Alejandro Jurado Gutiérrez

Java

Manual para programadores

Introducción

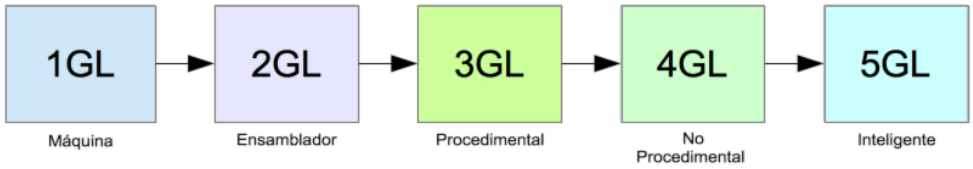
Lenguajes de programación

Un lenguaje de programación permite definir de manera formal un proceso que puede ser llevado a cabo por una máquina.

Un lenguaje está definido por una serie de símbolos, palabras clave y reglas sintácticas para la definición de instrucciones, que constituye principalmente la diferencia entre los distintos lenguajes existentes, además de su comportamiento en tiempo de ejecución, la gestión de memoria y procesos y el paradigma de programación en el cual se inscribe.

Evolución

La evolución de los lenguajes de programación facilita la abstracción del mundo de las máquinas hacia lenguajes más cercanos a las personas:

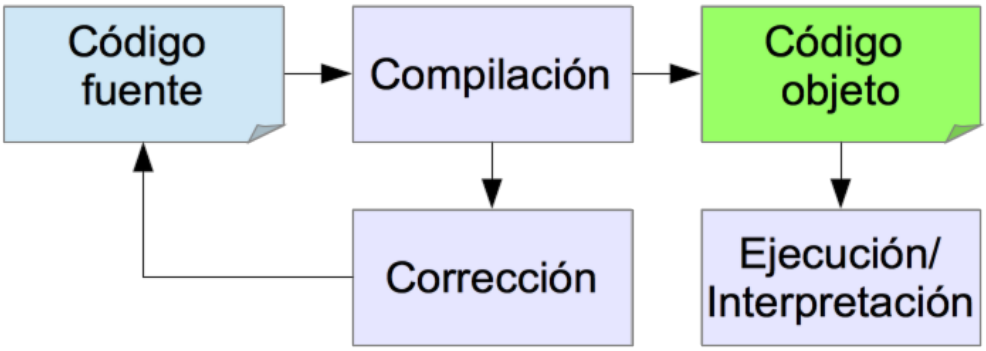


Algunos ejemplos de lenguajes:

- Fortran
- Cobol
- Basic
- Pascal
- C
- C++
- Haskell
- Visual Basic
- Delphi
- Java
- Python
- .NET
- PHP

Fases en la programación

Las fases por las que atraviesa un programa definido en un lenguaje concreto son las siguientes:



Cuando el lenguaje es interpretado, la fase de compilación realiza un análisis sintáctico inicial y la generación de código intermedio, y en fase de ejecución actúa un intérprete JIT.

Paradigmas de programación

Los paradigmas representan distintas formas de organizar y disponer el código para dar solución a un problema. Los distintos paradigmas que existen son:

Paradigma	Características
Procedimental	• Conjunto lineal de instrucciones • Múltiples procedimientos • Dificulta mantenibilidad y manejo de datos de entrada • Compilación y ejecución optimizadas
Declarativo	• Separa la definición del problema de las tareas a realizar • No mejora mantenibilidad • Dificulta la depuración • Ejecución basada en reglas o consultas
Orientado a Objetos	• Objetos y relaciones entre ellos • Mejora mantenibilidad y modularidad • Mejora la gestión de datos de prueba • Ejecución basada en paso de mensajes
Dirigido por Eventos	• Eventos que disparan acciones en el programa • Interacción a través de GUI • Similar a Orientado a Objetos • Ejecución basada en eventos

Referencias

https://es.wikipedia.org/wiki/Lenguaje_de_programación

https://es.wikipedia.org/wiki/Anexo:Cronolog%C3%ADa_de_los_lenguajes_de_programación

El lenguaje Java

Java es un lenguaje de programación Orientado a Objetos cuya primera versión fue desarrollada en 1995 por la compañía Sun Microsystems, actualmente propiedad de Oracle Corporation.

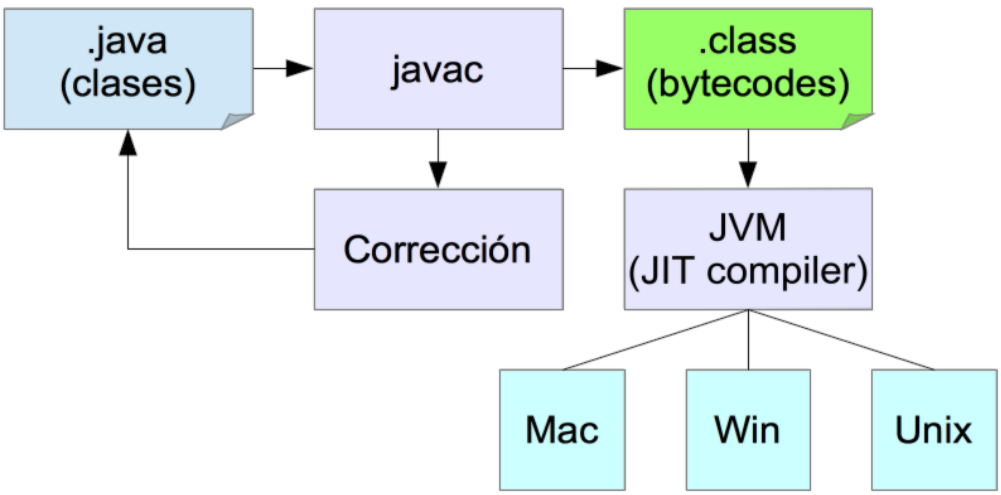
La principal característica de diseño de la plataforma Java fue la independencia de ejecución bajo la filosofía WORA (Write Once Run Anywhere), que evita la necesidad de recompilación del código para las distintas plataformas. Además de ésta, Java dispone de otra serie de características destacadas:

- Orientado a Objetos: sigue el paradigma orientado a objetos, incluyendo además tipos simples para la mejora del rendimiento en ejecución.
- Robusto y seguro: se trata de un lenguaje fuertemente tipado, con gestión automática de memoria y ejecución en entorno cerrado.
- Alto rendimiento: rendimiento optimizado a pesar de tratarse de un lenguaje interpretado, gracias a la gestión ligera de memoria y la interfaz nativa de código.
- Multitarea: permite la ejecución de múltiples hilos de forma simultánea.

La máquina virtual

La máquina virtual de Java es un software alojado en el dispositivo destino de nuestra aplicación, la cual se encarga de la traducción de las instrucciones del lenguaje origen (bytecodes) a instrucciones propias del sistema operativo subyacente.

El hecho de trabajar con una máquina virtual permite que el lenguaje Java sea multiplataforma, el código se escribe una vez y es posible ejecutarlo en cualquier sistema compatible.



JDK y JRE

El Java Development Kit es el conjunto de programas, herramientas y librerías que permite a los desarrolladores escribir, compilar y ejecutar aplicaciones Java.

Los componentes más destacados del JDK son:

- javac: compilador de clases Java
- java: intérprete de Java (JRE)
- javadoc: generador de documentación en formato API de Java
- jar: empaquetador de archivos y ejecutables
- keytool: gestión de claves y certificados

El JDK incluye un Java Runtime Environment, que contiene la máquina virtual para la ejecución de tales aplicaciones. El JRE se puede descargar de forma separada al JDK.

Existen distintas versiones del JDK que han ido surgiendo a lo largo de la historia de Java. Cada una de las versiones incluye nuevas clases, herramientas, utilidades o mejoras sobre las ya existentes. La compatibilidad hacia adelante normalmente no se ve comprometida, ya que aunque en ocasiones se marcan algunos elementos como no recomendados para su uso (deprecated) no se eliminan de la API. Por contra nuevos elementos o cambios en la forma de definir las aplicaciones hacen que el código más actualizado no sea totalmente compatible con las versiones anteriores.

Ventajas de Java

Existe una serie de ventajas adicionales a las capacidades avanzadas de Java:

- Fácil aprendizaje: a pesar de sus características orientadas a objetos, Java es un lenguaje sencillo de aprender y aplicar para la construcción de aplicaciones.
- Código compacto y optimizado: las APIs existentes en el JDK y la comunidad Java, características como la gestión de memoria simple y la capacidad de reutilización facilitan el ahorro de código necesario para implementar las soluciones.
- Portabilidad: la capacidad de Java para ejecutarse independientemente del sistema destino mejora el proceso de construcción y distribución de aplicaciones.

Referencias

https://docs.oracle.com/javase/tutorial/getStarted/index.html

https://es.wikipedia.org/wiki/Java_(lenguaje_de_programación)

http://www.oracle.com/technetwork/java/javase

https://docs.oracle.com/javase/6/docs/technotes/tools/

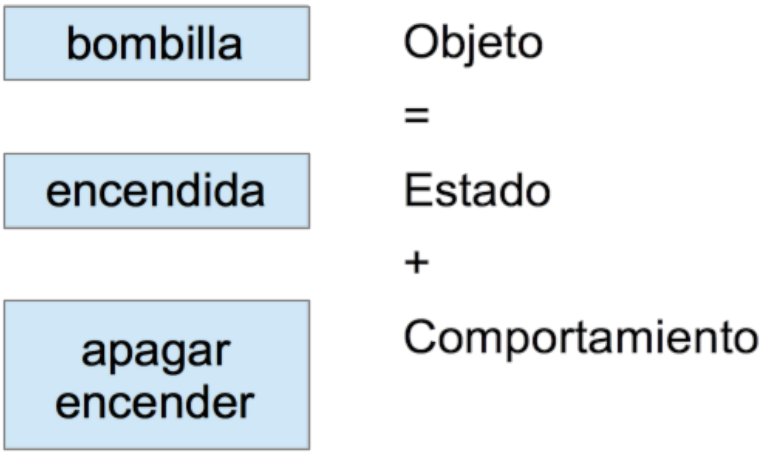
https://en.wikipedia.org/wiki/Java_version_history

Programación Orientada a Objetos

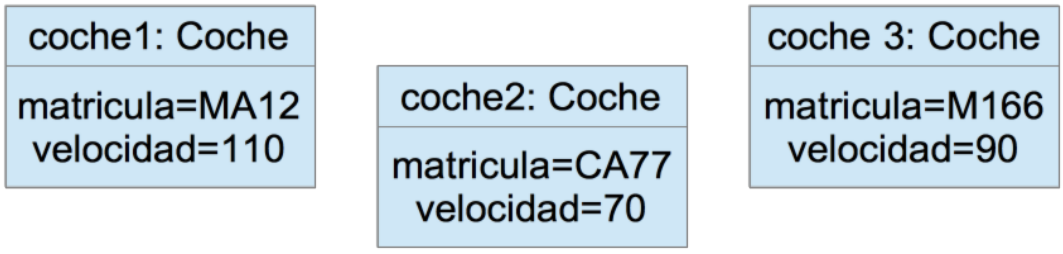
Clases y Objetos

La programación Orientada a Objetos, como su propio nombre indica está basada en la definición y construcción de objetos como una abstracción del mundo real.

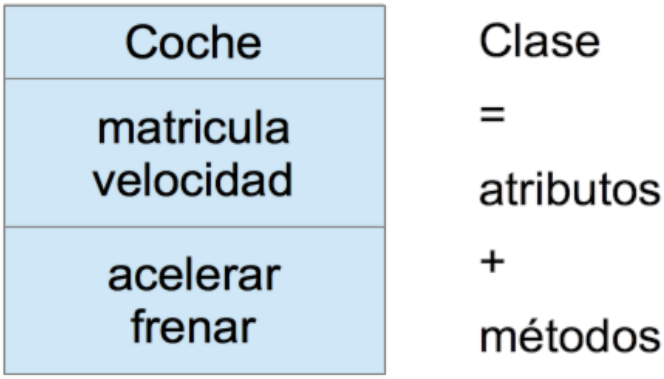
Cada objeto se puede considerar como una entidad compuesta por un estado interno actual y un comportamiento determinado por las operaciones que podemos realizar en cada momento con él.



Una de las principales características de la Programación Orientada a Objetos es el encapsulamiento, que oculta los detalles de implementación de los objetos. Como usuarios de la bombilla lo que nos interesa es lo que podemos hacer con la bombilla (encenderla o apagarla) o incluso su estado actual (encendida o no), pero no atendemos a por qué causa o tecnología interna la bombilla actúa de tal manera. Además el comportamiento interno de un objeto nunca debe ser modificado de forma directa, solamente a través de las operaciones que se permiten para el mismo.



En memoria durante la ejecución de la aplicación pueden existir múltiples objetos. Cada objeto representa una **instancia** (copia) distinta, con un estado interno posiblemente diferente al de otros objetos. Cada objeto se genera a partir de una plantilla común denominada **Clase**. Cada clase incluye la definición de los **atributos** que componen el estado de cada objeto, y el código de los **métodos** u operaciones que dispone la misma para poder modificar el estado interno de cada objeto.



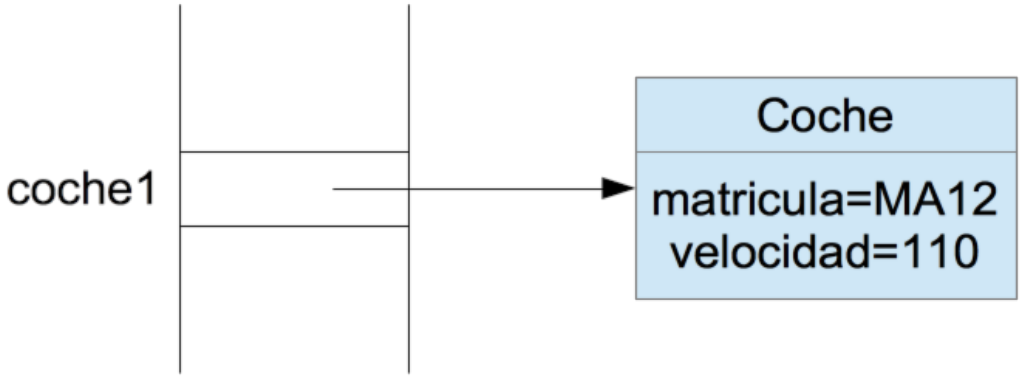
Cada instancia de la Clase dispone de una copia distinta de los atributos que determinan su estado, mientras que la definición de los métodos es compartida entre todos los objetos. De esta forma, existe un interfaz común para modificar el estado de cada objeto, cuyo comportamiento dependerá de su estado en cada momento. La velocidad final de un Coche que se frena dependerá de la velocidad que tenga en el momento determinado de invocar a esa operación.

Un hecho destacable en la Programación Orientada a Objetos es que las operaciones se invocan sobre instancias concretas, se trata de lanzar un mensaje al objeto para que éste modifique su estado: **coche1**.frenar().

Esto significa que el objeto que se frena es coche1 en función de su velocidad actual, el resto de los objetos se ve inalterado tras este paso de mensaje.

Gestión de memoria

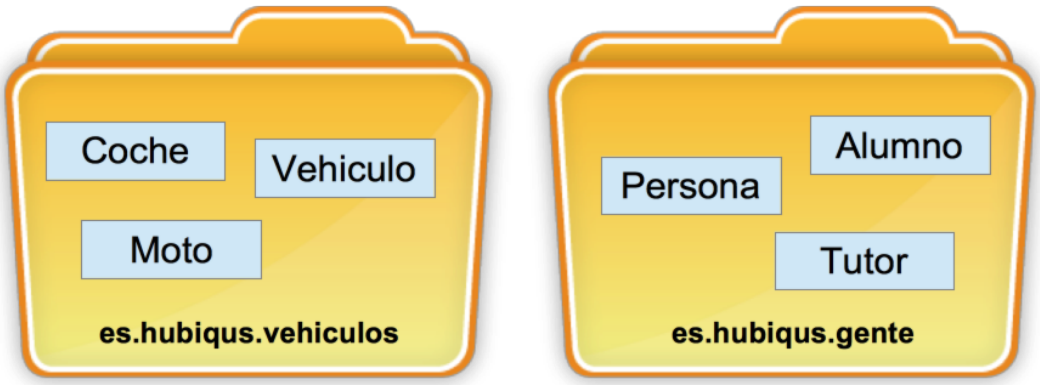
Para poder utilizar un objeto es necesario reservar memoria para el mismo a través del proceso de **instanciación** (creación de la instancia del objeto) o también denominado **construcción**, de forma que el sistema disponga de un emplazamiento físico para poder almacenar los valores de los atributos de dicho objeto.



De la misma forma, una vez el objeto no sea necesario se debe liberar de la memoria para evitar sobrepasar la ocupación máxima permitida, y problemas asociados.

Organización de clases

Habitualmente las clases que conforman un sistema Orientado a Objetos se suelen agrupar en unidades lógicas denominadas paquetes, que agrupan a los elementos que poseen un comportamiento, funcionalidad y características comunes.



Se puede pensar en un paquete como en una carpeta de la estructura de ficheros, donde se encontrarán físicamente los elementos que pertenecen al mismo. Los paquetes no solamente sirven para agrupar a las clases que se creen desde cero al construir el sistema, sino que es la organización habitual de las clases pertenecientes a las APIs propias del lenguaje empleado.

La forma de nombrar a los paquetes suele realizarse utilizando el dominio inverso de la organización que desarrolla el sistema.

Referencias

https://es.wikipedia.org/wiki/Programación_orientada_a_objetos

https://docs.oracle.com/javase/tutorial/java/javaOO/index.html

http://www.uml-diagrams.org/class-reference.html

https://docs.oracle.com/javase/tutorial/java/concepts/package.html

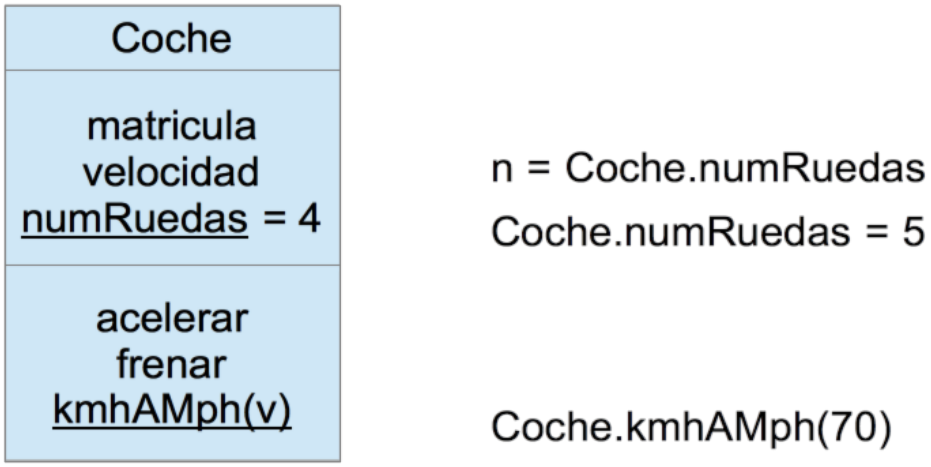
Referencias estáticas

Dentro de la definición de una clase pueden aparecer atributos y métodos estáticos o también denominados atributos y métodos de clase.

La definición de un miembro de clase se encuentra asociada a dicha clase, no a los objetos que se crean a partir de ella, es decir, no es necesario crear una instancia de la clase para poder acceder a sus miembros estáticos (de hecho el crear una instancia para acceder a cualquier elemento de este tipo se considera una mala práctica).

En la siguiente tabla se pueden observar las características principales que presentan este tipo de elementos:

Miembro estático	**Características**
Atributo	• Valor del atributo compartido por todos los objetos • Accedido a través del nombre de la clase • Todos los objetos comparten una misma localización de memoria para el valor del atributo • Si el valor del atributo se modifica, queda modificado para cualquier objeto que acceda al atributo posteriormente • Se suele emplear para valores o constantes compartidas
Método	• Definición del método compartida por todos los objetos • Accedido a través del nombre de la clase • El método no se aplica sobre ninguna instancia en concreto, recibe como parámetros los elementos sobre los que trabaja • Se suele emplear en métodos cuya naturaleza es independiente del estado del objeto en cuestión • También se suele emplear para acceder a los atributos estáticos



Existe una serie de reglas de aplicación para los miembros de una clase:

- Los métodos de instancia de una clase pueden acceder a las variables y otros métodos de instancia de la misma clase de forma directa, ya que una instancia siempre tiene acceso a sus miembros en tiempo de ejecución.

- Los métodos de instancia de una clase pueden acceder a las variables y métodos estáticos de la misma clase de forma directa, ya que se encuentran definidos dentro del cuerpo de la clase.
- Los métodos de clase pueden acceder a las variables y métodos estáticos de la misma clase de forma directa por el mismo motivo.
- Los métodos de clase NO pueden acceder a las variables y métodos de instancia de la propia clase de forma directa, ya que no representan una instancia de dicha clase.
- Los métodos de una clase para poder acceder a las variables y métodos de instancia de otra clase necesitan un objeto creado de dicha clase.
- Los métodos de una clase para poder acceder a las variables y métodos estáticos de otra clase necesitan anteponer el nombre de dicha clase.

Referencias

https://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html

Definición de métodos

El código que ejecuta la funcionalidad de las aplicaciones escritas en lenguajes Orientados a Objetos suele encontrarse incluido dentro de los métodos definidos en las clases.

Para determinar un método de forma completa es necesario especificar su **signatura**, compuesta por:

- Tipo de datos devuelto: indica el tipo de datos del resultado producido por el método. Es posible que un método no devuelva nada como resultado de su ejecución.
- Nombre: identificador del método.
- Argumentos: cada uno de los parámetros que recibe el método, además dentro de la lista de parámetros resulta significativo el número, orden y tipo de datos de dichos argumentos. Es posible que un método no reciba parámetros.

Identificadores

Un método tiene un nombre concreto a través del cual puede ser invocado en el código de la aplicación como llamada sobre un objeto o referencia estática de una clase. Los identificadores de los métodos suelen denominarse en letras minúsculas (**acelerar**), a menos que se trate de palabras compuestas donde se seguirá el criterio CamelCase (comenzar cada nueva palabra con letra mayúscula - **cambiarMarcha**).

Cada método debe poder ser invocado de forma inequívoca, para ello no es necesario que dos métodos distintos tengan nombres distintos bajo los siguientes supuestos:

- Los métodos de clases distintas pueden recibir el mismo nombre. La distinción se realizará por el tipo de la variable sobre la cual se invoca a dicho método (coche.acelerar() - particula.acelerar()).
- Los métodos de una misma clase pueden recibir el mismo nombre, la distinción en este caso es posible siempre que la lista de argumentos sea distinta (coche.acelerar() - coche.acelerar(10)). NO se consideran dos métodos distintos si teniendo el mismo nombre lo que distingue a ambos es que el tipo devuelto sea diferente.

El hecho de que dos o más métodos reciban el mismo nombre dentro de una clase se denomina en Programación Orientada a Objetos **sobrecarga** (del inglés overload). La sobrecarga tiene sentido siempre que los métodos que reciben el mismo nombre lo hagan porque desempeñan una tarea o funcionalidad similar, no simplemente por economía en el criterio de nombrado de los métodos.

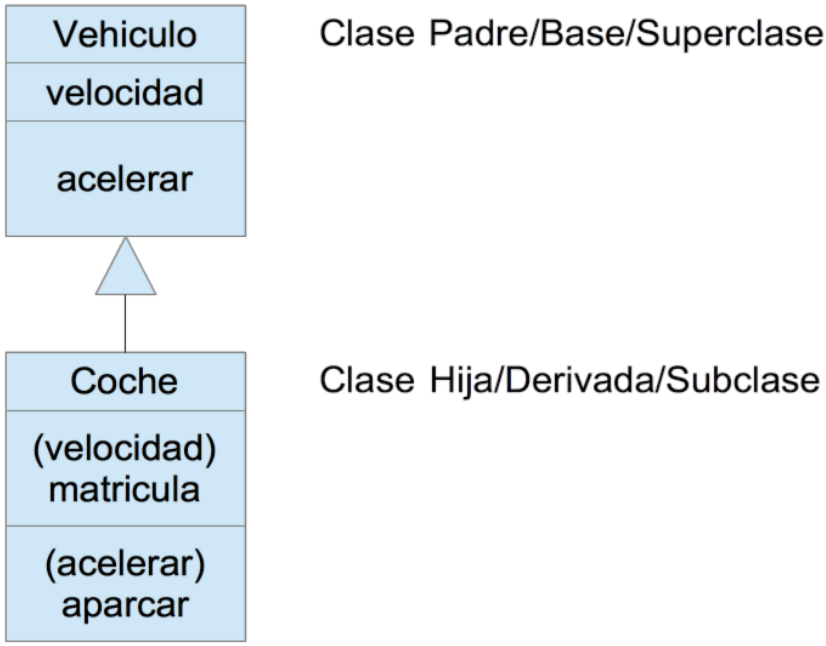
Referencias

https://docs.oracle.com/javase/tutorial/java/javaOO/methods.html

https://es.wikipedia.org/wiki/Método_(informática)

Herencia

La herencia es uno de los principales mecanismos proporcionados por la Programación Orientada a Objetos. Permite que una clase pueda ser definida a partir de otras clases existentes, lo cual representa una forma de reutilización de código. La herencia configura una relación de jerarquía entre las clases del sistema.



La herencia se puede establecer siempre que entre dos clases exista una relación del tipo es-un, donde la clase hija debe considerarse una especialización de la clase padre:

"Un Coche **es un** tipo especial de Vehículo".

Es un error muy habitual el intentar forzar la herencia entre dos clases para reutilizar código existente, nunca se debe realizar por el mero hecho de la reutilización del código o nombre común de los métodos si no se cumple la regla anterior:

"Una Partícula se puede acelerar, una Partícula **NO es un** Vehículo".

Tipos de herencia

Existen distintos tipos de relaciones de herencia entre clases:

Tipo	Características	Estructura
Simple	Una clase puede heredar de una sola clase	Árbol simple
Multinivel	Una clase hereda los elementos de sus antecesores	Árbol varios niveles
Múltiple	Una clase puede heredar de más de una clase	Grafo
Jeráquica	Una clase puede ser heredada por más de una clase	Árbol varias ramas

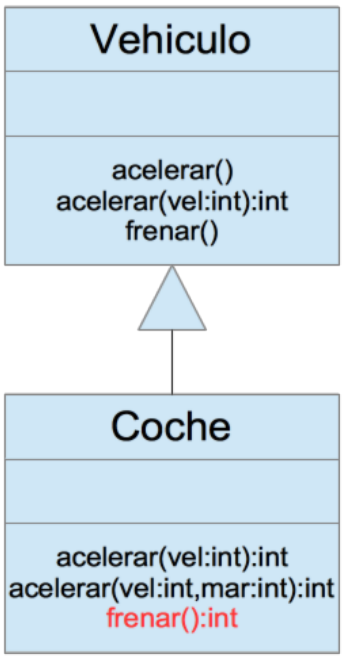
Sobrescritura

La sobrescritura (del inglés override) es la redefinición de un método heredado de la clase padre indicando una implementación específica para la clase hija y sus sucesores. Es decir, si en una clase hija no se especifica ninguna nueva versión para un método heredado, por defecto se utiliza la copia de la implementación de la clase padre, en caso contrario, la nueva implementación definida en la clase hija se utilizará.

Para que un método se considere sobrescrito, la signatura del mismo debe cumplir las siguientes reglas:

- El nombre del método de la clase hija es el mismo que en la clase padre.
- La lista de argumentos tiene que ser exactamente la misma (en número de parámetros, orden y tipos de datos de éstos).
- El tipo de datos que devuelve el método es el mismo (o compatible).

Si no se cumplen las reglas anteriores por completo entonces se tendrán en cuenta las reglas de sobrecarga para escritura de métodos con un mismo nombre dentro de la clase.



Referencias

https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html

https://es.wikipedia.org/wiki/Herencia_(informática)

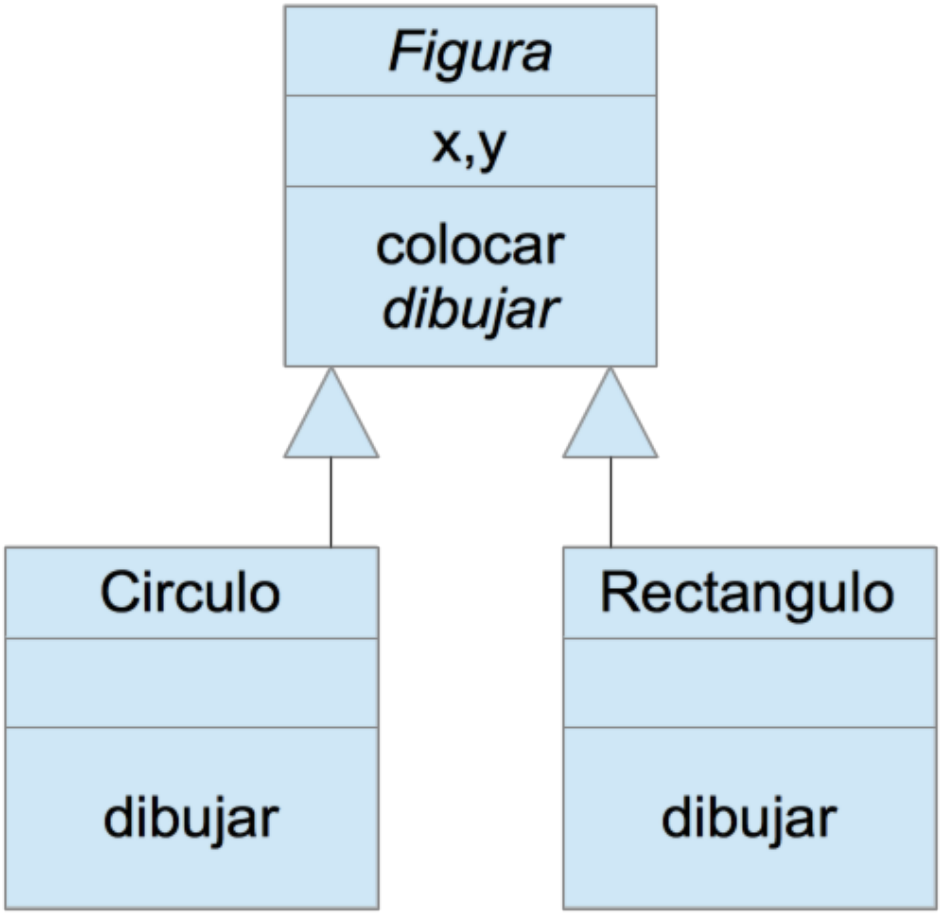
https://en.wikipedia.org/wiki/Method_overriding

Clases Abstractas

Una clase abstracta es principalmente aquella en la que alguno de sus métodos no se encuentra definido, es decir no dispone de una implementación asociada. Por este motivo no se pueden crear instancias de una clase abstracta.

Las clases que heredan de una clase abstracta están obligadas a definir una implementación de los métodos abstractos incluidos en la clase padre. La naturaleza de este tipo de métodos es obligar a disponer de un comportamiento determinado que deberán definir las clases sucesoras en la jerarquía pero sin conocer de antemano cuál será su implementación en detalle, ya que dependerá de cada caso concreto. Este hecho adquiere un mayor sentido en conjugación con el mecanismo de **Polimorfismo**.

Tanto el método sin cuerpo, como la clase se deben marcar como abstractas. Podríamos tener una clase abstracta sin ningún método abstracto en su definición.



Referencias

https://es.wikipedia.org/wiki/Tipo_abstracto

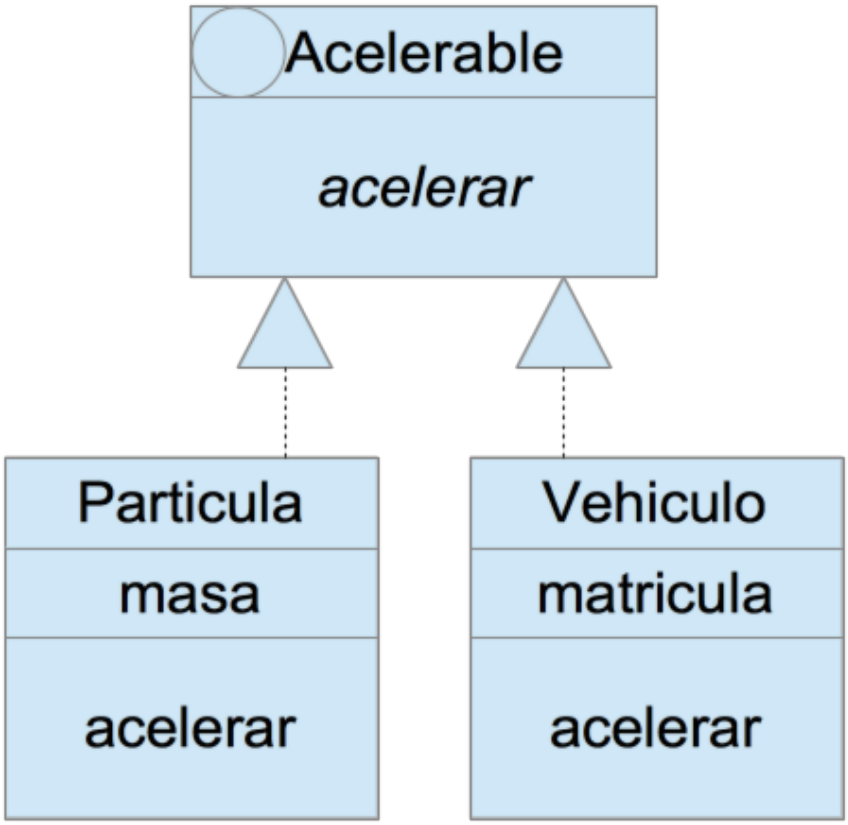
https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html

Interfaces

Se puede considerar a un interfaz como un elemento completamente abstracto, ya que ninguno de los métodos que posee dispone de implementación.

La naturaleza de un interfaz es la definición de comportamientos que deberán disponer las clases que lo implementen - la relación entre clases e interfaces es de implementación, no de herencia. Una clase que implementa a un interfaz está obligada a especificar el código de cada uno de los métodos de dicho interfaz, a menos que esta clase estuviera marcada como abstracta.

Tampoco se podrán crear instancias de los interfaces, en primer lugar por su naturaleza abstracta y además por no considerarse clases al uso.



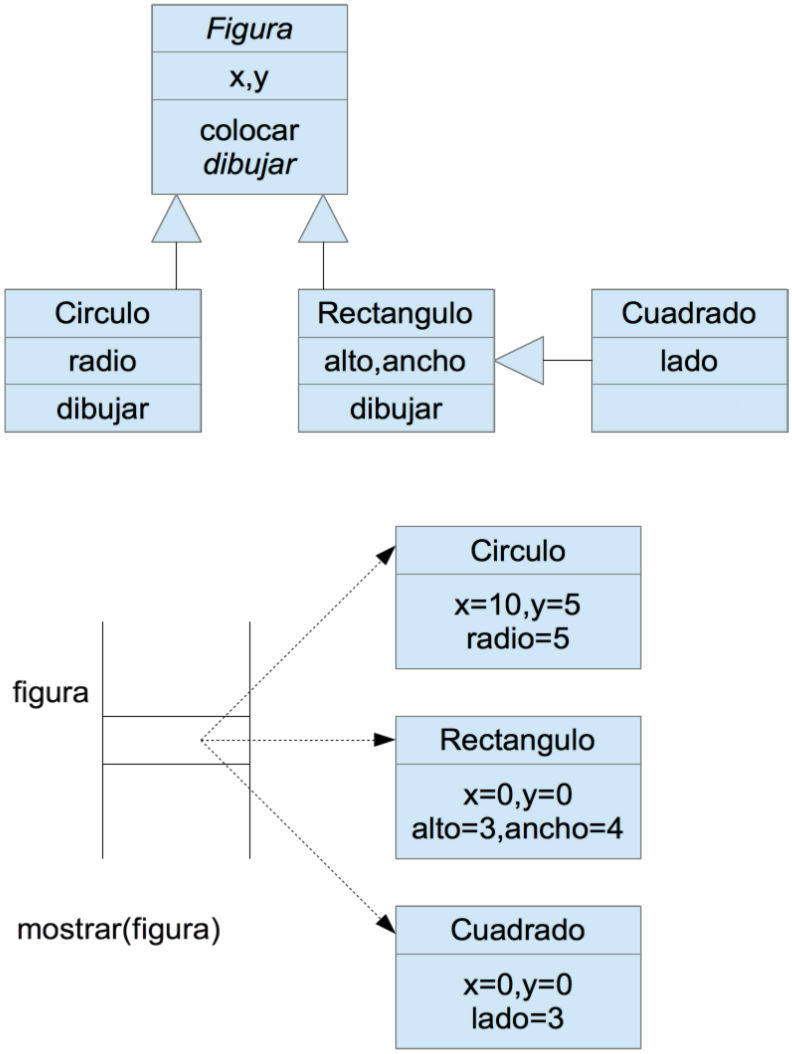
Referencias

https://docs.oracle.com/javase/tutorial/java/landl/createinterface.html

Polimorfismo

El Polimorfismo representa una de las mayores capacidades proporcionadas por la Programación Orientada a Objetos, ya que va a permitir la reutilización de métodos y estructuras a través de las relaciones de herencia e implementación de interfaces que existen entre los objetos del sistema.

Se puede definir el polimorfismo como la capacidad de una referencia (o variable) de comportarse de forma distinta en función de la instancia u objeto que aloja en cada momento en tiempo de ejecución.



En la figura anterior el método **mostrar** recibe como argumento un parámetro de tipo Figura. El método **mostrar** puede incluir dentro de su código una llamada al método **dibujar** (figura.dibujar()), debido a que la variable pertenece a la clase Figura y ésta dispone de ese método en su definición.

Gracias a la relación de herencia existente, el método **mostrar** podrá recibir como argumento tanto objetos de tipo Circulo como de tipo Rectangulo, incluso de Cuadrado (un Cuadrado es un tipo especial de Rectángulo). Dependiendo la instancia que se pase al método **mostrar** el resultado será distinto, ya que cada clase posee una implementación distinta de **dibujar**.

Como se puede observar el polimorfismo presenta dos comportamientos:

Comportamiento	Fase	Características
Estático	Compilación	• La invocación del miembro es correcta si la clase de la variable posee el atributo o método invocado • La invocación del miembro es correcta si la clase de la variable hereda una clase que lo posee • La invocación del miembro es correcta si la clase de la variable implementa un interfaz que lo posee
Dinámico	Ejecución	• La invocación del miembro se realizará sobre la versión disponible en el objeto referenciado • Si el objeto no dispone para el miembro de versión propia se asciende en la jerarquía hasta localizarlo

Una gran ventaja relativa a lo comentado del polimorfismo es la flexibilidad, ya que ofrece capacidad de reutilización del código en los métodos, manipulando objetos de diversas clases sin necesidad de conocer a qué tipo pertenecen de antemano.

En ocasiones también se considera un tipo especial del polimorfismo a la sobrecarga, que representa la ejecución de un método específico en tiempo de ejecución basándose en la signatura del mismo (en los parámetros que recibe).

Conversión de tipos

Existen dos tipos de conversiones de tipos que se pueden realizar gracias al polimorfismo:

- Conversiones implícitas: una conversión implícita se produce cuando a una variable de una clase de jerarquía superior se asigna un objeto de una clase inferior de la jerarquía. Es el mecanismo de conversión que hemos aplicado hasta el momento (Figura f = circulo).

- Conversiones explícitas (**casting**): una conversión explícita se hace necesaria cuando se quiere realizar un cambio del tipo de referencia a un objeto desde una variable de jerarquía inferior, lo cual no se permite de forma directa. Para ello habitualmente se indica entre paréntesis el tipo de conversión de tipo a realizar (Circulo c = (Circulo) f). La conversión que se realiza es exclusivamente del tipo de variable o referencia, nunca se realizará una transformación del objeto interno por lo que solamente estará permitido para variables que realmente encierren un objeto compatible con la conversión, en caso contrario se producirá un error en tiempo de ejecución. Un casting explícito se puede establecer siempre que sean elementos relacionados en la jerarquía.

Referencias

https://es.wikipedia.org/wiki/Polimorfismo_(informática)

https://docs.oracle.com/javase/tutorial/java/landl/polymorphism.html

https://en.wikipedia.org/wiki/Type_conversion

Diseño Orientado a Objetos

Un sistema Orientado a Objetos se puede considerar como una abstracción del mundo real representada por un conjunto de instancias de clases que intercambian mensajes entre ellas. Antes de proceder a la programación de estos objetos y clases es necesario realizar un proceso previo de diseño de las clases y las relaciones que existen entre ellas, de forma que el sistema se adecue al modelo real y además lo haga de la forma más óptima posible cumpliendo los principios de encapsulamiento, abstracción y reutilización entre otros.

Como norma general, cuando se realiza un diseño Orientado a Objetos sobre un enunciado inicial, los nombres o grupos de nombres que aparecen en el mismo definen las clases o los atributos de éstas, mientras que las acciones determinan posibles métodos:

"Para **acceder** al sistema, el **usuario** deberá escribir su cuenta de **mail** y una **contraseña**".

Para definir las relaciones entre clases es necesario comprender cómo se asocian en el entorno real las entidades del modelo del dominio. La herencia se puede establecer siempre que entre dos clases exista una relación del tipo es-un, donde la clase hija debe considerarse una especialización de la clase padre:

"Un Coche **es un** tipo especial de Vehículo".

Es un error muy habitual el intentar forzar la herencia entre dos clases para reutilizar código existente u obtener beneficios del polimorfismo. Nunca se debe establecer herencia por el mero hecho de la reutilización del código, nombre común de los métodos o paso de parámetros más generales si no se cumple la regla anterior:

"Una Partícula se puede acelerar, una Partícula **NO es un** Vehículo".

Se denomina **composición** al hecho de que una clase posea atributos que son referencias a otras clases del sistema. En este caso la relación entre dichas clases es una relación **tiene-un**:

"Un Coche **tiene un** Pasajero".

Como norma general, si entre dos clases no existe una relación es-un (herencia) la opción más válida será la de composición.

Por último, la relación básica que existe entre dos clases es la de **utilización**, ya que en el código que determina el comportamiento de un método será frecuente que se haga referencia a más de una clase del sistema sin que necesariamente exista ninguna de las relaciones explicadas anteriormente, simplemente como variables necesarias para cumplimentar la tarea.

El lenguaje Java

Variables y Tipos de datos

En Programación Orientada a Objetos, como en el resto de paradigmas, una variable es un identificador que almacena un valor determinado en tiempo de ejecución. Las variables son necesarias como elemento de almacenamiento temporal de resultados dentro del código de los métodos (variables locales), pero también se consideran variables los parámetros que reciben dichos métodos, o los atributos que se encuentran dentro de una clase.

En Java existen dos clases de variables en función del tipo de dato que almacenan: primitivas y referencias a objetos.

Para poder emplear una variable en Java es necesario declararla previamente, indicando cuál es su tipo de datos y su identificador. A menudo se realizan tanto la declaración de la variable como su inicialización en la misma **expresión**, es decir la asignación del primer valor que tomará dicha variable:

```
int x;
x = 5;
int y = 8;
Coche coche = new Coche();
```

Identificadores en Java

En Java los identificadores de variables válidos deben cumplir con las siguientes reglas:

- No se trata de palabras reservadas del lenguaje.
- Comienzan por letra, _ o $.
- No contienen espacios.
- No contienen caracteres especiales, tales como operadores o signos de puntuación.

Además de las reglas propias del lenguaje se debe tener en cuenta la nomenclatura CamelCase para el nombrado de identificadores, y que además se debe tratar de nombres explicativos, concretos y concisos.

Tipos de datos primitivos

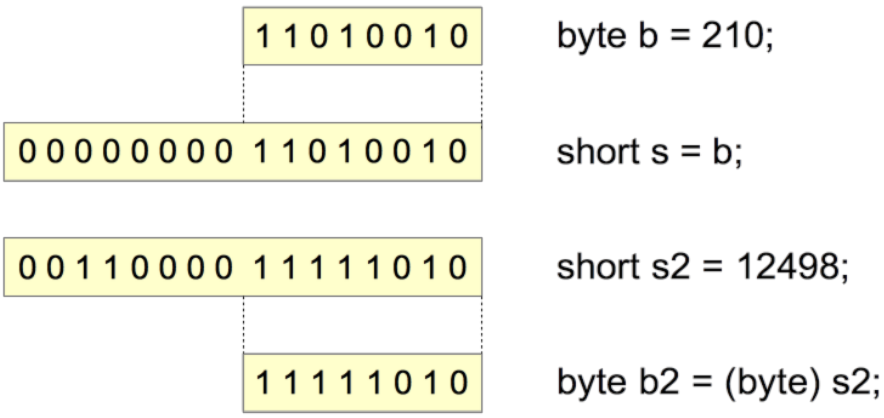
Los tipos primitivos en Java representan elementos básicos para las operaciones que se ejecutan en los métodos, o también se emplean como bloques elementales en la definición de clases. En un lenguaje Orientado a Objetos puro cualquier elemento que se maneje dentro de la programación debe ser una instancia de clase, los tipos primitivos no son instancias ni se gestionan como tales en memoria, de forma que representan una manera más eficiente de conseguir representar valores básicos y operaciones entre ellos.

Los tipos de datos primitivos en Java son los siguientes:

Tipo	Descripción	Ejemplos de Literales
boolean	Valores lógicos	true false
char	Caracteres simples formato UNICODE (16 bits)	'a' '\n' '\u03A9'
byte	Enteros de 8 bits	7
short	Enteros de 16 bits	17
int	Enteros de 32 bits	27 0x1b 0b11011
long	Enteros de 64 bits	37L
float	Decimales de 32 bits	47.0F
double	Decimales de 64 bits	57.0 57.0D 0.67e2

No es posible realizar conversiones en Java entre números y valores booleanos o viceversa, pero sí resulta posible realizarlo entre las variables que representan valores numéricos. Si la asignación es hacia variables que albergan una cantidad mayor de bits que el valor original la conversión es siempre posible de forma implícita manteniendo el valor original:

```
int x = 5;
long l1 = x;
long l2 = 7;
float f = x;
char c = 'a';
x = c;
```



En el caso de realizar asignaciones hacia variables con una capacidad menor en memoria será necesario realizar una conversión explicita (casting). En este caso el valor resultante de la asignación no puede determinarse de forma sencilla ya que Java recortará el contenido original de la variable para ajustarlo al tamaño.

La representación binaria de números negativos en Java es en complemento a 2.

Ámbito y visibilidad de variables

El ámbito de una variable es el fragmento de código en el que ésta tiene vigencia, es decir, una vez declarada la variable en qué líneas del código que se escriba posteriormente va a poder ser empleado. Las reglas del ámbito de una variable son las siguientes:

Variable	Inicialización	Ámbito
Atributo	Automática (0, false, null)	• Toda la clase • Clases hijas en función de visibilidad
Parámetro	Llamada	• Método actual
Local	Manual (obligatoria)	• Bloque de código ({ })

En Java existen 4 opciones para establecer la visibilidad de miembros de una clase, que determinan el tipo de acceso que se puede realizar sobre un atributo o método y cuyo significado se encuentra relacionado con el principio de encapsulamiento. Ordenados de menor a mayor visibilidad:

- **private**: un miembro privado solamente dispone de visibilidad dentro de la clase en la cual se define.
- (package): es la visibilidad por defecto, es decir, si no se determina ningún modificador de acceso para un miembro éste dispone de visibilidad de este tipo, solamente disponible en la clase actual y clases del mismo paquete.
- **protected**: visibilidad en la clase actual y clases que hereden de ésta, además de las clases del mismo paquete.
- **public**: visibilidad sin restricciones desde cualquier clase.

A efectos prácticos, los dos modificadores más empleados serán:

- **private**: habitualmente todos los atributos de una clase se marcan como private para proteger la estructura interna del objeto frente a accesos externos, nunca se debe modificar un atributo si no es a través de los métodos que definen el comportamiento de la clase. Además se establecen como private los métodos auxiliares cuya implementación también determina el funcionamiento interno de la clase, el cual no debe ser expuesto.

- **public**: los métodos que determinan el interfaz de comunicación con la clase deben ser públicos para permitir su invocación desde cualquier código que haga uso de un objeto de dicha clase.

Referencias

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Operadores

Los operadores son símbolos que van a permitir realizar determinadas funciones y operaciones entre las variables y valores que se utilizan en la aplicación.

Existen varios tipos de operadores en función del tipo de variable y resultado obtenido, los cuales aparecen agrupados en la siguiente tabla por orden de precedencia (orden de aplicación en las operaciones):

Tipo	Grupo	Operadores
Aritméticos	Postfijo	++ --
	Unarios	++ -- + -
	Unarios nivel bit	~
	Multiplicación	* / %
	Adición	+ -
	Desplazamiento	<< >> >>>
	Binarios nivel bit	& ^ |
Lógicos	Unarios	!
	Relacionales	< > <= >= instanceof
	Igualdad	== !=
	Binarios	&& & || |
	Ternarios	? :
Asignación	Asignación	= += -= *= %= &= ^= |= <<= >>= >>>=

La precedencia de una operación siempre se puede modificar utilizando paréntesis:

```
int x = -5+3*8;
int y = -(5+3)*8;
```

Referencias

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/opsummary.html

Estructuras de Control

Cada método en Java está determinado por una serie de expresiones o sentencias que se ejecutan de forma secuencial. Las estructuras de control modifican este flujo de ejecución por defecto a través de condiciones o iteraciones.

Sentencias condicionales

Se trata de sentencias basadas en condiciones booleanas, que al evaluarse ejecutarán ciertos bloques de código en función del cumplimiento o no de las mismas. Por tanto las sentencias condicionales sirven para establecer flujos alternativos de ejecución.

Las sentencias condicionales se pueden anidar entre ellas o con otro tipo de sentencia.

Sentencia	Sintaxis	Características

if	**if** (condicion) {
	…
	}
		• Se ejecutan las sentencias dentro de if si la condición se evalúa a true
		• En otro caso no se ejecuta nada
		• Si solamente contiene una sentencia interna las llaves no son obligatorias, pero sí recomendadas

if…else	**if** (condicion) {
	…
	} **else** {
	…
	}
		• Similar al anterior, pero en caso de que la condición se evalúe a false se ejecutan las sentencias dentro de else

if…else if	**if** (condicion1) {
	…
	} **else if** (condicion2) {
	…
	} …
		• Similar al anterior, pero en caso de que una condición se evalúe a false se evalúa la siguiente sucesivamente
		• Puede contener un else final

switch	**switch** (expresion){
	case valor1: … **break**;
	case valor2: … **break**;
	case …
	default: …
	}
		• Se evalúa la expresión y se ejecutan las sentencias del primer case cuyo valor coincide
		• En caso de no coincidir con ningún case se ejecutan las sentencias de default (opcional)
		• Se ejecutan todas las sentencias hasta encontrar un break o final del bloque
		• No pueden existir 2 case con el mismo valor, y su valor debe ser de tipo int, String o enum, coincidente con el tipo de datos de la expresión
		• El orden de los case y default no es significativo

Iteraciones

Las sentencias de iteración o bucles se emplean para ejecutar instrucciones de forma repetida mientras que se cumpla la condición de control.

Dentro de los bucles se pueden encontrar en Java dos expresiones especiales, se trata de prácticas de programación poco recomendadas ya que siempre resulta más elegante la salida del bucle a través de la condición de control establecida:

- **continue**: si la ejecución de la aplicación encuentra una instrucción continue dentro del bucle, saltará automáticamente a evaluar de nuevo la condición de salida del mismo (también la expresión de incremento en el caso de for).
- **break**: si la ejecución de la aplicación encuentra una instrucción break dentro del bucle se cancelará directamente la ejecución del mismo.

Las dos instrucciones anteriores pueden emplearse con etiquetas en el caso de anidamiento de bucles para especificar cuál de ellos es al que afecta la ejecución de dicha etiqueta.

```
salida:
      for (int i=0; i<10; i++){
            for (int j=0; j<10; j++){
                  System.out.println(i+j);
                  if (i+j >= 5){
                        break salida;
                  }
            }
      }
```

Los bucles también se pueden anidar entre ellos o con sentencias condicionales.

Sentencia	Sintaxis	Características

for	**for** (inic;condicion;expresion) {
	…
	}
		• Se ejecuta la inicialización
		• Se ejecutan las sentencias dentro de for si la condición se evalúa a true
		• Se ejecuta la expresión y se vuelve a evaluar la condición
		• Si sólo contiene una sentencia las llaves no son obligatorias, pero sí recomendadas

for mejorado	**for** (Clase tmp: Iterable){
	…
	}
		• Se ejecuta para cada elemento del iterable (arrays, colecciones)

while	**while** (condicion) {
	…
	}
		• Se ejecutan las sentencias dentro de while si la condición se evalúa a true
		• La inicialización y expresión de incremento son responsabilidad del programador

do while	**do** { … } **while** (condicion);	• Similar al anterior pero la condición se evalúa al final • Asegura al menos una ejecución de las sentencias

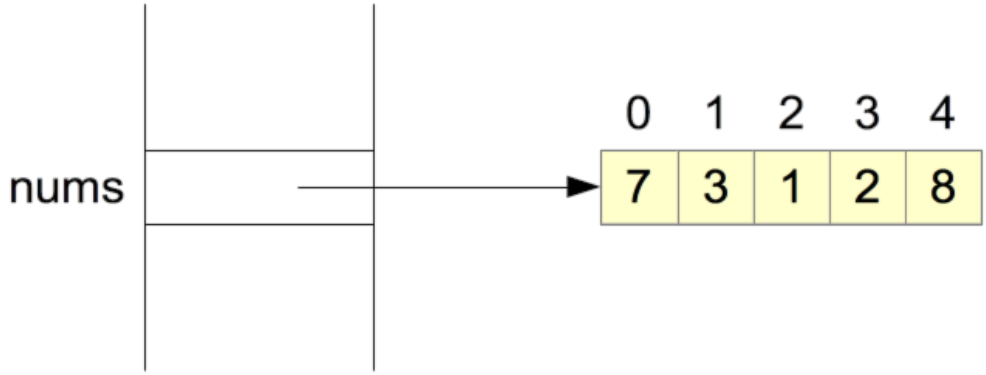
Referencias

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/flow.html

https://blogs.oracle.com/CoreJavaTechTips/entry/using_enhanced_for_loops_with

Arrays

Un array es un objeto que actúa como contenedor para un número fijo predeterminado de valores que pertenecen a un tipo concreto. Los valores de un array se encuentran indexados.

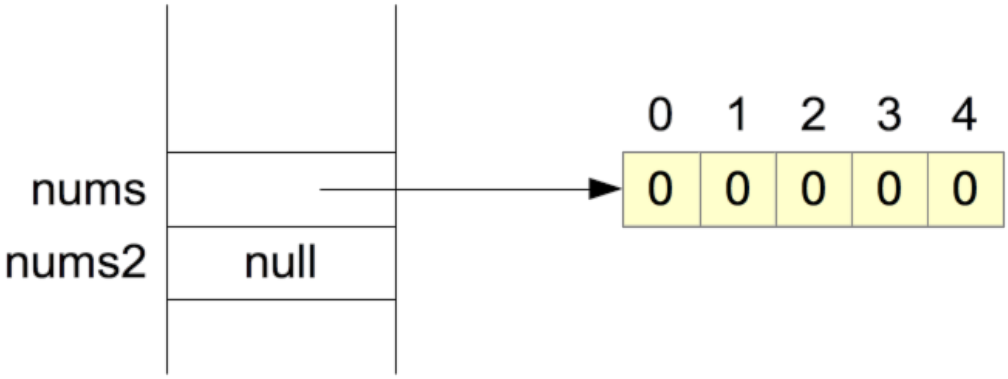


La declaración de una variable para un array por tanto debe determinar cuál es el tipo de datos base para el mismo, añadiendo [] para indicar a Java que se trata de una estructura como la que hemos comentado:

```
int[] nums;
int nums2[];
```

Para inicializarlo es necesario indicar el número de elementos que va a contener. De esta forma se realiza la creación y reserva de memoria para cada uno de los elementos del array, inicializando cada posición al valor por defecto en función del tipo de datos base. No se puede emplear un array sin crearlo, en caso contrario se producirá un error en tiempo de ejecución:

```
nums = new int[5];
```



Para acceder a una posición determinada del array se emplea su índice. Cabe destacar que los índices comienzan en la posición 0 y llegarán hasta el tamaño del array -1. Si se accede a una posición no válida del array también se producirá un error en tiempo de ejecución, circunstancia con la cual se debe tener precaución a la hora de iterar sobre ellos en bucles:

```
int[] nums = new int[5];
nums[3] = 7;
nums[5] = 16; //Error en tiempo de ejecución

for (int i=0; i<nums.length; i++){
      System.out.println(nums[i]);
}
```

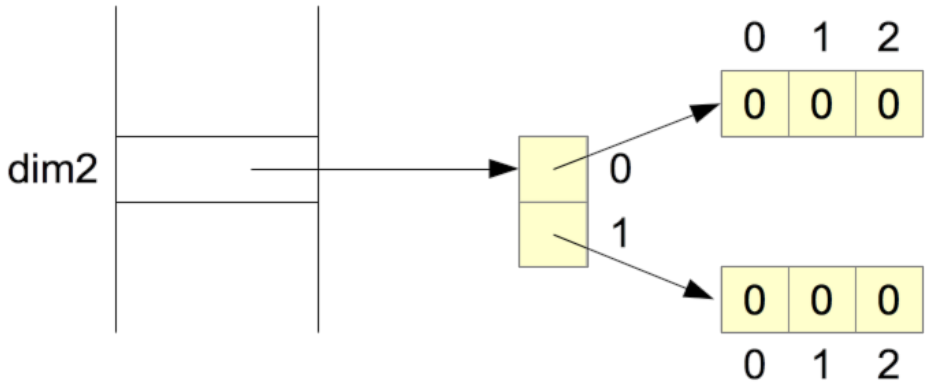
Los objetos de tipo array incluyen un atributo **length** que indica el número de elementos del mismo. Un array en Java podría tener longitud 0.

Un array se puede inicializar directamente con los valores que ocuparán cada una de sus posiciones, en este caso no es necesario establecer el tamaño del mismo ya que directamente se obtiene del número de valores indicados:

```
int[] nums = {7, 3, 1, 2, 8};
```

Por último indicar que los arrays pueden contener n dimensiones en Java:

```
int[][] dim2 = new int[2][3]; //int[] dim2[]
int[][] dim2_1 = {{1, 2, 3}, {4, 5, 6}};
```

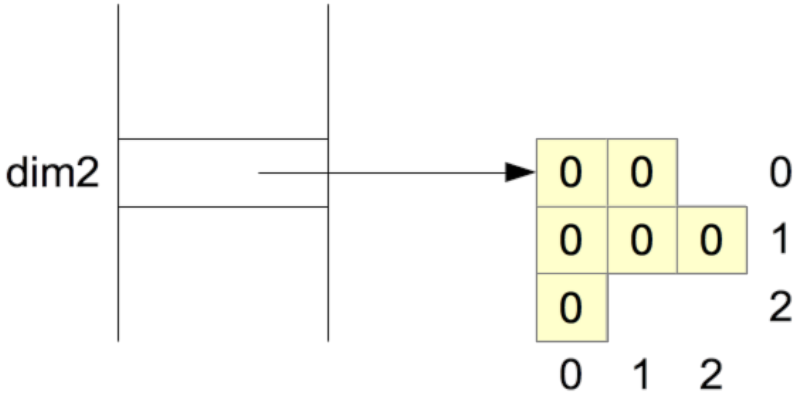


En el caso anterior se definen matrices de 2 dimensiones, creando un espacio de memoria de 2 filas y 3 columnas para cada una de las filas. En el siguiente ejemplo se genera una estructura con 3 dimensiones:

```
int[][][] dim3 = new int[2][3][3];
```

Los arrays no tienen por qué definir dimensiones con tamaño homogéneo, es decir cada una de las posiciones del array podría configurar una longitud distinta:

```
int[][] dim2 = new int[3][];

dim2[0] = new int[2];
dim2[1] = new int[3];
dim2[2] = new int[1];
```



Referencias

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html

Programación Orientada a Objetos con Java

Clases y Objetos

Cada clase Java se define habitualmente en un fichero con extensión .java con el mismo nombre de la clase. Las clases Java poseen una estructura determinada, donde el orden de las secciones opcionales **package** e **import** es significativo:

```
package es.hubiqus.gente;

import java.util.Date;

public class Persona {
	//Atributos

	//Métodos
}
```

En el momento que una clase se encuentra definida en el sistema se puede hacer uso de ella para la declaración de variables que contendrán referencias a objetos de la misma. La creación de una instancia de la clase es necesaria para poder invocar métodos sobre el objeto:

```
Persona persona = new Persona();
persona.comer();
```

Como todas las variables en Java, las referencias a objetos también necesitan ser inicializadas. En el caso de variables locales como en el ejemplo anterior la inicialización se hace obligatoria, en caso contrario se produciría un error de compilación. Es necesario tener precauciones cuando las variables se refieren a parámetros recibidos en un método y sobre todo cuando se refiere a atributos de una clase, ya que en este caso la inicialización por defecto se realiza al valor null, e intentar realizar cualquier tipo de acceso sobre una variable nula producirá un error en tiempo de ejecución.

```
public class Coche {

	private Motor motor;
	…

	public void arrancar(){
		motor.iniciar();
		//NullPointerException si motor no ha sido inicializado
	}

}
```

Constructores

Un constructor es un tipo especial de método empleado en la creación de instancias. Cuando el constructor es invocado se inicializa el proceso de reserva de memoria. Los constructores permiten la creación de objetos de una clase empleando la palabra reservada **new**. No es obligatorio indicar un constructor para todas las clases ya que Java introduce un constructor por defecto, pero en caso de que exista un constructor definido será obligatorio emplearlo para la creación de instancias.

Los constructores son métodos cuyo nombre es el mismo nombre de la clase y no pueden devolver nada, ni siquiera **void**. Se pueden sobrecargar, es decir, se pueden tener varios constructores en la misma clase siempre que su lista de argumentos difiera de unos a otros.

La naturaleza de un constructor debe ser la de configurar los atributos de la clase con valores iniciales por defecto o valores recibidos a través de parámetros.

```
public class Persona {

	private String nombre;
	private int edad;

	public Persona(String n, int e){
		nombre = n;
		edad = e;
	}

}

public class Coche {

	public void acelerar(){}

}

Persona persona = new Persona("Ana", 86);

//Coche no define constructor
Coche coche = new Coche();
```

El constructor por defecto posee las siguientes características:

- No recibe argumentos (parámetros).
- No ejecuta ningún código, de forma que los atributos quedarán inicializados con sus valores por defecto.
- Java introduce el constructor por defecto exclusivamente si no se ha definido otro constructor para la clase independientemente del número de argumentos de éste.

```
public class Persona {

	private String nombre;
	private int edad;

	public Persona(){
		nombre = "Sin nombre";
		edad = 0;
	}

	public Persona(String n, int e){
		nombre = n;
		edad = e;
	}

}
```

En el ejemplo anterior existen dos constructores para la clase, un constructor sin argumentos (no es el constructor por defecto de Java) y un constructor que recibe nombre y edad de la persona.

Cabe destacar que los constructores no se ven afectados por el mecanismo de herencia, es decir, los constructores ni se heredan ni se pueden sobrescribir.

Gestión de memoria

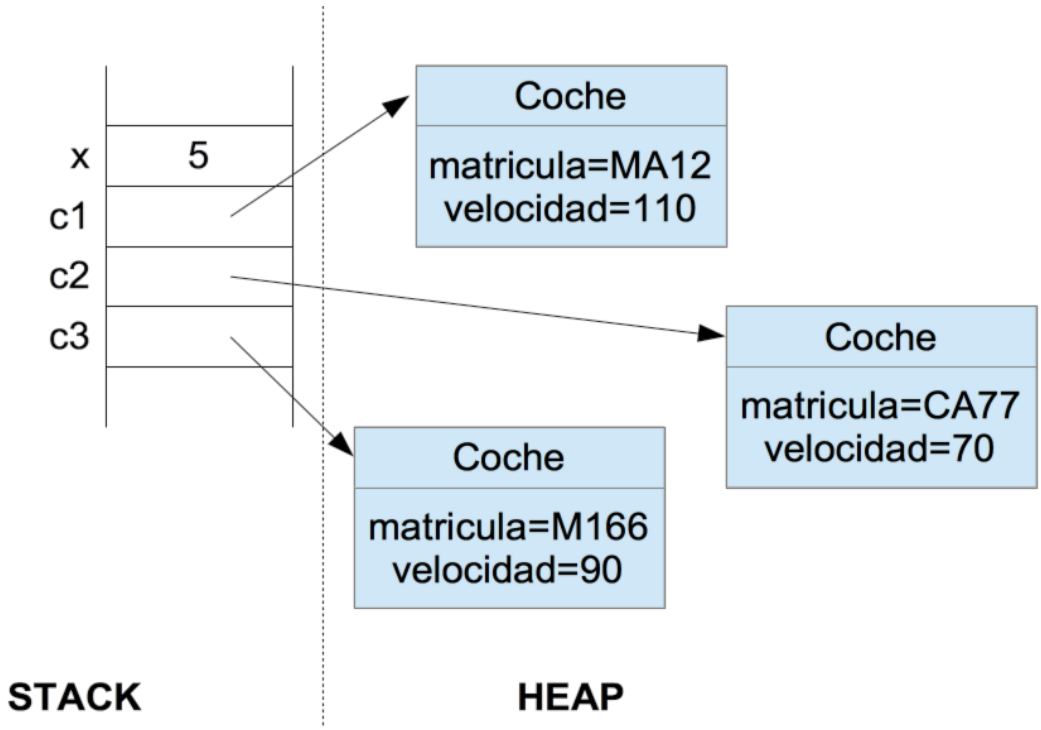
En cada ocasión en la que se genera una nueva instancia de cualquier objeto se realiza un proceso de reserva de memoria necesaria para la ocupación por parte de los valores de los atributos que componen a dicho objeto, incluyendo los atributos definidos por esa clase y por todos los antecesores a ésta en función de la jerarquía.

Todas las instancias de objetos existentes en memoria en un momento determinado componen lo que se denomina el espacio Heap de Java. El resto de elementos de la ejecución del JRE (variables y llamadas a métodos) se encuentra en un espacio Stack. Ambos espacios ocupan un tamaño inicial fijo y crecen en sentido contrapuesto de forma que si alguno de ellos llega a invadir al otro espacio, en el sistema ocurrirá un error de memoria. El tamaño de memoria que ocupa en ejecución la máquina virtual en su totalidad, así como cada uno de los espacios mencionados depende del sistema en el cual se ejecute, siendo además configurable a través de parámetros de inicialización.

Se pueden producir problemas de memoria por tanto si ocurre alguno de estos eventos:

- El número de instancias de objetos existentes en memoria alcanza el límite del espacio Heap.
- El número de llamadas a métodos o de variables en uso es tan grande que alcanza el límite del espacio Stack. Precaución en iteraciones o llamadas recursivas.

- Otros problemas derivados como el desborde del espacio para referencias estáticas de Java (espacio PermGem).



La gestión de memoria en Java es automática, de la misma forma que no es necesario preocuparse sobre la cantidad de memoria reservada para cada objeto, tampoco habrá que destruirla de forma manual cuando la tarea para la cual se han creado los objetos ya haya concluido. En Java existe un mecanismo de recolección de basura (**Garbage Collector**) encargado de liberar automáticamente la memoria que ya no es necesaria para la aplicación.

El recolector de basura es un proceso que se inicia de forma automática en el sistema a intervalos indeterminados, y que no puede ser invocado de forma manual. De cualquier forma existen llamadas del sistema que pueden levantar un aviso sobre la necesidad de recolección de basura tras la ejecución de código muy intensivo en el consumo de memoria, que no lo activan directamente pero sugieren a la máquina virtual que lance el proceso de limpieza cuando sea posible.

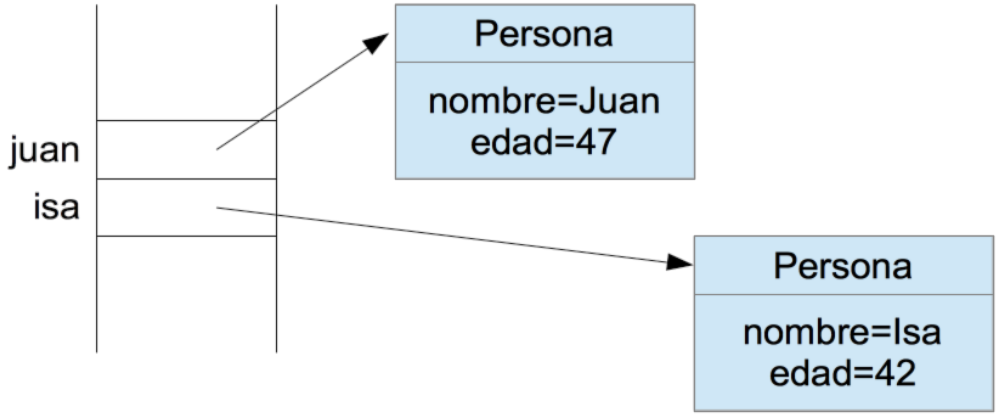
```
System.gc();
Runtime.getRuntime().gc();
```

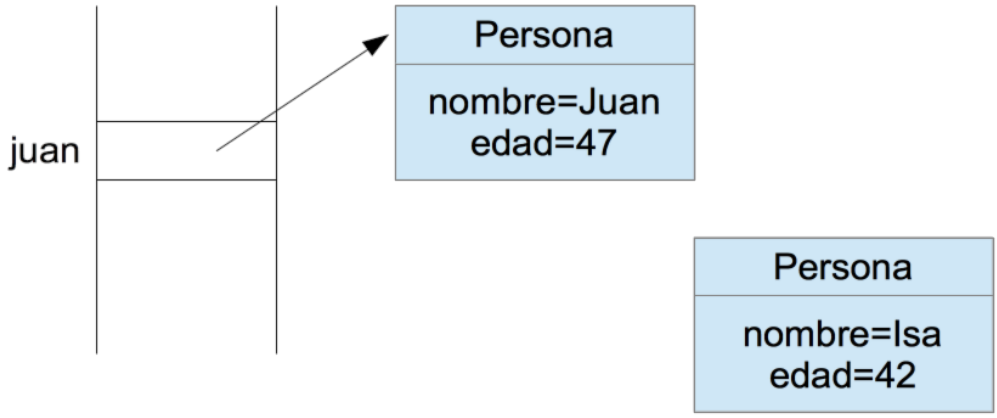
Un objeto es susceptible de ser recolectado por el Garbage Collector si ninguna variable se encuentra haciendo referencia al mismo en un momento determinado. Hay que tener en cuenta que cuando un bloque donde una variable ha sido declarada concluye su ejecución, las variables se eliminan de forma que las referencias a

objetos se pierden y éstos quedan en memoria ocupando el espacio original en espera de ser liberados.

```
28  Persona juan = new Persona("Juan", 47);
29  if (fiesta){
30     Persona isa = new Persona("Isa", 42);
31     juan.bailar(isa);
32  }
```

Después de ejecutar la línea 31, la variable isa ya no se emplea más y es eliminada, por lo que el objeto al que apuntaba originalmente sería recolectable. Una vez que concluya la ejecución del bloque de código completo ocurrirá lo mismo con el objeto juan.





Las clases en Java pueden definir un método **finalize** para la liberación de recursos justo antes de que los objetos sean eliminados de la memoria. Su finalidad no es la de destruir memoria manualmente, es la de ser invocado por el recolector de basura antes de eliminar al objeto de forma definitiva, es decir, es el sistema el que gestiona la ejecución de este método.

Variables y Métodos

Las variables en Java pueden ser globales a una clase (atributos) o locales a un método o bloque de código. Los atributos se definen de la siguiente forma:

- Modificador de acceso: los atributos de una clase deben indicar un modificador de acceso, habitualmente se marcan como **private** para preservar el principio de encapsulamiento. No indicar un modificador de acceso define el nivel de visibilidad como package.
- Otro tipo de modificadores: opcionalmente **static** o **final.**
- Tipo de datos de la variable: tipo básico o referencia de objeto.
- Identificador: etiqueta que delimita el uso y carácter de la variable de forma concreta y concisa, siguiendo las convenciones de nombrado ya comentadas.
- Inicialización: en el caso de los atributos la inicialización es opcional, ya que Java establece un valor por defecto para éstos (0 en caso de tipos numéricos, false para atributos de tipo **boolean** y **null** para cualquier tipo de referencia).

```
private String nombre;
private int edad;
```

Las variables locales no definen modificadores de acceso, ni de otro tipo. Su inicialización es obligatoria por lo que suele realizarse directamente en la misma línea donde se declaran:

```
boolean fiesta = false;
Persona juan = new Persona("Juan", 47);
```

La signatura de un método en Java se encuentra definida por los siguientes elementos:

- Modificador de acceso: los métodos de utilidad de la clase se deben marcar como **public**, mientras que los métodos auxiliares como **private**. No indicar un modificador de acceso define el nivel de visibilidad como package.
- Otro tipo de modificadores: opcionalmente **static**, **final** o **abstract**.
- Tipo devuelto: es obligatorio definir el tipo que devuelve un método, **void** para el caso que no devuelva nada. En caso distinto a void el método siempre tiene que concluir con una sentencia **return**. Los constructores no establecen tipo devuelto.
- Nombre del método: los identificadores de métodos deben seguir las mismas convenciones que los identificadores de variables.
- Lista de argumentos: parámetros que recibe el método, cada uno de ellos definido por su tipo de datos Java y su identificador. Es posible que un método no reciba parámetros, o que reciba número indeterminado de parámetros (…).

```
public int getEdad() {
	return edad;
}


public void setEdad(int edad) {
	this.edad = edad;
}
```

Los métodos que aparecen en el ejemplo anterior pertenecen a un tipo de método especial en Java empleado para el acceso a los atributos de forma estándar, sin necesidad de recurrir a otros métodos del interfaz de la clase o a constructores. Su nomenclatura sigue siempre el mismo criterio, anteponiendo el prefijo get/set al nombre del propio atributo, y por ello reciben el nombre de **getters** y **setters**.

La variable this

La variable **this** representa a la instancia actual del objeto cuando se encuentra en ejecución, y se puede emplear dentro de una clase para diversos usos:

- Acceso a métodos: en general toda llamada a un método Java (que no se refiera a una referencia estática) debe estar referenciada sobre un objeto, de esta forma se cumple con el modelo teórico de paso de mensajes. De la misma forma, cuando un método pertenece a la propia clase, el objeto sobre el cual se realiza esta llamada es this. En este caso la utilización de la variable this es opcional, ya que Java entiende que cualquier llamada a método sin referencia de objeto se refiere a la instancia actual.

```
public void arrancar(){
	motor.iniciar();   //iniciar es método public de Motor
	this.setMarcha(1); //setMarcha método de Coche, this opcional
	setVelocidad(0);   //this.setVelocidad(0)
}
```

- Acceso a atributos: el uso de this para el acceso a atributos es similar al caso de los métodos. Un atributo siempre pertenece a un objeto, si no se indica ninguno Java hará referencia al atributo sobre la instancia actual. El uso de this se hace obligatorio cuando el nombre de uno de los parámetros coincide con el nombre de un atributo de la clase, como ocurre habitualmente en setters o constructores.

```
public Coche(Motor motor){
	this.motor = motor; //this obligatorio
	velocidad = 0;      //this.velocidad = 0
	this.deportivo = motor.cc > 2500; //cc public en Motor
}
```

- Acceso a constructores: this se emplea habitualmente para realizar una llamada desde un constructor a otro constructor para aprovechar el código existente. En este caso la llamada a this siempre tiene que ocupar la primera línea.

```
public Coche(Motor motor){
	this(0);
	this.motor = motor;
}

public Coche(int velocidad){
	this.velocidad = velocidad;
}
```

Paso de parámetros

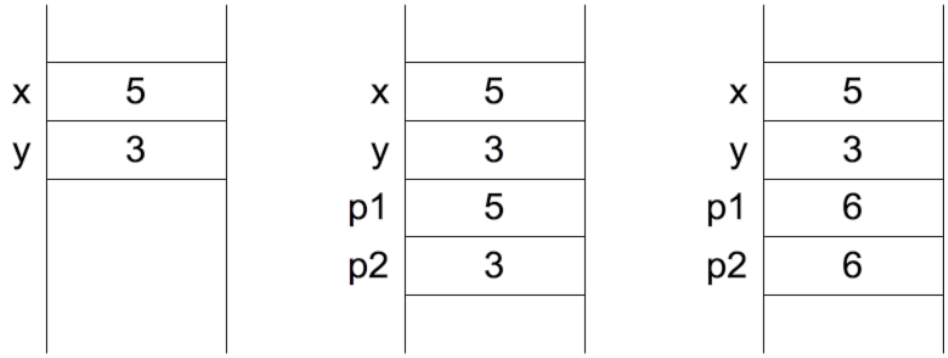
Para invocar a una subrutina, o función auxiliar, o método si se trata de un lenguaje de programación orientado a objetos, en el caso de que exista una lista de parámetros en su definición necesarios para poder efectuar su ejecución, será obligatorio realizar un paso de valores o variables que contengan a estos valores.

```
public int m(int p1, int p2){
	p1 = p1 + 1;
	p2 = p2 * 2;
	return p1 + p2;
}

int x = 5;
int y = 3;
m(x, y);
```

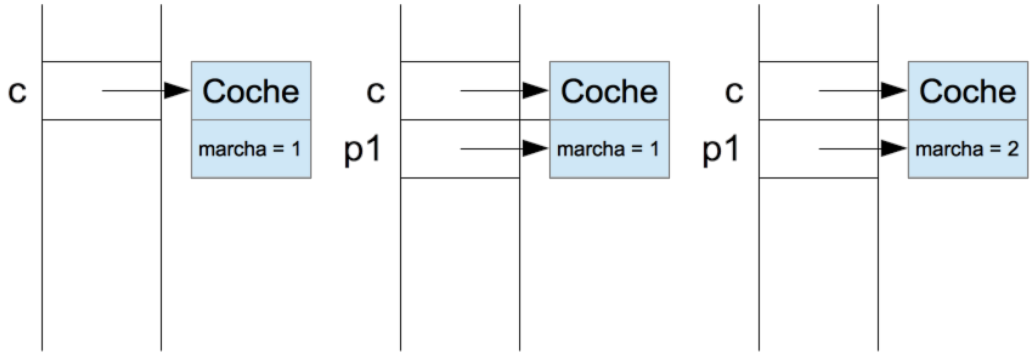
El paso de variables como parámetros dentro de los lenguajes de programación se puede realizar generalmente de dos maneras: por **valor** donde se realiza una copia del valor inicial para manipularla en la subrutina, o por **referencia**, caso en el que se pasa un puntero a la variable original de forma que cualquier cambio sobre ésta será irreversible.

En el caso de Java el paso de parámetros siempre se realiza por valor, es decir, aunque dentro de un método manipulemos el valor de una variable nunca se pierde el valor original de la misma al trabajar con una copia. De cualquier forma hay que tener en cuenta que al pasar a un método una referencia a un objeto, lo que se pasa es una copia de la dirección de memoria que ocupa dicho objeto, por lo que si se realiza una modificación de los valores de los atributos estos cambios serán permanentes.

```java
public int m(int p1, int p2){
	p1 = p1 + 1;
	p2 = p2 * 2;
	return p1 + p2;
}

int x = 5;
int y = 3;
m(x, y);
System.out.println(x); //Imprime 5
```




```java
public void m(Coche p1){
	p1.setMarcha(2);
}

Coche c = new Coche(1);
m(c);
System.out.println(c.getMarcha()); //Imprime 2
```



Referencias

https://docs.oracle.com/javase/tutorial/java/javaOO/index.html

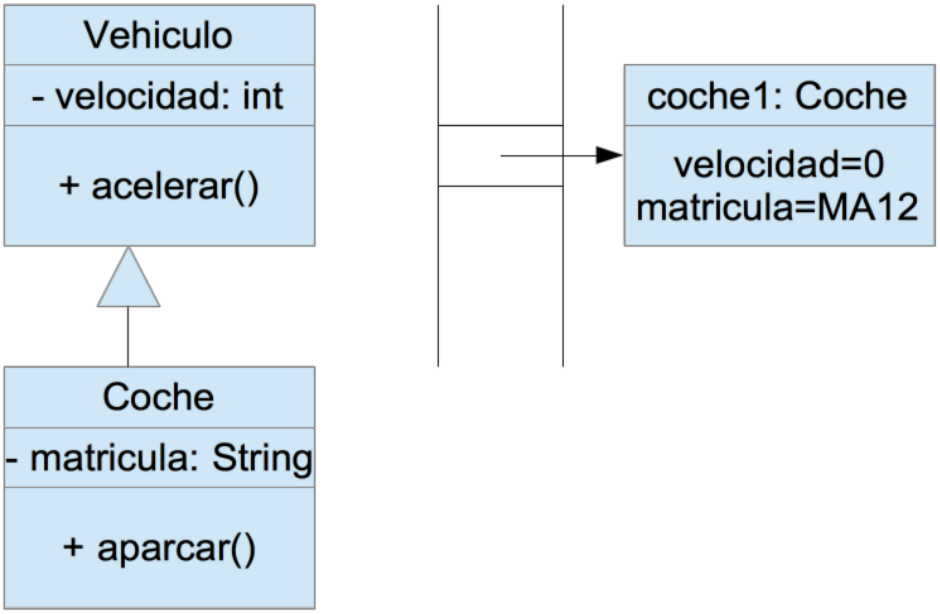
Herencia

Una clase Java puede heredar de otra clase cualquiera indicándolo a través de la palabra clave **extends**. En Java no se permite herencia múltiple:

```
public class Coche extends Vehiculo {
```

La palabra extends indica extensión, significa que la naturaleza de una clase Java siempre debe ser la de extender el comportamiento de la clase padre. Cuando una clase se hereda, todos los elementos no privados son heredados y accesibles desde la clase hija.

En el caso de los métodos se traduce en que se dispondrá de acceso a los métodos de la clase padre, o también que se permitirá sobrescribirlos para extender o modificar su comportamiento por completo. En el caso de los atributos las reglas de acceso son idénticas, pero aunque una clase hija no disponga de acceso directo a los atributos privados de la clase padre, a nivel de ocupación del espacio en memoria del objeto se tendrán en cuenta todos los atributos de todos los antecesores.



En Java todas las clases heredan de forma directa o indirecta de la clase **Object**, la cual representa la raíz de la jerarquía de todas las clases. Si para una clase no se especifica relación de herencia alguna, es como si heredara directamente de Object. En esta clase se definen algunos métodos muy usuales en Java, que por mecanismo de herencia podrán ser empleados en todas las clases sucesoras.

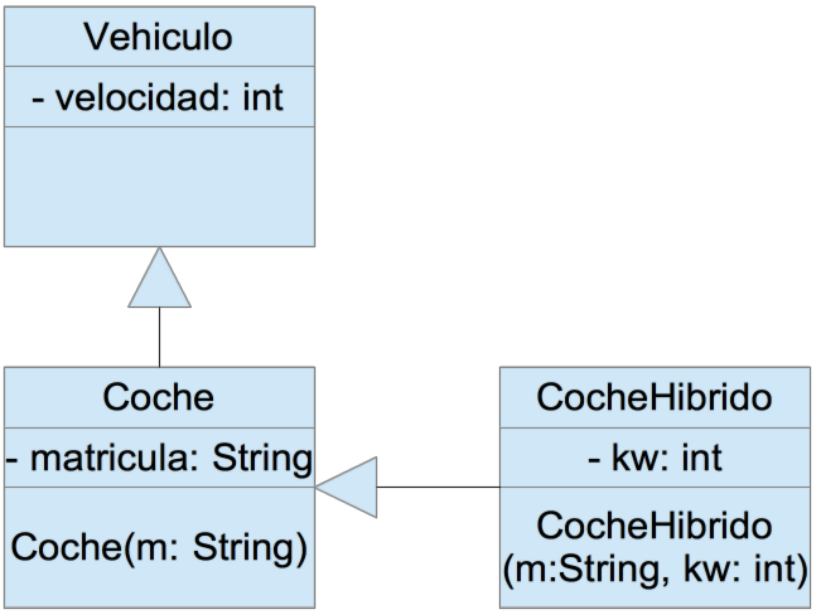
```
@Override
public String toString() {
	return "Mi nombre es " + nombre +
		", y tengo " + edad + " años";
}

System.out.println(juan);
```

El método **toString** sirve para representar a un objeto en forma de cadena de caracteres. Se suele sobrescribir en las clases para modificar el comportamiento por defecto de dicho método en la clase Object que es devolver la dirección de memoria del objeto.

Constructores

Los constructores no se incluyen dentro del mecanismo de herencia, ni se heredan ni se sobrescriben. A la hora de crear una instancia de un objeto se tiene en cuenta la jerarquía de clases del mismo, ya que se debe producir una reserva de memoria adecuada para almacenar toda su información en forma de los atributos que lo componen. Este hecho se traduce en la necesidad de invocar a los constructores de las clases antecesoras desde el constructor de la clase actual.

En Java tanto la reserva de memoria como la invocación de constructores hacia arriba en la jerarquía se produce de forma automática siempre que en las clases que intervienen exista un constructor sin argumentos (o el constructor por defecto).



En el caso de que en la clase padre no exista un constructor sin argumentos será necesario invocarlo de forma explícita:

```
public class Vehiculo {

}

public class Coche extends Vehiculo {

	private String matricula;

	public Coche(String matricula){
		this.matricula = matricula;
	}

}

Coche c = new Coche("MA12");
```

En el ejemplo anterior, al invocar al constructor de la clase Coche lo primero que se produce dentro del mismo es una llamada al constructor de la clase Vehiculo, aunque no aparezca especificado en el código directamente. Es como si en la primera línea se produjera una llamada escrita de la siguiente manera:

```
public Coche(String matricula){
	super(); //No obligatorio
	this.matricula = matricula;
}
```

Esa primera línea es introducida por Java de forma automática siempre que en el padre exista el constructor sin argumentos. En caso contrario habrá que especificar la llamada al constructor pasando por parámetros los valores necesarios, ya que en este caso Java no puede determinar por sí mismo cuál será el valor que se desea indicar para éstos. La llamada a **super** si aparece siempre tiene que ocupar la primera línea del constructor.

```
public class Hibrido extends Coche{

	private int kw;

	public Hibrido(String matricula, int kw){
		super(matricula);
		this.kw = kw;
	}

}

Hibrido h = new Hibrido("MA77", 220);
```

La variable super

La variable **super** representa a la instancia del padre del objeto cuando se encuentra en ejecución, y se puede emplear dentro de una clase para diversos usos:

- Acceso a métodos: de la misma forma que con this se accede a los métodos de la clase actual, con super se accede a los métodos heredados de la clase padre. Tampoco en este caso es necesario escribir super para los métodos heredados ya que Java realizará una búsqueda del método escrito en la clase actual, y si no lo localiza irá ascendiendo en la jerarquía hasta encontrarlo, o en caso contrario se producirá un error de compilación. Su utilización es necesaria en el caso de sobrescritura, para distinguir la versión del método de la clase padre. No se permite ascender más de un nivel en la jerarquía (super.super).

```
public void acelerar(int velocidad){
      super.acelerar();        //aprovechar comportamiento del padre
      setVelocidad(velocidad); //extenderlo con otras acciones
}
```

- Acceso a atributos: acceso a los atributos visibles de la clase padre. Tampoco es imprescindible especificarlo a menos que exista el mismo nombre de atributo en las clases padre e hija, hecho poco habitual y no recomendable.
- Acceso a constructores: para invocar de forma explícita o implícita al constructor de la clase padre.

El modificador final

El modificador final indica que un elemento no es modificable:

Elemento	Características
Variable	La variable no se puede modificar
Método	El método no puede ser sobrescrito
Clase	No se puede heredar de la clase

Las constantes en Java se definen como atributos de clases, empleando los modificadores static y final para indicar que se trata de una referencia compartida que además no es modificable. Las constantes se nombran en mayúsculas, utilizando el guión bajo para hacer la separación en caso de palabras compuestas:

```
public static final int NUM_RUEDAS = 4;
```

Referencias

https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html

Abstracción

Las clases abstractas en Java se identifican con el modificador **abstract**, así como los métodos abstractos que incluyan. Una clase tiene que marcarse como abstract obligatoriamente en el caso de que en la misma se encuentre un método definido como abstracto:

```
public abstract class Figura {

	public abstract void dibujar();

}
```

Los métodos abstractos no tienen cuerpo en Java y acaban en ;, no confundir con métodos con cuerpo vacío que significa que no ejecutan ninguna instrucción:

```
public void vacio() { }
```

Una clase abstracta puede incluir constructores aunque no se puedan crear instancias de ellas. El sentido de disponer de un constructor dentro de una clase abstracta es el de obligar a completar parámetros de inicialización que vendrán proporcionados desde las clases hijas, de forma análoga a como éstas estarán obligadas a implementar los comportamientos abstractos definidos en la clase padre.

```
public abstract class Figura {
	…

	public Figura(String color){
		this.color = color;
	}

}


public class Circulo extends Figura{

	public Circulo(String color) {
		super(color); //Obligatorio
	}

	@Override
	public void dibujar() {
		...
	}

}
```

La definición de interfaces en Java se realiza al igual que las clases en un fichero .java, ocupado por un elemento de tipo **interface**:

```
public interface Acelerable {

	public void acelerar();

}
```

Un interfaz solamente puede contener en Java métodos public y abstract, aunque no se utilicen dichos modificadores Java los considera por defecto. Un interfaz puede definir atributos, en este caso Java los marcará como final static (constantes).

Una clase puede implementar todos los interfaces que necesite, la implementación de interfaces es la manera más aproximada que existe en Java a la herencia múltiple, no se permite heredar de más de una clase pero se pueden implementar diversos comportamientos para dotar a la clase de mayor flexibilidad y realizar un mayor aprovechamiento del código a través del polimorfismo.

```
public class Coche extends Vehiculo implements Acelerable, Comparable
```

La relación que puede existir entre interfaces es de herencia. Un interfaz que hereda de uno o varios interfaces obliga a las clases que lo implementen a definir todos los métodos que se derivan de la relación:

```
public interface Acelerable extends B, C, D
```

Si una clase abstracta realiza una implementación de un interfaz no es necesario que implemente todos los métodos definidos por el mismo, será responsabilidad de las clases sucesoras el completar la definición de todos los métodos necesarios.

Polimorfismo

En Java las relaciones entre clases concretas o abstractas, y la implementación de interfaces, fomentan el empleo de polimorfismo. Una buena práctica de programación es realizar la declaración de las variables o parámetros de métodos empleando el elemento más abstracto posible, de esta forma el código resulta más flexible puesto que gracias a la relación de herencia o implementación de un interfaz en esa variable se podrá realizar referencia a objetos de diversas clases existentes, o incluso clases que se definan en el futuro sin necesidad de alterar otras secciones de código fuente.

```
public interface Acelerable {

	public void acelerar();
}

public interface Aparcable {

	public void aparcar();
}

public abstract class Vehiculo implements Acelerable{ }

public class Coche extends Vehiculo implements Aparcable{

	@Override
	public void acelerar() {
		System.out.println("Acelerando");
	}

	@Override
	public void aparcar() {
		System.out.println("Aparcao");
	}

}

public class Bici extends Vehiculo{

	@Override
	public void acelerar() {
		System.out.println("Pedaleando");
	}

}
```

En el esquema anterior existen dos interfaces y una clase abstracta. Como ya sabemos no se pueden crear instancias de este tipo de elemento pero nada impide declarar variables o parámetros de métodos con estas clases.

```
Vehiculo v;
Acelerable a;
```

Se podrán asignar a las variables anteriores instancias de todas las clases que se encuentran relacionadas siguiendo el orden descendente de la jerarquía, teniendo en cuenta que sobre éstas se podrán invocar únicamente a aquellos métodos que son

exclusivos del tipo de la variable, aún cuando el objeto referenciado disponga de otros métodos adicionales:

```
Vehiculo v = new Coche();
Acelerable a = new Bici();

a.acelerar();
v.aparcar(); //Error de compilación
```

La variable v contiene un Coche que incluye en su definición el método aparcar, pero en tiempo de compilación para Java solamente serán válidos los métodos de Vehiculo. En el caso de la variable a no existe ningún problema al invocar al método acelerar. La versión de acelerar que se ejecutará corresponderá a la del objeto al cual haga referencia la variable en tiempo de ejecución, es decir a la versión contenida en el objeto de la clase Bici.

Para relacionar variables en orden no descendente de la jerarquía es necesario realizar una conversión explícita o casting. Un casting siempre es posible entre elementos relacionados, y permite realizar cambios en las variables que apuntan a los objetos, de forma que para el compilador de Java sí que se permitan las operaciones propias del tipo de la variable. Es necesario en este caso tener precaución de no realizar conversiones no permitidas.

```
Aparcable b = (Aparcable) v;
b.aparcar();

b = (Aparcable) a; //Error de ejecución
b.aparcar();
```

El mayor exponente del polimorfismo en Java es el empleo de variables de entidad superior de la jerarquía como parámetros de métodos o como tipos **genéricos** en la definición de clases.

```
public void mover(Acelerable[] items){
      for (Acelerable a: items){
            a.acelerar();
      }
}
```

Referencias

https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html

https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html

https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html

Clases y APIs básicas en Java

Cadenas de caracteres

En Java se emplea la clase **String** para la gestión de cadenas de caracteres. A diferencia de otros lenguajes, una cadena de caracteres NO es lo mismo que un array de caracteres, aunque la disposición de los caracteres internamente también sigue un esquema indexado.

Para definir una cadena de caracteres se pueden emplear diversos constructores:

```
//Cadena vacía: ""
String s1 = new String();

String s2 = new String("HOLA");

//Excepción al uso de new: asignación de literal
String s3 = "HOLA";
```

Las cadenas de caracteres en Java son sensibles a mayúsculas (**case-sensitive**).

Existe una serie de métodos para el manejo de cadenas muy usuales:

Método	**Descripción**
int charAt(int index)	Obtener el carácter que ocupa un índice
String concat(String str)	Concatenar con str, mismo resultado que +
int length()	Longitud de la cadena
int indexOf(String str)	Primer índice de la subcadena, -1 si no está
int lastIndexOf(String str)	Último índice de la subcadena, -1 si no está
String substring(int begin)	Subcadena a partir de un índice
String substring(int begin, int end)	Subcadena desde un índice hasta otro (-1)
String replace(String old, String new)	Reemplazar las ocurrencias de old por new
String replaceAll(String regex, String new)	Reemplazar las coincidencias según expresión
String trim()	Eliminar espacios en blanco de inicio y fin
String toLowerCase()	Pasar a minúsculas
String toUpperCase()	Pasar a mayúsculas
int compareTo(String str)	Comparar con otra cadena: • 0 si son iguales • <0 si es menor en orden alfabético a str • >0 si es mayor en orden alfabético a str
…	

Ejemplo de uso de los métodos de String:

```
public class StringTest
{
  public static void main(String[] args)
  {
    String s1 = new String();
    String s2 = new String(" ababABAB ");
    String s3 = "Ejemplo";

    System.out.println(s1.compareTo(" "));
    System.out.println(s2.concat((s3)));
    System.out.println(s1.indexOf('h'));
    System.out.println(s2.trim());
    System.out.println(s2.toLowerCase());
    System.out.println(s3.toUpperCase());
    System.out.println(s2.indexOf('b'));
    System.out.println(s2.lastIndexOf('b'));
    System.out.println(s3.length());
    System.out.println(s3.charAt(1));
    System.out.println(s2.replace('A','x'));
  }
}
```

Una característica a tener en cuenta en las cadenas de caracteres de tipo String es que se trata de objetos inmutables. Esto significa que los métodos de la clase String jamás modifican su contenido, sino que generan una nueva cadena en cada ocasión en la que sean invocados. Este hecho requiere de precaución a la hora de manejar objetos String por la sobrecarga de memoria relacionada.

```
String s = "";
for (int i=0; i<1000; i++){
      s = s + i;
}
```

Además de lo anterior hay que tener en cuenta que en Java existe una sección de memoria reservada para los literales de cadenas de caracteres denominada String Pool. Cada vez que se inicializa un objeto de tipo String con un literal de cadena de caracteres la JVM comprueba si ya existe, y en tal caso lo reutiliza. Esta zona de la memoria Heap nunca está sujeta a recolección de basura.

Comparación de cadenas

Dentro de la definición de la clase String nos encontramos con el método **compareTo** proveniente del interfaz **Comparable** cuya funcionalidad será la comparación en orden natural.

Para la comparación de igualdad en Java existe el método **equals** que pertenece a la clase **Object**. Es decir, cualesquiera dos objetos en Java se pueden comparar con equals.

```
String s1 = new String("HOLA");
String s2 = new String("HOLA");
System.out.println(s1 == s2);
System.out.println(s1.equals(s2));
```

El operador de igualdad (==) realiza la comparación de los valores de las variables (como puede ocurrir en el caso de enteros), por lo tanto el valor de la comparación será true solamente en el caso de que se trate de la misma referencia de memoria. La naturaleza de equals es la de comparar los valores de los objetos, aunque es necesario tener en cuenta que la implementación por defecto del método realiza la comparación de referencias.

Concatenación

El operador + en el caso de cadenas de caracteres exclusivamente se emplea para un uso distinto a la suma, que es la concatenación. En Java siempre que en una operación + aparezca involucrado un String todo el resto de elementos se transforma a cadenas de caracteres bajo el siguiente criterio:

- Los tipos básicos de Java se transforman a su representación en formato cadena.
- Lo mismo ocurre para null.
- Los objetos recurren a la representación en forma de cadena que proporciona el método toString.

```
String str = "El resultado es: ";
str = str + x;
str = 5 + " es el resultado";
str = "Mi coche es: " + coche1; //Llamada implícita a toString
str = coche1 + coche2; //Error de compilación
```

StringBuffer y StringBuilder

Ambas clases Java representan versiones mutables de cadenas de caracteres, poseen métodos iguales pero diferencia a ambas el hecho de que StringBuilder no tiene sus métodos sincronizados, por lo que las operaciones se realizan de forma más rápida, por lo tanto es la clase más recomendada como cadena mutable.

En la siguiente tabla aparecen algunos métodos destacados de StringBuilder:

Método	**Descripción**
StringBuilder(String str)	Constructor a partir de String
StringBuilder(int capacity)	Constructor con capacidad inicial
int charAt(int index)	Obtener el carácter que ocupa un índice
StringBuilder append(String str)	Añadir caracteres al final
int length()	Longitud de la cadena
int capacity()	Capacidad actual
int indexOf(String str)	Primer índice de la subcadena, -1 si no está
int lastIndexOf(String str)	Último índice de la subcadena, -1 si no está
String substring(int begin)	Subcadena a partir de un índice
String substring(int begin, int end)	Subcadena desde un índice hasta otro (-1)
StringBuilder replace(int b, int e, String new)	Sustituir caracteres entre b y e-1 por new
StringBuilder insert(int offset, String new)	Insertar new comenzando en offset
StringBuilder delete(int begin, into end)	Eliminar caracteres entre b y e-1
StringBuilder reverse()	Invertir la cadena
void trimToSize()	Ajustar la capacidad a la ocupación
…	

Referencias

https://docs.oracle.com/javase/7/docs/api/java/lang/String.html

http://www.journaldev.com/797/what-is-java-string-pool

https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuffer.html

https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html

Enumerados

Los tipos enumerados en Java sirven para agrupar valores constantes relacionados a través de una referencia similar a una clase.

La definición de un enumerado puede ser tan simple como indicar las constantes que lo componen. Hay que tener en cuenta que una constante de un enumerado no tiene correspondencia directa con un entero o una cadena de caracteres, sino que los valores de enumerados serán referencias a este nuevo tipo de datos creado.

```
public enum Dia {
	LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO
}

class Fecha{
	private Dia dia;
	public Fecha(Dia dia){
		this.dia = dia;
	}
}

Fecha f = new Fecha(Dia.LUNES);
```

Normalmente un enumerado se define en su propio archivo .java, y sigue todas las reglas de acceso para cualquier otro elemento como una clase o interfaz.

Las variables que contienen un valor de un enumerado serán del tipo Java que define dicho enumerado, y solamente podrán recibir valores determinados por el mismo. Los valores de un enumerado se pueden emplear en un **switch**. También se puede tener acceso a todos los valores de un enumerado empleando su método estático values():

```
for (Dia d: Dia.values()){
	System.out.println(d);
}
```

Los valores contenidos en un enumerado a priori no tienen asociados literales numéricos como suele ocurrir con las constantes, pero se pueden establecer a través de un constructor:

```
public enum Moneda {
	PESETA(1), EURO(166.386);

	public double valor;
	private Moneda(double valor){
		this.valor = valor;
	}
}
```

En el ejemplo anterior podemos acceder al valor asociado al enumerado gracias a que el atributo se establece como público:

```
double precioEuros = 3;
double precioPesetas = precioEuros * Moneda.EURO.valor;
System.out.println(precioPesetas);
```

Referencias

https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

System

La clase **java.lang.System** contiene utilidades generales del sistema. Todos sus métodos y atributos son estáticos, destacando de éstos los siguientes:

- public static final PrintStream out: salida estándar, por defecto consola.
- public static final PrintStream err: salida estándar de error, por defecto consola (rojo).
- public static final InputStream in: entrada estándar, por defecto teclado.

La salida por consola a través de un PrintStream se emplea para imprimir valores en pantalla, empleando alguno de los métodos habituales:

```
//Métodos para imprimir por pantalla cualquier tipo Java
System.out.print("Hola ");
System.out.println(nombre);
//Método para imprimir con formato, con número variable de argumentos
System.out.printf("La edad de %s es %d", nombre, edad);
```

La entrada estándar se empleará para la lectura de valores de teclado desde un flujo de bytes, aunque en este caso resulta más indicado utilizar la clase Scanner por sencillez de uso.

Algunos métodos definidos en System como static son los siguientes:

Método	Descripción
long currentTimeMillis()	Fecha actual en ms (desde 01/01/1970)
void exit(int status)	exit(0) salida inmediata, no recomendado
void gc()	Solicitar recolección de basura
void setOut(PrintStream out)	Establecer salida estándar
void setErr(PrintStream out)	Establecer salida de error estándar
void setIn(InputStream out)	Establecer entrada estándar
…	

Referencias

https://docs.oracle.com/javase/7/docs/api/java/lang/System.html

Scanner

La clase **java.util.Scanner** se introduce en Java 1.5 para facilitar la lectura desde teclado de valores que pueden corresponder con tipos básicos de Java o con valores de tipo String.

```
Scanner sc = new Scanner(System.in);
String nombre = sc.next();
int edad = sc.nextInt();
sc.close();
```

Además de los métodos para obtener valores básicos existen métodos adicionales para explorar por completo la entrada:

Método	**Descripción**
boolean hasNext()	Indica si hay un siguiente elemento en la entrada
boolean hasNextInt()	Indica si el siguiente elemento es de tipo int
boolean hasNext…	
boolean hasNextLine()	Indica que hay una nueva línea
Scanner setDelimiter(String exp)	Modificar el delimitador por defecto (espacio)
void close()	Cerrar el Scanner
…	

```
//Leer hasta conseguir un entero
sc.useDelimiter(System.lineSeparator());
System.out.println("Introduzca un entero:");
while (!sc.hasNextInt()){
	sc.next();
	System.out.println("Introduzca un entero:");
}
System.out.println(sc.nextInt());
```

Referencias

https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html

Wrappers

En ocasiones es necesario disponer de una representación en forma de objeto de un tipo básico de Java:

- Paso de parámetro a métodos que reciben un Object.
- Inclusión en estructuras que requieren un Object (colecciones).
- Comprobación de valores nulos.

En el paquete java.lang existe una clase para representar a cada uno de los tipos básicos de Java: Byte, Short, **Integer**, Long, Float, Double, **Character**, Boolean.

Para todas estas clases existe una serie de métodos usuales, entre los cuales destacan aquellos que permiten la conversión desde y hacia cadenas de caracteres. En el siguiente listado se encuentran los principales métodos de Integer que se encuentran en forma análoga en el resto de clases wrapper:

Método	**Descripción**
Integer(int value)	Constructor a partir del tipo básico
Integer(String str)	Constructor a partir de cadena de caracteres
static Integer valueOf(int value)	Obtiene una instancia de Integer a partir de int
static Integer valueOf(String str)	Obtiene una instancia de Integer a partir de String
static int parseInt(String str)	Obtiene un int a partir de una cadena de caracteres
static String toString(int value)	Obtiene una cadena a partir del valor entero
static String toBinaryString(int value)	Obtiene una cadena binaria a partir del valor entero
int intValue()	Obtener el valor int del objeto
short shortValue()	Obtener el valor del objeto como short
long longValue()	Obtener el valor del objeto como long
float floatValue()	Obtener el valor del objeto como float
boolean equals()	Comparar dos objetos Integer
…	

Todos los métodos que reciben una cadena de caracteres son susceptibles de producir un error en tiempo de ejecución si lo que reciben no es exactamente un número.

A partir de la versión 1.5 de Java se introducen dos mecanismos de conversión automática:

- Auto-boxing: generación de wrapper desde tipo básico.
- Auto-unboxing: extracción del valor contenido en un wrapper.

```
Integer a = new Integer(2);
Integer b = 2;
int x = a.intValue();
int y = b;
int z = a + b;
System.out.println(a == x);
System.out.println(a == b);
System.out.println(a.intValue() == b.intValue());
```

En el ejemplo anterior se pueden observar conversiones automáticas que se aplican en todos los casos, excepto en la comparación con == que seguirá ejecutándose sobre los objetos.

Los booleanos también se ven favorecidos por estos mecanismos de conversión.

```
Boolean b1 = new Boolean("tRue");
Boolean b2 = true;
if (b1 && b2){
	System.out.println("OK");
}
```

En el caso de sobrecarga de métodos, la versión ejecutada será siempre aquella donde Java tenga que realizar una operación menos costosa, es decir, no se tendrá en cuenta la conversión automática realizando un casting de tipo simple.

```
public void metodo(double param){
	System.out.println("double");
}
public void metodo(Integer param){
	System.out.println("Integer");
}
int valor = 5;
c.metodo(valor);
```

Referencias

https://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html

https://docs.oracle.com/javase/7/docs/api/java/lang/Boolean.html

Math

Math es una clase para realizar operaciones matemáticas básicas a través de métodos estáticos.

Posee dos constantes PI y E y una lista de métodos entre los cuales destacan los que aparecen a continuación. Algunos métodos se encuentran sobrecargados para poder aplicar el cálculo correspondiente a todos los tipos básicos de valores numéricos de Java:

Método	**Descripción**
int abs(int a)	Valor absoluto de a
double ceil(double a)	Menor double mayor o igual que a con valor entero
double floor(double a)	Mayor double menor o igual que a con valor entero
double cos(double a)	Coseno de a, también disponible sin y tan
double acos(double a)	Arco coseno de a, también disponible asin y atan
double exp(double a)	Potencia de E elevado al valor de a
double hypot(double x, double y)	Raíz de la suma del cuadrado de x e y
double log(double a)	Logaritmo de a
double log10(double a)	Logaritmo en base 10 de a
int min(int a, int b)	Mínimo entre a y b
int max(int a, int b)	Máximo entre a y b
double pow(double a, double b)	Potencia de a elevado a b
double random()	Número aleatorio mayor o igual a 0.0 y menor a 1.0
long round(double a)	Redondeo al long más cercano
int round(float a)	Redondeo al int más cercano
double sqrt(double a)	Raíz cuadrada de a
…	

Referencias

https://docs.oracle.com/javase/7/docs/api/java/lang/Math.html

Fechas

En Java las fechas se representan con la clase **java.util.Date**. Esta clase simplemente actúa como referencia para almacenar un valor de tipo fecha. Para poder trabajar con fechas, consultar datos y realizar alguna operación con ellas es necesario acudir a la clase Calendar.

```
Date fecha = new Date();
//Mostrar la fecha en formato por defecto del sistema
System.out.println(fecha);
//Obtiene un Calendar con la fecha actual
Calendar cal = Calendar.getInstance();
System.out.println(cal.get(Calendar.DAY_OF_MONTH));
//La semana empieza en domingo (1)
System.out.println(cal.get(Calendar.DAY_OF_WEEK));
//Modificar la fecha de un Calendar
cal.setTime(fecha);
//Modificar el mes de la fecha (JANUARY == 0)
cal.set(Calendar.MONTH, Calendar.JANUARY);
//Mostrar la fecha del calendario
System.out.println(cal.getTime());
```

La clase Calendar posee constantes que representan días de semana y meses del año, así como otras constantes para poder solicitar información a través del método get o set(DAY, MONTH, YEAR, HOUR, …)

Algunos métodos destacados de Calendar:

Método	**Descripción**
void add(int field, int amount)	Añadir la cantidad indicada al campo
boolean after(Object when)	Representa un periodo posterior al dado
boolean before(Object when)	Representa un periodo anterior al dado
int get(int field)	Obtener el valor de un campo
void set(int field, int value)	Establecer el valor de un campo
String getDisplayName(int fld, int sty, Locale loc)	Representación en formato cadena
int getFirstDayOfWeek()	Día en el que comienza la semana
void setFirstDayOfWeek(int value)	Cambiar día de comienzo de la semana
int getActualMaximum(int value)	Valor máximo de un campo
Calendar getInstance()	Obtener un calendar por defecto
Calendar getInstance(Locale loc)	Obtener un calendar de una localización
Calendar getInstance(TimeZone zone)	Obtener calendar de una zona horaria
Calendar getInstance(TimeZone zone, Locale loc)	Obtener calendar de localización y zona
Date getTime()	Obtener la fecha actual

void setTime(Date time)	Modificar la fecha actual
TimeZone getTimeZone()	Obtener la zona horaria
void setTimeZone(TimeZone zone)	Modificar la zona horaria
boolean isLenient()	El calendar se comporta como tolerante
void setLenient(boolean value)	Modificar tolerancia del calendar
…	

```
Calendar cal = Calendar.getInstance(Locale.getDefault());
cal.set(Calendar.MONTH, 13);
```

Formato de fecha

La representación interna de la clase Date corresponde con la cantidad de milisegundos transcurridos desde el 01/01/1970. A la hora de imprimir una instancia de Date por pantalla nos encontramos con el problema de que en Java la implementación de toString no devuelve un formato simple por defecto:

```
Mon Feb 26 13:24:29 CET 2018
```

Además de ser un formato poco legible también existirán posibles implicaciones sobre la localización de los usuarios. A través de los métodos de Calendar podemos obtener los campos de la fecha e imprimirlos en formato más adecuado, pero existe una forma más simple que es emplear la clase **java.text.SimpleDateFormat** que a través de un patrón permite proporcionar una salida adecuada:

```
SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
System.out.println(sdf.format(fecha));
```

También se puede emplear una instancia de esta clase para obtener una fecha a partir de una cadena de caracteres:

```
fecha = sdf.parse("27/05/2017");
```

Referencias

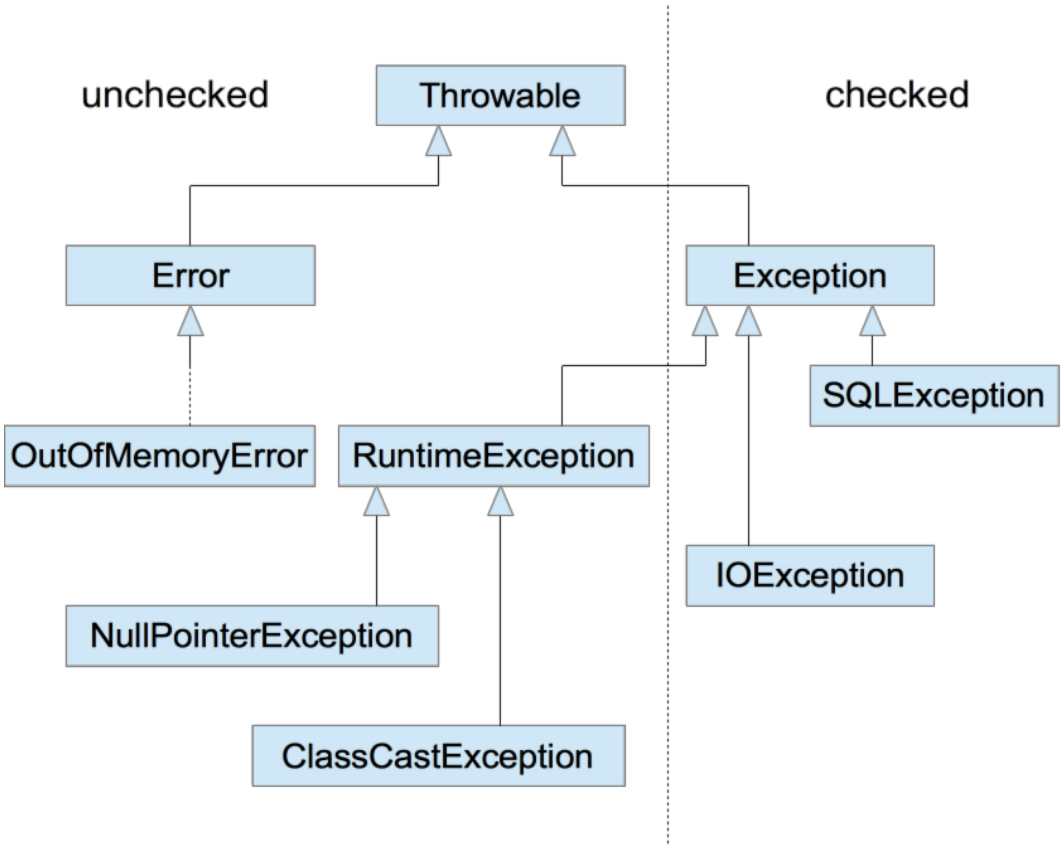
https://docs.oracle.com/javase/7/docs/api/java/util/Date.html

https://docs.oracle.com/javase/7/docs/api/java/util/Calendar.html

http://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html

Excepciones

Una excepción representa una condición anormal ocurrida durante la ejecución de una aplicación que provoca que ésta no pueda continuar convenientemente. Las excepciones modifican por tanto el flujo normal de instrucciones de la aplicación. En la siguiente jerarquía aparecen algunas clases involucradas:



Las excepciones en Java pueden pertenecer a dos grandes grupos:

- **unchecked**: se trata de aquellos errores cometidos por los programadores en la escritura de código, como falta de inicializaciones o paso de parámetros incorrectos. Debido a que un código correctamente implementado no contendrá este tipo de excepciones, no es obligatorio tenerlas en cuenta en tiempo de compilación, aunque recomendable el uso de buenas prácticas como proteger al código frente a posibles excepciones en tiempo de ejecución.

- **checked**: se trata de problemas relacionados con circunstancias ajenas al código escrito, pero que pueden suceder en multitud de ocasiones en función del entorno o de las variables de ejecución. Debido a que la aplicación no podría ejecutarse convenientemente ante una excepción de este tipo, el sistema alerta de esta posibilidad obligando a realizar la acción correspondiente ante dicha excepción.

Además de excepciones en Java se pueden producir otro tipo de problemas en tiempo de ejecución, los errores. Un error habitualmente detendrá la ejecución de la aplicación debido un mal código o diseño de la solución, lo cual provoca fallos en el entorno de ejecución tales como problemas relacionados con la memoria.

Si un bloque de código contiene una o varias sentencias que pueden lanzar excepciones de tipo checked estaremos obligados a realizar alguna acción con ellas para que el código pueda compilar. La signatura de un método determinado de Java indica si éste puede o no lanzar excepciones.

```
public FileInputStream(String name) throws FileNotFoundException
```

La captura de excepciones se realiza en Java con un bloque **try…catch**:

```
//1: Antes de try..catch: código no protegido
try {
	//2: Código protegido
} catch (SQLException ex) {
	//3: Código ejecutado en caso de SQLException
} catch (IOException ex) {
	//4: Código ejecutado en caso de IOException
} catch (Exception ex) {
	//5: Código ejecutado en caso de cualquier otra excepción
} finally {
	//6: Siempre se ejecuta: opcional
}
//7: Después de try..catch: código no protegido
```

Dentro de un catch habitualmente se introduce el código para gestionar la excepción, que habitualmente consistirá en mostrar el error en consola:

```
} catch (Exception ex) {
	System.out.println("Error al conectar");
	System.out.println("Error: " + ex);
	System.out.println("Error: " + ex.getMessage());
	ex.printStackTrace();
}
```

Mostrar la traza de error tiene la gran ventaja de permitirnos visualizar en la consola las sucesivas llamadas a métodos que han originado la excepción actual, incluyendo las líneas de código por las que el error se va propagando. No es recomendable incluir este tipo de llamadas en código de producción.

```
java.io.FileNotFoundException: in.txt (No such file or directory)
	at java.io.FileInputStream.open(Native Method)
	at java.io.FileInputStream.<init>(FileInputStream.java:146)
	at java.io.FileInputStream.<init>(FileInputStream.java:101)
	at es.hubiqus.exc.Conexion.conectar(Conexion.java:75)
	at es.hubiqus.exc.Test.main(Test.java:57)
```

El bloque **finally** es opcional y se suele emplear para la liberación o cierre de recursos. Es posible tener un try..finally sin ningún bloque catch.

El orden en el cual se capturan las excepciones viene determinado por la relación de jerarquía, es decir, un bloque catch no puede preceder a otro bloque catch de una clase de excepción cuyo orden sea inferior en la jerarquía ya que el código siempre entraría por el primer catch y el siguiente sería inalcanzable.

```
try {
      conectar();
} catch (Exception ex) {
} catch (IOException ex) { //Error de compilación
}
```

También se puede escribir código para realizar una captura de alguno de los otros elementos de la jerarquía que hemos visto como Error o clases derivadas o incluso Throwable. Un bloque de código Java NO puede capturar tipos de excepciones que no se puedan producir en el mismo, a menos que se trate de excepciones de tipo Runtime o directamente se realice un catch de Exception o Throwable.

```
try {
      int x = 2 + 3;
} catch (IOException ex) { //Error de compilación
}
```

Lanzamiento de excepciones

Otra opción para poder gestionar una excepción es realizar un lanzamiento en lugar de capturarla con un try…catch:

```
public void metodo() throws SQLException, IOException{
      …
      conectar();
      …
}
```

Un método de Java puede lanzar excepciones siempre que se cumplan las reglas de sobrescritura consistentes en no hacer al método nunca más restrictivo que en la clase padre. En el caso de excepciones estas reglas son las siguientes:

- El método que sobrescribe puede no lanzar ninguna excepción.
- El método que sobrescribe puede lanzar algunas de las excepciones que lanza el método sobrescrito, o alguna subclase de las mismas.
- El método que sobrescribe puede lanzar cualquier tipo de excepción Runtime.

Siempre que se cumplan las condiciones anteriores, un lanzamiento de excepción es más aconsejable que una captura en aquellas clases que no realicen la interacción con el usuario, quedando como responsabilidad de los programadores de estas clases del interfaz de usuario la gestión de errores, proporcionando una salida adecuada.

Un lanzamiento de excepción se puede producir debido a invocación de métodos que lancen excepciones en su definición, pero a través de código es posible lanzar de forma directa una excepción ante situaciones anómalas.

```
if (Math.sqrt(vector.length) != matriz.length){
   throw new Exception("Tamaño de fila incompatible");
}
```

El código donde se produzca un lanzamiento de una nueva excepción a través de **throws** estará sujeto a control de la misma ya sea a través de try..catch o habitualmente produciendo un relanzamiento de la misma.

Excepciones propias

En ocasiones es necesario definir excepciones personalizadas para nuestro código Java. Por ejemplo si estamos desarrollando una API de funciones para matrices, cada método de la clase Matriz sujeto a control de errores podría producir una excepción del tipo MatrizException, muy adecuado por ser totalmente específico de dicha API.

Para escribir una excepción propia en Java simplemente hay que heredar de alguna de las excepciones existentes en la jerarquía. Es necesario tener en cuenta que si nuestra excepción hereda de RuntimeException se definirá como una excepción de tipo unchecked, pero si se hereda de Exception o alguna de sus otras clases hija directas se comportará en tiempo de compilación como checked.

```
public class MatrizException extends Exception{

	//Constructor sin argumentos
	public MatrizException(){
	}

	//Constructor habitual: incluye mensaje de error
	public MatrizException(String msg){
		super(msg);
	}

	//Constructor de encapsulamiento: incluye la excepción real
	public MatrizException(Exception ex){
		super(ex);
	}

}
```

Resulta una práctica habitual encapsular errores producidos dentro del código con excepciones propias, de forma que aunque en la ejecución se produzca una excepción de un tipo determinado de Java, se encapsule en alguna otra excepción para satisfacer el diseño del sistema:

```
try{
   …
}catch(ArithmeticException ex){
      throw new MatrizException(ex);
}
```

De esta forma, aunque a nivel de código se deba tratar a la MatrizException, en tiempo de ejecución en caso de error en la traza se podrá contemplar el tipo de excepción aritmética producida realmente.

Referencias

https://docs.oracle.com/javase/tutorial/essential/exceptions/

https://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html

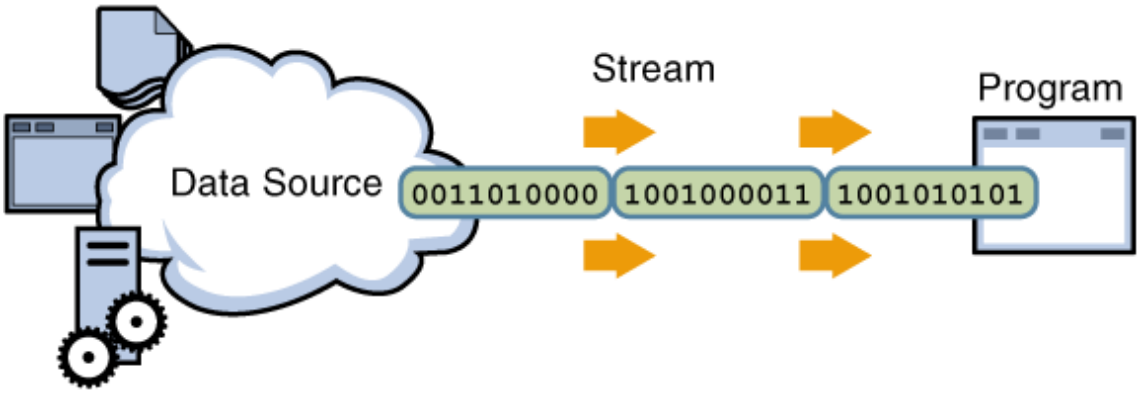
https://docs.oracle.com/javase/7/docs/api/java/lang/Error.html

https://docs.oracle.com/javase/7/docs/api/java/lang/Exception.html

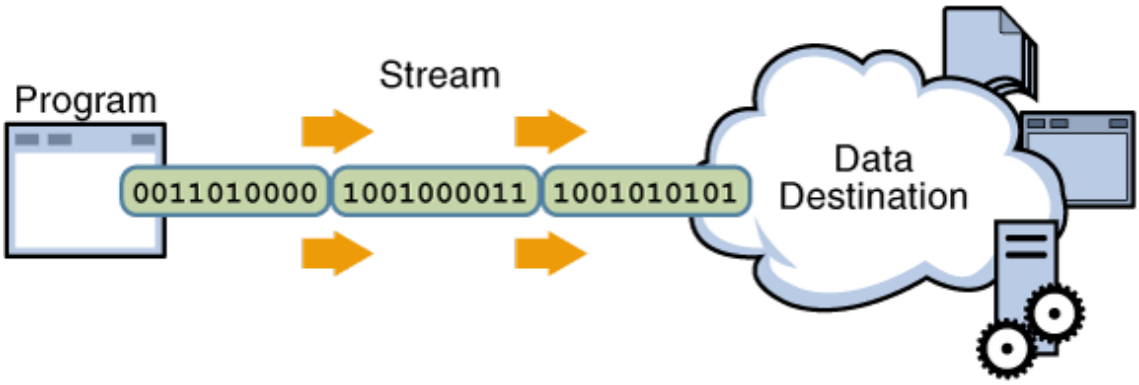
https://docs.oracle.com/javase/7/docs/api/java/lang/RuntimeException.html

Entrada/Salida

La entrada/salida en Java se gestiona a través de Streams. Un Stream representa un recurso de entrada o un recurso de salida independientemente de su naturaleza, por tanto la gran ventaja de la API de Java para esta funcionalidad es la versatilidad que presenta para adaptarse a distintos tipos de flujos y la similitud de utilización en cualquiera de los escenarios posibles.

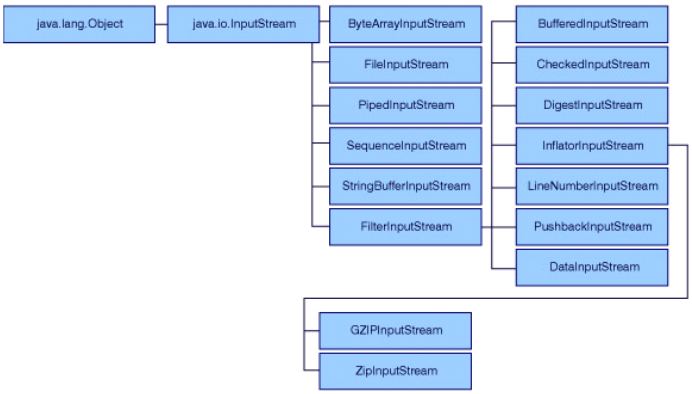


Los Streams permiten por tanto leer o escribir flujos secuenciales y continuos de información en formato binario (bytes). En Java, en función de la dirección del flujo reciben el nombre de **InputStream** y **OutputStream** respectivamente.



Gracias al uso de Streams se pueden realizar operaciones básicas de entrada/salida con elementos tales como ficheros, interfaces de entrada o salida del equipo o recursos remotos, y además a través de las distintas clases de la API se proporcionan funcionalidades extendidas como: buffering, filtrado y parseo, lectura de texto o de tipos primitivos, …

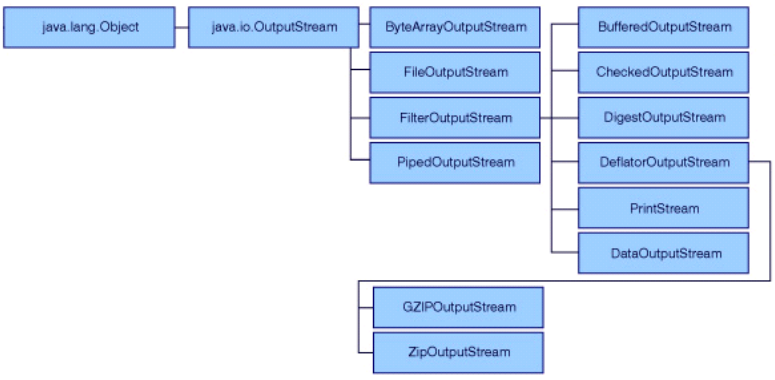
En la siguiente jerarquía aparecen algunas de las clases derivadas de InputStream que permitirán añadir funcionalidad a los streams empleando el patrón de diseño Decorador. En este tipo de patrón se promueve la incorporación de nuevos comportamientos a través de la inclusión de la clase original en un envoltorio, gracias al tipo de constructor que se utiliza que toma como base siempre a una clase del nivel superior.




```
URL url = new URL("http://www.hubiqus.es");
//Abrir stream de entrada desde la url
InputStream is = url.openStream();
byte b = (byte) is.read();
//Lectura con buffer intermedio
BufferedInputStream bis = new BufferedInputStream(is);
//Lectura directa de tipos Java
DataInputStream dis = new DataInputStream(bis);
String str = dis.readUTF();
```

Si examinamos los constructores de las clases que aparecen en el ejemplo podemos ver que cada uno de ellos recibe como argumento un InputStream. Como todas las clases de la jerarquía heredan de InputStream se podrá usar como entrada del constructor cualquier instancia de cualquier clase de la jerarquía, por lo que podemos montar unos tipos de streams en base a otros ya existentes decorándolos con nuevas funcionalidades (métodos disponibles) y características relacionadas.

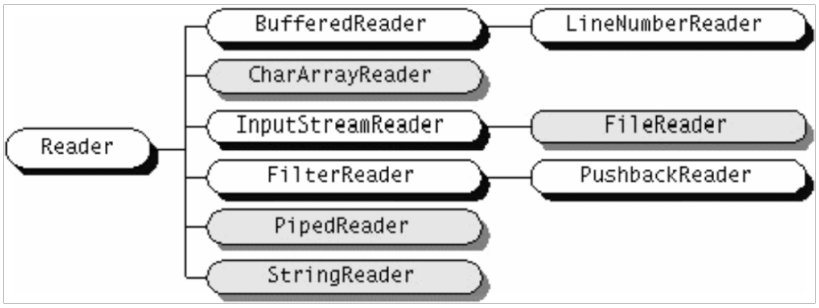
```
public BufferedInputStream(InputStream in)
public DataInputStream(InputStream in)
```



La jerarquía para OutputStream es similar.

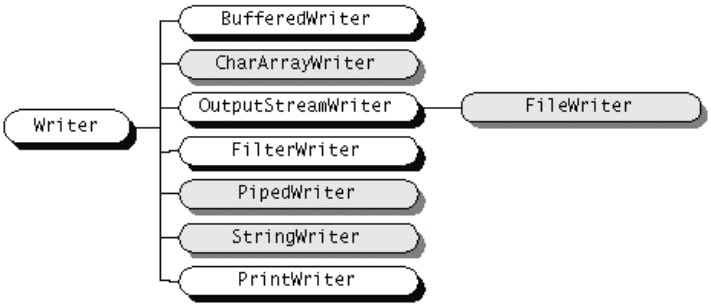
Flujos de caracteres

Muchos de los tipos de contenidos que se van a gestionar para lectura o escritura desde aplicaciones Java serán textuales. En dichos casos en lugar de realizar conversiones directas de los bytes a caracteres, en Java se pueden usar algunas de las clases basadas en **Reader** o **Writer**.



Un Reader lee caracteres de la entrada.

```
URL url = new URL("http://www.hubiqus.es");
//Abrir stream de entrada desde la url
InputStream is = url.openStream();
byte b = (byte) is.read();
//Abrir un reader para el InputStream
InputStreamReader r = new InputStreamReader(is);
char c = (char) r.read();
//Lectura con buffer intermedio
BufferedReader br = new BufferedReader(r);
//Lectura directa de líneas completas
String str = br.readLine();
```



Es muy importante cerrar los flujos de entrada y salida, o los readers y writers, al finalizar el trabajo, empleando el método **close**. El cierre de un objeto empleado para lectura o escritura debe implicar el cierre de los objetos encapsulados.

La clase File

Se trata de una clase para permitir la interacción de la aplicación con el sistema de archivos. Cabe recordar que en los sistemas operativos basados en Unix el concepto de fichero y directorio es el mismo.

La clase File contiene una constante **separator** para especificar el separador de ruta de directorios, ya que es distinto de unos sistemas a otros.

Algunos métodos de la clase File:

Método	**Descripción**
File(String path)	Crear un nuevo File a través de su ruta
File(File parent, String child)	Crear un nuevo File a partir de un File padre y ruta
boolean canRead()	Comprobar si el File puede leerse
boolean canWrite()	Comprobar si el File puede escribirse
boolean canExecute()	Comprobar si el File puede ejecutarse
boolean createNewFile()	Crear fichero en el path del File si no existe
boolean delete()	Eliminar File (directorio vacío) del path si existe
boolean exists()	Comprobar existencia del File en el path
String getName()	Nombre del File
String getPath()	Path del File
String getAbsolutePath()	Path absoluto del File
File getParent()	Obtener File padre del actual
boolean isDirectory()	Comprobar si es un directorio
boolean isFile()	Comprobar si es un fichero
boolean isHidden()	Comprobar si es File oculto
long length()	Longitud del archivo
File[] listFiles()	Lista de Files que componen un directorio
boolean mkdir()	Crear un directorio con el path del File
boolean renameTo(File dest)	Renombrar un File
…	

Es posible además emplear un objeto File como base para la creación de un flujo de entrada:

```
public FileInputStream(File file)
                throws FileNotFoundException
```

Serialización

El procedimiento de **serialización** se ideó en Java para poder transformar objetos en cadenas de bytes, las cuales pueden ser transmitidas a través de un Stream. En el otro extremo de la comunicación, el destino recibirá dicha secuencia y podrá **deserializarla** para reconstruir el objeto.

Para que un objeto pueda convertirse en serializable debe implementar al interfaz **Serializable**. Este interfaz de Java es especial ya que no dispone de ningún método, por lo que implementarlo es como marcar a la clase para que el sistema conozca de esta característica.

Cuando se produce la serialización de un objeto se siguen las siguientes reglas con respecto a los atributos que posee el mismo:

- Todos los atributos se serializan de forma automática excepto aquellos marcados como **transient**. En este caso en la deserialización se inicializan a su valor por defecto.
- Todos los atributos que correspondan con instancias de objetos Java tienen que ser serializables a su vez, si no se producirá una excepción.
- Todas las clases de la jerarquía implicadas también deben ser serializables, en caso contrario los atributos que pertenezcan a clases superiores no serán serializados.

Además las clases serializables habitualmente definirán un atributo **serialVersionUID**, que debe representar un identificador único para que el sistema se asegure en el destino de que dispone de la clase cargada que específicamente permite recuperar el objeto.

Para poder escribir un objeto a un flujo de salida se construye un Stream empleando una instancia de la clase **ObjectOutputStream** y para recuperarlo desde un flujo de entrada se empleará **ObjectInputStream**.

Referencias

https://docs.oracle.com/javase/tutorial/essential/io/

https://docs.oracle.com/javase/7/docs/api/java/io/InputStream.html

https://docs.oracle.com/javase/7/docs/api/java/io/OutputStream.html

http://howtodoinjava.com/java-io-tutorial/

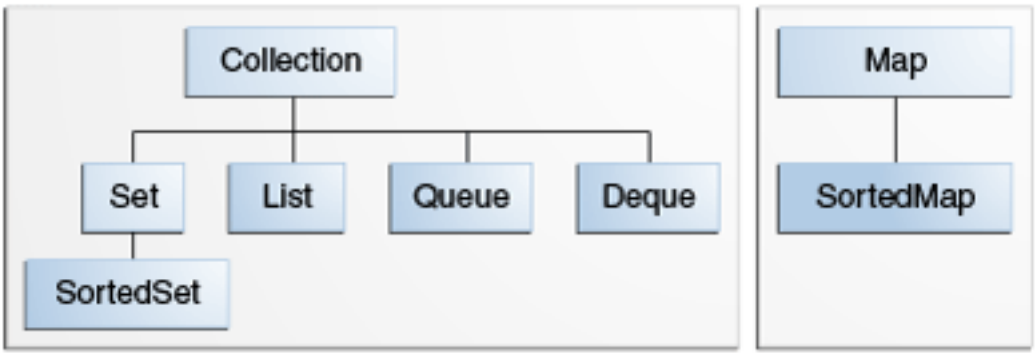
https://en.wikipedia.org/wiki/Decorator_pattern

Colecciones

Una colección es un objeto Java que agrupa a otros elementos en una unidad con una estructura de almacenamiento y características determinadas.

La utilización de colecciones está muy indicada como contenedores de información empleando estándares de Java, y además reduce los esfuerzos de programación en aspectos como almacenamiento u ordenación de información en tiempo de ejecución. La organización de los elementos relativos a colecciones, que se encuentran en el paquete **java.util** define una serie de interfaces, implementaciones y algoritmos que podemos emplear para aprovechar tales ventajas.

La jerarquía de interfaces relativos a colecciones es la siguiente, donde las versiones **Sorted** indican que internamente dispondrán de elementos ordenados:



El interfaz **Collection** ocupa un orden superior y dispone de los métodos básicos comunes a todas las colecciones, excepto en el caso de los **Map**, que por trabajar con pares clave/valor en lugar de con elementos unitarios ocupa otro orden en la jerarquía. Los métodos principales de Collection son:

Método	**Descripción**
boolean add(E e)	Añadir un elemento a la colección si es posible
boolean remove(Object o)	Eliminar un elemento de la colección si es posible
int size()	Número actual de elementos de la colección
boolean isEmpty()	Comprobar si la colección está vacía
boolean contains(Object o)	Comprobar si la colección contiene al elemento
void clear()	Eliminar todos los elementos de la colección
Object[] toArray()	Obtener un array con los elementos de la colección
…	

Para el interfaz Map los métodos distintivos son **put** y **get** para insertar y obtener objetos del map respectivamente.

Los interfaces que heredan de Collection además de disponer de los métodos anteriores poseen otros métodos adicionales en función del tipo de colección:

Interfaz	Colección	Características	Métodos específicos
Set	Conjunto	• No permite duplicados (null)	
List	Lista	• Permite duplicados • Elementos indexados (order)	• add(int, E) • get(int) • remove(int) • set(int, E)
Queue	Cola	• Estructura FIFO	• add(E)/offer(E) • remove/poll • element/peek
Deque	Cola doble	• Acceso a ambos extremos	• addFirst(E)/offerFirst(E) • removeFirst/pollFirst • elementFirst/peekFirst • addLast(E)/offerLast(E) • removeLast/pollLast • elementLast/peekLast

```
lista.add(new Persona("alejandro", 30));
lista.add(new Persona("javier", 32));
//Se puede acceder a un elemento en concreto
System.out.println(lista.get(1));
//Se puede sobrescribir una posición o eliminarla
lista.set(1, new Persona("juan", 23));
```

Implementaciones

Para poder trabajar con las colecciones necesitamos clases que implementen los interfaces anteriores. En Java existen varias implementaciones tipo que determinan el comportamiento interno:

Implementación	Descripción	Características	Clases
Array	Array redimensionable	• Velocidad inserción final • Alta velocidad búsqueda	ArrayList
Linked	Lista enlazada	• Velocidad inserción • Búsqueda secuencial	LinkedList
Hash	Tabla hash	• Alta velocidad inserción • Alta velocidad búsqueda • Posibles colisiones	HashSet HashMap
LinkedHash	Tabla hash encadenada	• Linked + Hash	LinkedHashSet LinkedHashMap
Tree	Árbol binario	• Mantiene orden (sort) • Búsqueda optimizada • Inserción costosa	TreeSet TreeMap

```
//Creamos un TreeSet, que es una implementación de conjunto ordenado
SortedSet<Persona> conjunto = new TreeSet<Persona>();
conjunto.add(new Persona("alejandro", 30));
conjunto.add(new Persona("javier", 32));
conjunto.add(new Persona("ana", 29));
//En SortedSet tenemos acceso a posición primera y última
System.out.println(conjunto.first());
Persona tmp = conjunto.last();
//En todas las colecciones podemos eliminar elementos
conjunto.remove(tmp);
```

Un Map utiliza pares clave/valor. Como clave y como valor se puede emplear cualquier clase Java.

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("alejandro", 30);
map.put("javier", 32);
map.put("ana", 29);
System.out.println(map.get("javier"));
```

Iteradores

Para poder visualizar los elementos que una colección contiene es necesario iterar sobre todos ellos en la mayoría de los casos, ya que por ejemplo en los Set no disponemos de ningún acceso directo a los elementos que contiene (a menos que se exporte a un array), o si se trata de una colección ordenada convendrá obtener los elementos en el orden establecido.

En todas las colecciones se dispone de un método **iterator** que devuelve una instancia del interfaz **Iterator** que permite recorrer los elementos de la colección:

```
Iterator<Persona> iterador = lista.iterator();
while (iterador.hasNext()){
	System.out.println(iterador.next());
}
```

Gracias al bucle for mejorado esto también es posible de manera sencilla sin necesidad de invocar directamente al iterador:

```
for (Persona p: lista){
	System.out.println(p);
}
```

Para recorrer un map, habitualmente se realiza una iteración sobre el **keySet**, que contiene el conjunto de claves (ordenado en caso de un SortedMap) del map:

```
for (String k: map.keySet()){
      System.out.println(map.get(k));
}
```

Ordenación

Para poder ordenar objetos en una colección existen distintas alternativas, la más habitual es recurrir a una colección ordenada. Para que los objetos que se introducen en una colección puedan ordenarse es imprescindible que la clase implemente el interfaz **Comparable**, que tiene como único método **compareTo**:

```
int compareTo(T obj)
```

Cuando una clase implementa el interfaz Comparable, se dice que implementa orden natural. El método devuelve un entero con los siguientes posibles valores:

- 0 si ambos objetos son iguales en orden.
- <0 si el objeto es menor en orden que el objeto obj con el cual se compara.
- >0 si el objeto es mayor en orden que el objeto obj con el cual se compara.

En clases como String o los tipos wrapper numéricos el método compareTo está implementado, pero en las clases de nuestro código Java será necesario implementarlo para conseguir la ordenación. Una práctica muy habitual es basar el resultado del método en la comparación entre atributos comparables:

```
public class Persona implements Comparable{
      …

      @Override
      public int compareTo(Object otro) {
            Persona persona = (Persona) otro;
            //Orden alfabético por nombre
            return this.nombre.compareTo(persona.nombre);
      }
}
```

Cabe destacar que Comparable, al igual que en el caso de las colecciones, es un interfaz configurable con tipo genérico, es decir, que a la hora de definir que una clase lo implementa se puede incluir el tipo base de forma que el método compareTo

reciba una instancia de la clase concreta que se necesita para la comparación, evitando problemas de casting.

```
public class Persona implements Comparable<Persona>{
	…

	@Override
	public int compareTo(Persona persona) {
		//Orden alfabético por nombre
		return this.nombre.compareTo(persona.nombre);
	}
}
```

Otra alternativa para ordenar colecciones en Java es recurrir a la clase **Collections** (no confundir con el interfaz Collection). Esta clase dispone de algunos métodos de utilidad para colecciones, como **sort** que permite ordenar listas. En este caso también es necesario que los objetos que están dentro de la lista pertenezcan a una clase que implementa Comparable, en caso contrario se producirá excepción.

```
Collections.sort(lista);
```

El método sort también dispone de una versión que recibe una instancia de **Comparator**. Este interfaz contiene un método **compare** que realiza una comparación entre dos objetos, devolviendo un resultado similar a compareTo. De esta forma podemos obtener los objetos ordenados aunque una clase no implemente orden natural, o si necesitamos una alternativa de ordenación para la misma.

```
int compare(T o1, T o2)
```

Para implementar este interfaz se puede recurrir a una clase concreta, o directamente generar una definición anónima para pasar al método sort. A partir de la versión 1.8 de Java es posible incluir también como expresión lambda:

```
Comparator<Persona> comp = new Comparator<Persona>(){
	@Override
	public int compare(Persona p1, Persona p2) {
		return p1.getEdad() - p2.getEdad();
	}
};
Collections.sort(lista, comp);
//Con expresión lambda
Collections.sort(lista,
		(Persona p1, Persona p2) -> p1.getEdad() - p2.getEdad());
```

Los métodos equals y hashCode

En las colecciones, en función del tipo de implementación seleccionada para contener objetos existen dos métodos heredados de **Object** que adquieren gran importancia: **hashCode** para almacenamiento y recuperación en implementaciones hash, y **equals** para las comparaciones de instancias.

El método equals determina cuándo dos objetos son iguales en cuanto a su contenido. También en este caso es recomendable basar su implementación en la comparación entre elementos Java que ya dispongan de la capacidad de comparación, como por ejemplo las cadenas de caracteres:

```
@Override
public boolean equals(Object obj) {
	return (obj instanceof Persona) && (nombre != null) &&
				(nombre.equals(((Persona) obj).nombre));
}
```

Si equals no está bien implementado es posible que no se encuentren los objetos a la hora de buscar o eliminar, o que se admitan duplicados en conjuntos. El método equals debe cumplir las siguientes propiedades:

- Reflexiva: x.equals(x) es true.
- Simétrica: si x.equals(y) es true entonces y.equals(x) es true.
- Transitiva: si x.equals(y) es true, y.equals(z) es true, entonces x.equals(z) es true.
- Consistente: si x.equals(y) es true lo será en sucesivas invocaciones.
- Comparación con null: x.equals(null) es false.

Hay que tener en cuenta que en colecciones ordenadas, para la comparación en lugar de equals se suele emplear compareTo internamente.

Para la implementación de hashCode la norma básica es que se tiene que basar en los mismos atributos en los cuales se basa equals, ya que se debe cumplir que si x.equals(y) su código hash es el mismo:

```
@Override
public int hashCode() {
	return (nombre == null) ? 0 : nombre.hashCode();
}
```

Es posible que dos objetos devuelvan el mismo código hash. Si hashcode no está bien implementado puede ocurrir que se produzcan más colisiones de las habituales.

Genéricos

En la definición de variables de tipo colección, en la creación de instancias para éstas y en aspectos como la comparación de objetos resulta muy útil desde el punto de vista de la programación emplear un tipo base que indique cuál es el contenido de los métodos que se van a emplear, ya que en tiempo de compilación Java examinará si los objetos son compatibles con el tipo base.

Realmente una colección Java es un elemento genérico que admite cualquier clase Java, ya que todos sus métodos están basados en Object:

```
List lista = new ArrayList();
lista.add(new Persona("alejandro", 30));
lista.add(3); //Integer
lista.add("OTRA COSA");
//Recuperamos una persona de la lista (casting)
Persona p = (Persona) lista.get(1); //Error en tiempo de ejecución
```

En el momento que se establece el tipo base en la creación de la colección, ni se podrán agregar elementos de otro tipo Java a la misma, ni será necesario realizar un casting para recuperarlos:

```
List<Persona> lista = new ArrayList<Persona>();
lista.add(new Persona("alejandro", 30));
lista.add(3); //Error de compilación
lista.add("OTRA COSA"); //Error de compilación
//Recuperamos una persona de la lista (casting)
Persona p = lista.get(0);
```

Se puede crear una clase Java genérica determinando en su definición el tipo base:

```
public class Generica<E> implements Comparable<E>{
	private E atributo;

	public E get(){
		return atributo;
	}

	public void add(E e){
		this.atributo = e;
	}

	@Override
	public int compareTo(E o) {
		return 0;
	}
}
```

Cualquier clase construida de la forma anterior admite la inclusión del tipo base a la hora de definir variables para la misma:

```
Generica<Coche> g = new Generica<Coche>();
g.add(coche);
```

También es posible en la programación de clases o métodos genéricos incluir restricciones del tipo de clase Java que se puede emplear en su definición:

```
//Solamente admite clases que sean Number
public class Generica<E extends Number>{
```

Por último, si en la determinación del tipo base de una clase se emplean comodines (**wildcards**) se indica al compilador que los objetos admitidos pueden pertenecer a distintos órdenes de jerarquía:

```
List<?> lista1 = new ArrayList<Persona>();
List<? extends Number> lista2 = new ArrayList<Integer>();
List<? super Number> lista3 = new ArrayList<Number>();
```

Operaciones de agregado

A partir de Java 1.8 se introduce la capacidad de trabajar con colecciones a través de métodos de agregado, de forma que encadenando operaciones sobre el resultado de otras operaciones se realiza un procesamiento de forma más sencilla y optimizada que el realizado habitualmente a través de iteradores, ya que se comprime el código y se favorece la delegación de estas operaciones al propio JDK que empleará procesos concurrentes.

Se define un **pipeline** como la secuencia de operaciones de agregado aplicadas sobre un **stream**, el cual representa a su vez la secuencia de elementos sobre los que se produce.

Hay varias funciones que se pueden aplicar de esta manera a un stream:

Método	Descripción	Ejemplo
filter	Aplica un filtro para cada elemento	`filter(p -> p.getEdad() > 18)`
distinct	Obtener elementos distintos (equal)	`distinct()`
forEach	Aplica un método a cada elemento	`forEach(System.out::println)`
map	Extrae un valor determinado de cada uno de los elementos	`map(Persona::getNombre)`
mapToInt	Obtener un IntStream	`mapToInt(Persona::getEdad)`
count	Cuenta el número de elementos	`count()`

sum
average
max min
	Operaciones sobre un stream numérico	`sum()`
		`average()`
		`max()`

allMatch anyMatch	Comprobación sobre el conjunto de elementos	`allMatch(p -> p.getEdad() > 18)`
sorted	Obtiene el stream ordenado, a través de Comparable o Comparator	`sorted()`
collect	Obtener el resultado en forma de lista	`collect(Collectors.toList())`
limit	Limita el número de resultados	`limit(5)`
skip	Descarta los n primeros resultados	`skip(5)`
…		

En la definición de este tipo de operaciones y el paso de condiciones a las mismas es muy habitual el empleo de expresiones lambda como forma compacta de escritura de código y acceso a métodos:

```
lista
      .stream()
      .filter(p -> p.getEdad() > 18)                    //Predicate
      .sorted((p1, p2) -> p1.getEdad() - p2.getEdad()) //Lambda
      .forEach(System.out::println);                   //Consumer
```

Referencias

https://docs.oracle.com/javase/tutorial/collections/

https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html

https://docs.oracle.com/javase/7/docs/api/java/util/Map.html

https://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html

https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html

https://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html

https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html

https://docs.oracle.com/javase/tutorial/java/generics/

http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html

https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

Acceso a Bases de Datos desde Java

Bases de datos

Una base de datos es un conjunto organizado y estructurado de información. Los SGBD surgen como evolución de la utilización de ficheros como base de almacenamiento de la información de un sistema, presentando las siguientes ventajas principales:

- Acceso lógico a través del lenguaje de consulta
- Acceso seguro, concurrente y optimizado
- Integridad y aseguramiento de la información

Tablas y columnas

La información en las bases de datos se organiza en **registros** dentro de **tablas.** Una tabla es un contenedor de información relacionada que se estructura en **columnas**.

Nombre	Profesión	País	Población
Andrés Iniesta	Futbolista	España	47M
Alejandro Jurado	Programador	España	47M
Julia Roberts	Actriz	USA	326M
…			

Cada columna viene determinada por un tipo de datos concreto:

Tipo	Descripción	Ejemplos
Texto	Cadena de texto de longitud determinada	CHAR(N), VARCHAR(N)
Texto largo	Texto de longitud ilimitada	TEXT
Numérico	Valores numéricos	INT, DECIMAL(I, D), FLOAT
Fecha	Fechas	DATETIME, TIMESTAMP
Binario	Información formato binario	BLOB

Dentro de las columnas de una base de datos relacional existen tipos de columnas especiales, denominadas **claves**:

- Clave primaria: determina un registro de forma única e inequívoca.
- Clave foránea: determina una relación entre tablas.

Cabe destacar la existencia de columnas con valores autonuméricos, y la posibilidad de disponer de columnas que permiten valores nulos.

Modelo de datos

Un modelo de datos es una representación de la estructura de una base de datos, donde se muestran las tablas y relaciones que existen entre ellas.

Una relación se produce cuando los registros de una tabla realizan alguna referencia a registros de una segunda tabla como consecuencia del diseño del modelo. Las relaciones que se pueden producir entre tablas de una base de datos son:

- Muchos a uno: en la primera tabla pueden existir muchos registros relacionados con la segunda, es la relación habitual entre tablas.
- Muchos a muchos: entre ambas tablas pueden existen varios registros relacionados, se suele resolver con una tabla intermedia que recoge los emparejamientos.
- Uno a uno: entre ambas tablas solamente puede existir la relación a través de un único registro.

En el diseño del modelo de datos donde se define la información que aparecerá en cada columna de cada tabla y las relaciones que surgen entre ellas, es necesario tener en cuenta el proceso de normalización:

- 1FN: elimina registros duplicados
  - Todas las columnas son atómicas
  - No existe variación en el número de columnas
  - Existe una clave primaria
  - Dependencia funcional de la clave
  - Independencia del orden

Nombre	Profesión	País	Población
Andrés Iniesta	Futbolista	España	47M
Alejandro Jurado	Programador, Formador	España	47M
Julia Roberts	Actriz	USA	326M
…			

Nombre PK	Profesión1	Profesión2	País	Población
Andrés Iniesta	Futbolista		España	47M
Alejandro Jurado	Programador	Formador	España	47M
Julia Roberts	Actriz		USA	326M
…				

Nombre PK	Profesión	País	Población
Andrés Iniesta	Futbolista	España	47M
Alejandro Jurado	Programador	España	47M
Alejandro Jurado	Formador	España	47M
Julia Roberts	Actriz	USA	326M
…			

- 2FN: elimina dependencias parciales
  - Se encuentra en 1FN
  - Todas las columnas dependen de forma única de la clave primaria

Nombre PK	Profesión PK	País	Población
Andrés Iniesta	Futbolista	España	47M
Alejandro Jurado	Programador	España	47M
Alejandro Jurado	Formador	España	47M
Julia Roberts	Actriz	USA	326M
…			

Una tabla 1FN con clave simple se encuentra siempre en 2FN.

- 3FN: elimina anomalías de actualización
  - Se encuentra en 2FN
  - No hay dependencias entre columnas no primarias

Nombre PK	Profesión PK	País	Población
Andrés Iniesta	Futbolista	España	47M
Alejandro Jurado	Programador	España	47M
Alejandro Jurado	Formador	España	49M
Julia Roberts	Actriz	USA	326M
…			

Existen más niveles de formas normales destinados a la reducción de la redundancia de la información que aparece en las tablas, pero que en raras ocasiones serán detectadas o alcanzables en aplicaciones empresariales.

Elementos de una base de datos

Un SGBD suele recoger varias bases de datos o **esquemas**, que vendrán determinados por sus tablas y relaciones.

Adicionalmente en cada base de datos pueden existir otra serie de elementos. Algunos de éstos requieren del empleo de un lenguaje de programación propio del SGBD para permitir la definición de cursores, variables, condiciones, excepciones, …

Elemento	Descripción
Vistas	Se trata de consultas que se almacenan con un nombre, debido a que su utilización sea frecuente en el esquema o para facilitar el acceso desde aplicaciones externas
Índices	Elementos de utilización interna para la mejora de eficiencia en búsquedas en tablas, o cumplimiento de restricciones de unicidad. Algunos índices se generan de forma automática (PK, FK)
Triggers	Rutina asociada a la ejecución de una determinada acción (INSERT, UPDATE, DELETE) sobre una tabla (BEFORE, AFTER). Dispone de acceso además sobre las filas que se están alterando (OLD, NEW) de forma que puede modificar el comportamiento de la tabla sobre la que se aplica o tablas adicionales.
Procedimientos	Rutina que aglutina instrucciones y procesamiento sobre resultados de consultas. Facilita el acceso a operaciones en bloque desde aplicaciones externas
Funciones	Similar al anterior pero produciendo una salida determinada. Las funciones se suelen emplear dentro de consultas

Ejemplo de trigger:

```
DELIMITER //
CREATE TRIGGER empleados_insert AFTER INSERT ON empleados
FOR EACH ROW
BEGIN
-- seleccionar salario mínimo del departamento
DECLARE minimo INT;
SELECT MIN(salario) INTO minimo FROM empleados
WHERE dnumero = NEW.dnumero;

IF minimo < 1000 THEN
      SET minimo = 1000;
END IF;

-- insertar nómina con salario base
INSERT INTO nomina
SET enumero = NEW.numero,
salario = minimo,
fecha = NEW.fechaAlta;
END;//
```

Ejemplo de procedimiento almacenado con cursor:

```
DELIMITER //
CREATE PROCEDURE actualizarNominas()
BEGIN
DECLARE done INT DEFAULT FALSE;
DECLARE num INT;
DECLARE sal INT;

-- cursor de empleados
DECLARE eCursor CURSOR FOR
SELECT numero,salario FROM empleados;

-- se encarga de la salida del bucle cuando no queden datos
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

OPEN eCursor;

read_loop:
LOOP
FETCH eCursor INTO num, sal;
IF done THEN
      LEAVE read_loop;
END IF;

UPDATE nomina SET salario = sal WHERE enumero = num;
END LOOP;

CLOSE eCursor;

END;//
```

Referencias

https://es.wikipedia.org/wiki/Base_de_datos

https://es.wikipedia.org/wiki/Modelo_de_base_de_datos

https://es.wikipedia.org/wiki/Normalización_de_bases_de_datos

https://dev.mysql.com/doc/refman/5.7/en/

Revisión de SQL

El **Structured Query Language** es un lenguaje declarativo que permite indicar a una base de datos sentencias para conseguir obtener un resultado concreto. Las operaciones que se pueden especificar con SQL pertenecen a dos grupos principales:

- **DML**: sentencias de manipulación de datos, para recuperación o almacenamiento de la información. A este grupo pertenecen: **SELECT**/**INSERT**/**UPDATE**/**DELETE**.
- **DDL**: definición y modificación de los posibles elementos existentes en una base de datos. En este caso las sentencias son de la forma: **CREATE**/**ALTER**/**DROP**.

SQL es un estándar seguido por la mayoría de sistemas gestores de bases de datos (ANSI-SQL), pero que en algunos de los casos introduce algunas peculiaridades, sobre todo en el uso de funciones específicas.

SELECT

Permite definir consultas para la recuperación de la información. La forma general de una sentencia de este tipo es:

```
SELECT columnas
FROM tablas
WHERE condiciones
GROUP BY columnas de agrupación
HAVING condiciones sobre grupos
ORDER BY columnas
```

En algunos sistemas es posible limitar el número de resultados a la salida con **LIMIT**.

Las condiciones pueden unirse a través de **AND** o **OR**, o negarse con **NOT**, y pueden contener algunas funciones estandarizadas como las siguientes:

Función	Descripción
columna = valor	Comparación directa de un valor. También >, <, >=, <=, <>
columna LIKE 'valor'	Comparación parcial de cadenas, puede contener %
columna IS NULL	Comparación con el valor NULL
DISTINCT columna	Distintos valores de una columna
COUNT, SUM, MIN, MAX, AVG	Funciones sobre grupos
columna + valor	Se pueden realizar operaciones entre columnas o con valores
…	

Una característica del lenguaje SQL es que se encuentra basado en el álgebra relacional. En algunos casos se puede recurrir a una operación de unión para agrupar dos conjuntos de resultados distintos (pero compatibles):

```
SELECT nombre, dnumero, 'Revisar salario'
FROM empleados WHERE salario < 1500
UNION
SELECT nombre, dnumero, 'Mantener salario'
FROM empleados WHERE salario >= 1500
```

Una operación muy importante en las consultas es la combinación, ya que para obtener datos involucrados en una relación de tablas es necesario indicar el criterio de emparejamiento. Una combinación se puede indicar explícitamente con un JOIN o a través de una condición en WHERE:

```
SELECT empleados.nombre, departamentos.nombre
FROM empleados, departamentos
WHERE departamentos.numero = empleados.dnumero;

SELECT empleados.nombre, departamentos.nombre
FROM empleados
INNER JOIN departamentos ON departamentos.numero = empleados.dnumero;
```

Existen distintos tipos de JOIN:

Combinación	**Descripción**
CROSS JOIN	Producto cartesiano, necesario evitarlo
INNER JOIN JOIN	Combinación exclusivamente de los registros que cumplan el criterio
LEFT JOIN LEFT OUTER JOIN	El más común, obtiene todos los datos de la tabla principal y la combinación con los que cumplan el criterio
RIGHT JOIN RIGHT OUTER JOIN	Obtiene todos los datos de la tabla secundaria y la combinación con los que cumplan el criterio
FULL JOIN FULL OUTER JOIN	Unión de los anteriores

INSERT

Permite almacenar datos en las tablas definidas en la base de datos.

```
INSERT INTO tabla (columnas)
VALUES (valores)
```

A la hora de definir las columnas para inserción se deben seguir las siguientes reglas, en caso contrario se producirá un error en la base de datos:

- Es posible no indicar ningún nombre de columna, en tal caso se establecerán valores para todas las columnas de la tabla en orden de definición.
- Es posible no indicar algún nombre de columna, en tal caso se establecerán valores para todas las columnas de la tabla que no admitan valores nulos.
- En caso de columnas auto-numéricas no se deben indicar en el insert para que el sistema las gestione de forma automática.
- Los valores establecidos deben ser compatibles con la definición de columnas de la tabla.

Es posible establecer una sentencia INSERT con múltiples valores o desde un SELECT para copiar valores desde otra tabla:

```
INSERT INTO departamentos (numero,nombre,ciudad) VALUES
 (10,'Contabilidad','Málaga'),
 (20,'Desarrollo','Madrid');

INSERT INTO departamentos
 SELECT numero, nombre, ciudad
 FROM copia.departamentos;
```

UPDATE

Permite actualizar datos de una tabla. También en este caso es necesario indicar qué columnas van a ser actualizadas, además de incluir un WHERE que determine cuáles de las filas serán modificadas (en caso contrario modificará todas).

```
UPDATE tabla
SET columna = valor,
columna = valor …
WHERE condicion
```

DELETE

Permite eliminar filas de una tabla, en este caso es mucho más importante el uso de WHERE.

```
DELETE FROM tabla
WHERE condicion
```

Transacciones

Una transacción incluye operaciones sobre una o más tablas que se deben comportar de forma atómica, es decir, o todo el grupo de operaciones se ejecuta o ninguna de ellas se habrá producido finalmente en la base de datos.

Se trata de un mecanismo de protección para mantener la coherencia de los datos. Las transacciones en un sistema gestor de bases de datos pueden ser de dos tipos:

- Explícitas: el usuario debe marcar el comienzo de la transacción (begin/start), y cerrarla confirmando el éxito de las actualizaciones (**commit**) o deshacerla ante cualquier tipo de problema (**rollback**).
- Implícitas: cada vez que un usuario lanza una actualización a la base de datos el sistema abre una transacción. Habitualmente la confirmación se realiza de forma automática, si no el usuario tendrá la responsabilidad de cerrarla.

Referencias

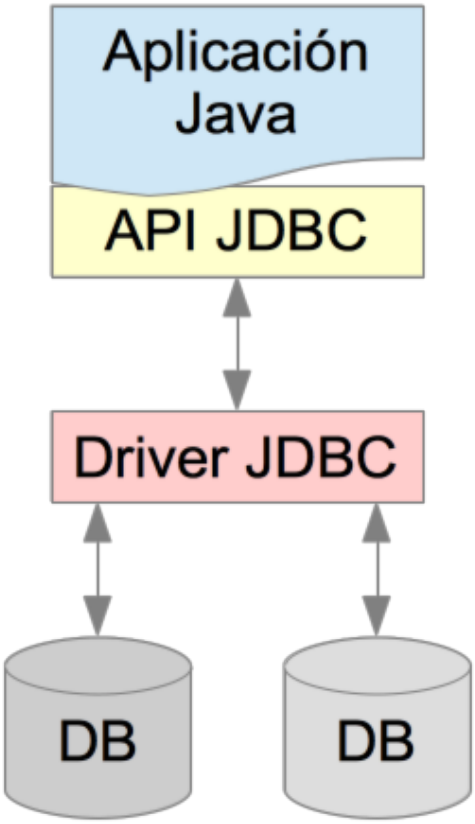
https://en.wikipedia.org/wiki/SQL

https://dev.mysql.com/doc/refman/5.7/en/sql-syntax.html

http://docs.oracle.com/cd/B28359_01/server.111/b28286/toc.htm

JDBC

JDBC (Java DataBase Connectivity) es la API estándar de Java para la conectividad con sistemas gestores de bases de datos. Su principal característica es que el código de la aplicación es totalmente independiente del tipo de sistema de base de datos empleado, normalizando la forma de acceso, el lanzamiento de sentencias SQL y el procesamiento de los resultados.



Los drivers JDBC son proporcionados por los fabricantes de cada sistema de bases de datos, y son los que realmente proporcionan la conexión con cada uno de ellos implementando los protocolos necesarios, permitiendo la interacción desde la aplicación Java a través de las clases de la API JDBC. Los drivers JDBC pueden pertenecer a una de estas familias:

- Puente JDBC-ODBC: gestionado por el sistema operativo.
- API Nativa: gestionado por el sistema gestor de base de datos completamente.
- Middleware: gestionado por un servidor de aplicaciones.
- Driver Java: gestionado por completo desde Java.

Para poder programar el acceso desde Java a una base de datos es necesario realizar una serie de pasos, que son los siguientes:

1. Cargar el driver correspondiente al SGBD con el que vamos a conectar. Es necesario disponer del archivo .jar configurado en el path del proyecto.
2. Definir los parámetros de conexión, habitualmente url en formato JDBC, usuario y contraseña de acceso a la base de datos.
3. Establecer la conexión con la base de datos.
4. Definir la sentencia SQL.
5. Ejecutarla y procesar los resultados a través de las clases de la API JDBC.
6. Cerrar la conexión con la base de datos.

```
//Cargar driver
Class.forName("com.mysql.jdbc.Driver");
//Conectar con la base de datos
Connection conn = DriverManager.getConnection(
      "jdbc:mysql://127.0.0.1/inventario", "root", "root");
//Obtener objeto Java para lanzar sentencia SQL
Statement st = conn.createStatement();
…
//Cerrar el statement
st.close();
//Cerrar la conexión
conn.close();
```

Para comunicarnos con la base de datos se pueden emplear dos tipos de objetos:

Interfaz	Descripción	Métodos básicos
Statement	• se crea con createStatement • indicado para lanzar varias consultas	int executeUpdate(sql) ResultSet executeQuery(sql)
PreparedStatement	• se prepara con prepareStatement(sql) • indicado para consultas repetitivas o complejas ya que precompila la sql • permite incluir comodines (?)	int executeUpdate() ResultSet executeQuery() setString/setInt/setDate/…

Transacciones

En JDBC es posible realizar las operaciones de actualización dentro de una transacción de la base de datos. Para ello es necesario desactivar el modo autocommit de la conexión empleada:

```
//Obligará a hacer commit para confirmar la transacción
conn.setAutoCommit(false);
```

De esta forma en el momento que se lance la primera actualización se generará una transacción implícita. Para confirmar o cancelar la transacción existen métodos convenientes:

```
//Confirmar transacción
conn.commit();
//Cancelar transacción
conn.rollback();
```

Procesamiento de resultados

La forma de acceder a la información de una consulta es utilizando un ResultSet, se trata de una representación iterable de cada una de las filas recuperadas, que permite consultar el valor de cada una de sus columnas:

```
ResultSet rs = st.executeQuery("select * from departamentos");
//Iterar sobre todos los registros
while (rs.next()){
	rs.getString(1); //Los índices de columnas empiezan en 1
	rs.getInt("numero"); //Acceso por nombre más indicado
}

rs.close();
```

La consulta SQL empleada puede incluir JOINs para la unión entre tablas, en tal caso si existe conflicto de nombres se pueden emplear alias individualizados, pero también es permitida la inclusión del nombre de tabla como prefijo:

```
System.out.println(rs.getString("producto.nombre"));
```

Referencias

https://docs.oracle.com/javase/tutorial/jdbc/

Desarrollo de Aplicaciones Web con Java

HTTP

HTTP (**Hypertext Transfer Protocol**) es el protocolo de comunicación del modelo TCP/IP para transferir información por Internet.

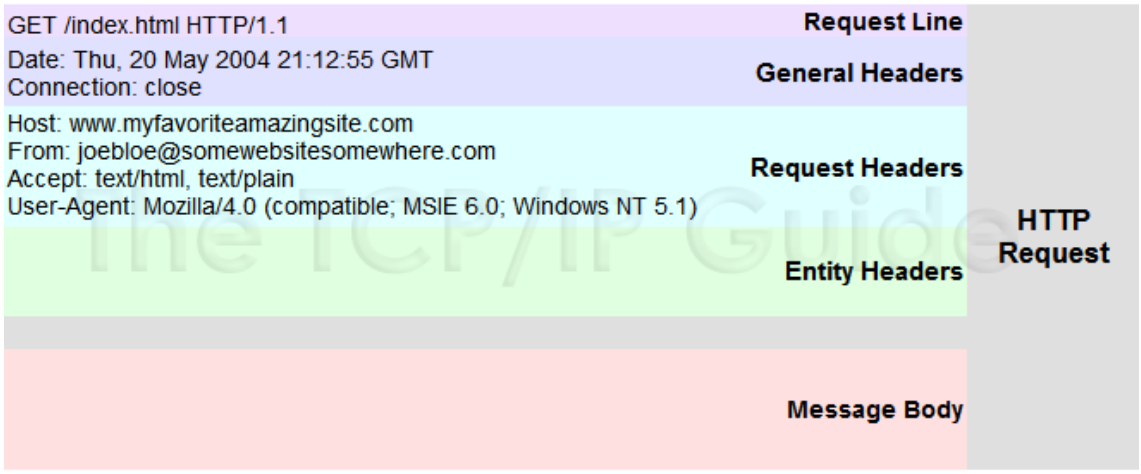
El protocolo HTTP es un protocolo sin estado, es decir, cada petición recibe una respuesta del servidor independiente de las peticiones y respuestas que se hayan producido anteriormente, aunque existen mecanismos adicionales para el mantenimiento de sesiones, como las cookies.

Cada petición está formada principalmente por los siguientes elementos:

- **Método HTTP**: método del protocolo que define la acción a realizar.
- **URL**: url relativa a la cual se realiza la petición.
- **Versión HTTP:** versión del protocolo (HTTP/1.1).
- **Cabeceras**: que contienen meta información (host, lenguaje, cliente, cookie, …).
- **Parámetros**: opcionalmente valores usados para el procesamiento en el servidor.

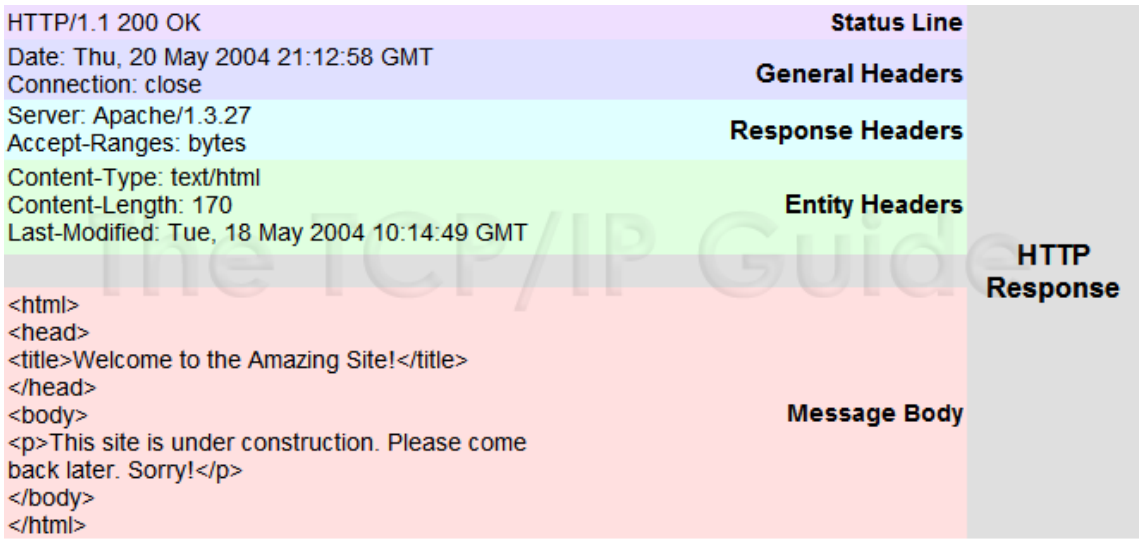
La serie de métodos disponibles en HTTP son los siguientes:

Método	Acción	Tipo	Usos
GET	Recuperar un recurso de una URL, los parámetros se incluyen en la misma URL	• Seguro • Idempotente	• Dirección en navegador • Link de página web • Formulario por defecto • Formulario method="GET"
POST	Enviar datos al servidor, la información viaja en el cuerpo de la petición		• Formulario method="POST"
PUT	Modificar un recurso de la URL	• Idempotente	
DELETE	Eliminar un recurso de la URL	• Idempotente	
HEAD	Obtener la cabecera de una respuesta (GET)	• Seguro • Idempotente	
TRACE	Obtener auto respuesta del envío, para comprobar disponibilidad		
OPTIONS	Lista de métodos disponibles para una URL		
CONNECT	Conectar en procesos de tunneling (SSL)		



Y las respuestas:

- **Versión HTTP:** versión del protocolo (HTTP/1.1).
- **Código de estado**: indica si la respuesta es satisfactoria (OK - 200) o no (404, …).
- **Cabeceras**: la respuesta también puede llevar cabeceras de control o información.
- **Tipo de contenido**: indica si la respuesta es una página, o contenido de otro tipo.
- **Contenido**: contenido según lo determinado por content-type.



HTTPS es la versión segura de HTTP, representa un protocolo seguro sobre conexión HTTP encriptada en ambos sentidos.

Utiliza SSL en la capa de transporte sobre IP. Resulta transparente a nivel de aplicación, y requiere del lado del servidor de certificados y autenticación con autoridades.

Referencias

https://www.w3.org/Protocols/

https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html

https://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html

https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

http://www.tcpipguide.com/free/t_TCPIPHypertextTransferProtocolHTTP.htm

Servlets

Un **HttpServlet** es una clase Java que se ejecuta en un servidor de aplicaciones. Los Servlets son capaces de recibir solicitudes HTTP y responder a las mismas habitualmente con contenido HTML, ya sea de forma directa o mediante algún mecanismo de redirección.

El servidor de aplicaciones es el encargado de la gestión del ciclo de vida de los Servlets que contiene, gestionando además la concurrencia de las múltiples solicitudes que pueden recibir de manera simultánea. El ciclo de vida de un Servlet comprende las siguientes fases:

1. **Creación**: se emplea el constructor de la clase Java para crear una nueva instancia del Servlet si éste se había desplegado (**deploy**) en el servidor pero aún no se encontraba en servicio.

2. **Inicialización**: tras la creación de la instancia se procede a la inicialización, este hecho ocurre una única vez a lo largo de todo el ciclo de vida del Servlet. A través del método **init** se pueden inicializar los aspectos necesarios.

3. **Servicio**: el Servlet queda a la espera de solicitudes HTTP para dar respuesta adecuada.

4. **Destrucción**: si la aplicación web asociada es replegada (**undeploy**) o el servidor se desactiva de forma ordenada todos los Servlets son destruidos, invocando antes al método **destroy** para la liberación de recursos asociados.

Un HttpServlet posee métodos para atender a las peticiones que reciba a través de un cliente web, habitualmente un navegador. Los métodos principales son los siguientes:

Método	Descripción
doGet(HttpServletRequest req, HttpServletResponse res)	Peticiones GET
doPost(HttpServletRequest req, HttpServletResponse res)	Peticiones POST

Desde cualquiera de estos métodos, o desde ambos, se define la respuesta que obtendrá la petición del usuario. Se puede establecer la salida desde el Servlet directamente indicando el tipo de contenido a devolver (no es lo más indicado por mezclar la lógica de ejecución con la presentación del resultado):

```
response.setContentType("text/html; charset=UTF-8");
PrintWriter out = response.getWriter();
try{
	out.println("HOLA<br>SERVLET");
}finally{
	out.close();
}
```

HttpServletRequest y **HttpServletResponse** representan la petición recibida y la respuesta enviada desde el Servlet respectivamente.

Peticiones

Las peticiones a un Servlet a través del navegador o cualquier otro tipo de cliente pueden incluir parámetros en su query string.

```
http://localhost:8080/Ejemplo/MiServlet?nombre=ana&edad=54
```

Para recibir un parámetro de la petición desde un Servlet se emplea el método **getParameter** del request, tanto para solicitudes GET como POST. Devuelve una cadena de caracteres con el valor del parámetro o null si no lo localiza.

```
String nombre = request.getParameter("nombre");
```

Otros métodos para recuperar valores de parámetros:

Método	**Descripción**
String[] getParameterValues(String name)	Si un parámetro tiene más de un valor
Enumeration<String> getParameterNames()	Nombres de parámetros recibidos

También es posible acceder a las cabeceras del protocolo HTTP, que incluyen información como el tipo de contenido solicitado, lenguaje, cliente que solicita, …

Método	**Descripción**
String getHeader(String name)	Obtener valor de una cabecera
Enumeration<String> getHeaderNames()	Nombres de cabeceras recibidas

Respuestas

Aunque en una arquitectura MVC la respuesta no se debe producir de forma directa desde el Servlet, es posible que necesitemos configurar un resultado de algún tipo de datos determinado empleando código Java en el Servlet.

Los métodos **setContentType** y **getWriter** del response se emplearán entonces para definir el contenido y escribir datos a la salida.

También en el caso de las respuestas se tiene acceso a las cabeceras incluidas, tanto a través de métodos genéricos, como a través de métodos específicos para algunas cabeceras destacadas:

Método	Descripción
setHeader(String name, String value)	Establecer el valor de una cabecera
setContentType(String type)	Establecer tipo MIME de respuesta
setContentLength(int length)	Indicar la longitud del cuerpo de la respuesta

Se puede emplear response también para indicar un código de estado distinto al OK (200) en caso de error o cualquier otro tipo de situación desde el servidor:

```
response.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
```

Redirecciones

A través del response es posible realizar una redirección hacia otro recurso, ya sea interno a la aplicación web o directamente una url externa:

```
response.sendRedirect("lista.jsp");
```

Este tipo de redirección es una respuesta completa al cliente, que provoca que éste tenga que realizar una segunda petición al recurso indicado. Para evitar la nueva solicitud en los Servlets se suele emplear el mecanismo de forwarding, que permite realizar todas las redirecciones internas necesarias hasta llegar al recurso que finalmente produce la respuesta:

```
request.getRequestDispatcher("lista.jsp").forward(request, response);
```

Después de cualquier tipo de redirección debemos evitar incluir ninguna línea de código.

Referencias

http://docs.oracle.com/javaee/6/tutorial/doc/bnafd.html

http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html

http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletResponse.html

JSP

Una página JSP (**Java Server Page**) es un documento que puede contener elementos HTML (también Javascript o CSS), etiquetas determinadas o código Java. Una JSP realmente es un Servlet solamente que a la hora de escribirlo se realiza en formato documento para facilitar la presentación de información en vistas de aplicaciones web.

Cuando el servidor se encuentra con una página JSP implementa un ciclo de vida similar a los Servlets:

1. **Traducción**: el documento se traduce a Servlet, incluyendo el código necesario para producir la salida de la página (out).
2. **Compilación**: el Servlet se compila para poder ejecutarlo.
3. **Carga**: carga de la clase en el servidor.
4. Creación, Inicialización, Servicio y Destrucción: son las mismas fases del ciclo de vida del Servlet.

Una página JSP puede contener los siguientes elementos principalmente:

Etiqueta	Elemento	Descripción
<%@ directiva %>	Directiva	Instrucciones para el servidor
<%! declaración %>	Declaración	Declaración de atributos o métodos
<%= expresión %>	Expresión	Expresión que se toma como un String
<% código %>	Scriptlet	Código Java
<%-- comentario --%>	Comentario	Comentario JSP, no produce salida
<jsp:action />	Acciones	Implementan determinadas acciones JSP

Las etiquetas JSP se convierten a código Java en tiempo de traducción y se ejecutan en el servidor produciendo el resultado de la página, no forman parte de éste.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<html>
<body>
	<%! int count = 0; %>
	<% count++; %>
	Bienvenido, eres el visitante número
	<%= count %>
</body>
</html>
```

Elemento	Descripción	Ejemplos

Directiva	• Instrucciones para el contenedor en tiempo de traducción
	• Pueden incluirse varias directivas, o una con varios atributos
	• Preferiblemente al inicio
	• include: inclusión estática
	• taglib: biblioteca de etiquetas
		`<%@ page language="java"%>`
		`<%@ page import="es.paq.Clase"%>`
		`<%@ page errorPage="error.jsp"%>`
		`<%@ page isErrorPage="true"%>`
		`<%@ include file="url"%>`
		`<%@ taglib uri="uri"`
		`prefix="pre"%>`

Declaración	• Declaración de atributos o métodos del Servlet	`<%! int count = 0; %>`
Expresión	• Devuelve a la salida un String • NO incluir ; al final	`<%=p.getNombre()%>`

Scriptlet	• Sentencias de código Java
	• Acceso a objetos implícitos
	• Se puede mezclar con la salida
		`<%`
		`String[] tipos =`
		`{"Zapato", "Camisa", "Falda"};`
		`for (String s: tipos){`
		`out.println(s);`
		`}`
		`%>`

Acciones	• Ejecutar acciones desde JSP • include: inclusión dinámica • forward: redirección	`<jsp:include page="listar.jsp"/>` `<jsp:forward page="listar.jsp"/>`

Objetos Implícitos

A través de las etiquetas se puede acceder a los elementos que se encuentran disponibles en una página JSP denominados objetos implícitos, los cuales representan elementos del Servlet:

Objeto	Descripción
response	Acceso a la respuesta
out	Acceso a la salida de la página
request	Acceso a la petición
page	Acceso al ámbito de la página actual
session	Acceso al ámbito de la sesión
application	Representa a ServletContext, acceso al ámbito de la aplicación
exception	Excepción en páginas de error
config	Representa a ServletConfig, acceso a parámetros con **getInitParameter**

Además del empleo de estos objetos para realizar cualquier tipo de acción necesaria a través de código, los objetos que representan un ámbito concreto permiten leer o escribir atributos. Esto permite la comunicación entre Servlets y páginas JSP a través de la transferencia de objetos Java de cualquier naturaleza.

```
//Guardar atributo desde el Servlet
request.setAttribute("persona", p);

<%-- Recoger atributo desde JSP --%>
<%
	Persona p = (Persona) request.getAttribute("persona");
%>
```

Salida en JSP

Para producir la salida dentro de un JSP se puede emplear una expresión, o utilizar el objeto out directamente desde un Scriptlet:

```
<%
	String[] tipos = {"Zapato", "Camisa", "Falda"};
	out.println("<table>");
	for (String s: tipos){
		out.println("<tr><td>");
		out.println(s);
		out.println("</td></tr>");
	}
	out.println("</table>");
%>
```

En JSP se permite realizar una mejora del código anterior en cuanto a legibilidad y simplicidad mezclando las líneas de salida que pertenecen al documento, con las líneas de código que ejecutará el Servlet:

```
<table>
	<% for (String s: tipos){ %>
		<tr>
		   <td><%=s%></td>
		</tr>
	<% } %>
</table>
```

Referencias

http://docs.oracle.com/javaee/5/tutorial/doc/bnagx.html

EL

El lenguaje EL (**Expression Language**) permite acceder a **beans** de las páginas JSP de forma sencilla, evitando en muchos casos el uso de scripts.

```
Bienvenido <strong>${usuario.nombre}</strong>
```

La utilización es muy similar al caso de las expresiones JSP, pero presenta bastantes ventajas sobre éstas:

Ventaja	Requerimiento
No necesario incluir el import de las clases	
Realiza búsqueda del bean en todos los ámbitos	
La notación . es más cómoda	La clase debe incluir getters
En caso de no localizar el bean no devuelve null	Las propiedades invocadas deben existir
Mantiene acceso a objetos implícitos	Se trata de maps que incluyen valores

Una expresión EL se puede utilizar con notación . o notación [] para acceder a las propiedades de un objeto, a los valores de un map, o a los índices de un List o array. Las expresiones se pueden anidar.

```
${p.nombre}
${p.ana}
${p["ana"]}
${p[0]}
${p["nombre"]}
${p.tipo.descripcion}
${p[nombres[0]]}
```

Operadores

Dentro de las expresiones además se pueden emplear operadores:

Tipo	Grupo	Operadores
Aritméticos	Multiplicación	* / div % mod
	Adición	+ -
Lógicos	Unarios	! not
	Relacionales	< lt > gt <= le >= ge
	Igualdad	== eq != ne empty
	Binarios	&& and || or

Objetos Implícitos

Los objetos implícitos en EL son maps que contienen los valores o atributos que se pueden examinar sobre cada tipo de elemento, no son el objeto Java en sí mismo:

Tipo de objeto	Objeto
Ámbito	pageScope requestScope sessionScope applicationScope
Parámetro	param paramValues
Cabecera	header headerValues
Cookie	cookie
Configuración	initParam
Acceso a objetos de la página	pageContext

El objeto **pageContext** permite acceder a los objetos implícitos JSP (request, session, …). Estos objetos permiten invocar métodos necesarios para obtener información de interés en la página:

```
<%-- Incluir ruta de contexto en el enlace --%>
<a href="${pageContext.request.contextPath}/guardar">...
```

Referencias

http://docs.oracle.com/javaee/5/tutorial/doc/bnahq.html

JSTL

Java Standard Tag Library es una API perteneciente a la especificación Java EE que define una serie de etiquetas simples que encapsulan la funcionalidad principal que es usada comúnmente para escribir páginas JSP, evitando el uso de Scriptlets. Para poder emplear JSTL en una aplicación web Java es necesario incluirla en el path del mismo.

El contenido JSP de una página, aún incluyendo un código muy concreto que realiza poca funcionalidad presenta una serie de problemas:

- El código de Scriptlets es desordenado.
- Actualizar el diseño de una página por parte de terceros es complejo debido al tipo de contenidos al cual se enfrentan.
- El código dentro de una página no es reutilizable por otras páginas JSP.
- La lógica de recuperación de información y procesamiento requiere cierta complejidad e importación de los elementos relacionados desde las JSP.

Las etiquetas JSTL se definen en formato XML, lo cual permite dotar de mucha más claridad y orden a las páginas en las cuales se incluye, integrando además EL para acceder a los objetos necesarios.

La API JSTL incluye cuatro paquetes de etiquetas:

Paquete	Uso	URI
core	Funciones básicas	http://java.sun.com/jsp/jstl/core
fmt	Formato e internacionalización	http://java.sun.com/jsp/jstl/fmt
xml	Procesamiento XML	http://java.sun.com/jsp/jstl/xml
sql	Acceso a bases de datos	http://java.sun.com/jsp/jstl/sql

Para poder emplear un paquete JSTL (xml y sql no recomendados para la capa de vista), es necesario incluir una directiva taglib al comienzo de la página:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
```

Paquete core

El paquete core incluye acciones que facilitan la inclusión de lógica en la página JSP. En la siguiente tabla se encuentran todas las etiquetas de este paquete, que incluyen acciones de propósito general, etiquetas para establecer condiciones lógicas o iteraciones, y etiquetas que permiten definir acciones concretas sobre urls, de forma complementaria a las acciones JSP o directivas ya comentadas para tales efectos.

Etiqueta	Uso	Ejemplo
c:out	Devolver el valor de una expresión a la salida	`<c:out value="${x}"/>`
c:set	Establecer el valor de una variable o propiedad. Permite definir el ámbito	`<c:set var="hello" value="Hola!"/>`
c:remove	Eliminar una variable	`<c:remove var="hello"/>`

c:catch	Captura un Throwable. Permite definir una variable que contendrá el elemento capturado	`<c:catch var="ex">`
		`  ...`
		`</c:catch>`

c:if	Sentencia condicional. Permite definir una variable que contendrá el resultado evaluado	`<c:if test="${empty p}">`
		`  ...`
		`</c:if>`

c:choose	Define condiciones exclusivas (if…else if)	

c:when	Condiciones incluidas en un c:choose	`<c:choose>`
		`  <c:when`
		`  test="${x eq 2}">`
		`  ...`
		`  </c:when>`
		`  ...`
		`</c:choose>`

c:otherwise	Alternativa por defecto para c:choose	

c:forEach	Permite iterar sobre colecciones, iteradores, arrays, enumeraciones o cadena de caracteres separada por comas. Permite definir inicio, fin y salto del bucle, así como una varStatus para examinar el estado de cada iteración	`<c:forEach`
		`  items="${lista}"`
		`  var="p">`
		`  ... ${p.nombre}`
		`</c:forEach>`

c:forTokens	Iterar sobre un conjunto de tokens separados por un delimitador	`<c:forTokens`
		`  items="${cadena}"`
		`  delims="-" var="t">`
		`  ...`
		`</c:forTokens>`

c:redirect	Realiza un redirect completo	`<c:redirect` `  url="lista.jsp"/>`
c:import	Importar un recurso desde URL, como cadena de caracteres o como Reader	`<c:import url="..."` `  var="c"/>`
c:url	Construir una URL con reescritura	`<c:url` `  value="lista.jsp"/>`
c:param	Indicar parámetros para las etiquetas con acciones de url	`<c:param name="name"` `  value="value"/>`

Paquete fmt

En el paquete fmt existe una serie de etiquetas destinadas a internacionalización que permiten definir recursos de idiomas, y también otras etiquetas para parseo y formato de valores como números o fechas. Las etiquetas de formato son las siguientes:

Etiqueta	Uso	Ejemplo
fmt:formatNumber	Formatear un valor numérico con estilo o patrón. Permite definir una variable que contendrá el valor formateado	`<fmt:formatNumber value="1212.21" type="currency"/>` `<fmt:formatNumber value="1212.21" pattern="#,#00.0#"/>`
fmt:parseNumber	Parsear un valor numérico a través de un formato determinado. Permite definir una variable que contendrá el valor parseado	`<fmt:parseNumber value="${valor}" type="currency"/>`
fmt:formatDate	Formatear una fecha con estilo o patrón. Permite la definición de una variable que contendrá el valor formateado	`<fmt:formatDate value="${now}" timeStyle="long" dateStyle="long"/>` `<fmt:formatDate value="${now}" pattern="dd/MM/yy"/>`
fmt:parseDate	Parsear un valor de fecha a través de un formato determinado. Permite definir una variable que contendrá el valor parseado	`<fmt:parseDate value="12/12/12" pattern="dd/MM/yy" var="fecha" />`

Para emplear este tipo de etiquetas es necesario incluir en la página JSP la directiva taglib correspondiente:

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
```

Referencias

http://www.oracle.com/technetwork/java/index-jsp-135995.html

Sesiones

El protocolo HTTP es un protocolo sin estado. Para poder enlazar las acciones que se producen entre distintas peticiones es necesario manejar algún mecanismo artificial:

- **Campos hidden**: no es la mejor solución por la necesidad de acumular información oculta para ser enviada de forma automática por los formularios de la aplicación.
- **Cookies**: representa un mecanismo válido de mantenimiento de sesión muy empleado en HTTP, ya que el servidor incluye un archivo con la información de la sesión que viaja automáticamente en las sucesivas llamadas que se produzcan al mismo.
- **Sesiones**: HttpSession es un interfaz Java creado para evitar la necesidad de trabajar a bajo nivel con las cookies, aunque basado en este mecanismo y que permite además la inclusión de cualquier tipo de objeto Java para la transmisión de información.
- **Reescritura de URLs**: alternativa para el uso de sesiones sin necesidad del empleo directo o indirecto de cookies. Basado en la inclusión de un identificador en todas las llamadas que se produzcan al servidor.

Trabajar con HttpSession es muy sencillo, simplemente se necesita acceder a la sesión actual y almacenar un atributo para que ya se encuentre disponible en cualquier otro elemento de la aplicación web que también acceda a la misma:

```
//Obtiene la sesión actual, si no existe la crea
HttpSession session = request.getSession();
session.setAttribute("carrito", carrito);
```

Todos los elementos que se almacenen en la sesión se encontrarán disponibles mientras ésta se encuentre vigente. Una sesión se destruye en los siguientes casos:

- Se cierra el navegador.
- Se agota el timeout de la sesión (configurable en el descriptor o el servidor).
- Se cierra la sesión a través de código.

```
//Eliminar el atributo solamente
session.removeAttribute("carrito");
//Invalidar sesión completa
session.invalidate();
```

Referencias

http://docs.oracle.com/javaee/6/api/?javax/servlet/http/HttpSession.html