servicio para descubrir proveedores.

Módulos (v9)

```
module com.example.infraestructura.data {
    requires com.example.domain.entities;
    exports com.example.infraestructura.unitofwork;
    exports com.example.infraestructura.repositories to
        com.example.domain.services;
    opens com.example.infraestructura.validators;

    // Común para proveedores y consumidores de servicios.
    requires com.example.domain.contracts;
    // Exponer el proveedor de servicios
    provides com.example.domain.contracts.MyService with
        com.example.repositories.MyServiceImpl;
    // Exponer el consumo de servicios
    uses com.example.domain.contracts.MyService;
}
```

Módulos (v9)

Con Service Loader se puede tener una interfaz de proveedor de servicios (SPI), Servicio, y múltiples implementaciones de la SPI, Proveedores de servicios.

Para evitar referencias circulares, un módulo de contratos exporta el o los interfaces de los servicios (aunque no recomendable, pueden ser clases abstractas o concretas).

Los módulos que deseen proveer implementaciones del servicio deben:

- Requerir el módulo de contratos.
- Disponer de una clase publica que implemente el interfaz del servicio (proveedor).
- Exportar la paquete del proveedor.
- Exponer al proveedor como implementación del servicio.

El modulo consumidor de los servicios debe:

- Requerir el módulo de contratos.
- Opcionalmente, requerir uno o varios módulos proveedores del servicio.
- Declarar el uso de los servicios.
- Consumir los servicios:

```
ServiceLoader<MyService> services = ServiceLoader.load(MyService.class);
services.forEach(srv -> srv.action());
```


Módulos (v9)

Para enumerar todos los módulos JDK

- java --list-modules

Para enumerar todos los módulos de un determinado modulo:

- java --describe-module java.sql
- jar --describe-module -f modulo.principal.jar

Para crear un módulo empaquetado:

- jar --create --file foo.jar --main-class com.foo.Main --module-version 1.0 -C foo/ classes resources

Para consultar las dependencias:

- jdeps --module-path . com.example.main.jar

Para ejecutar una aplicación modular:

- java --module-path ./ --module com.example.main/com.example.Principal

Para generar un JRE personalizado con lo mínimo imprescindible para poder ejecutar el módulo suministrado:

- jlink --module-path ./ --add-modules "com.example.main" --output build/
- build/bin/java --list-modules

PAQUETE JAVA.LANG

La clase Object

Todos las clase en Java derivan de la clase Object:

- Implícitamente: si no se define herencia, el sistema automaticamente le define como subclase de Object.
- Explícitamente: La jerarquía de clases tiene como origen a la clase Object.

Todas las clases heredan, por lo tanto, con los siguientes métodos:

- clone() : Crea un duplicado del objeto.
- equals(Object obj) : Comparacion de dos objetos.
- toString() : Devuelve una cadena con la representación del objeto.
- getClass() : Devuelve objeto Class de la clase (final).
- finalize() : Implica que todos los objetos tienen un destructor.
- hashCode() : Devuelve un código hash para el objeto.
- notify(), notifyAll() y wait() : Métodos relacionados con threads (finales).

Clases envolventes

Clases diseñadas para ser un complemento de los tipos primitivos. Las clases son:

(Byte, Short, Integer, Long, Float, Double): Number, Character, Boolean

Permiten implementar un mecanismo para pasar los tipos primitivos por referencia a los métodos.

Amplían la funcionalidad de los tipos primitivos.

El autoboxing/unboxing es una característica aparecida en JDK 1.5, que realiza automáticamente la conversión de tipo valor a clase y viceversa en función al tratamiento que se de al dato.

La clase Math (final) encapsula las funciones matemáticas y cuenta con las constantes PI y el número E.

Clases para Cadenas

La clase String

- Permite el manejo de cadenas de longitud fija.
- Cuenta con métodos que permiten la comparación, manipulación y conversión de cadenas y subcadenas.
- Aceptan el operador + como operador de concatenación.

La clase StringBuilder

- Permite el manejo de cadenas de longitud variable.
- Cuenta con métodos que permiten la manipulación y cambio de tamaño de cadenas.
- Aceptan el operador + como operador de concatenación.

La clase StringBuffer

- Versión síncrona de StringBuilder.

La clase System

Clase estática final que permite acceder a los recursos del sistema.

Propiedades:

- in : fichero de entrada estándar (InputStream).
- out : fichero de salida estándar (PrintStream).
- err : fichero de salida de error estándar (PrintStream).

Métodos:

- arraycopy(…) : Copia tablas elemento a elemento.
- currentTimeMillis(): Devuelve la hora actual en milisegundos (1/1/1970).
- exit(…) : Termina la ejecución de la aplicación.
- gc(), runFinalization() y runFinalizersOnExit() : Gestionan la memoria.
- setIn(), setOut() y setErr() : Cambian la entrada/salida estándar.
- load() y loadLibrary() : Cargan métodos nativos.
- getenv(), getProperties(), getProperty(), setProperty() : Permite la gestión de propiedades del sistema.

La clase Runtime

Clase estática final que permite acceder a la JVM.

Métodos:

- exec(...) : Permite ejecutar comandos y aplicaciones del S.O.
- exit(...) : Termina la ejecución y retorna al proceso padre.
- freeMemory() y totalMemory() : Informan de la disponibilidad de memoria.
- getRuntime() : Devuelve la instancia asociada a la aplicación.

PAQUETE JAVA.UTIL

Clases auxiliares

Random
- Generador de números pseudo-aleatorios.

BitSet
- Manejo de conjuntos de bits sin limitación de longitud.

Locale
- Representa las bases de internacionalización en programas Java.

Date, DateFormat, Calendar, GregorianCalendar, TimeZone, SimpleTimeZone
- Permiten manejar fechas y horas de distintos formatos y usos horarios.

java.time (v8)

Este paquete es una extensión de la versión Java 8 a las clases java.util.Date y java.util.Calendar que eran un poco limitadas para el manejo de fechas, horas y localización.

Las clases definidas en este paquete representan los principales conceptos de fecha - hora, incluyendo instantes, fechas, horas, periodos, zonas de tiempo, etc. Están basados en el sistema de calendario ISO (aaaa-mm-ddThh:mm:ss) que es el calendario mundial de-facto y sigue las reglas del calendario Gregoriano.

Cuenta con:

- Enumerados de mes y de día de la semana
- Las clases de fecha como el java.time.LocalDate manejan la fecha, pero, a diferencia del java.util.Date, es que es solo la fecha, y no hora.
- La clase java.time.LocalTime es similar java.util.Date pero se centra únicamente en la hora.
- La clase java.time.LocalDateTime manipula la fecha y la hora sin importar la zona horaria. Esta clase es usada para representar la fecha (año, mes, día) junto con la hora (hora, minuto, segundo, nanosegundo) y es, en efecto, la combinación de LocalDate y LocalTime.

Clases para Cadenas

La clase StringTokenizer

- Ayuda a dividir un string en substrings o tokens, en base a otro string (normalmente un carácter) separador entre ellos denominado delimitador.
- Implementa el interface Enumeration, por tanto define las funciones nextElement y hasMoreElements.

La clase Scanner

- Es un iterador que permite analizar una fuente de texto y extraer los tipos primitivos y cadenas usando expresiones regulares.
- Cuenta con los métodos hasNextXXX() y nextXXX(), donde XXX es el nombre del tipo primitivo.

Clases para Cadenas

La clase Formatter

- Permite formatear cadenas al estilo de printf.
- Similar a String.format pero permite globalización.

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

%b Booleano
%h Hashcode
%s Cadena
%c Caracter unicode
%d Entero decimal
%o Entero octal
%x Entero hexadecimal
%f Real decimal
%e Real notación científica
%g Real notación científica o decimal
%a Real hexadecimal con mantisa y exponente
%t Fecha u hora

Manipulaciones de tablas

La tablas, como objetos en Java, permiten:

- clone: crear una copia superficial de la tabla.
- length: informa de la dimensión de la tabla

La clase java.util.Arrays proporciona algunas otras operaciones útiles:

- binarySearch: Buscar en una matriz un valor específico para obtener el índice en el que se coloca.
- compare, compareUnsigned, equals, deepEquals: Comparar dos matrices para determinar si son iguales o no.
- mismatch: busca la primera discrepancia.
- fill: Rellenar una matriz introduciendo un valor específico.
- copyOf, copyOfRange: Devuelve una copia de la matriz.
- sort, parallelSort: Ordenar secuencialmente o simultáneamente una matriz en orden ascendente.

Colecciones

El Java da un amplio soporte para la creación y manejo de estructuras abstractas de datos.

Una colección es una estructura que permite el almacenamiento de un conjunto de objetos.

Colecciones vs Tablas:

- Los elementos no tienen que ser del mismo tipo.
- La capacidad es variable.

Para implementar las colecciones aporta:

- Interfaces.
- Genéricos
- Clases abstractas.
- Clases ya implementadas.

Interfaz Collection

Define una serie de métodos que nos servirán para acceder a los elementos de cualquier colección de datos, sea del tipo que sea.

Métodos:

- boolean add(Object o): Añade un elemento (objeto) a la colección.
- void clear(): Elimina todos los elementos de la colección.
- boolean contains(Object o): Indica si la colección contiene el elemento (objeto) indicado.
- boolean isEmpty(): Indica si la colección está vacía (no tiene ningún elemento).
- Iterator iterator(): Proporciona un iterador para acceder a los elementos de la colección.
- boolean remove(Object o): Elimina un determinado elemento (objeto) de la colección, devolviendo true si dicho elemento estaba contenido en la colección, y false en caso contrario.
- int size(): Nos devuelve el número de elementos que contiene la colección.
- Object [] toArray(): Nos devuelve la colección de elementos como un array de objetos.

Colecciones Genéricas

Las versiones iniciales de las colecciones, en la mayoría de los casos, utilizaban como tipo base el Object dado que permiten almacenar referencias a cualquier clase.

Podemos tener colecciones de tipos concretos de datos, lo que permite asegurar que los datos que se van a almacenar van a ser compatibles con un determinado tipo o tipos.

Además, con esto nos ahorramos las conversiones cast al tipo que deseemos, puesto que la colección ya se asume que será de dicho tipo.

A partir de JDK 1.5 deberemos utilizar genéricos siempre que sea posible.

Si creamos una colección sin especificar el tipo de datos que contendrá normalmente obtendremos un warning.

Inferencia de tipos genéricos (v7)

Inferencia de tipos para la creación de instancia genérico:

- Se puede sustituir los argumentos de tipo necesarios para invocar el constructor de una clase genérica con los paréntesis angulados de tipo vacíos (<>), informalmente llamados operador diamante, porque el compilador puede inferir los argumentos de tipo del contexto.

La expresión:

```
Map<String, List<String>> myMap = new HashMap<String,
List<String>>();
```

Equivale a:

```
Map<String, List<String>> myMap = new HashMap<>();
```

Tipos de colecciones

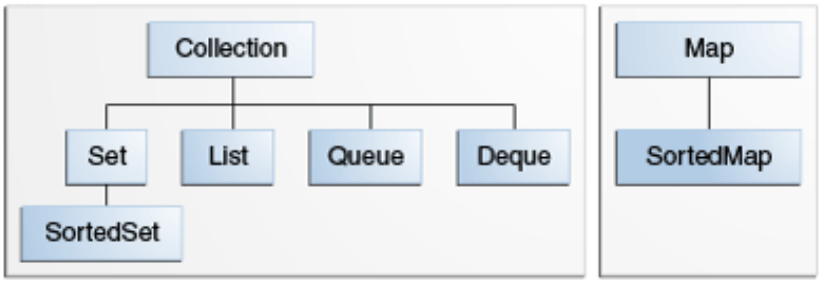
Listas: Una lista ordenada o secuencia. Normalmente permiten duplicados y tienen acceso aleatorio (es decir, puedes obtener elementos alojados en cualquier índice como si de un array se tratase).

Sets: Conjuntos que no admiten dos elementos iguales. Es decir, colecciones que no admiten que un nuevo elemento B pueda añadirse a una colección que tenga un elemento A cuando A.equals(B).

Pilas y Colas: Colecciones ordenadas que permiten crear colas LIFO o FIFO. No permiten acceso aleatorio, solo pueden tomarse objetos de su principio, final o ambos, dependiendo de la implementación.

Maps: Diccionario que asocian un valor a una clave. Parecido a la estructura de "array asociativo" que se encuentra en otros lenguajes. Un Map no puede tener dos claves iguales.

Interfaces



Implementaciones

Las **implementaciones de propósito general** son las implementaciones más utilizadas, diseñadas para el uso diario.

Las **implementaciones de propósito especial** están diseñadas para su uso en situaciones especiales y muestran características de rendimiento, restricciones de uso o comportamiento no estándar.

Las **implementaciones concurrentes** están diseñadas para admitir una alta concurrencia, generalmente a expensas del rendimiento.

Las **implementaciones de Wrapper** se utilizan en combinación con otros tipos de implementaciones, a menudo las de propósito general, para proporcionar una funcionalidad adicional o restringida.

Las **implementaciones de conveniencia** son mini implementaciones, generalmente disponibles a través de métodos de fábrica estáticos, que brindan alternativas convenientes y eficientes a las implementaciones de propósito general para colecciones especiales (por ejemplo, conjuntos de singleton).

Las **implementaciones abstractas** son implementaciones base que facilitan la construcción de implementaciones personalizadas.

Listas de elementos

Este tipo de colección se refiere a listas en las que los elementos de la colección tienen un orden (agregación), existe una secuencia de elementos en la que cada elemento estará en una determinada posición (índice) de la lista.

Las listas vienen definidas en la interfaz **List**.

Métodos:

- void add(int indice, Object obj): Inserta un elemento (objeto) en la posición de la lista dada por el índice indicado.
- Object get(int indice): Obtiene el elemento (objeto) de la posición de la lista dada por el índice indicado.
- int indexOf(Object obj): Nos dice cual es el índice de dicho elemento (objeto) dentro de la lista. Nos devuelve -1 si el objeto no se encuentra en la lista.
- Object remove(int indice): Elimina el elemento que se encuentre en la posición de la lista indicada mediante dicho índice, devolviéndonos el objeto eliminado.
- Object set(int indice, Object obj): Establece el elemento de la lista en la posición dada por el índice al objeto indicado, sobrescribiendo el objeto que hubiera anteriormente en dicha posición. Nos devolverá el elemento que había previamente en dicha posición.

Tipos de Lista

- **ArrayList**: Muy rápida accediendo a elementos, relativamente rápida agregando elementos si su capacidad inicial y de crecimiento están bien configuradas. Es la lista que deberías usar casi siempre.
- **LinkedList**: Una lista que también es una cola. Más rápida que ArrayList añadiendo elementos en su principio y eliminando elementos en general. Utilízala en lugar de ArrayList si realizas más operaciones de inserción (en posición 0) o de eliminación que de lectura. La diferencia de rendimiento es enorme.
- **Vector**: Vector es una colección deprecated (obsoleta), así que usadla únicamente si necesitáis un ArrayList concurrente. El rendimiento de Vector es superior al de Collections.syncronizedList(new ArrayList()).
- **CopyOnWriteArrayList**: Colección concurrente que es muy poco eficiente en operaciones de escritura, pero muy rápida en operaciones de lectura. Úsala sobre Vector (o synced ArrayList) cuando el número de lecturas concurrentes sea mucho mayor al número de escrituras.

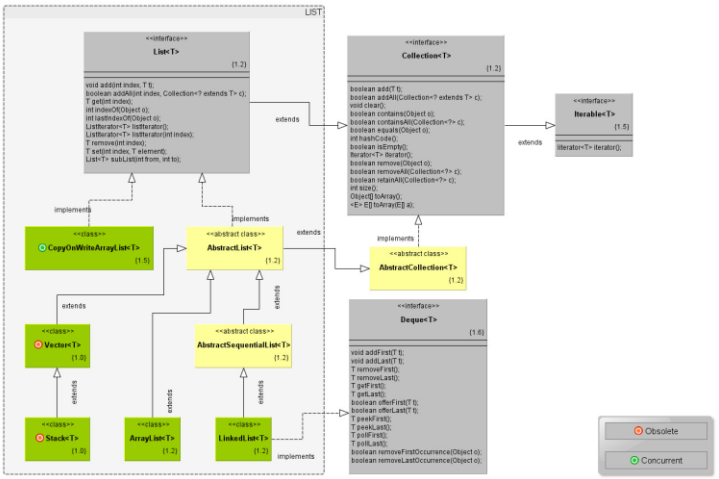
Clase Vector

Matriz de longitud variable de referencias a objetos.

Métodos:

- addElement(), insertElementAt(), setElementAt() :
  - Permiten introducir elementos en el vector.
- removeElement(), removeAllElements(), removeElementAt():
  - Eliminan elementos del vector.
- capacity(), ensureCapacity(), isEmpty(), setSize(), size():
  - Recuperan y manipulan el tamaño del vector.
- contains(), elementAt(), indexOf(), firstElement(), lastElement():
  - Buscan y recuperan elementos.
- elements():
  - Devuelve una enumeración para el recorrido del vector.

Jerarquía de Listas



Conjuntos

Los conjuntos son grupos de elementos en los que no encontramos ningún elemento repetido.

Consideramos que un elemento está repetido si tenemos dos objetos o1 y o2 iguales, comparándolos mediante el operador o1.equals(o2).

De esta forma, si el objeto a insertar en el conjunto estuviese repetido, no nos dejará insertarlo.

El método add devolverá un valor booleano, true si el elemento a añadir no estaba en el conjunto y ha sido añadido, o false si el elemento ya se encontraba dentro del conjunto.

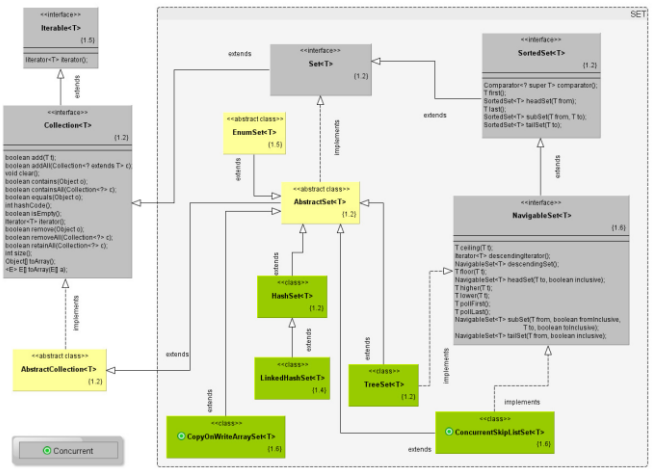
Un conjunto podrá contener como máximo un elemento null.

Los conjuntos se definen en la interfaz Set

Tipos de Set

- **HashSet**: La implementación más equilibrada de la interfaz Set. Es rápida y no permite duplicados, pero no tiene ningún tipo de ordenación. Utilízala si necesitas un control de duplicados pero ningún tipo de ordenación o acceso aleatorio.
- **LinkedHashSet**: Un HashSet que incluye ordenación de elementos por orden de entrada, una velocidad de iteración mucho mayor y un rendimiento mucho peor a la hora de añadir elementos. Utilízala si necesitas un Set ordenado por orden de inserción o si vas a usar un Set que vaya a realizar solamente operaciones de iteración.
- **TreeSet**: Un set que incluye una implementación de un árbol binario de búsqueda equilibrado. Este Set puede ser ordenado, pero su rendimiento es mucho peor en cualquier operación (menos iteración) respecto aun HashSet. Utilízalo solo si necesitas un Set con un criterio de ordenación específico y ten cuidado con las inserciones.
- **EnumSet**: La mejor implementación de Set para tipos enumerados (Enum).
- **CopyOnWriteArraySet**: Set concurrente que tiene un gran rendimiento de lectura, pero pésimo de escritura, eliminado y busquedas. Úsalo solo en Sets concurrentes que apenas tengan estas operaciones.
- **ConcurrentSkipListSet**: Set concurrente y ordenable. Utilízalo solo cuando requieras un Set ordenable (como TreeSet) en entornos de concurrencia. En Sets de tamaños muy grandes su rendimiento empeora notablemente.

Jerarquía de Conjuntos



Pilas y Colas

Las colas son estructuras que ofrecen un gran rendimiento al obtener elementos de su principio o de su final, representando colas LIFO / FIFO, aunque también veremos colas ordenadas en función de otros criterios.

Deberás usar una cola cuando vayas a recuperar siempre el primer o último elemento de una serie. Se usan para implementar las mencionadas colas LIFO / FIFO, así como colas de prioridades (como puede ser un sistema de tareas o de procesos).

Cabe destacar que hay dos tipos de colas, Queues y Deques. Las primeras solo proporcionan métodos para acceder al último elemento de la cola, mientras que las Deques permiten acceder a cualquiera de los dos extremos.

Tipos de colas

ArrayDeque: Una implementación de Deque de rendimiento excepcional. Implementa tanto cola LIFO como FIFO al ser una Deque y es la cola que deberías usar si quieres implementar una de estas dos estructuras.

ArrayBlockingQueue: Cola limitada (FIFO) con bloqueo, una vez creada su capacidad no se puede cambiar. Los intentos put de colocar un elemento en una cola llena bloquean la operación y los intentos take en una cola vacía se bloquearán de manera similar.

DelayQueue: Cola con bloqueo de elementos demorados, en la que un elemento solo se puede retirar una vez ha expirado un timeout.

LinkedBlockingDeque: Una Deque concurrente que has de usar cuando quieras usar un ArrayDeque en entornos multihilo.

LinkedList: LinkedList, anteriormente mencionada en la sección de listas, también es una Deque, sin embargo, su rendimiento es muy inferior al de ArrayDeque. No deberías usar LinkedList cuando quieras usar una cola.

PriorityQueue: Una cola que se ordena mediante un Comparator, permitiendo crear una Cola donde el primer elemento no dependerá de su tiempo de inserción, sino de cualquier otro factor (tamaño, prioridad, etc). Deberemos usarlo cuando necesitemos este comparator, ya que ArrayDeque no lo permite.

PriorityBlockingQueue: La versión concurrente de PriorityQueue.

Interface Queue

El interface Queue permite implementar colas de tipo FIFO (primero en entrar, primero en salir).

Métodos:

- add(): Añade un elemento a la cola
- poll(): Devuelve el primer elemento de la cola y lo elimina de ella
- peek(): Devuelve el primer elemento de la cola sin sacarlo
- size(): Número de elementos de la cola
- iterator(): Devuelve un elemento Iterator para poder recorrer la cola.
- isEmpty(): Indica si la cola está vacía o no.
- remove():Elimina el primer elemento de la Cola

Implementaciones:

- java.util.LinkedList
- java.util.PriorityQueue

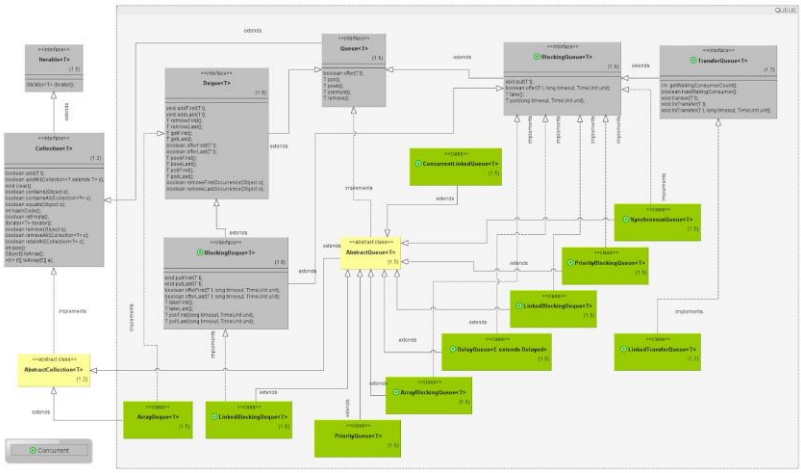
Clase Stack

Subclase de Vector que implementa una pila de tipo LIFO (último en entrar, primero en salir).

Métodos:

- empty(): Indica que la pila está vacía.
- peek(): Devuelve el elemento superior de la pila.
- pop(): Devuelve el elemento superior de la pila y lo elimina.
- push(): Añade un elemento.
- search(): Busca un elemento

Jerarquía de Colas



Mapas

Aunque muchas veces se hable de los mapas como una colección, en realidad no lo son, ya que no heredan de la interfaz Collection. Los mapas se definen en la interfaz Map.

Un mapa es un diccionario que relaciona una clave con un valor. Contendrá un conjunto de claves y a cada clave se le asociará un determinado valor.

En versiones anteriores este mapeado entre claves y valores lo hacía la clase Dictionary, que ha quedado obsoleta. Tanto la clave como el valor puede ser cualquier objeto.

Métodos:

- Object get(Object clave): Nos devuelve el valor asociado a la clave indicada
- Object put(Object clave, Object valor): Inserta una nueva clave con el valor especificado. Nos devuelve el valor que tenía antes dicha clave, o null si la clave no estaba en la tabla todavía.
- Object remove(Object clave): Elimina una clave, devolviendonos el valor que tenía dicha clave.
- Set keySet(): Nos devuelve el conjunto de claves registradas
- int size(): Nos devuelve el número de parejas (clave,valor) registradas.

Tipos de Map

- **HashMap**: La implementación más genérica de Map. Un array clave→valor que no garantiza el orden de las claves (de la misma forma que un HashSet). Si necesitas un Map no-concurrente que no requiera ordenación de claves, este es el tuyo. Si os fijáis en el código de HashSet, veréis que curiosamente utiliza un HashMap internamente.
- **LinkedHashMap**: Implementación de map que garantiza orden de claves por su momento de inserción; es decir, las claves que primero se creen serán las primeras. También puede configurarse para que la orden de claves sea por tiempo de acceso (las claves que sean accedidas precederán a las que no son usadas). Itera más rápido que un HashMap, pero inserta y elimina mucho peor. Utilízalo cuando necesites un HashMap ordenado por orden de inserción de clave.
- **TreeMap**: Un Map que permite que sus claves puedan ser ordenadas, de la misma forma que TreeSet. Úsalo en lugar de un HashMap solo si necesitas esta ordenación, ya que su rendimiento es mucho peor que el de HashMap en casi todas las operaciones (excepto iteración).

Tipos de Map

- **EnumMap**: Un Map de alto rendimiento cuyas claves son Enumeraciones (Enum). Muy similar a EnumSet. Usadlo si vais a usar Enums como claves.
- **WeakHashMap**: Un Map que solo guarda referencias blandas de sus claves y valores. Las referencias blandas hacen que cualquier clave o valor sea elegible por el recolector de basura si no hay ninguna otra referencia al mismo desde fuera del WeakHashMap. Usa este Map si quieres usar esta característica, ya que el resto de operaciones tienen un rendimiento pésimo. Comúnmente usado para crear registros que vayan borrando propiedades a medida que el sistema no las vaya necesitando y vaya borrando sus referencias.
- **HashTable**: Map deprecated y concurrente. Básicamente, es un HashMap concurrente que no debes usar nunca. En su lugar, utiliza ConcurrentHashMap.
- **ConcurrentHashMap**: Un Map concurrente que no permite valores nulos. Sustitución básica de HashTable. Usala si necesitas un HashMap concurrente.

Clase abstracta Dictionary

Clase abstracta que permite el almacenamiento de conjuntos de pares clave/valor.

Métodos:

- put(): Permiten introducir un elemento.
- get(): Recupera un valor por su clave.
- remove(): Elimina un elemento.
- isEmpty(), size(): Recuperan el tamaño del vector.
- elements(): Devuelve una enumeración de los valores.
- keys(): Devuelve una enumeración de las claves.

Clase Hashtable

Subclase instanciable de la clase Dictionary.

Implementa el mecanismo de hashing (valor único) de claves para el almacenamiento y recuperación de elementos.

La clase de los objetos utilizados como clave deben sobreescribir los métodos hashCode() y equals(). La clase String implementa los dos métodos.

Métodos adicionales:

- clear(): Borra toda la tabla.
- contains(), containsKey(): Busca si contiene el valor o la clave.
- rehash(): Recalcula la tabla y sus claves.

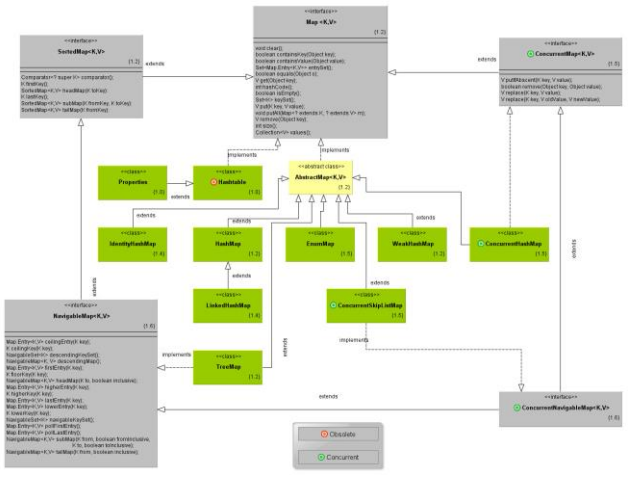
Clase Properties

Subclase persistente de Hashtable, donde las claves y los valores son cadenas.

Métodos adicionales:

- getProperty(): Devuelve el valor de una propiedad.
- list(): Envía una lista de las propiedades y sus valores
- load(): Recupera la tabla.
- save(): Guarda la tabla.
- propertyNames(): Devuelve una enumeración de las claves.

Jerarquía de Mapas



Wrappers

La clase Collections aporta una serie métodos para cambiar ciertas propiedades de las listas. Los wrappers son objetos que 'envuelven' al objeto de nuestra colección, pudiendo de esta forma hacer que la colección esté sincronizada o que pase a ser de solo lectura.

Podemos conseguir que las operaciones se ejecuten de forma sincronizada envolviendo nuestro objeto de la colección con un wrapper, que será un objeto que utilice internamente nuestra colección encargándose de realizar la sincronización cuando llamemos a sus métodos. Para obtener estos wrappers utilizaremos los siguientes métodos estáticos de Collections:

- Collection synchronizedCollection(Collection c), List synchronizedList(List l), Set synchronizedSet(Set s), Map synchronizedMap(Map m), SortedSet synchronizedSortedSet(SortedSet ss), SortedMap synchronizedSortedMap(SortedMap sm)

Para obtener versiones de sólo lectura de nuestras colecciones:

- Collection unmodifiableCollection(Collection c), List unmodifiableList(List l), Set unmodifiableSet(Set s), Map unmodifiableMap(Map m), SortedSet unmodifiableSortedSet(SortedSet ss), SortedMap unmodifiableSortedMap(SortedMap sm)

Tipos primitivos en las colecciones

Los tipos primitivos del Java son: boolean, int, long, float, double, byte, short, char.

Cuando trabajamos con colecciones los elementos que contienen éstas son siempre objetos, por lo que en un principio no podríamos utilizar tipos valor.

Para hacer esto posible tenemos una serie de clases envolventes que se encargarán de envolver a estos tipos primitivos, permitiéndonos tratarlos como objetos y por lo tanto utilizarlos como elementos de colecciones. Las clases e envolventes tienen nombre similares al del tipo primitivos que encapsulan, empezando con mayúscula: Boolean, Integer, Long, Float, Double, Byte, Short, Character.

El autoboxing/unboxing es una característica aparecida en JDK 1.5, que realiza automáticamente la conversión de tipo valor a clase y viceversa en función al tratamiento que se de al dato.

Métodos factoría para colecciones (v.9)

Se incorporan a las colecciones métodos de clase con argumentos variables que simplifican su creación con una lista de valores:

```
// Java 8
List<String> list = Collections.unmodifiableList(Arrays.asList("a", "b",
        "c"));
Set<String> set = Collections.unmodifiableSet(new
        HashSet<>(Arrays.asList("a", "b", "c")));
Map<String, Integer> map = new HashMap<>();
map.put("a", 1);

// Java 9
List<String> list = List.of("a", "b", "c");
Set<String> set = Set.of("a", "b", "c");
Map<String, Integer> map = Map.of("a", 1);
```


Interfaces Enumeration e Iterator

Enumeration: Define los métodos que permiten recorrer los elementos de un conjunto de objetos.

- boolean hasMoreElements()
  - Indica si quedan elementos por recorrer.
- Object nextElement()
  - Devuelve el siguiente elemento del conjunto.

Iterator<E>: Define los métodos que permiten recorrer los elementos de una colección o un array. A diferencia de las enumeraciones, los iteradores permiten al llamador eliminar elementos de la colección subyacente durante la iteración.

- boolean hasNext()
  - Indica si quedan elementos en la iteración.
- E next()
  - Devuelve el siguiente elemento de la iteración.

Recorrer las colecciones

Originalmente, mediante la interfaz Enumeration:

```
while (enum.hasMoreElements()) {
  Object item = enum.nextElement();
  // Hacer algo con el item leido
}
```

Posteriormente, mediante la interfaz Iterator:

```
while (iter.hasNext()) {
   Object item = iter.next();
   if(condicion_borrado(item))
      iter.remove();
}
```

A partir de JDK 1.5:

```
ArrayList<String> a = new ArrayList<String>();
for(String cadena: a)
   System.out.println (cadena);
```


Comparación de objetos

Comparar objetos es fundamental para hacer ciertas operaciones y manipulaciones en estructuras de datos.

Disponemos de diferentes mecanismos:

- Sobrecarga de equals
- Implementación de Comparable
- Implementación de Comparator

Sobre escritura de equals

Todas las clases heredan de la clase Object el método equals(Object o) que es sobre escribible (también conviene sobrecargar el método hashCode):

```
@Override
public boolean equals(Object o) {
   // return true o false, según un criterio
}
```

Cuando se sobrecarga el método equals se deben cumplir las siguientes propiedades:

- Reflexividad: x.equals(x) devuelve siempre verdadero, si no es nulo.
- Simetría: para cualquier par de instancias no nulas, x.equals(y) devuelve verdadero si y sólo si y.equals(x) también devuelve verdadero.
- Transitividad: si x.equals(y)==true y y.equals(z)==true, entonces x.equals(z) también será verdadero, para cualesquiera instancias no nulas.
- Consistencia: múltiples llamadas al método con las mismas instancias devuelven el mismo resultado.
- Comparación con null falsa: x.equals(null) devuelve falso.

Sobre escritura de hashCode

hashCode devuelve un valor de resumen hash para el objeto. Este método es utilizado en tablas hash como las proporcionadas por HashMap.

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

Cuando se sobrecarga el método hashCode se deben cumplir las siguientes normas:

- Siempre que se invoca en el mismo objeto más de una vez durante la ejecución de una aplicación Java, el método hashCode debe devolver el mismo número entero, siempre que no se modifique la información utilizada en las comparaciones equals del objeto. Este número entero no necesita permanecer consistente de una ejecución de una aplicación a otra ejecución de la misma aplicación.
- Si dos objetos son iguales, según el método equals, sus hashCode deben ser iguales.
- No es obligatorio que dos objetos distintos generen hasCode distintos, pero en caso de serlo pueden mejorar el rendimiento de las tablas hash.

Implementación de Comparable

Los algoritmos, como Collections.sort( ), requieren que los objetos implementen la interfaz Comparable para que tengan un método compareTo() que devuelva un número negativo, positivo o cero, según si un objeto es menor, mayor o igual que el otro (0 debería equivaler a equals).

Por supuesto, no todos los objetos se pueden comparar en términos de mayor o menor. Así que, el hecho de que una clase implemente Comparable, nos indica que una colección de dichos objetos tiene un "orden natural".

```
public class Persona implements Comparable<Persona> {
   public int id;
   public String apellido;
   ...
   @Override
   public int compareTo(Persona p) {
      return this.id - p.id;
   }
   …
}
```


Implementación de Comparator

En muchas colecciones la ordenación podría realizarse por diferentes criterios, por lo que se debe utilizar un comparador externo.

Para ello tenemos que implementar la interfaz Comparator que nos obliga a implementar el método compare. No debemos hacerlo dentro de la propia clase cuyas instancias vamos a comparar, es necesario crear una clase adicional que puede ser anónima:

```
public class ComparaPersonaPorNombre implements Comparator<Persona>{
   public int compare(Persona p1, Persona p2) {
      return p1.nombre.compareToIgnoreCase(p2.nombre);
   }
}
```

Para ordenar la colección:

```
Collections.sort(personas); //Comparable.compareTo
Collections.sort(personas, new ComparaPersonaPorNombre());
      //Comparator.compare
```

Collection Stream (v8)

En la versión 8 se agrego la implementación del interfaz Stream<T> a los array, enumeraciones y colecciones de la biblioteca estándar.

Permite acceder a sus elementos de una nueva forma distinta a los Iterator, recorrerlos como una secuencia de elementos con soporte para operaciones de agregación secuenciales y paralelas así como el uso de expresiones lambda.

Un flujo consiste una fuente (una colección), varias operaciones intermedias (de filtrado o transformación) y una operación final que produce un resultado (suma, cuenta...).

```
int sum = widgets.stream()
              .filter(w -> w.getColor() == RED)
              .mapToInt(w -> w.getWeight())
              .sum();
```


Collection Stream (v8)

Características:

- Sin almacenamiento. Un flujo no es una estructura de datos que almacena elementos, es un cursor o intermediario con una colección, una matriz, un canal de E/S, …
- Inmutable. El origen no es modificado.
- Consumible. Los elementos de una corriente solo se visitan una vez durante la vida de una corriente.
- Sólo se pueden «consumir» elementos que estén en el stream. Al terminar el recorrido, el stream desaparece.
- Una cadena de operaciones sólo puede ocurrir una vez en un stream determinado.
- Puede ser en serie (por defecto) o en paralelo

Las estructuras que soportan esta nueva API se encuentran en el paquete java.util.stream.

Se han modificado los paquetes de colecciones, IO, … para que incorporen el métodos stream() para poder usarlos con la nueva API.

Collection Stream (v8)

Java Stream nos permite realizar operaciones de tipo filtro/mapeo/reducción sobre colecciones de datos.

Modelo filtro/mapeo/reducción:

- Filtro: se seleccionan los datos que se van a procesar
- Mapeo: se convierten a otro tipo de dato
- Reducción: y al final se obtiene el resultado deseado.

El filtrado y el mapeo son operaciones intermedias con estado o sin estado: si crean o no nuevos valores.

Las operaciones con estado pueden necesitar procesar toda la entrada antes de producir un resultado.

Collection Stream (v8)

Para crear un Stream con sus métodos estáticos:

- iterate(): Devuelve un Stream producido por una iteración
- generate(): Devuelve un Stream producido por un Supplier.
- concat(): Crea un Stream secuencial con los elementos de los dos Stream.
- of(): Devuelve un Stream con los elementos especificados.
- empty(): Devuelve un Stream vacío .
- ofNullable(): Devuelve un Stream que contiene un solo elemento, si no es nulo o un Stream vacío si es nulo.

Se pueden obtener de:

- Collection, Arrays, BufferedReader, Files, Random y otras muchas clases.

Collection Stream (v8)

Para el filtro/mapeo:

- distinct(): Devuelve un Stream con los elementos distintos elementos.
- filter(): Devuelve un Stream con los elementos que coinciden con el predicado.
- dropWhile(): Devuelve un Stream sin los elementos que coinciden con el predicado.
- takeWhile(): Devuelve un Stream con los elementos que coinciden con el predicado hasta encontrar el primero que no coincide.
- map(): Devuelve un Stream con los elementos transformados por la función dada. [mapToDouble() , mapToInt() mapToLong()]
- flatMap(): Devuelve un Stream que consta de los resultados de reemplazar cada elemento de este flujo con el contenido de un Stream mapeado producido al aplicar la función de mapeo proporcionada a cada elemento. [flatMapToDouble(), flatMapToInt(), flatMapToLong()]
- peek() Devuelve el Stream después a aplicar la acción proporcionada en cada elemento a medida que se consumen elementos del flujo.
- sorted(): Devuelve un Stream con los elementos ordenados.
- skip(): Devuelve un Stream después de descartar los primeros n elementos.
- limit(): Devuelve un Stream con un número máximo de elementos.

Collection Stream (v8)

Para reducir:

- allMatch(): Devuelve true si todos los elementos la secuencia cumplen con el predicado.
- anyMatch(): Devuelve true si algún elemento de la secuencia cumplen con el predicado.
- findFirst(): Devuelve un Optional con el primer elemento del flujo.
- findAny(): Devuelve un Optional vacío si el flujo está vacío o con cualquier elemento si no.
- reduce(): Reduce la secuencia a un valor calculado con la función de acumulación asociativa proporcionada y devuelve un Optional.
- count(): Devuelve el recuento de elementos de esta secuencia.
- min(), max(): Devuelve un Optional con el elemento mínimo o máximo de la secuencia según Comparator proporcionado.
- forEach(): Realiza una acción para cada elemento de esta secuencia.
- collect(): Realiza una operación de reducción mutable en los elementos de este flujo usando Collectors. [Collectors.toList(), Collectors.toCollection(TreeSet::new)
- toArray(): Devuelve una matriz con los elementos de la secuencia.

Collection Stream (v8)

```
int sum = widgets.stream()
          .filter(b -> b.getColor() == RED)
          .map(b -> b.getWeight())
          .reduce(0, (a, item) -> a + item);
List<String> list = people.stream()
          .sorted(Person::getId)
          .skip(page * rows)
          .limit(rows)
          .collect(Collectors.toList());
Map<Department, Integer> totalByDept = employees.stream()
          .collect(Collectors.groupingBy(Employee::getDepartment,
                    Collectors.summingInt(Employee::getSalary)));
people.stream().map(Person::getName).forEach(System.out::println);
```


Collection Stream (v8)

Los streams actúan en modo perezoso: las operaciones solo se realizan cuando se llama a la operación final (reducción).

Son mas eficientes, en algunos casos no necesitando procesar todos los elementos del stream para devolver el resultado final.

Permiten dividir las operaciones en varias instrucciones, aplicando condicionales, devolviendo como valor de retorno y postergando la ejecución.

```
Stream<Person> query = lista.stream();
if(femaleOnly)
   query = query.filter(e -> e.getGender() == Person.Sex.FEMALE);
if(paged)
   query = query.sorted().skip(page * rows).limit(rows);
// ...
query.forEach(e -> e.setSalar(e.getSalar() * delta));
```

Parallel Stream (v8)

Las operaciones en un flujo paralelo se procesan utilizando varios subprocesos. No es necesario realizar pasos adicionales para procesar flujos paralelos.

La mayoría de los métodos de la API Streams producen flujos secuenciales de forma predeterminada. Para crear un flujo paralelo se usa el método parallelStream() de la interfaz Collection (List, Set) o el método parallel() de una secuencia secuencial para en una paralela. El método sequential() convierte una secuencia paralela en una secuencial.

```
List<Integer> listOfIntegers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);
System.out.println("Sequential Stream: ");
listOfIntegers.stream().forEach(e -> System.out.print(e + " "));
System.out.println("\nParallel Stream: ");
listOfIntegers.stream().parallel().forEach(e -> System.out.print(e + " "));
```