Java
Curso Básico

Francisco Philip

francisco.philip@gmail.com
@franciscophilip

Temario

Temario

- Introducción a java
- Clases y objetos
- Identificadores, palabras clave y tipos
- Operadores y control de flujo
- Arrays
- Clases avanzadas
- Excepciones y aserciones
- Colecciones y genéricos
- Garbage Collector
- Hilos

Introducción a Java

Introducción a Java

- Qué es Java
- Características de java
- JRE y JDK
- Ediciones
- Garbage Collector

Características de Java

Java es un lenguaje de programación creado por Sun Microsystems, (empresa que posteriormente fue comprada por Oracle) para poder funcionar en distintos tipos de procesadores.

Su sintaxis es muy parecida a la de C o C++, e incorpora como propias algunas características que en otros lenguajes son extensiones: gestión de hilos, ejecución remota, etc.

El código Java, una vez compilado, puede llevarse sin modificación alguna sobre cualquier máquina, y ejecutarlo. Esto se debe a que el código se ejecuta sobre una máquina hipotética o virtual, la Java Virtual Machine, que se encarga de interpretar el código (ficheros compilados .class) y convertirlo a código particular de la CPU que se esté utilizando (siempre que se soporte dicha máquina virtual).

Según Wikipedia

Java es un lenguaje de programación de propósito general, concurrente, orientado a objetos, que fue diseñado específicamente para tener tan pocas dependencias de implementación como fuera posible. Su intención es permitir que los desarrolladores de aplicaciones escriban el programa una vez y lo ejecuten en cualquier dispositivo (conocido en inglés como WORA, o "write once, run anywhere"), lo que quiere decir que el código que es ejecutado en una plataforma no tiene que ser re compilado para correr en otra.

Java es, a partir de 2012, uno de los lenguajes de programación más populares en uso, particularmente para aplicaciones de cliente-servidor de web, con unos diez millones de usuarios reportados. (2009)

El lenguaje de programación Java fue originalmente desarrollado por James Gosling, de Sun Microsystems (constituida en 1982 y posteriormente adquirida el 27 de enero de 2010 por la compañía Oracle),4 y publicado en 1995 como un componente fundamental de la plataforma Java de Sun Microsystems. Su sintaxis deriva en gran medida de C y C++, pero tiene menos utilidades de bajo nivel que cualquiera de ellos.

Las aplicaciones de Java son compiladas a bytecode (clase Java), que puede ejecutarse en cualquier máquina virtual Java (JVM) sin importar la arquitectura de la computadora subyacente.

Características de Java

Características de Java

Java tiene características que lo definen y lo han hecho uno de los lenguajes más utilizados.

- Orientado a objetos
- Independencia de la plataforma
- El recolector de basura o garbage collector (gc)

Orientado a objetos

- La primera característica, orientado a objetos se refiere a un método de programación y al diseño del lenguaje.
- Aunque hay muchas interpretaciones para OO, una primera idea es diseñar el software de forma que los distintos tipos de datos que usen estén unidos a sus operaciones.
- Así, los datos y el código (funciones o métodos) se combinan en entidades llamadas objetos. Un objeto puede verse como un paquete que contiene el “comportamiento” (el código) y el “estado” (datos).
- El principio es separar aquello que cambia de las cosas que permanecen inalterables, frecuentemente, cambiar una estructura de datos implica un cambio en el código que opera sobre los mismos, o viceversa.

- Esta separación en objetos coherentes e independientes ofrece una base más estable para el diseño de un sistema software.
- El objetivo es hacer que grandes proyectos sean fáciles de gestionar y manejar, mejorando como consecuencia su calidad y reduciendo el número de proyectos fallidos.

Independencia de la plataforma

- La característica de la independencia de la plataforma en Java, significa que programas escritos en este lenguaje pueden ejecutarse igualmente en cualquier tipo de hardware.
- Este es el significado de ser capaz de escribir un programa una vez y que pueda ejecutarse en cualquier dispositivo, tal como reza el axioma de Java, ‘’’write once, run anywhere’’’.
- Para ello, se compila el código fuente escrito en lenguaje Java, para generar un código conocido como “bytecode” (específicamente Java bytecode)—instrucciones máquina simplificadas específicas de la plataforma Java. Esta pieza está “a medio camino” entre el código fuente y el código máquina que entiende el dispositivo destino. El bytecode es ejecutado entonces en la máquina virtual (JVM), un programa escrito en código nativo de la plataforma destino (que es el que entiende su hardware), que interpreta y ejecuta el código.

- Además, se suministran bibliotecas adicionales para acceder a las características de cada dispositivo (como los gráficos, ejecución mediante hebras o threads, la interface de red) de forma unificada y aunque hay una etapa explícita de compilación, el bytecode generado es interpretado o convertido a instrucciones máquina del código nativo por el compilador JIT (Just In Time).
- El concepto de independencia de la plataforma de Java cuenta, con un gran éxito en las aplicaciones en el entorno del servidor, como los Servicios Web, los Servlets, los Java Beans, así como en sistemas empotrados basados en OSGi.

El recolector de basura o garbage collector (gc)

- En Java gran parte de los los memory leaks se evita en gran medida gracias a la recolección de basura (o automatic garbage collector).
- El programador determina cuándo se crean los objetos y el entorno en tiempo de ejecución de Java (Java runtime) es el responsable de gestionar el ciclo de vida de los objetos.
- El programa, u otros objetos, pueden tener localizado un objeto mediante una referencia a éste, cuando no quedan referencias a un objeto, el recolector de basura de Java borra el objeto, liberando así la memoria que ocupaba previniendo posibles fugas.
- Aun así, es posible que se produzcan fugas de memoria si el código almacena referencias a objetos que ya no son necesarios.

Programación Orientado a Objetos

Conceptos Fundamentales

Java es un lenguaje orientado a objetos, por lo cual Java admite los siguientes conceptos fundamentales:

- Polimorfismo
- Herencia
- Encapsulación
- Abstracción
- Clases
- Objetos
- Método

Ventajas

Reusabilidad:
Cuando hemos diseñado adecuadamente las clases, se pueden usar en distintas partes del programa y en numerosos proyectos.

Mantenibilidad:
Debido a las sencillez para abstraer el problema, los programas orientados a objetos son más sencillos de leer y comprender, pues nos permiten ocultar detalles de implementación dejando visibles sólo aquellos detalles más relevantes.

Modificabilidad:
La facilidad de añadir, suprimir o modificar nuevos objetos nos permite hacer modificaciones de una forma muy sencilla.

Fiabilidad:
Al dividir el problema en partes más pequeñas podemos probarlas de manera independiente y aislar mucho más fácilmente los posibles errores que puedan surgir.

Desventajas o inconvenientes

La programación orientada a objetos presenta también algunas desventajas como pueden ser:

- Cambio en la forma de pensar de la programación tradicional a la orientada a objetos.
- La ejecución de programas orientados a objetos es más lenta.
- La necesidad de utilizar bibliotecas de clases obliga a su aprendizaje y entrenamiento.

Ediciones

Java Ediciones

La plataforma Java, posee diferentes tipos de ediciones (no confundir con versiones). Cada una de estas ediciones fueron desarrolladas para atacar ciertos problemas sobre ambientes en particular.

Java es distribuido en tres diferentes ediciones.

- Java Micro Edition (Java ME)
- Java Standard Edition (Java SE)
- Java Enterprise Edition (Java EE)

Java Micro Edition (Java ME)

Java Micro Edition, también conocido como Java ME, es una versión reducida de la edición Java Standard Edition. Esta edición se encuentra enfocada para la creación de aplicaciones tanto en dispositivos móviles, como dispositivos integrados.

Con Java ME nosotros podemos desarrollar aplicaciones para diferentes dispositivos, no limitándose únicamente a teléfonos inteligente. Si así lo deseamos podemos crear aplicaciones para Televisores inteligentes, consolas de vídeo juegos, etc ...

Aunque su popularidad se vio reducida por el auge de Android, hoy en día se encuentra retomando terreno principalmente por el tema del Internet de las cosas.

Java Standard Edition (Java SE)

Java Standard Edition, también conocido como Java SE, es la edición estándar de Java, la versión original de Sun Microsystems. Con esta versión nosotros podemos crear tanto aplicaciones web, como aplicaciones de escritorio.

La edición cuenta con una amplia biblioteca de clases las cuales están pensadas para agilizar el proceso de desarrollo. Tenemos clases enfocadas en seguridad, red, acceso a base de datos, interfaces gráficas, conexión entre dispositivos, XML y otros.

Si tú quieres comenzar a desarrollar aplicaciones con Java es obligatorio que instales y comiences con esta edición, pues será esta, la que te provee de una base sólida del lenguaje, tocando temas como Java Virtual Machine, Java Runtime Environment, Java Development Kit, y API de Java. Expliquemos cada uno de ellos.

Java Enterprise Edition (Java EE)

Java Enterprise Edition, también conocido como Java EE, es la edición más grande de Java. Esta edición contiene toda la Standard Edition y mucho más. Por lo general es utiliza para crear aplicaciones con la arquitectura cliente servidor.

Java EE fue pensado para el mundo empresarial.

- Es portable y escalable.
- Posee una amplia biblioteca de clases con las cuales podemos trabajar con JSON, Email, base de datos, transacciones, Persistencia, envió de mensajes, etc.

JRE y JDK

Java Runtime Environment (JRE)

Java Runtime Environment (JRE) es un conjunto de herramientas que proporcionan un entorno en donde las aplicaciones Java pueden ser ejecutadas.

Cuando un usuario desea ejecutar un programa Java, este debe elegir el entorno que se adecue a sus necesidades (arquitectura y sistema operativo de la computadora).

Java Development Kit (JDK)

El Java Development Kit (JDK) es una extensión de JRE. Junto con los archivos y herramientas proporcionados por JRE, el JDK incluye compiladores y herramientas (como JavaDoc y Java Debugger) para crear programas Java.

Por esta razón, cuando uno quiere desarrollar una aplicación Java, necesitan instalar un JDK.

API Java

Java SE provee a una amplia biblioteca de clases las cuales están pensadas para agilizar nuestro proceso de desarrollo, son clases las cuales ya vienen con el lenguaje.

A esta biblioteca de clases se le denomina la API de JAVA. Y esta puede ser consultada de forma gratuita en la documentación oficial de Java.

https://docs.oracle.com/javase/8/docs/api/index.html?overview-summary.html

Compilador

Compilador

El compilador que forma parte del kit de desarrollo java (JDK) que es capaz de traducir el código fuente Java en posee múltiples argumentos de ejecución que son detallados en la documentación de oracle o de open jdk.

https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javac.html
https://docs.oracle.com/javase/tutorial/essential/environment/cmdLineArgs.html

Java Virtual Machine

Java Virtual Machine

Una máquina virtual Java (en inglés Java Virtual Machine, JVM) es una máquina virtual de proceso nativo, es decir, ejecutable en una plataforma específica, capaz de interpretar y ejecutar instrucciones expresadas en un código binario especial (el bytecode Java), el cual es generado por el compilador del lenguaje Java.

El código binario de Java no es un lenguaje de alto nivel, sino un verdadero código máquina de bajo nivel, viable incluso como lenguaje de entrada para un microprocesador físico. Como todas las piezas del rompecabezas Java, fue desarrollado originalmente por Sun.

La JVM es una de las piezas fundamentales de la plataforma Java. Básicamente se sitúa en un nivel superior al hardware del sistema sobre el que se pretende ejecutar la aplicación, y este actúa como un puente que entiende tanto el bytecode como el sistema sobre el que se pretende ejecutar. Así, cuando se escribe una aplicación Java, se hace pensando que será ejecutada en una máquina virtual Java en concreto, siendo ésta la que en última instancia convierte de código bytecode a código nativo del dispositivo final.

La JVM tiene instrucciones para los siguientes grupos de tareas:

- Carga y almacenamiento
- Aritmética
- Conversión de tipos
- Creación y manipulación de objetos
- Gestión de pilas (push y pop)
- Transferencias de control (branching)
- Invocación y retorno a métodos
- Excepciones

Ficheros java

Reglas de los ficheros Java

- Si las declaraciones de importación están presentes, entonces deben escribirse entre el extracto del paquete y la declaración de la clase. Si no hay instrucciones de paquete, la instrucción de importación debe ser la primera línea en el archivo fuente.
- Las sentencias **import** y **package** implicarán a todas las clases presentes en el archivo fuente. No es posible declarar diferentes declaraciones de importación y / o paquete a diferentes clases en el archivo fuente. Las instrucciones package e import se aplican a todas las clases del archivo.
- Un archivo puede tener más de una clase no pública.
- Los archivos sin clases public no tienen restricciones de nombres.

El método `main`

Ejecución de código

El método main proporciona el mecanismo para controlar la aplicación. Cuando se ejecuta una clase Java el sistema localiza y ejecuta el método main de esa clase.

El método main es el punto de partida de cualquier aplicación en Java.

En una aplicación Java, su clase principal debe contener el método main puesto que es necesario para ejecutar la aplicación de manera adecuada.

Éste método siempre se declara de la siguiente manera,
dentro de la clase:

```
public static void main(String[] args){
}
```

La clase `System`

System

El uso de Java System Class es algo de utilizamos muy a menudo , normalmente invocando `System.out.println(“hola mundo”)` o algo muy similar. Estamos muy acostumbrados a usar esta clase , pero muchas veces no entendemos a detalle que operaciones realiza.

La clase `System` pertenece al package `java.lang` y dispone de varias variables estáticas a utilizar. Las más utilizadas variables son `in`, `out` y `err` que hacen referencia a la entrada ,salida y manejo de errores respectivamente. De ahí que podamos invocar sin problema.

```
System.out.println(“hola mundo”)
```

Los métodos

La clase `System` tiene otros métodos muy útiles ya que es la encargada de interactuar en el sistema. Por ejemplo nos permite acceder a la propiedad de Java home, al directorio actual o la versión de Java que tenemos.

```
System.out.println(System.getProperty("java.home"));
```

Además de muchos otros métodos como arrayCopy(), para copiar arrays, urrentTimeMillis(), para obtener el tiempo en milisegundos o exit() que trmina el programa Java.

Se recomienda revisar el JavaDoc en

https://docs.oracle.com/javase/8/docs/api/java/lang/System.html

Herencia y polimorfismo

Herencia

Con la herencia podemos definir una clase a partir de otra que ya exista, de forma que la nueva clase tendrá todas las variables y métodos de la clase a partir de la que se crea, más las variables y métodos nuevos que necesite. A la clase base a partir de la cual se crea la nueva clase se le llama superclase.

Java permite el empleo de la herencia, característica muy potente que permite definir una clase tomando como base a otra clase ya existente. Esto es una de las bases de la reutilización de código, en lugar de copiar y pegar.

Por ejemplo, podríamos tener una clase genérica Animal, y heredamos de ella para formar clases más específicas, como Pato, Elefante, o León. Estas clases tendrán todo lo de la clase padre Animal, y además cada una podría tener sus propios elementos adicionales.

Una característica derivada de la herencia es que, por ejemplo, si tenemos un método dibuja (Animal a), que se encarga de hacer un dibujo del animal que se le pasa como parámetro, podremos pasarle a este método como parámetro tanto un Animal como un Pato, Elefante, o cualquier otro subtipo directo o indirecto de Animal.

Esto se conoce como polimorfismo.

Java permite el empleo de la herencia, característica muy potente que permite definir una clase tomando como base a otra clase ya existente. Esto es una de las bases de la reutilización de código, en lugar de copiar y pegar.

Herencia

- La herencia se puede definir como el proceso en el que una clase adquiere las propiedades (métodos y campos) de otra. Con el uso de la herencia, la información se hace manejable en un orden jerárquico.
- La clase que hereda las propiedades de otra se conoce como subclase (clase derivada, clase hija) y la clase cuyas propiedades se heredan se conoce como superclase (clase base, clase principal).
- La herencia permite que una clase sea una subclase de una superclase y por lo tanto hereda variables protegidas y métodos de la superclase, es un concepto clave que subyace al IS-A, el polimorfismo, la sobrecarga y el “casting”.
- Todas las clases (excepto la clase Object) son subclases de tipo Object, y por lo tanto heredan los métodos de “Object”

Abstracción

Del mismo modo, en la programación orientada a objetos, la abstracción es un proceso de ocultar los detalles de implementación del usuario, solo la funcionalidad se proporcionará al usuario. En otras palabras, el usuario tendrá la información sobre lo que hace el objeto en lugar de cómo lo hace.

Por ejemplo, cuando considera el caso del correo electrónico, detalles complejos como lo que sucede tan pronto como envía un correo electrónico, el protocolo que usa su servidor de correo electrónico está oculto para el usuario.

En Java, la abstracción se logra usando clases abstractas e interfaces.

Encapsulado y Subclases

- Encapsulado es uno de los cuatro conceptos fundamentales de OOP. Los otros tres son herencia, polimorfismo y abstracción.
- El encapsulado en Java es un mecanismo para envolver los datos (variables) y el código que actúa sobre los datos (métodos) juntos como una sola unidad.
- En el encapsulado, las variables de una clase se ocultan de otras clases, y solo se puede acceder a ellas a través de los métodos de su clase actual. Por lo tanto, también se conoce como ocultación de datos.
- Los campos de una clase se pueden hacer de solo lectura o sólo escritura.
- Una clase puede tener control total sobre lo que está almacenado en sus campos.

Encapsulado

- IS-A es indicativo de herencia (extends) o implementación (implements).
- IS-A, “hereda de,” y “es subtipo de”.
- HAS-A nos indicativo una instancia de una clase "tiene una" referencia a una instancia de otra clase u otra instancia de la misma clase.

Identificadores, palabras clave y tipos

Identificadores, palabras clave y tipos

- Reglas para nombrar un identificador
- Palabras clave
- Tipos primitivos
- Tipos Objeto

Reglas para nombrar un identificador

Cuando vayamos a dar un nombre a una variable deberemos de tener en cuenta una serie de normas. Es decir, no podemos poner el nombre que nos dé la gana a una variable.

Los identificadores son secuencias de texto unicode, sensibles a mayúsculas cuya primer carácter sólo puede ser una letra, símbolo $ o subrayado _ . Si bien es verdad que el símbolo dólar no es utilizado por convención.

Es recomendable que los nombres de los identificadores sean legibles y no acrónimos que no podamos leer. De tal manera que a la hora de verlos se auto-documenten por sí mismos. Además estos identificadores nunca podrán coincidir con las palabras reservadas.

Algunas reglas no escritas, pero que se han asumido por convención son:

- Los identificadores siempre se escriben en minúsculas. (pe. nombre). Y si son dos o más palabras, el inicio de cada siguiente palabra se escriba en mayúsculas (pe. nombrePersona)
- Si el identificador implica que sea una constante. Es decir que hayamos utilizado los modificadores final static, dicho nombre se suele escribir en mayúsculas (pe. LETRA). Y si la constante está compuesta de dos palabras, estas se separan con un subrayado (pe. LETRA_PI).

Palabras clave

En Java existen pocas palabras clave dentro del lenguaje.

abstract	boolean	break	byte	case	catch
char	class	const	continue	default	do
double	else	extends	final	finally	float
for	goto	if	implements	import	instanceof
int	interface	long	native	new	package
private	protected	public	return	short	static
strictfp	super	switch	synchronized	this	throw
throws	transient	try	void	volatile	while
assert	enum				

Operadores y control de flujo

Operadores y control de flujo

- Operadores de asignación
- Operadores matemáticos
- Operadores incremento y decremento
- Operadores bit a bit
- Operadores lógicos
- Operadores relacionales
- Condicional if -else
- Switch-case

- Bucle for
- Bucle for - each
- Bucle while
- Bucle do-while
- Sentencias break y continue

Operador de asignación

El operador asignación =, es un operador binario que asigna el valor del término de la derecha al operando de la izquierda. El operando de la izquierda suele ser el identificador de una variable. El término de la derecha es, en general, una expresión de un tipo de dato compatible; en particular puede ser una constante u otra variable. Como caso particular, y a diferencia de los demás operadores, este operador no se evalúa devolviendo un determinado valor.

```
a = b;
```

Operadores matemáticos

El lenguaje de programación Java tiene varios operadores aritméticos para los datos numéricos enteros y reales.

```
-    Operador unario de cambio de signo

+    Suma

-    Resta

*    Producto

/    División, entera y decimal

%    Resto de la división entera
```

Hay que tener en cuenta que el resultado exacto depende de los tipos de operando involucrados. Para ello se recomienda atender a las siguientes peculiaridades:

- El resultado es de tipo long si, al menos, uno de los operandos es de tipo long y ninguno es real (float o double).
- El resultado es de tipo int si ninguno de los operandos es de tipo long y tampoco es real (float o double).
- El resultado es de tipo double si, al menos, uno de los operandos es de tipo double.
- El resultado es de tipo float si, al menos, uno de los operandos es de tipo float y ninguno es double.

- El formato empleado para la representación de datos enteros es el complemento a dos. En la aritmética entera no se producen nunca desbordamientos (overflow) aunque el resultado sobrepase el intervalo de representación (int o long).
- La división entera se trunca hacia 0. La división o el resto de dividir por cero es una operación válida que genera una excepción ArithmeticException que puede dar lugar a un error de ejecución y la consiguiente interrupción de la ejecución del programa.
- La aritmética real (en coma flotante) puede desbordar al infinito (demasiado grande, overflow) o hacia cero (demasiado pequeño, underflow).
- El resultado de una expresión inválida, por ejemplo, dividir infinito por infinito, no genera una excepción ni un error de ejecución: es un valor NaN (Not a Number).

Operadores incremento y decremento

Los operadores aritméticos incrementales son operadores unarios (un único operando). El operando puede ser numérico o de tipo char y el resultado es del mismo tipo que el operando. Estos operadores pueden emplearse de dos formas dependiendo de su posición con respecto al operando.

Operadores aritméticos combinados

Combinan un operador aritmético con el operador asignación. Como en el caso de los operadores aritméticos pueden tener operandos numéricos enteros o reales y el tipo específico de resultado numérico dependerá del tipo de éstos.

Operadores bit a bit

Tienen operandos de tipo entero (o char) y un resultado de tipo entero. Realizan operaciones con dígitos (ceros y unos) de la representación binaria de los operandos. Exceptuando al operador negación, los demás operadores son binarios.

`~`	Negación o complemento unario
`|`	Suma lógica binaria
`^`	Suma lógica exclusiva (xor)
`&`	Producto lógico (and binario)
`<<`	Desplazamiento de A a la izquierda en B posiciones
`>>`	Desplazamiento de A a la derecha en B posiciones, tiene en cuenta el signo.
`>>>`	Desplazamiento de A a la derecha en B posiciones, no tiene en cuenta el signo.

Operadores lógicos

Realizan operaciones sobre datos booleanos y tienen como resultado un valor booleano. En la siguiente tabla se resumen los diferentes operadores de esta categoría.

`!`	Negación (not unario)
`|`	Suma lógica (or binario)
`^`	Suma lógica exclusiva (xor binario)
`&`	Producto lógico (and binario)
`||`	Suma lógica con cortocircuito.
`&&`	Producto lógico con cortocircuito

Para mejorar el rendimiento de ejecución del código es recomendable emplear en las expresiones booleanas el operador && en lugar del operador &.

Operadores relacionales

Realizan comparaciones entre datos compatibles de tipos primitivos (numéricos, carácter y booleanos) teniendo siempre un resultado booleano. Los operandos booleanos sólo pueden emplear los operadores de igualdad y desigualdad.

`>`	Mayor que
`>=`	Mayor o igual que
`<`	Menor que
`<=`	Menor o igual que
`==`	Igual
`!=`	Distinto de

Condicional if -else

La sentencia if es la declaración de toma de decisiones más simple. Se usa para decidir si una determinada declaración o bloque de enunciados se ejecutará o no; es decir, si una determinada condición es verdadera (true), se ejecutará un bloque de enunciado y, de ser falsa (false), no.

En el caso de querer ejecutar sentencia en el caso contrario, el lenguaje nos permite usar la sentencia else;

```
if (condición)
    declaración;
else if (condición)
    declaración;
```

Operador ternario

El operador ternario es una funcionalidad que está disponible en la mayoría de los lenguajes de programación, como por ejemplo C, C++ Y PHP entre otros.

Este operador toma 3 argumentos y retorna un valor, los cuales pueden ser de diferentes tipos, por lo general en los lenguajes de programación solamente se utiliza el operador ternario ?: para reemplazar la estructura de control IF.

```
expresionLogica ? expresion_1 : expresion_2
```

Switch-case

La instrucción switch es una declaración de bifurcación de múltiples vías (selección múltiple). Proporciona una forma sencilla de enviar la ejecución a diferentes partes del código en función del valor de la expresión.

- La expresión puede ser de tipo byte, short, int, char o una enumeración. A partir de JDK7, la expresión también puede ser de tipo String.
- Los valores duplicados de case no están permitidos.
- La declaración predeterminada default es opcional.
- La declaración de interrupción break; se usa dentro del switch para finalizar una secuencia de instrucción.
- La declaración break; es opcional. Si se omite, la ejecución continuará en el siguiente case.

```
switch (expresión) {
  case valor1:
      declaracion1;
      break;
  case value2:
      declaracion2;
      break;
  .   .
  case valorN:
      declaracionN;
      break;
  default:
      declaracionDefault
;
}
```

Bucles

Hasta Java 5 para hacer un bucle desde 0 a N elementos había que usar una variable para mantener un contador, hacer una comparación para comprobar si se había llegado al límite e incrementar la variable en la siguiente ejecución. El código era bastante verboso y dado que los bucles son una construcción básica de cualquier lenguaje de programación es empleada numerosas veces en cualquier algoritmo.

```
for (int i = 0; i < 5; ++i) {
    System.out.println(i);
}
```

Bucle for

La instrucción `for` proporciona una forma compacta de iterar sobre un rango de valores. Los programadores a menudo se refieren a él como el "bucle for" debido a la forma en que se repite repetidamente hasta que se cumple una condición particular. La forma general de la declaración for se puede expresar de la siguiente manera:

```
for (inicialización; terminación    incremento)
    sentencia / bloque de sentencias
```

Cuando use esta versión de la declaración for, tenga en cuenta que:

- La expresión de inicialización inicializa el bucle; Se ejecuta una vez, como comienza el bucle.
- Cuando la expresión de terminación se evalúa como falsa, el bucle termina.
- La expresión de incremento se invoca después de cada iteración a través del bucle; es perfectamente aceptable que esta expresión incremente o disminuya un valor.

Bucle for - each

En Java 5 el bucle for se enriqueció notablemente, con el foreach se puede recorrer una colección y cualquier objeto que implemente la interface Iterable. Con el bucle foreach una Collection se recorre de la siguiente manera.

```
for (int i : new int[]{8, 1, 4, 3, 422}) {
    System,out.println(“i?”+i);
}
```

Bucle while

La instrucción while continuamente ejecuta un bloque de instrucciones mientras que una condición particular es verdadera. Su sintaxis se puede expresar como:

```
while (expresión)
    sentencia / bloque de sentencias
```

La instrucción while evalúa la expresión, que debe devolver un valor booleano. Si la expresión se evalúa como verdadera, la instrucción while ejecuta la sentencia o bloquee. La instrucción while continúa probando la expresión y ejecutando su bloque hasta que la expresión se evalúa como falsa

Bucle do-while

El lenguaje de programación Java también proporciona una declaración de "`do-while`", que se puede expresar de la siguiente manera:

```
do
    sentencia / bloque de sentencias
while (expresión) ;
```

La diferencia entre do-while y while es que do-while evalúa su expresión en la parte inferior del bucle en lugar de la parte superior.

Sentencias break y continue

Java admite tres declaraciones de salto: break, continue y return. Estas tres declaraciones transfieren el control a otra parte del programa.

break se utiliza principalmente para:

- Terminar una secuencia en una instrucción switch (discutida arriba).
- Para salir de un bucle
- Como una forma “civilizada” de goto.
- Usar break para salir de un bucle

Utilizando el break, podemos forzar la terminación inmediata de un bucle, evitando la expresión condicional y cualquier código restante en el cuerpo del bucle.

Java no tiene una declaración goto porque proporciona una forma de bifurcar de manera arbitraria y no estructurada. Java usa etiquetas. Una etiqueta se usa para identificar un bloque de código.

continue

A veces es útil forzar una iteración temprana de un bucle. Es decir, es posible que desee continuar ejecutando el bucle, pero deje de procesar el resto del código (en su cuerpo) para esta iteración en particular. Esto es, en efecto, un goto pasando del cuerpo del bucle, al final del bucle. La instrucción continue realiza tal acción.

Tipos de datos

Tipos de datos

Todo programa de ordenador persigue ofrecer una funcionalidad determinada para la que, por regla general, necesitará almacenar y manipular información.

Dicha información, que son los datos sobre los que operaremos, deben almacenarse temporalmente en la memoria del ordenador. Para poder almacenar y recuperar fácilmente información en la memoria de un ordenador los lenguajes de programación ofrecen el concepto de variables, que no son más que nombres que "apuntan" a una determinada parte de la memoria y que el lenguaje utiliza para escribir y leer en esta de manera controlada.

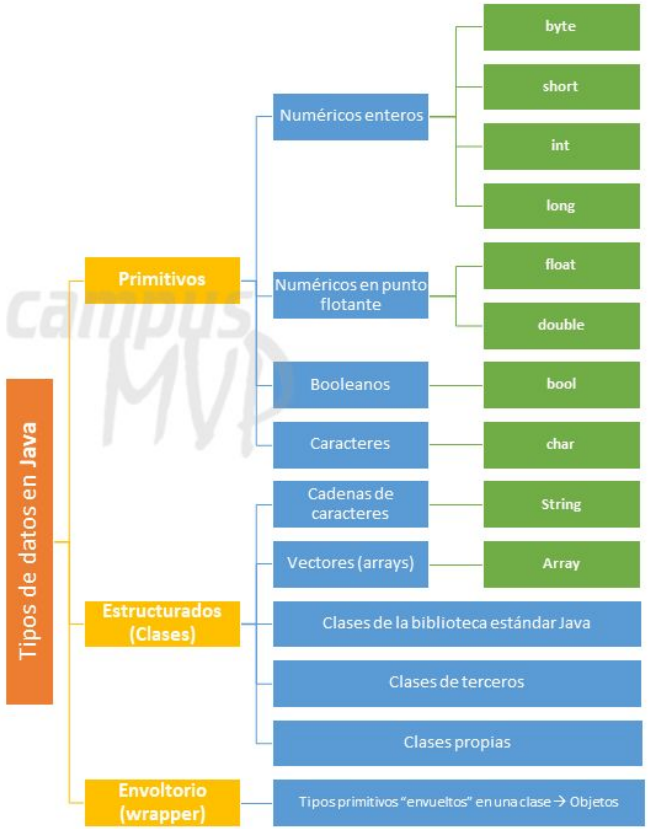
El acceso a esta información se puede mejorar dependiendo del tipo de información que almacenemos. Por ejemplo, no es lo mismo tener la necesidad de manejar números, que letras que conjuntos de datos. Y dentro de éstos no es igual tener que almacenar un número entero que uno decimal. Aunque al final todo son ceros y unos dentro de la memoria de nuestro ordenador, es la forma de interpretarlos lo que marca la diferencia, tanto al almacenarlos como al recuperarlos.

Este el motivo por el que los lenguajes de programación cuentan con el concepto de tipos de datos: se trata de distintas maneras de interpretar esos "ceros y unos" en función de ciertas configuraciones que establecen el espacio utilizado así como la representación aplicada para codificar y decodificar esa información.

Tipos de datos

Los tipos de datos en Java se pueden clasificar en 3 tipos.

Primitivos, estructurados y envoltorios.



Primitivos

Tipos primitivos

En Java existen ocho tipos de datos primitivos que se pueden clasificar en:

Números enteros (`byte, short, int, long`).
Números reales (`float, double`).
Carácter (`char`).
Booleano o lógico (`boolean`).

Lista de tipos de datos primitivos del lenguaje Java

Tipo	*Tamaño*	*Valor mínimo*	*Valor máximo*
byte	8 bits	-128	127
short	16 bits	-32768	32767
int	32 bits	-2147483648	2147483647
long	64 bits	-9223372036854775808	9223372036854775807
float	32 bits	-3.402823e38	3.402823e38
double	64 bits	-1.79769313486232e308	1.79769313486232e308
char	16 bits	'\u0000'	'\uffff'

Enteros Numéricos

En Java existen cuatro tipos destinados a almacenar números enteros, que varían en el número de bytes usados para su almacenamiento y, en consecuencia, el rango de valores que es posible representar con ellos. Todos ellos emplean una representación que permite el almacenamiento de números negativos y positivos. El nombre y características de estos tipos son los siguientes:

byte: como su propio nombre denota, emplea un solo byte (8 bits) de almacenamiento. Esto permite almacenar valores en el rango [-128, 127].
short: usa el doble de almacenamiento que el anterior, lo cual hace posible representar cualquier valor en el rango [-32.768, 32.767].

int: emplea 4 bytes de almacenamiento y es el tipo de dato entero más empleado. El rango de valores que puede representar va de -231 a 231-1.

long: es el tipo entero de mayor tamaño, 8 bytes (64 bits), con un rango de valores desde -263 a 263-1.

```
long distancia;
short altura;
int peso;
```

Numéricos Reales

En Java los representan los tipos: **float**, **double**.
El tipo de dato numérico real es un subconjunto finito de los números reales, los cuales siempre llevan un punto decimal y también pueden ser positivos o negativos. Los números reales tienen una parte entera y una parte decimal.

Numéricos Reales o en punto flotante

Los tipos numéricos en punto flotante permiten representar números tanto muy grandes como muy pequeños además de números decimales, pueden ser positivos o negativos, además poseen una parte entera y una parte decimal. Java dispone de 2 tipos concretos en esta categoría:

float: conocido como tipo de precisión simple, emplea un total de 32 bits. Con este tipo de datos es posible representar números en el rango de 1.4x10-45 a 3.4028235x1038.

double: sigue un esquema de almacenamiento similar al anterior, pero usando 64 bits en lugar de 32. Esto le permite representar valores en el rango de 4.9x10-324 a 1.7976931348623157x10308.

Ejemplo de declaración de variables reales:

```
float                                   peso;
double                                longitud;
float          altura          =          2.5F;
double          area           =          1.7E4;
double          z              =           .123;
```

Caracteres

El tipo de datos ***char*** es un simple carácter Unicode de 16 bits. Un ***char*** es un solo carácter.

Java fue diseñado para uso mundial, por lo tanto, necesita utilizar un juego de caracteres que pueda representar todos los idiomas del mundo.

Unicode es el conjunto de caracteres estándar diseñado expresamente para este fin.

Booleanos

Este tipo de datos tiene la finalidad de facilitar el trabajo con valores "verdadero/falso" (booleanos), resultantes por regla general de evaluar expresiones. Los dos valores posibles de este tipo son true y false.

```
boolean                          activo;
boolean          visible    =     true;
boolean evaluado = a>b;
```

Su valor por defecto es **false**

Literals and Primitive Casting

- Los literales int pueden ser binarios, decimales, octales (013), o hexadecimales (0x3d).
- Los literales para longs finalizan en L or l.
- Los literales float finalizan en F o f
- Los literales double literals finalizan en digito, D o d.
- Los literales boolean son true y false.
- Los literales char, son caracteres simples dentro de comillas simples: ‘d’.

Tipos Estructurados

Tipos de datos estructurados

Los tipos de datos primitivos que acabamos de ver se caracterizan por poder almacenar un único valor. Salvo este reducido conjunto de tipos de datos primitivos, que facilitan el trabajo con números, caracteres y valores booleanos, todos los demás tipos de Java son objetos, también llamados tipos estructurados o "Clases".

Los tipos de datos estructurados se denominan así porque en su mayor parte están destinados a contener múltiples valores de tipos más simples, primitivos. También se les llama muchas veces "tipos objeto" porque se usan para representar objetos. La mismas podríamos distinguirlas en 3 grupos:

Cadena de caracteres, definidos por el usuario y envoltorios o wrappers.

Cadenas de caracteres

Aunque las cadenas de caracteres no son un tipo simple en Java, sino una instancia de la clase String, *el lenguaje otorga un tratamiento bastante especial a este tipo de dato, lo cual provoca que, en ocasiones, nos parezca estar trabajando con un tipo primitivo*.

Aunque cuando declaramos una cadena estamos creando un objeto, su declaración no se diferencia de la de una variable de tipo primitivo de las que acabamos de ver:

```
String curso = "Iniciación a Java";
```

Y esto puede confundir al principio. Recuerda: Las cadenas en Java son un objeto de la clase String, aunque se declaren de este modo.

Las cadenas de caracteres se delimitan entre comillas dobles, en lugar de simples como los caracteres individuales. En la declaración, sin embargo, no se indica explícitamente que se quiere crear un nuevo objeto de tipo String, esto es algo que infiere automáticamente el compilador.

Las cadenas, por tanto, son objetos que disponen de métodos que permiten operar sobre la información almacenada en dicha cadena. Así, encontraremos métodos para buscar una subcadena dentro de la cadena, sustituirla por otra, dividirla en varias cadenas atendiendo a un cierto separador, convertir a mayúsculas o minúsculas, etc.

Vectores o arrays

Los vectores son colecciones de datos de un mismo tipo. También son conocidos popularmente como arrays e incluso como "arreglos" (aunque se desaconseja esta última denominación por ser una mala adaptación del inglés).

Un vector es una estructura de datos en la que a cada elemento le corresponde una posición identificada por uno o más índices numéricos enteros.

También es habitual llamar matrices a los vectores que trabajan con dos dimensiones.

Los elementos de un vector o array se empiezan a numerar en el 0, y permiten gestionar desde una sola variable múltiples datos del mismo tipo.

Por ejemplo, si tenemos que almacenar una lista de 10 números enteros, declararemos un vector de tamaño 10 y de tipo entero, y no tendríamos que declarar 10 variables separadas de tipo entero, una para cada número.

Tipos definidos por el usuario

Además de los tipos estructurados básicos que acabamos de ver (cadenas y vectores) en Java existen infinidad de clases en la plataforma, y de terceros, para realizar casi cualquier operación o tarea que se pueda ocurrir: leer y escribir archivos, enviar correos electrónicos, ejecutar otras aplicaciones o crear cadenas de texto más especializadas, entre un millón de cosas más.

Todas esas clases son tipos estructurados también.

Y por supuesto puedes crear tus propias clases para hacer todo tipo de tareas o almacenar información. Serían tipos estructurados definidos por el usuario.

Envoltorios o wrappers

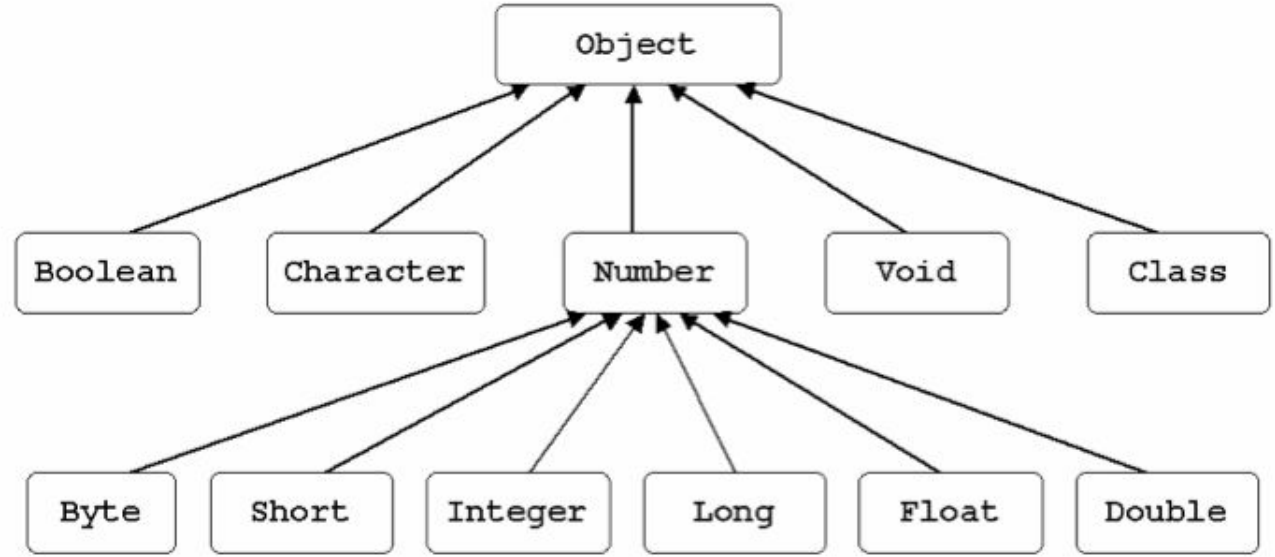
Tipos de datos envoltorio (wrappers)

Java cuenta con tipos de datos estructurados equivalentes a cada uno de los tipos primitivos que hemos visto. ***Byte, Short, Integer, Long, Float, Double, Boolean y Character .***

Así, por ejemplo, para representar un entero de 32 bits (int) de los que hemos visto al principio, Java define una clase llamada Integer que representa y "envuelve" al mismo dato pero le añade ciertos métodos y propiedades útiles por encima.

Además, otra de las finalidades de estos tipos "envoltorio" es facilitar el uso de esta clase de valores allí donde se espera un dato por referencia (un objeto) en lugar de un dato por valor, como los contenedores que veremos posteriormente.

Jerarquía de Envoltorios



Objetos

Objetos

- El objeto es la entidad en torno a la cual gira la POO.
- Un objeto es un ejemplar concreto de una clase, como por ejemplo el curso de metodología de la programación es un curso concreto dentro de todos los tipos de cursos que pueden existir.
- Un objeto pertenece a una clase, por lo tanto dispondrá de los atributos (datos) y operaciones (métodos) de la clase a la que pertenece.
- Un objeto responde al comportamiento definido por las operaciones de la clase a la que pertenece. Es decir, si la clase coche dispone del atributo color y del método arrancar, un coche concreto tendrá un color, y podrá arrancar, exclusivamente.
-
- Una clase proporciona los planos para los objetos, entonces, básicamente, un objeto se crea a partir de una clase.

- En Java, la keyword que se usa para crear nuevos objetos es new. Los objetos tienen estados y comportamientos.
- Un objeto es una instancia de una clase. Los objetos de software también tienen un estado y un comportamiento.
- El estado de un objeto de software se almacena en campos y el comportamiento se muestra a través de métodos.
- Entonces, en el desarrollo de software, los métodos operan en el estado interno de un objeto y la comunicación de objeto a objeto se realiza a través de métodos.
- Un objeto es un conjunto de variables junto con los métodos relacionados con éstas. Contiene la información (las variables) y la forma de manipular la información (los métodos).

Creación de Objetos

Hay tres pasos al crear un objeto de una clase: declaración, Instanciación e Inicialización.

Declaración:
Una declaración de variable con un nombre de variable con un tipo de objeto.

Instanciación:
Mediante la palabra clave **new** se usa para crear el objeto.

Inicialización:
Al usar la palabra clave **new** es seguida por una llamada a un constructor. Esta llamada inicializa el nuevo objeto.

Arrays

Que es un Array?

Una array o arreglo es una colección de variables del mismo tipo, a la que se hace referencia por un nombre común. En Java, los arrays pueden tener una o más dimensiones, aunque el array unidimensional es el más común.

Los arrays se usan para una variedad de propósitos porque ofrecen un medio conveniente de agrupar variables relacionadas. Por ejemplo, puede usar una matriz para mantener un registro de la temperatura alta diaria durante un mes, una lista de promedios de precios de acciones o una lista de tu colección de libros de programación.

La ventaja principal de un array es que organiza los datos de tal manera que puede ser manipulado fácilmente. Por ejemplo, si tiene un array que contiene los ingresos de un grupo seleccionado de hogares, es fácil calcular el ingreso promedio haciendo un ciclo a través del array. Además, los arrays organizan los datos de tal manera que se pueden ordenar fácilmente.

Aunque los arrays en Java se pueden usar como matrices en otros lenguajes de programación, tienen un atributo especial: **se implementan como objetos**. Este hecho es una de las razones por las que la discusión de los arrays se pospuso hasta que se introdujeron los objetos. Al implementar arrays como objetos, se obtienen varias ventajas importantes, una de las cuales es que los arrays no utilizados pueden ser recolectados.

Un array o matriz es simplemente una variable que puede contener valores múltiples, a diferencia de una variable regular que solo puede contener un único valor.

- En Java, todas las matrices se asignan dinámicamente. (Se analiza a continuación)
- Como las matrices/arrays son objetos en Java, cada array tiene asociado una variable de instancia de longitud (length) que contiene la cantidad de elementos que la matriz puede contener. (En otras palabras, length contiene el tamaño de la matriz.)
- Una variable array en Java se declara como otras variables con corchetes [] después del tipo de datos.
- Las variables en el array están ordenadas y cada una tiene un índice que comienza desde 0.
- El array Java también se puede usar como un campo estático, una variable local o un parámetro de método.

- El tamaño de un array debe especificarse mediante un valor int y no, long o short.
- La superclase directa de un tipo de array es Object.
- Cada tipo de array implementa las interfaces Cloneable y java.io.Serializable.
- El array puede contener tipos de datos primitivos así como también objetos de una clase según la definición del array. En el caso de los tipos de datos primitivos, los valores reales se almacenan en ubicaciones de memoria contigua. En el caso de los objetos de una clase, los objetos reales se almacenan en heap.
- Los arrays pueden contener primitivas u objetos, pero la propia matriz es siempre un objeto.
- Cuando se declara un array, los corchetes pueden estar a la izquierda oa la derecha del nombre de la variable.
- Nunca es legal incluir el tamaño de una matriz en la declaración.

Declaración

Cuando un array se declara, solo se crea una referencia del array. Para realmente crear o dar memoria al array, puede crear un array de la siguiente manera:

```
tipo                                    nombre-array[];
```

o

```
tipo [] nombre-array;
```

La declaración de un array tiene dos componentes: el **tipo** y el **nombre**, donde se declara el tipo de elemento del array, determina el tipo de datos de cada elemento que comprende la matriz.

Una dimensión

Cuando un array se declara, solo se crea una referencia del array. Para realmente crear o dar memoria al array, puede crear un array de la siguiente manera:

```
nombre-array = new tipo [tamaño];
```

Los elementos en la matriz asignada por new se inicializará automáticamente a cero (para tipos numéricos), **false** (para boolean) o **null** (para tipos de referencia). Obtener un array es un proceso de dos pasos. Primero, debe declarar una variable del tipo de array deseado. En segundo lugar, debe asignar la memoria que mantendrá el array, usar new y asignarla a la variable del array. Por lo tanto, en Java, todos los arrays se asignan dinámicamente.

Varias dimensiones

En Java es posible crear arrays con más de una dimensión, pasando de la idea de lista, vector o matriz de una sola fila a la idea de matriz de m x n elementos, estructuras tridimensionales, tetradimensionales, etc.

La sintaxis será:

```
tipo nombre-array[][];
```

o

```
tipo [][] nombre-array;
```

Clases y objetos

Clases y objetos

- Declaración de clases
- Declaración de propiedades
- Creación de objetos y administración de memoria
- Acceso a miembros
- Encapsulación

Declaración de clases

Estructura de una Clase

Una clase se estructura mediante

- package
- imports
- tipo clase

En resumen una clase se puede definir como una plantilla o plan que describe el comportamiento o estado que admite el objeto de su tipo.

Una clase es un plano a partir del cual se crean objetos individuales.

```
package my.project;
public class Dog {
   String breed;
   int age;
   String color;

   void barking() {
   }
   void hungry() {
   }
   void sleeping() {
   }
}
```

Ámbito de las Variables

El alcance, ámbito o scope de la variable es simplemente el área de una programa en el que existe una variable y se puede ver o utilizar y puede limitarse a unas pocas líneas o puede incluir toda la clase.

Variables locales:
Las variables definidas dentro de los métodos, constructores o bloques de código se denominan variables locales. Esta se declarará e inicializará dentro del método y la variable se destruirá cuando el método se haya completado.

Las variables locales viven mientras su método esté en la pila; sin embargo, si su método invoca otro método, están temporalmente no disponibles.

Variables de instancia:
Las variables de instancia son variables dentro de una clase pero fuera de cualquier método. Estas variables se inicializan cuando se crea una instancia de la clase. Se puede acceder a las variables de instancia desde cualquier método, constructor o bloque de esa clase en particular.

Las variables de instancia viven mientras su objeto vive.

Variables de clase:
Las variables de clase son variables declaradas dentro de una clase, fuera de cualquier método, con la palabra clave *static*.

Las variables estáticas viven básicamente mientras su clase vive.

Variables de bloque
Las variables de bloque (por ejemplo, en un `for` o un `if`) en vivo hasta que el bloque complete.

Shadow Variable o Variables Sombra:

En Java las Shadow variables son aquellas que tienen el mismo nombre pero diferente ámbito. Por ejemplo en el diagrama tenemos dos variables nombre que se encuentran en distinto nivel de la jerarquía.

```
public class Animal {
   String name = "Animal Name";
   String getName() {
       return name;
   }
   void setName(String name) {
       this.name=name;
   }
   void showName() {
       System.out.println(name);
   }
}
```

```
public class Dog extends Animal {
   String name = "Dog Name";
   String getName() {
       return name;
   }
   void setName(String name) {
       this.name=name;
   }
}

public static void main(String[] args) {
       Dog d= new Dog();
       d.showName();
}
```

Que pasaría ejecutaremos el código?

Class Static Variables

- Las variables de clase también conocidas como variables estáticas se declaran con la palabra clave static en una clase, pero fuera de un método, constructor o bloque.
- Solo habría una copia de cada variable de clase por clase, independientemente de cuántos objetos o instancias se crearán a partir de ella.
- Las variables estáticas rara vez se usan aparte de declararse como constantes. Las constantes son variables que se declaran como public / private, final y static. Las variables constantes (final) nunca cambian de su valor inicial.
- Las variables estáticas se almacenan en la memoria estática. Es raro usar variables estáticas distintas a las declaradas como finales y usadas como constantes públicas o privadas.

- Las variables estáticas se crean cuando el programa accede a la clase y se destruye cuando el programa se detiene .
- La visibilidad es similar a las variables de instancia. Sin embargo, la mayoría de las variables estáticas se declaran públicas, ya que deben estar disponibles para los usuarios de la clase.
- Los valores predeterminados son los mismos que las variables de instancia. Para números, el valor predeterminado es 0; para booleanos, es falso; y para las referencias de objeto, es nulo. Los valores se pueden asignar durante la declaración o dentro del constructor. Además, los valores pueden asignarse en bloques de inicializadores estáticos especiales.

- Se puede acceder a las variables estáticas llamando a `ClassName.VARIABLE_NAME`.
- Al declarar las variables de clase como estática pública final, los nombres de las variables (constantes) están todos en mayúsculas. Si las variables estáticas no son públicas y finales, la sintaxis de denominación es la misma que la instancia y las variables locales.

Declaración de propiedades

Propiedades o atributos

En Java, las propiedades de un objeto son determinadas por una serie de variables que definen las características de los objetos de la clase, y que podrán tener o no, métodos del tipo getter y setter para acceder a ellas, como lo hacen los JavaBeans.

Las mismas se declaran dentro del bloque de de la clase y fuera de los métodos de la misma.

Poseen visibilidad y accesibilidad, dependiendo de los modificadores del tipo **public**, **protected, private, <default>**, etc.

```
public class Customer {
  private String name;
  private String email;

  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
    this.email = name+”@domain.com”;
  }

  public String getEmail() {
    return email;
  }
  public void setEmail(String email) {
    this.email = email;
  }

}
```

Otras consideraciones

- Los enteros literales son implícitamente int .
- Las expresiones enteras siempre dan como resultado un int-size, nunca menor.
- Los numeros Floating-point son implícitamente doubles (64 bits).
- Al estrechar (narrowing) una primitiva se trunca los bits de orden alto.
- Las asignaciones compuestas (como + =) realizan un cast automático.
- Una variable de referencia contiene los bits que se utilizan para referirse a un objeto.
- Las variables de referencia pueden referirse a subclases del tipo declarado pero no a las superclases.

Acceso a miembros

Accesos a miembros

- Hay tres modificadores de acceso: public, protected y private, sin embargo existen 4 tipos de niveles de acceso: `public`, `protected`, `default`, y `private`.
- Los métodos y variables de instancia (no locales) se conocen como "miembros" (member), estos pueden utilizar los cuatro niveles de acceso.
- El acceso a los “miembros” tiene dos formas:
  - El código de una clase puede acceder a un miembro de otra clase.
  - Una subclase puede heredar un miembro de su superclase.
- Los miembros públicos pueden ser accedidos por todas las otras clases, incluso en otros paquetes.
- Si un miembro de superclase es público, la subclase lo hereda, independientemente del paquete.

- Los miembros a los que se accede sin el operador punto (.) Deben pertenecer a la misma clase. (this). Siempre hace referencia al objeto que se está ejecutando actualmente.
- `this.aMethod()` es lo mismo que invocar `aMethod()`..
- Se puede acceder a miembros privados sólo por código en la misma clase.
- Los miembros privados no son visibles para las subclases, por lo que los miembros privados no pueden ser heredados.
- Si no se puede acceder a una clase, no se puede acceder a sus miembros, por lo cual es muy recomendable determinar correctamente la visibilidad de la clase antes de determinar la visibilidad del miembro.

Tablas de accesos

Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Clases abstractas

Clase abstracta

Si desea que una clase contenga un método particular pero desea que la implementación real de ese método sea determinada por clases secundarias, puede declarar el método en la clase principal como un resumen.

Debe colocar la palabra clave `abstract` antes del nombre del método en la declaración del método.

En lugar de llaves, un método abstracto tendrá un punto y coma (;) al final.

Se utiliza mucho en la construcción de jerarquías donde muchas subclases comparten gran parte de los atributos o funcionalidades, escalando las misma a una superclase común, dejando las características únicas en cada subclase.

Clases anidadas

Clases anidadas

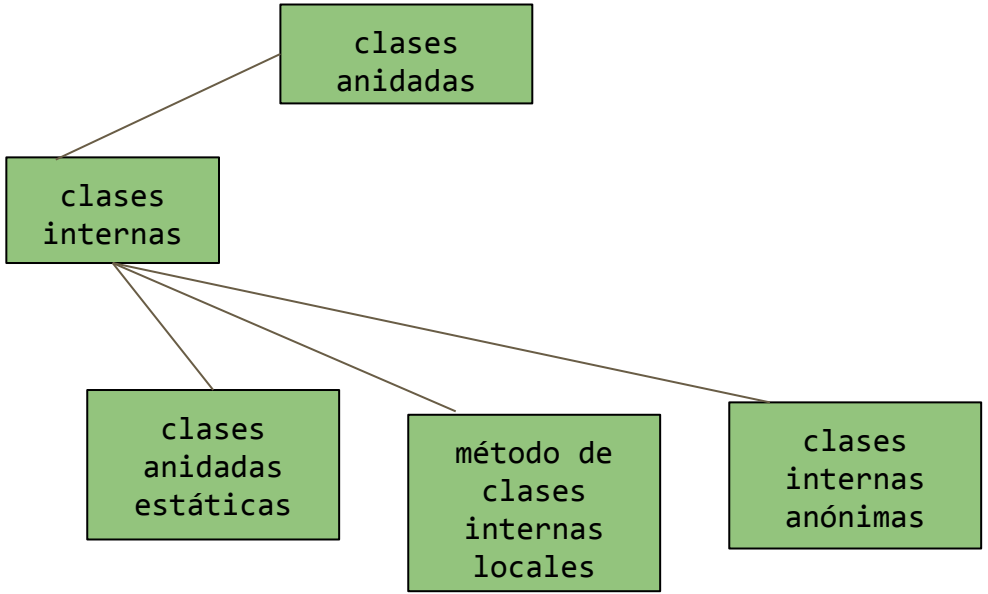
En Java, al igual que los métodos, las variables de una clase también pueden tener otra clase como miembro. Escribir una clase dentro de otro está permitido en Java. La clase escrita dentro se llama clase anidada, y la clase que contiene la clase interna se llama clase externa.

```
class Outer_Demo {
   class Nested_Demo {

   }

}
```

Tipos de clases anidadas

Las clases anidadas se dividen en dos tipos

- Clases anidadas no estáticas
- Clases anidadas estáticas



Clases internas (anidadas no estáticas)

Las clases internas son un mecanismo de seguridad en Java. Sabemos que una clase no puede asociarse con el modificador de acceso privado, pero si tenemos la clase como miembro de otra clase, entonces la clase interna puede hacerse privada. Y esto también se usa para acceder a los miembros privados de una clase outer o cointenedora.

Las clases internas son de tres tipos según cómo y dónde las defina. Ellos son:

- Clase interna
- Clase interna local del método
- Clase interna anónima

Clase interna (Inner class)

Crear una clase interna es bastante simple. Solo necesitas escribir una clase dentro de una clase. A diferencia de una clase, una clase interna puede ser privada y una vez que declaras una clase interna privada, no se puede acceder desde un objeto fuera de la clase.

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

Clase interna anónima

Una clase interna declarada sin un nombre de clase se conoce como clase interna anónima. En el caso de clases internas anónimas, las declaramos y las creamos al mismo tiempo.

En general, se usan siempre que necesite anular (override) el método de una clase o una interface.

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

Clase interna anónima como argumento

Generalmente, si un método acepta un objeto de una interface, una clase abstracta o una clase concreta, entonces podemos implementar la interface, extender la clase abstracta y pasar el objeto al método. Si es una clase, podemos pasarla directamente al método.

Clase estática anidada

Una clase interna estática son aquellas que son un miembro estático de la clase externa. Se puede acceder sin instanciar la clase externa, usando otros miembros estáticos.

Al igual que los miembros estáticos, una clase anidada estática no tiene acceso a las variables de instancia y los métodos de la clase externa.

Reglas aplicables a clase

Reglas de clase

- Las clases no inner únicamente pueden tener acceso `public` o `default`.
- Una clase con acceso default sólo puede ser vista por clases dentro del mismo package.
- Una clase con acceso public puede ser vista por todas las clases de todos los paquetes.
- La visibilidad de la clase gira en torno a si el código de una clase puede
  - Crear una instancia de otra clase
  - Extender (o subclase) otra clase
  - Métodos de acceso y variables de otra clase.
- Las clases también pueden ser modificadas con final, abstract, o strictfp.
- Una clase no puede ser final y abstracta.

- Una clase final no puede ser sub clasificada (heredada).
- **Una clase abstract no se puede instanciar.**
- Un solo método abstract en una clase, obliga a que toda la clase debe ser abstract.
- Una clase abstract puede tener tanto métodos abstractos como no abstractos.
- La primera clase concreta (no abstracta) para extender una clase abstracta debe implementar todos sus métodos abstractos.

Clases avanzadas

Temario

- Herencia
- Sobreescritura de métodos
- Sobrecarga de métodos
- Sobrecarga de constructores
- La clase object
- Clases
- Interfaces
- Polimorfismo
- Clases envolventes
- Autoboxing
- Recursos estáticos
- Palabra clave final
- Tipos enumerados
- Argumentos Variables

Clases

Interfaces

Que son?

Las interfaces son clases totalmente abstractas, es decir una clase que posee todos sus métodos abstractos. Son contratos para lo que una clase puede hacer, pero no dicen nada sobre la forma en que la clase debe hacerlo. Las mismas pueden ser implementados por cualquier clase desde cualquier árbol de herencia. Una interface es como una clase abstracta de 100 por ciento y es implícitamente abstracta si escribe el modificador abstract en la declaración o no.

No se permiten métodos concretos.

```
interface Animal {
   int weight();
}
```

Los métodos de interface son por defecto público y la declaración abstracta-explícita de estos modificadores es opcional.

Reglas

- En el encabezado se usa la palabra clave interface en lugar de class o abstract class. Por ejemplo public interface NombreDelInterface {...}
- Todo método es abstracto y público sin necesidad de declararlo, es decir, no hace falta poner abstract public porque por defecto todos los métodos son abstract public. Por lo tanto un interface en Java no implementa ninguno de los métodos que declara: ninguno de sus métodos tiene cuerpo.
- Las interfaces no tienen ningún constructor.
- Un interface solo admite campos de tipo “public static final”, es decir, campos de clase, públicos y constantes. No hace falta incluir las palabras public static final porque todos los campos serán tratados como si llevaran estas palabras. Recordemos que **static** equivalía a “de clase” y final a “constante”. Las interfaces pueden ser un lugar interesante para declarar constantes que van a ser usadas por diferentes clases en nuestros programas.

- Una clase puede derivar de un interface de la misma manera en que puede derivar de otra clase. No obstante, se dice que el interface se implementa (implements), no se extiende (extends) por sus subclases. Por tanto para declarar la herencia de un interface se usa la palabra clave implements en lugar de extends
- Las interfaces pueden tener constantes, que siempre son implícitamente public, static y final.
- Las declaraciones constantes de interfaces public, static y final son opcionales en cualquier combinación.
- Debe mantener la firma exacta (que permite los retornos covariantes) y el tipo de retorno de los métodos que implementa (pero no tiene que declarar las excepciones de la interface).
- Una clase que implementa una interface puede ser abstract.
- Una clase de implementación abstracta no tiene que implementar los métodos de interface (pero la primera subclase concreta debe).

- Una clase puede extender sólo una clase (no hay herencia múltiple), pero puede implementar muchas interfaces.
- Las interfaces pueden extender una o más interfaces.
- Las interfaces no pueden extender una clase o implementar una clase o interface.
- Al realizar el examen, verifique que las declaraciones de interface y clase sean legales antes de verificar otra lógica de código.
- Una clase de implementación legal no abstracta tiene las siguientes propiedades:
    - Debe proporcionar implementaciones concretas para los métodos de la interface.
    - Debe seguir todas las reglas legales de anulación para los métodos que implementa.
    - No debe declarar ninguna nueva excepción comprobada para un método de implementación.
    - No debe declarar excepciones comprobadas que sean más amplias que las excepciones declaradas en el método de interface.

```
public class MyException extends Exception {}
public class MySubException extends MyException
{}

public interface IFace1 {
  void method1() throws Exception;
  void method2() throws Exception;
  void method3() throws MyException;
  void method4() throws MyException;
  void method5() throws MyException;
}
```

```
public class Class1 implements IFace1 {
  public void method1()  {}   <- Ok
  public void method2() throws MyException { } <- Ok
  public void method3() throws Exception { }  <- Error
  public void method4() throws MyException, Exception { }  <- Error
  public void method5() throws MyException, MySubException { }  <- Ok
}
```

J8 Interfaces recargadas

Java SE 8 hace un cambio grande a las interfaces con el fin de que las librerías puedan evolucionar sin perder compatibilidad. A partir de esta versión, las interfaces pueden proveer métodos con una implementación por defecto. Las clases que implementen dichas interfaces heredarán automáticamente la implementación por defecto si éstas no proveen una explícitamente:

- Llamados métodos por defecto, métodos virtuales o métodos defensores , son especificados e implementados en la interface. Usan la nueva palabra reservada default antes del tipo de retorno.
- La implementación por defecto es usada solo cuando la clase implementadora no provee su propia implementación .
- Desde el punto de vista de quién invoca al método, es un método más de la interface.Se crea un conflicto cuando se implementen interfaces con métodos por defecto con el mismo nombre.

```
interface A {
     void method();
     void method2();
}
interface C extends A {
     @Override
     default void method() {
          System.out.println("C");
     }
}
interface B extends A {
     @Override
     default void method() {
        System.out.println("B");
     }
}
```

```
/*Error*/
interface D extends B, C {}

/* OK */
interface D extends B, C {
     @Override
     default void method() {
          B.super.method(); //<<<<<<
     }

     @Override
     default void method2() {
      }
}

class DImpl implements D {}
```

- Los métodos default pueden ayudarnos a extender interfaces garantizando funcionalidades en las implementaciones.
- Los métodos por defecto funcionan como puente por las diferencias entre las interfaces y clases abstractas.
- Los métodos por defecto nos ayudarán a evitar clases de utilidad, como la clase Collections puede ser proporcionada en la propia interface.
- Los métodos por defecto nos ayudará en la eliminación de clases de implementación de base, podemos proporcionar implementación por defecto y las clases de implementación pueden elegir cuál de ellos para anular.
- Una de las principales razones para la introducción de métodos por defecto es para mejorar la API Colecciones en Java 8 para apoyar las expresiones lambda.
- Los métodos default también se conocen como Defender Method o Virtual Extension Method

Constructores

Constructores

Al hablar sobre las clases, uno de los temas más importantes sería el de los constructores. Cada clase tiene un constructor. Si no escribimos explícitamente un constructor para una clase, el compilador de Java crea un constructor predeterminado para esa clase.

Los constructores, al igual que los métodos poseen la capacidad de sobrecargarse.

La regla principal de los constructores es que deberían tener el mismo nombre que la clase. Una clase puede tener más de un constructor.

```
public class Puppy {

   public Puppy() {
   }

   public Puppy(String name) {
   }

}
```

- Los constructores pueden tener cualquier modificador de acceso.
- El compilador creará un constructor predeterminado si uno no crea ningún constructor en su clase.
- El constructor default es un constructor no-arg con una llamada no-arg a super().
- La primera declaración de cada constructor debe ser una llamada a **this()** (un constructor sobrecargado) o a **super().**
- El compilador agrega una llamada a super() a menos que ya haya realizado una llamada a this() o super().
- Los miembros de instancia sólo son accesibles después de ejecutarse el super constructor.
- Las clases abstract tienen constructores que se llaman cuando se crea una instancia de una subclase concreta.
- Las interfaces no tienen constructores.

- Si su superclase no tiene un constructor vacío, debe crear un constructor e insertar una llamada a **super**() con argumentos que coinciden con los del constructor de la superclase.
- Los constructores nunca se heredan, por lo tanto no se sobreescriben.
- Un constructor puede ser invocado de manera directa por otro constructor via **super**() y **this**();
- Asuntos a tener encuentra con la llamada **this()**:
    - Debe aparecer como primera sentencia dentro de un constructor.
    - Las lista de argumentos determinara que constructor sobrecargado se llama.
    - Los constructores pueden llamar a constructores, y así sucesivamente, pero tarde o temprano uno de ellos mejor llamará super() o la pila explotará Stackoverflow.
    - Las llamadas a super() y this() no pueden estar en el mismo constructor. Usted puede tener uno o el otro, pero nunca ambos.

Bloques estáticos y de inicialización

Bloques de inicialización

Un bloque de inicialización es simplemente un par de llaves ({}) colocado dentro de la clase (pero no dentro de un método esos son simples bloques), las instrucciones colocadas allí son ejecutadas inmediatamente después de la llamada a `super()` de nuestro constructor.

Es decir, son bloques de código que deseamos que se ejecuten para cada nueva instancia, justo después de todos los constructores padres se hayan ejecutado.

Estos bloques se ejecutan desde la parte superior de la clase hacia abajo.

Bloques estáticos

Al igual que existen variables estáticas, Java nos permite crear bloques estáticos para ser invocado cuando la clase es cargada por la memoria mediante el ClassLoader.

Múltiples bloques se ejecutan desde arriba hacia abajo.

Ciclo y Herencia

Como comentamos, los bloques estáticos se ejecutan únicamente cuando la clase es cargada por la memoria , seguido el bloque de inicialización, para luego los constructores. Por lo tanto el orden es Bloque Estático, Bloque de Inicialización, Constructor y en el caso de que se tenga herencia se da el siguiente caso:

- Bloque de estático de la SuperClase
- Bloque de estático de la SubClase
- Bloque de inicialización de la SuperClase
- Constructor de la SuperClase
- Bloque de inicialización de la SubClase
- Constructor de la SubClase

Sobreescritura de métodos

Sobreescritura / Overriding

- Si una clase hereda un método de su superclase, existe la posibilidad de sobreescribir el método siempre que no esté marcado como final.
- El beneficio de sobrescribir es la capacidad de definir un comportamiento que sea específico para el tipo de subclase, lo que significa que una subclase puede implementar un método de clase padre en función de sus requisitos.
- En términos orientados a objetos, sobrescribir significa anular la funcionalidad de un método existente.
- La lista de argumentos debe ser exactamente la misma que la del método reemplazado.
- El tipo de devolución debe ser el mismo o un subtipo del tipo de devolución declarado en el método reemplazado original en la superclase. Covariant return Types (subtipo)

- El nivel de acceso no puede ser más restrictivo que el nivel de acceso del método reemplazado. Por ejemplo: si el método de superclase se declara público, el método de anulación en la subclase no puede ser privado o protegido.
- Los métodos de instancia sólo pueden anularse si la subclase los hereda.
- Un método declarado final no puede ser sobrescrito.
- Un método declarado estático no se puede sobrescribir, pero se puede volver a declarar.

Sobrecarga de métodos

Sobrecarga / Overloading

Sobrecarga significa reutilizar un nombre de método pero con argumentos diferentes, estos deben tener listas de argumentos diferentes y puede tener diferentes tipos de devolución, si las listas de argumentos también son diferentes. A su vez, puede tener diferentes modificadores de acceso y lanzar diferentes excepciones.

- Los métodos de una superclase pueden sobrecargarse en una subclase.
- **El polimorfismo se aplica a la sobrescritura, no a la sobrecarga.**
- Tipo de objeto (no el tipo de la variable de referencia) determina qué método sobreescrito se utiliza en tiempo de ejecución. El tipo de referencia determina qué método sobrecargado se utilizará durante la compilación.

Sobrecarga de constructores

La clase Object

hashCode

Comparando Objetos

Comparando java == vs equals

Polimorfismo

Polimorfismo

El polimorfismo es la capacidad de un objeto para tomar muchas formas. El uso más común de polimorfismo en OOP ocurre cuando se usa una referencia de clase principal para referirse a un objeto de clase hijo.

```
Object o = new Integer(7)
```

Cualquier objeto Java que pueda pasar más de una prueba IS-A se considera polimórfico. En Java, todos los objetos Java son polimórficos, ya que cualquier objeto pasará la prueba IS-A para su propio tipo y para la clase Object.

Es importante saber que la única forma posible de acceder a un objeto es a través de una variable de referencia, la cual puede ser de un solo tipo y una vez declarado, el tipo de una variable de referencia no se puede cambiar.

Java Runtime Polymorphism o Casting

Downcasting : Si tiene una variable de referencia que hace referencia a un objeto de subtipo, se puede asignar a una variable de referencia del subtipo. Se debe hacer una conversión explícita, y el resultado es que puede acceder a los miembros del subtipo con esta nueva variable de referencia.

```
Object o = "a string";
String s = (String) o; //IS-A funciona
```

Upcasting : asignar una variable de referencia a una variable de referencia de supertipo explícita o implícitamente. Esto es una operación inherentemente segura porque la asignación restringe las capacidades de acceso de la nueva variable.

```
Object o = new String("a string");
```

Instanceof

En Java, podemos preguntar a un objeto su es una instancia de una clase o interface mediante el uso del operador `instanceof`

- `instanceof` es sólo para variables de referencia; Comprueba si el objeto es de un tipo particular.
- El operador instanceof sólo puede usarse para probar objetos (o nulos) en los tipos de clase que están en la misma jerarquía de clases.
- Para las interfaces, un objeto pasa la prueba instanceof si alguna de sus superclases implementa la interface en el lado derecho del operador `instanceof`.

```
if(o instanceof Number){
}
```

Clases envolventes

Clases Envoltorio o Wrappers

Como ya comentamos, j ava cuenta con tipos de datos estructurados equivalentes a cada uno de los tipos primitivos ***Byte, Short, Integer, Long, Float, Double, Boolean y Character .***

Además, comentamos que estas clases poseen finalidades en las funcionalidades donde se requieren objetos como los son los Collections

```
List<Double> y;
```

Las clases envolventes tienen dos principales funciones en java:

- Proveen un mecanismo para envolver (wrap) valores primitivos en un objeto, de esta manera los datos primitivos pueden tener actividades o comportamientos que son reservados sólo para los objetos (como ser agregados a una colección o ser retornados de un método como simple Object).
- Proveen útiles funciones para realizar conversiones: a cadena, cambiar de base numérica(octal, hexadecimal), u otros tipos primitivos.

Existe para cada tipo primitivo una clase envolvente, nombrada igual que su dato primitivo pero su nombre empieza con mayúscula.

Autoboxing

Autoboxing

En Java, se denomina `autoboxing` a conversión de un valor primitivo en un objeto de la clase envoltorio /wrapper correspondiente. Por ejemplo, convertir int a clase Integer.

El compilador de Java aplica autoboxing cuando un valor primitivo es:

- Pasado como un parámetro a un método que espera un objeto de la clase envoltura correspondiente.
- Asignado a una variable de la clase envoltura correspondiente.

Unboxing

A su vez, la conversión de un objeto de un tipo de envoltorio a su valor primitivo correspondiente se denomina `unboxing`. Por ejemplo, la conversión de entero a int.

El compilador de Java aplica unboxing cuando un objeto de una clase contenedora es:

- Pasado como un parámetro a un método que espera un valor del tipo primitivo correspondiente.
- Asignado a una variable del tipo primitivo correspondiente.

Recursos estáticos

Recurso estáticos

- Use métodos estáticos cuando debe implementar comportamientos que no deba ser afectado por el estado de las instancias.
- Use variables estáticos para almacenar datos que son específicos de clase, no de instancia. Solo habr una copia de esta variable.
- Todos los miembros estáticos pertenecen a una clase no a su instancia.
- Un método estático no puede acceder directamente a una variable de instancia.
- Utilice el operador “.” para acceder a los miembros estáticos, pero recuerde que el uso de una variable de referencia con el operador “.” es realmente un truco de sintaxis y el compilador sustituirá el nombre de la clase por la variable de referencia; por ejemplo:
    - d.doStuff(); será Dog.doStuff();
- métodos estáticos no pueden ser sobreescritos pero pueden ser redefinidos.

Palabra clave final

Tipos enumerados

Enum

Un enumerado (o Enum) es una clase "especial" (tanto en Java como en otros lenguajes) que limitan la creación de objetos a los especificados explícitamente en la implementación de la clase. La única limitación que tienen los enumerados respecto a una clase normal es que si tiene constructor, este debe de ser privado para que no se puedan crear nuevos objetos.

```
enum Animal {
   DOG,CAT,BIRD
}
```

Argumentos
Variables

Argumentos Variables o varargs

Algunas veces querrá crear un método que tome una cantidad variable de argumentos, según su uso preciso. Por ejemplo, un método que abre una conexión a Internet puede tomar un nombre de usuario, contraseña, nombre de archivo, protocolo, etc., pero proporcionará los valores predeterminados si no se proporciona parte de esta información.

En esta situación, sería conveniente pasar solo los argumentos a los que no se aplicaron los valores predeterminados. Crear dicho método implica que debe haber alguna forma de crear una lista de argumentos que sea de longitud variable, en lugar de fija.

En el pasado, los métodos que requerían una lista de argumentos de longitud variable podían manejarse de dos maneras, ninguna de las cuales era particularmente agradable.

- Primero, si la cantidad máxima de argumentos era pequeña y conocida, entonces podría crear versiones sobrecargadas del método, una para cada forma en que se podría llamar al método. Aunque esto funciona y es adecuado para algunas situaciones, solo se aplica a una clase estrecha de situaciones.
- En los casos en que la cantidad máxima de argumentos potenciales es mayor o incognoscible, se utilizó un segundo enfoque en el que los argumentos se colocaron en una matriz y luego la matriz se pasó al método. Francamente, ambos enfoques a menudo dieron lugar a soluciones torpes, y se reconoció ampliamente que era necesario un mejor enfoque.

Sintaxis de Varargs

Un argumento de longitud variable se especifica por tres puntos (...).

```
static void vaTest(int ... v){
      for (int i=0; i<v.length;i++)
            System.out.println(" arg "+i+": "+v[i]);
}
```

Esta sintaxis le dice al compilador qu más argumentos. Además, hace que v se iz de tipo int[]. Por lo tanto, dentro de vaTest(), se accede a v usando la sintaxis de matriz normal

Recuerde, el parámetro varargs debe ser el último y debe haber solo un parámetro varargs.

Sobrecarga de los métodos Varargs

Hay dos formas en que se puede sobrecargar un método varargs.

- Primero, los tipos del parámetro vararg pueden diferir. Puede sobrecargar métodos utilizando diferentes tipos de parámetros de matriz, puede sobrecargar métodos varargs utilizando diferentes tipos de variables. En este caso, Java usa la diferencia de tipo para determinar qué método sobrecargado llamar.
- La segunda forma de sobrecargar un método varargs es agregar uno o más parámetros normales. En este caso, Java usa tanto el número de argumentos como el tipo de argumentos para determinar a qué método llamar.

Varargs y error por ambigüedad

Se pueden producir errores algo inesperados al sobrecargar un método que toma un argumento de longitud variable. Estos errores implican ambigüedad porque es posible crear una llamada ambigua a un método varargs sobrecargado

```
void method(int ... v){

void method(int n, int ... v) {
```

No puede resolver llamadas como `method(1)`

Excepciones y aserciones

Temas

- Capturar excepciones
- Manejo de excepciones múltiples
- Api exception
- Propagar excepciones
- Excepciones personalizadas
- Gestión de aserciones
- Buenas prácticas con aserciones
- Habilitar y deshabilitar aserciones

Excepciones

Qué son las excepciones

El lenguaje Java™ utiliza excepciones para proporcionar posibilidades de manejo de errores para sus programas. Una excepción es un evento que se produce cuando se ejecuta el programa de forma que interrumpe el flujo normal de instrucciones.

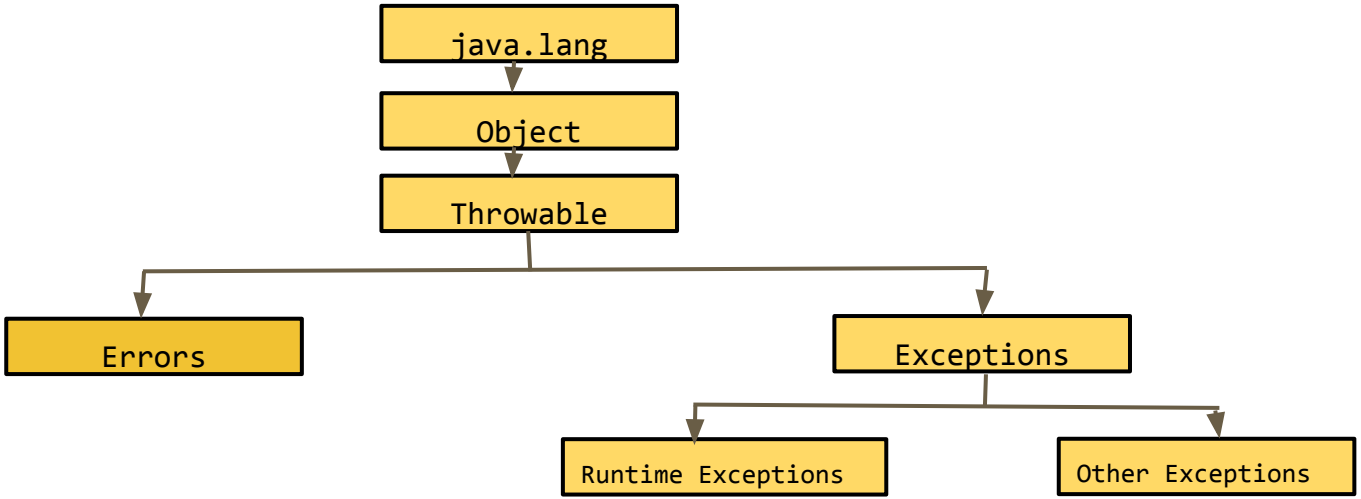
Las excepciones son un tipo de clase que la máquina virtual o las aplicaciones lanzan cuando desean interrumpir la ejecución. Algunas excepciones son creadas por programadores, y algunas por la JVM.

Podríamos decir que una excepción (o evento excepcional) es un problema que surge durante la ejecución de un programa. Cuando se produce una excepción, el flujo normal del programa se interrumpe y el programa / aplicación finaliza de manera anormal, lo que no se recomienda, por lo tanto, estas excepciones deben

Una excepción puede ocurrir por muchas razones diferentes, los siguientes son algunos escenarios donde ocurre una excepción. Por ejemplo, un usuario ha ingresado datos no válidos, no se puede encontrar un archivo que debe abrirse o se ha perdido una conexión de red en medio de las comunicaciones o se ha agotado la memoria de la JVM.

Jerarquía de Excepciones

Todas las clases de excepción son subtipos de java.lang.Exception class. La clase de excepción es una subclase de la clase Throwable.



Tipos de Excepciones

Excepciones comprobada (checked): es una excepción que se produce en el momento de la compilación, también se denominan excepciones de tiempo de compilación. Estas excepciones no pueden simplemente ignorarse al momento de la compilación, el programador debe encargarse de estas excepciones.

Excepciones no verificadas (unchecked): es una excepción que se produce en el momento de la ejecución. Estos también se llaman excepciones de tiempo de ejecución, incluyen errores de programación, como errores de lógica o uso incorrecto de una API. Las excepciones de tiempo de ejecución se ignoran al momento de la compilación.

Errores: no son excepciones en absoluto, sino problemas que surgen más allá del control del usuario o del programador. Generalmente, se ignoran los errores en su código porque rara vez puede hacer algo acerca de un error.

- Las excepciones se agrupan en dos tipos: checked y unchecked.
- Las checked incluyen todo las heredadas de Exception, excluyendo las que lo hacen de RuntimeException.
- Las excepciones de tipo checked están sujetas a la regla de “declarar y gestionar”; declaramos mediante throws y gestionamos con el uso del try/catch.
- Los subtipos de Error o RuntimeException son unchecked. Si bien se pueden declarar o gestionar, aunque no es requerido.
- Si se utiliza, de manera opcional, el bloque finally, hay que tener en cuenta que siempre se invocará, independientemente de que se lance o no una excepción en el intento correspondiente, e independientemente de si se captura o no una excepción lanzada.
- La única excepción a la regla finally-siempre-será-invocado es que cuando la JVM se cierra. Esto podría ocurrir si el código de los bloques try o catch llama a System.exit().
- Sólo porque finalmente se invoca no significa que se complete. Código en el bloque finalmente podría plantear una excepción o emitir un System.exit ().

- Las excepciones no “catch”ed se propagan de nuevo a través de la pila de llamadas, comenzando desde el método donde se genera la excepción y terminando con el primer método que tiene una captura correspondiente para ese tipo de excepción o un cierre de JVM.
- Puede crear sus propias excepciones, normalmente extendiendo Exception o uno de sus subtipos. Su excepción se considerará una excepción checked (a menos que se extienda desde RuntimeException) y el compilador aplicará la regla declarar y gestionar para esa excepción.
- Todos los bloques de captura deben ser ordenados de la forma más específica a la más general. Si tiene una cláusula catch para IOException y Exception, debe poner el catch para IOException primero en su código. De lo contrario, el IOException sería capturado por catch(Exception e), porque un argumento catch puede capturar la excepción especificada o cualquiera de sus subtipos. El compilador le impedirá definir cláusulas de captura que nunca se puedan alcanzar.

A partir de JDK 7, el mecanismo de manejo de excepciones de Java se ha ampliado con la adición de tres características.

- El primero es compatible con la gestión automática de recursos, que automatiza el proceso de liberación de un recurso, como un archivo, cuando ya no es necesario. Se basa en una forma expandida de try, llamada declaración try-with-resources (try con recursos).
- La segunda característica nueva se llama multi-catch.
- Y la tercera a veces se llama final rethrow o more precise rethrow. Estas dos características se describen aquí.

Capturar excepciones

Como comentamos, las excepciones se agrupan en dos tipos: checked y unchecked.

Las checked incluyen todo las heredadas de Exception, excluyendo las que lo hacen de RuntimeException, estas están sujetas a la regla de “declarar y gestionamos”; declaramos mediante throws y gestionamos con el uso del try/catch.

Los subtipos de Error o RuntimeException son unchecked. Si bien se pueden declarar o gestionar, aunque no es requerido. Si se utiliza, de manera opcional, el bloque finally, hay que tener en cuenta que siempre se invocará, independientemente de que se lance o no una excepción en el intento correspondiente, e independientemente de si se captura o no una excepción lanzada.

Uso de throws

Las excepciones son “lanzadas” mediante la sentencia throw y pueden ser “capturadas” con la cláusula catch.

Las excepciones se crean como cualquier objeto y lanzar o se pueden capturar y volver a lanzar la misma capturada.

Finally

Un bloque finally siempre se ejecuta si se maneja una excepción o no. Es un bloque opcional, y si hay uno, va después de un bloque try o catch.

```
try {
  // code that might throw an exception
} catch(Exception e) {
  // handle exception
} finally {
  // code that it's executed no matter what
}
try {
  // code that might throw an exception
} finally {
  // code that it's executed no matter what
}
```

Propagar excepciones

En Java, las excepciones se capturan o se propagan, es decir o las procesas o dejas que el elemento superior del stack realice la misma operación. Como sabemos, se capturan mediante `catch` y mediante throws, se declara que el método es capaz de lanzar una excepción de ese tipo o incapaz de capturar una que sus llamadas a otros métodos lancen.

En caso que ningún elemento del stack sea capaz de capturar esa excepción, máquina virtual aborta la ejecución.

```
void mymethod() throws MyProblematicException
```

Manejo de excepciones múltiples

El multi-catch permite capturar dos o más excepciones mediante la misma cláusula catch. Como aprendió anteriormente, es posible (de hecho, común) que un intento sea seguido por dos o más cláusulas catch. Aunque cada cláusula catch a menudo proporciona su propia secuencia de código única, no es raro tener situaciones en las que dos o más cláusulas catch ejecutan la misma secuencia de código aunque atrapen diferentes excepciones.

En lugar de tener que capturar cada tipo de excepción individualmente, puede usar una única cláusula de catch para manejar las excepciones sin duplicación de código.

Para crear un multi-catch, especifique una lista de excepciones dentro de una sola cláusula catch. Para ello, separe cada tipo de excepción en la lista con el operador OR. Cada parámetro multi-catch es implícitamente final. (Puede especificar explícitamente final, si lo desea, pero no es necesario.) Debido a que cada parámetro multi-catch es implícitamente final, no se le puede asignar un nuevo valor.

Aquí se ex capturar ArithmeticEx a única cláusula catch.

```
try {
      ...
}catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {
}
```

try con recursos

Otra opción es utilizar try con recursos, el cual puede utilizar si el recurso implementa la interface AutoCloseable. Esto es lo que la mayoría de los recursos Java hace. Cuando abres un recursos en la sentencia try, este se cierra automáticamente cuando el bloque try se ejecuta o cuando se maneja una excepción.

```
public void automaticallyCloseResource() {
      File file = new File("./tmp.txt");
      try (FileInputStream inputStream = new FileInputStream(file);) {
            // Uso de InputStream para leer un archivo

      } catch (FileNotFoundException e) {
      } catch (IOException e) {     }
}
```

Las excepciones son “lanzadas” mediante la sentencia `throw` y pueden ser “capturadas” con la clausula `catch`.

```
if(error){
      throw new NotEnoughMoneyException();
}
```

```
try {
      if(error){
            throw new NotEnoughMoneyException();
      }
} catch(Exception e){
} finally {
}
```

Buenas practicas

- No mostrar por consola los errores, si no, usa un sistema de log. Es decir, nunca uses System.out o System.err.
- Pon un sólo try y varios catch en el código siempre que sea posible, y en los catch captura y traza todas las excepciones. Ordenalos desde el más específico hasta el menos específico y usa la parte finally para liberar recursos, como conexiones o ficheros.
- Cuando escribas en el log, escribe información significativa, como los parámetros recibidos por el método, lo trivial. El nombre del método, por ejemplo, ya se verá en la traza.
- Al trazar, no concatenes el mensaje de la excepción, vuelca la excepción en sí (toda la traza).
- Un método no debería declararse como throws Exception, ya que enmascara todas las excepciones, sin permitir gestionar el error concreto. No pasa nada, por ejemplo, por que se lancen varias excepciones: throws IOException, MiOtraException... De esta forma, el que la invoque sabrá a qué tipos de error se debe enfrentar.

Asserts

Que es un assert?

Las aserciones son afirmaciones que puede usar para evaluar sus suposiciones sobre el código durante el desarrollo. Si la afirmación resulta ser falsa, se lanza un AssertionError.
Puede usar aserciones en dos formas:

```
private method(int i) {
  assert i > 0;
  //or
  assert i > 0 : "Parameter i must be a positive value"

  // Do something now that we know i is greater than 0
}
```

Habilitar y deshabilitar aserciones

El siguiente conmutador permite afirmaciones en varias granularidades:

```
java [ -enableassertions | -ea  ] [:<package name>"..." | :<class name> ]
```

Sin argumentos, el conmutador habilita aserciones de forma predeterminada. Con un argumento que termina en "...", las aserciones se activan en el paquete especificado y en los subpaquetes. Si el argumento es simplemente "...", las aserciones se habilitan en el paquete sin nombre en el directorio de trabajo actual. Con un argumento que no termina en "...", las aserciones están habilitadas en la clase especificada.:

```
java [ -disableassertions | -da ] [:<package name>"..." | :<class name> ]
```

Si una sola línea de comandos contiene varias instancias de estos conmutadores, se procesan en orden antes de cargar cualquier clase. Por ejemplo, para ejecutar un programa con aserciones activadas sólo en el paquete com.wombat.fruitbat (y cualquier subpaquete), se podría utilizar el siguiente comando:

```
java -ea:com.wombat.fruitbat... java -ea:com.wombat.fruitbat... <Main class>
```

Hay otra forma de controlar las aserciones: mediante programación, enganchando al objeto ClassLoader. JDK 1.4 agregó varios métodos nuevos a ClassLoader que permiten la activación y desactivación dinámica de aserciones, in
estado de aserci

```
public class LoaderAssertions {
  public static void main(String[] args) {
       ClassLoader.getSystemClassLoader().setDefaultAssertionStatus(true);
       new Loaded().go();
  }
}

class Loaded {
  public void go() {
       assert false: "Loaded.go()";
  }
}
```

Dado que se puede simular el efecto de aserciones utilizando otras construcciones de programación, se puede argumentar que el punto de agregar afirmaciones a Java es que son fáciles de escribir. Las declaraciones de aserción vienen en dos formas:

```
assert boolean-expression;
assert boolean-expression: information-expression;
```

Ambas declaraciones dicen "Afir
un valor verdadero". Si no es así
AssertionError. Esta es una subc
una especificación de excepción.

```
class Assert2 {
 public static void main(String[] args) {
     assert false: "Here's a message saying what happened";
  }
}
```

Las instrucciones de verificación son una adición valiosa a su código. Dado que las aserciones pueden ser deshabilitadas, las instrucciones de verificación deben utilizarse siempre que tenga conocimiento no obvio sobre el estado de su objeto o programa.

```
static {
	boolean assertionsEnabled = false;
	assert assertionsEnabled = true;
	if (!assertionsEnabled) {
		throw new RuntimeException("Assertions disabled");
	}
}
```

Colecciones y genéricos

Colecciones y genéricos

- API collections
- Set
- List
- Map
- Ordenar colecciones
- Genéricos

API Collections

El framework de las colecciones fue diseñado para cumplir varios objetivos, tales como:

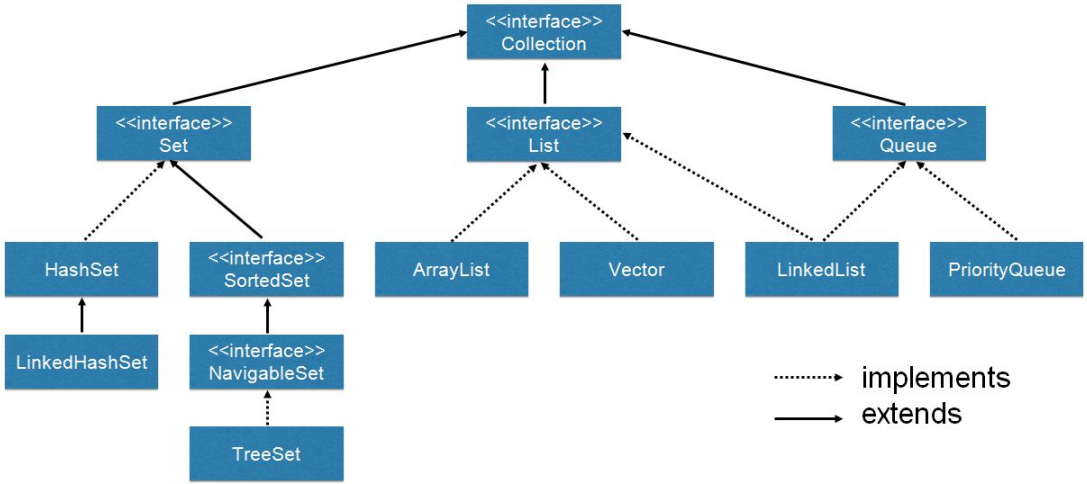
- Tener alto rendimiento. Las implementaciones para las colecciones fundamentales (matrices dinámicas, listas enlazadas, árboles y tablas hash) debían ser altamente eficientes.
- Permitir que diferentes tipos de colecciones funcionaran de manera similar y con un alto grado de interoperabilidad.
- Ampliar y / o adaptar una colección fácilmente.

Con este fin, todo el marco de colecciones está diseñado en torno a un conjunto de interfaces estándar. Se proporcionan varias implementaciones estándar, como LinkedList, HashSet y TreeSet , de estas interfaces que puede usar como están y también puede implementar su propia colección, si lo desea.

Un framework de colecciones es una arquitectura unificada para representar y manipular colecciones, donde todos contienen:

- **Interfaces** : son tipos de datos abstractos que representan colecciones. Las interfaces permiten manipular las colecciones independientemente de los detalles de su representación. En los lenguajes orientados a objetos, las interfaces generalmente forman una jerarquía.
- **Implementaciones**, es decir, clases - Estas son implementaciones concretas de las interfaces de colección. En esencia, son estructuras de datos reutilizables.
- **Algoritmos** : estos son los métodos que realizan cálculos útiles, como la búsqueda y clasificación, en objetos que implementan interfaces de colección. Se dice que los algoritmos son polimórficos: es decir, se puede usar el mismo método en muchas implementaciones diferentes de la interface de colección apropiada.

Además de las colecciones, el framework define varias interfaces y clases de mapas. Los mapas almacenan pares clave / valor. Aunque los mapas no son colecciones en el uso adecuado del término, pero están completamente integrados con las colecciones.


<<interface>>
Collection
<<interface>>
Set
<<interface>>
List
<<interface>>
Queue
HashSet
<<interface>>
SortedSet
ArrayList
Vector
LinkedList
PriorityQueue
LinkedHashSet
<<interface>>
NavigableSet
TreeSet
implements
extends

Tipos de colecciones

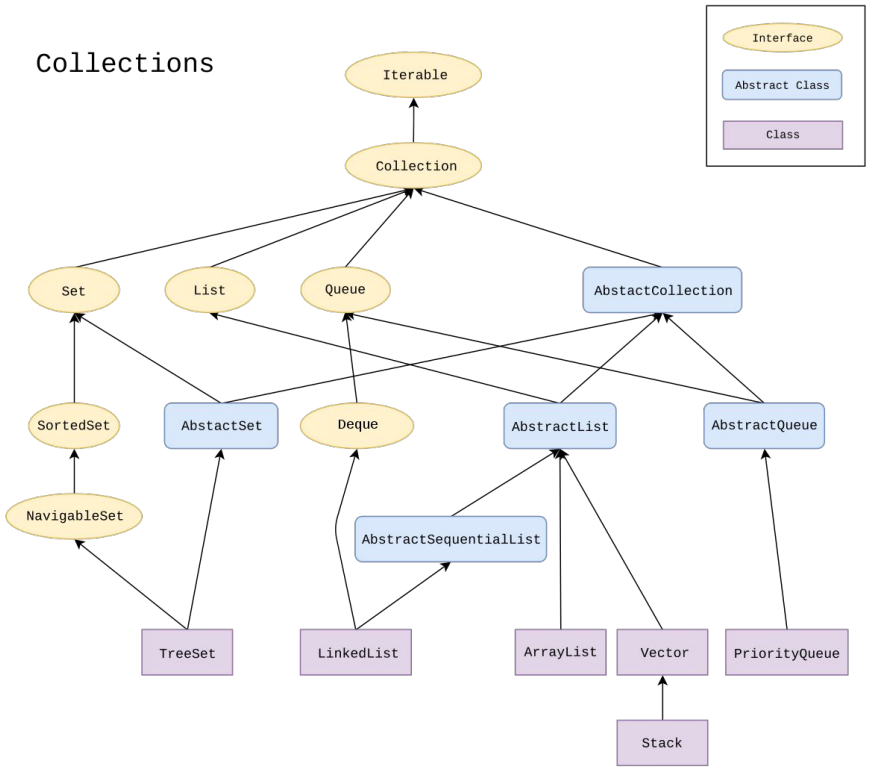
Hay tres tipos genéricos de colección: listas ordenadas, diccionarios/mapas y conjuntos. Las listas ordenadas permiten al programador insertar elementos en un cierto orden y recuperar esos elementos en el mismo orden. Un ejemplo es una lista de espera. Se incluyen dos interfaces en las listas ordenadas que son la interface List y la interface Queue.

Los diccionarios/mapas almacenan referencias a objetos con una clave de búsqueda para acceder a los valores del objeto. Un ejemplo de una clave es una tarjeta de identificación. La interface Map está incluida en los diccionarios/mapas.

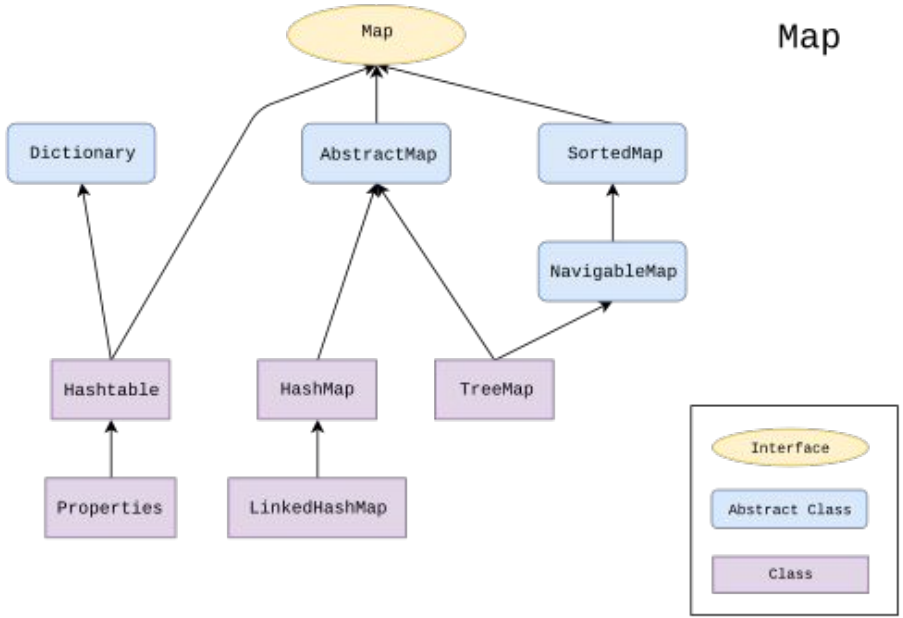
La interface Set, son colecciones desordenadas que pueden ser iteradas y donde no se permiten objetos similares. La interface Set está incluida.

- **Collection**: Esto le permite trabajar con grupos de objetos; está en la parte superior de la jerarquía de colecciones.
- **List**: Esto amplía la Colección y una instancia de Lista almacena una colección ordenada de elementos.
- **Set**: Esto amplía la Colección para manejar conjuntos, que deben contener elementos únicos.
- **SortedSet**: Esto extiende el Conjunto para manejar conjuntos ordenados.
- **Map**: Esto mapea claves únicas a los valores.
- **Map.Entry**: Esto describe un elemento (un par clave / valor) en un mapa. Esta es una clase interna de Mapa.
- **SortedMap**: Esto extiende el Mapa para que las claves se mantengan en orden ascendente.
- **Enumeration**: Esta es una interface heredada que define los métodos mediante los cuales puede enumerar (obtener uno a la vez) los elementos en una colección de objetos. Esta interface heredada ha sido superada por Iterator.

Jerarquía del framework.



Jerarquía del framework para mapas.



Implementaciones

- **AbstractCollection** : Implementa la mayor parte de la interface Collection.
- **AbstractList** : extiende de AbstractCollection e implementa la mayor parte de la interface List.
- **AbstractSequentialList**: extiende AbstractList fpara su uso por una colección que usa acceso secuencial en lugar de aleatorio de sus elementos.
- **LinkedList** : Implementa una lista vinculada extendiendo AbstractSequentialList..
- **ArrayList**: Implementa una matriz dinámica extendiendo AbstractList.
- **AbstractSet** : extiende AbstractCollection e implementa la mayoría de la interfaz Set.
- **HashSet** : extiende AbstractSet para usar con una tabla hash.
- **LinkedHashSet** : extiende HashSet para permitir iteraciones de orden de inserción.
- **TreeSet**: Implementa un conjunto almacenado en un árbol. Extiende AbstractSet.

- **AbstractMap**: Implementa la mayor parte de la interfaz del mapa.
- **HashMap**: extiende AbstractMap para usar una tabla hash.
- **TreeMap**: extiende AbstractMap para usar un árbol.
- **WeakHashMap**: extiende AbstractMap para usar una tabla hash con claves débiles.
- **LinkedHashMap** extiende HashMap para permitir iteraciones de orden de inserción.
- **IdentityHashMap** extiende AbstractMap y utiliza la igualdad de referencia al comparar documentos.

Las clases AbstractCollection, AbstractSet, AbstractList, AbstractSequentialList y AbstractMap proporcionan implementaciones esqueléticas de las interfaces de recopilación principales, para minimizar el esfuerzo requerido para implementarlas.

List

Las listas se utilizan para almacenar una secuencia de elementos. Puede insertar un elemento del contenedor en una posición específica utilizando un índice, y recuperar el mismo elemento más tarde (es decir, mantiene el orden de inserción). Puede almacenar duplicados elementos en una lista. Entre las implementaciones, hay dos clases concretas ArrayList y LinkedList.

ArrayList

Implementa una matriz redimensionable. Cuando se crea un array nativo (digamos, new String [10];), el tamaño del array es conocido (fijo) en el momento de la creación. Sin embargo, esta es una matriz dinámica, puede crecer en tamaño según sea necesario. Internamente, un ArrayList asigna un bloque de memoria y lo aumenta según sea necesario. Por lo tanto, acceder a los elementos del array es muy rápido en un ArrayList. Sin embargo, cuando agrega o elimina elementos, internamente el resto de los elementos se copian; así que la suma / eliminación de elementos es una operación costosa.

LinkedList

Utiliza internamente una lista doblemente vinculada. Por lo tanto, la inserción y eliminación es muy rápido en LinkedList.

Sin embargo, acceder a un elemento implica recorrer los nodos uno por uno, por lo que es lento. Cuando desee agregar o eliminar elementos con frecuencia en una lista de elementos, es mejor usar una LinkedList. Verá un ejemplo de LinkedList junto con la interfaz ListIterator.

Set

Set a diferencia de las listas, no contiene duplicados, no recuerdan donde insertó el elemento, es decir, no recuerda el orden de inserción. Hay dos clases concretas importantes para Set, HashSet y TreeSet.

Un HashSet es para insertar y recuperación de elementos; no mantiene ningún orden de clasificación de los elementos que contiene.

Un TreeSet almacena los elementos en un orden (e implementa la interfaz SortedSet).

HashSet

Set no permite duplicados, y puede ser utilizado para la inserción y la búsqueda rápidas. Así que puede utilizar un HashSet para resolver problemas de donde los elementos no se deben repetir.

TreeSet

Si recordamos que esta implementación almacena los elementos en un orden podemos resolver el problema de ordenamiento de elementos mediante este contenedor.

Map

Un Map almacena los pares clave y valor. La interfaz de Map no extiende la interfaz de Collection. Sin embargo, existen en la interfaz nombres de métodos similares para problemas similares, por lo que es fácil de entender y utilizar Map.

Existen dos importantes clases concretas de Map que son HashMap y TreeMap.

HashMap

Un HashMap utiliza una estructura de datos de tabla hash internamente. En HashMap, buscar (o acceder a elementos) es una operación rápida. Sin embargo, HashMap no recuerda el orden en el que inserta elementos ni los mantiene ordenados.

TreeMap

Un TreeMap utiliza internamente una estructura de datos red-black tree. A diferencia de HashMap, TreeMap mantiene los elementos ordenados (por sus claves). Por lo tanto, buscar o insertar es algo más lento que el HashMap.

NavigableMap

La interfaz NavigableMap extiende la interfaz de SortedMap. En la jerarquía Collection, la clase TreeMap es una clase ampliamente utilizada que implementa NavigableMap. Como indica el nombre, se puede navegar por el Map fácilmente. Tiene muchos métodos que facilitan la navegación por mapas. Puede obtener el valor más cercano que coincida con la clave valores menores que la clave dada, todos los valores mayores que la clave dada, etc.

Queue

Una Queue sigue el mecanismo FIFO: el primer elemento insertado se eliminará primero. Para obtener un comportamiento de cola, puede crear un objeto LinkedList y referir a través de una referencia de Queue.

Deque (Doubly ended queue)

Es una estructura de datos que le permite insertar y eliminar elementos de ambos extremo, la misma se introdujo en Java 6 en el paquete java.util.collection y extiende de la Interfaz Queue. Por lo tanto, todos los métodos proporcionados por Queue también están disponibles en la interfaz Deque.

Hashcode y equals

La sobreescritura de los métodos equals and hashCode de manera correcto es importante en los contenedores (de manera particular, HashMap and HashSet)

```
class Circle {
        private int xPos, yPos, radius;
        public Circle(int x, int y, int r) {
                xPos = x;
                yPos = y;
                radius = r;
        }
        @Override
        public boolean equals(Object arg) {
                if(arg == null) return false;
                if(this == arg) return true;
                if(arg instanceof Circle) {
                        Circle that = (Circle) arg;
                    if( (this.xPos == that.xPos) && (this.yPos == that.yPos)
                          && (this.radius == that.radius )) {
                          return true;
                    }
                }
                return false;
       }
}
```

Si invocamos este el código habiendo sobrescrito este método podemos comprobar casos como :

```
class TestCircle {
    public static void main(String []args) {
        Set<Circle> circleList = new HashSet<Circle>();
        circleList.add(new Circle(10, 20, 5));
        System.out.println(circleList.contains(new Circle(10, 20, 5)));
     }
}
```

Los métodos hashCode() y equals() necesitan ser consistentes para una clase. Para fines prácticos, asegúrese de que siga esta regla: el método hashCode()debe devolver el mismo valor hash para dos objetos si el método equals()devuelve true para ellos.

La clase Collections

Es una clase de utilidad de el API Collections.

Mediante métodos de utilidades estáticos, nos permite realizar operaciones comunes sobre las colecciones del framework, por ejemplo, los ordenamientos de listas.

```
void sort(List<T> list)
void sort(List<T> list, Comparator<? super T> c)
```

https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html

La clase Arrays

Similar a las Collections, Arrays es también una clase de utilidad (es decir, la clase sólo tiene métodos estáticos). Los métodos en Collections son también muy similares a los métodos en Arrays. La clase Collections es para clases de contenedor; la clase Arrays es para arrays nativos (es decir, matrices con [] sintaxis).

```
List<T> asList(T . . . a)
int binarySearch(Object[] objArray,Object key)
boolean equals(Object[] objArray1, Object[] objArray2)
void fill(Object[] objArray, Object val)
void sort(Object[] objArray)
String toString(Object[] a)
```

Ordenar colecciones

Mediante la clase de utilidad Collections o mediante el uso comparadores, el API framework nos permite ordenar de manera eficientes.

Para ello, utiliza componentes externos al mismo para realizar la comparación, mediante interfaces como Comparator.

Comparador y comparable

Interface Comparator

Tanto TreeSet como TreeMap almacenan elementos ordenados. Sin embargo, es el comparador el que define con precisión qué significa el orden ordenado.

La interface Comparator define dos métodos principales a implementar:

`int compare(obj1, obj2)`, donde obj1 y obj2 son comparables, retornando cero si son iguales, positivo si obj1 es mayor que obj2 y negativo sino.

`boolean equals(obj),` que sobrecarga al de Object y solo se aplica cuando el objeto es comparador de si mismo. Sigue el mismo criterio que el de Object.

Interface Comparable

La interface Comparable define un método `compareTo(obj1),` que permite comparar el objeto con otro del tipo o subtipo.

Al igual que en el comparador, retornando un entero negativo, cero o un entero positivo si el objeto (this) es menor que, igual o mayor que el objeto especificado

Genéricos

Genéricos

Los métodos genéricos de Java y las clases genéricas permiten a los programadores especificar, con una única declaración de método, un conjunto de métodos relacionados, o con una única declaración de clase, un conjunto de tipos relacionados, respectivamente.

Los genéricos también proporcionan seguridad de tipo en tiempo de compilación que permite a los programadores detectar tipos no válidos en tiempo de compilación.

Usando el concepto genérico de Java, podríamos escribir un método genérico para clasificar una matriz de objetos, luego invocar el método genérico con matrices de enteros, matrices dobles, matrices de cadenas, etc., para ordenar los elementos de la matriz.

Métodos genéricos

Puede escribir una única declaración de método genérico que se puede llamar con argumentos de diferentes tipos. En función de los tipos de argumentos pasados al método genérico, el compilador maneja cada llamada de método de manera apropiada. Las siguientes son las reglas para definir métodos genéricos:

- Todas las declaraciones de métodos genéricos tienen una sección de parámetros de tipo delimitada por corchetes angulares (<y>) que preceden al tipo de retorno del método (<E> en el siguiente ejemplo).
- Cada sección de parámetros de tipo contiene uno o más parámetros de tipo separados por comas. Un parámetro de tipo, también conocido como variable de tipo, es un identificador que especifica un nombre de tipo genérico.

- Los parámetros de tipo se pueden usar para declarar el tipo de retorno y actuar como marcadores de posición para los tipos de argumentos pasados al método genérico, que se conocen como argumentos de tipo real.
- El cuerpo de un método genérico se declara como el de cualquier otro método. Tenga en cuenta que los parámetros de tipo solo pueden representar tipos de referencia, no tipos primitivos (como int, double y char).

- Generics se asegurará de que cualquier intento de agregar elementos de tipos distintos de los especificados tipo (s) se capturará en tiempo de compilación. Por lo tanto, los genéricos ofrecen una implementación con seguridad de tipo.
- Java 7 introdujo la sintaxis del diamante donde los parámetros del tipo pueden omitirse. El compilador inferirá los tipos de la declaración de tipado.
- Los genéricos no son covariantes. Es decir, el subtipado no funciona con los genéricos; no puede asignar una subtipo a un parámetro de tipo base.
- El <?> Especifica un tipo desconocido en genéricos y se conoce como comodín. Por ejemplo, List <?> Se refiere a la lista de desconocidos.
- Los comodines pueden ser limitados (bounded wildcars). Por ejemplo, <? extends Runnable> especifica que puede coincidir cualquier tipo, siempre y cuando sea Runnable o cualquiera de sus tipos derivados. Tenga en cuenta que se extiende es inclusivo, por lo que puede reemplazar X en? se extiende X. Sin embargo, en <? super Runnable>,? coincidiría sólo con la super tipos de Runnable y Runnable no coinciden (es decir, es una cláusula exclusiva).

I/O API

Qué es Java IO?

El lenguaje Java proporciona un modelo simple para entrada y salida (E / S). Todo I/O es realizado escribiendo y leyendo de secuencias de datos. Los datos pueden existir en un archivo o una matriz, se canaliza desde otra secuencia, o incluso proviene de un puerto en otra computadora.

La flexibilidad de este modelo lo convierte en una poderosa abstracción de cualquier entrada y salida requerida.

El paquete java.io contiene casi todas las clases que pueda necesitar para realizar entradas y salidas (E / S) en Java. Todas estas secuencias representan una fuente de entrada y un destino de salida. La secuencia en el paquete java.io admite muchos datos, como primitivas, objetos, caracteres localizados, etc.

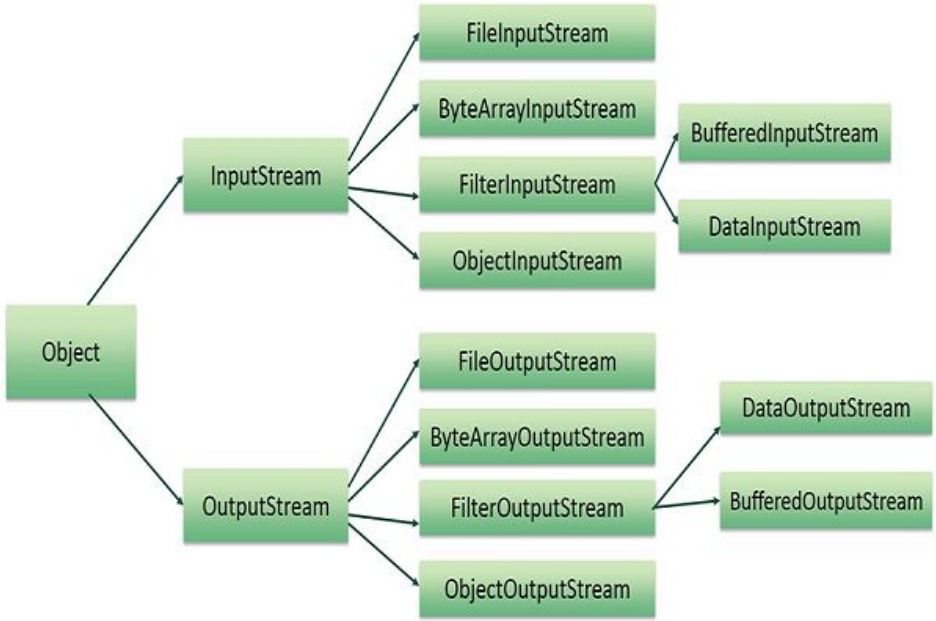
Tipos de Streams

Una secuencia se puede definir como una secuencia de datos. Hay dos tipos de IO Streams:

InputStream: se utiliza para leer datos de una fuente.

OutputStream: se utiliza para escribir datos en un destino.

Java proporciona soporte fuerte pero flexible para E / S relacionadas con archivos y redes



Jerarquía del API

Implementaciones

Byte Streams

Las secuencias de bytes de Java se utilizan para realizar entradas y salidas de bytes de 8 bits. Aunque hay muchas clases relacionadas con las secuencias de bytes, las clases más utilizadas son, FileInputStream y FileOutputStream.

Character Streams

Las secuencias de Java Byte se utilizan para realizar entradas y salidas de bytes de 8 bits, mientras que las secuencias de Java Character Streams se utilizan para realizar entradas y salidas para unicode de 16 bits.

Standard Streams

Todos los lenguajes de programación brindan soporte para E / S estándar donde el programa del usuario puede tomar entrada desde un teclado y luego producir una salida en la pantalla de la computadora. Si conoce los lenguajes de programación C o C ++, debe tener en cuenta tres dispositivos estándar STDIN, STDOUT y STDERR. Del mismo modo, Java proporciona las siguientes tres corrientes estándar:

- Entrada estándar: se usa para alimentar los datos al programa del usuario y generalmente se usa un teclado como flujo de entrada estándar y se representa como `System.in`.
- Salida estándar: se utiliza para generar los datos producidos por el programa del usuario y, por lo general, se utiliza una pantalla de computadora para el flujo de salida estándar y se representa como `System.out`.
- Error estándar: se utiliza para generar los datos de error producidos por el programa del usuario y, por lo general, se utiliza una pantalla de computadora para el flujo de error estándar y se representa como `System.err`.

Leyendo y grabando ficheros

Una secuencia se puede definir como una secuencia de datos. InputStream se utiliza para leer datos de una fuente y OutputStream se utiliza para escribir datos en un destino.

java.io

La clase File

File

Es una clase asociada a las operaciones de gestión de ficheros. Proporciona información sobre los ficheros a nivel de sistema operativo, como nombres, directorios, distintos tipos de rutas, o atributos. Permite realizar operaciones globales sobre ficheros y directorios, como por ejemplo, borrados o cambios de nombre.

Dispone de varios constructores para crear objetos de tipo File, donde necesario pasarles el nombre del fichero y/o el del directorio donde se va a crear.

```
File (String path_and_filename)
File (String path, String filename)
File (File path, String filename)
```

Métodos

getName(), devuelve el nombre del fichero o directorio.
getPath(), devuelve la ruta relativa.
getAbsolutePath(), devuelve la ruta absoluta.
getParent(), devuelve el directorio padre o null.
canRead(), devuelve cierto si el fichero se puede leer.
canWrite(), devuelve cierto si el fichero se puede escribir.
length(), devuelve el tamaño del fichero en bytes.
createNewFile(), crea un nuevo fichero si no existe.
mkdir(), crea un directorio.

delete(), borra el fichero o directorio.
renameTo(), cambia el nombre con el nuevo argumento.
exists(), devuelve cierto si el fichero o directorio existe.
isDirectory(), devuelve cierto si el objeto File está asociado a un directorio.
isFile(), devuelve cierto si el objeto File está asociado a un fichero.
list(), devuelve una lista con los nombres de los ficheros o directorios.

Atributos de la clase File

Hay que tener en cuenta los distintos separadores de directorios o carpetas que utilizan los diferentes sistemas operativos, por ejemplo en Linux será “/”, mientras que en Windows es “\”. También los sistemas operativos difieren en el carácter utilizado para separar rutas, por ejemplo, en variables del sistema.

Para ayudar a las aplicaciones para que puedan intentar ser independientes del sistema operativo, se proporcionan los siguientes atributos de File:

```
File.pathSeparator Separador de rutas.
File.pathSeparatorChar. Separador de rutas.
File.separator.  Separador de carpetas.
File.separatorChar. Separador de carpetas.
```

Directorios en Java

Directorios

- Un directorio es un archivo que puede contener una lista de otros archivos y directorios.
- Utiliza el objeto **File** para crear directorios, para listar los archivos disponibles en un directorio.
  Para obtener detalles completos, consulte una lista de todos los métodos a los que puede llamar en el objeto Archivo y que están relacionados con los directorios.

Crear directorios

Hay dos métodos útiles de utilidad de archivos, que se pueden usar para crear directorios, **mkdir** que crea un directorio, devuelve true en caso de éxito y false en caso de error cuando la ruta especificada en el objeto **File** ya existe o que el directorio no se puede crear porque toda la ruta aún no existe y el método **mkdirs**() crea un directorio y todos los padres del directorio.

Listando directorios

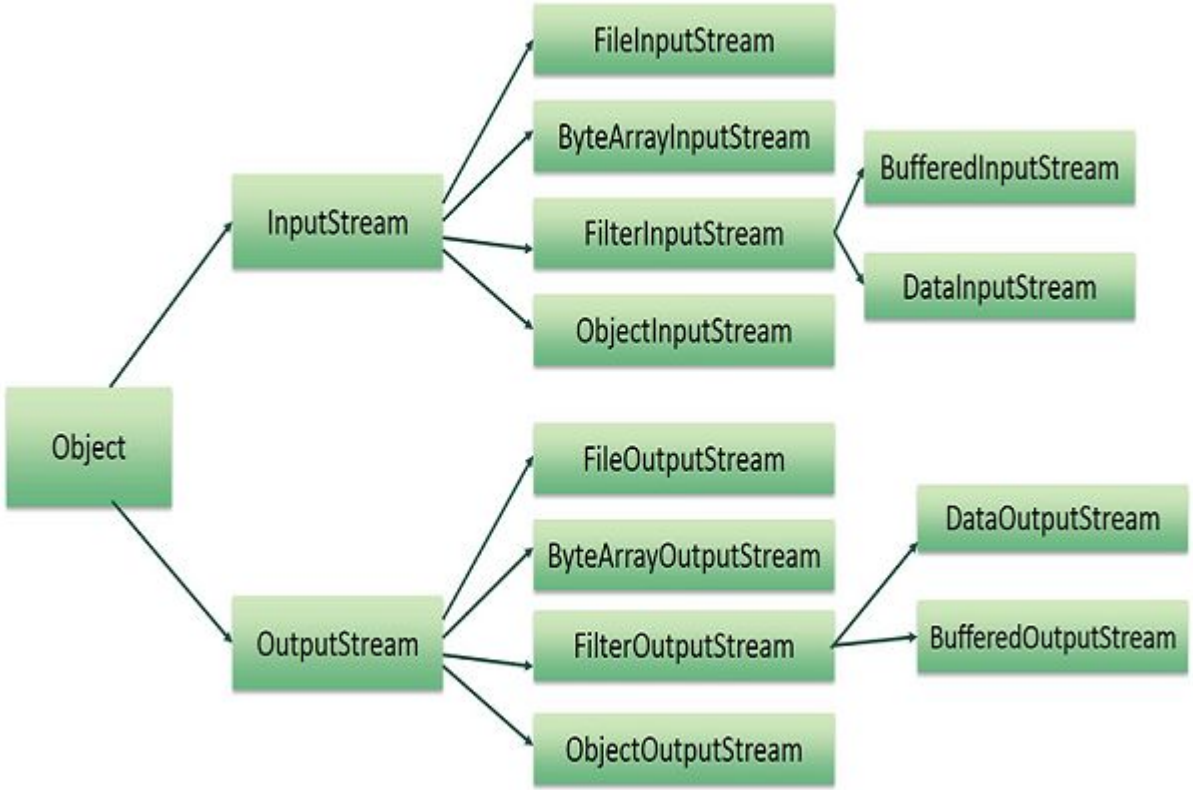
Puede usar el método list() proporcionado por el objeto File para listar todos los archivos y directorios disponibles en un directorio.

Input y Output Stream

El paquete IO de Java se ocupa principalmente de la lectura de datos sin procesar de un origen y la escritura de datos sin procesar en un destino.

Las fuentes y destinos de datos más típicos son estos:

```
Files
Pipes
Network Connections
In-memory Buffers
System.in, System.out, System.error
```



Jerarquía de IO Streams

Una secuencia se puede definir como una secuencia de datos. `InputStream` se utiliza para leer datos de una fuente y `OutputStream` se utiliza para escribir datos en un destino.

IO Streams es un concepto central en Java IO. Un flujo es un flujo de datos conceptualmente sin fin. Puedes leer desde una secuencia o escribir en una secuencia. Un flujo está conectado a una fuente de datos o un destino de datos. Las transmisiones en Java IO pueden estar basadas en bytes (lectura y escritura de bytes) o en caracteres (lectura y escritura de caracteres).

InputStream, OutputStream, Reader y Writer
Un programa que necesita leer datos de alguna fuente necesita un InputStream o un Reader . Un programa que necesita escribir datos en algún destino necesita un

Implementaciones

BufferedInputStream	Contiene métodos para leer bytes del búfer (área de memoria).
BufferedOutputStream	Contiene métodos para escribir bytes en el búfer.
ByteArrayInputStream	Contiene métodos para leer bytes de un array de bytes
ByteArrayOutputStream	Contiene métodos para escribir bytes en un array de bytes.
DataInputStream	Contiene métodos para leer tipos de datos primitivos de Java.
DataOutputStream	Contiene métodos para escribir tipos de datos primitivos de Java.
FileInputStream	Contiene métodos para leer bytes de un archivo.
FileOutputStream	Contiene métodos para escribir bytes en un archivo.
FilterInputStream	Contiene métodos para leer bytes de otros flujos de entrada que utiliza como fuente de datos.
FilterOutputStream	Contiene métodos para escribir en otros flujos de salida.
InputStream	Clase abstracta que describe la entrada del stream.

OutputStream	Clase abstracta que describe la salida del stream.
ObjectInputStream	Contiene métodos para leer objetos.
ObjectOutputStream	Contiene métodos para escribir objetos.
PipedInputStream	Un flujo de entrada canalizado debe conectarse a un flujo de salida canalizado.
PipedOutputStream	Contiene métodos para escribir en un flujo de salida canalizado.
PrintStream	Flujo de salida que contiene print() y println()
PushbackInputStream	Flujo de entrada que permite que los bytes se devuelvan al stream.
SequenceInputStream	Contiene métodos para concatenar múltiples flujos de entrada y luego leer de el flujo combinado.

O podríamos agruparlos de esta manera.

	Byte Based		Character Based	
	Input	Output	Input	Output
Basic	InputStream	OutputStream	Reader InputStreamReader	Writer OutputStreamWriter
Arrays	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
Files	FileInputStream RandomAccessFile	FileOutputStream RandomAccessFile	FileReader	FileWriter
Pipes	PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
Buffering	BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
Filtering	FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
Parsing	PushbackInputStream StreamTokenizer		PushbackReader LineNumberReader	
Strings			StringReader	StringWriter
Data	DataInputStream	DataOutputStream		
Data - Formatted		PrintStream		PrintWriter
Objects	ObjectInputStream	ObjectOutputStream		
Utilities	SequenceInputStream			

File IO Stream

FileOutputStream

La clase FileOutputStream hace posible escribir un archivo como un flujo de bytes. La clase FileOutputStream es una subclase de OutputStream lo que significa que puede usar un FileOutputStream como OutputStream .

```
OutputStream output =
        new FileOutputStream("c:\\data\\output-text.txt");

while(moreData) {
  int data = getMoreData();
  output.write(data);
}
output.close();
```

FileInputStream

La clase Java FileInputStream permite leer el contenido de un archivo como un flujo de bytes. La clase Java FileInputStream es una subclase de Java InputStream . Esto significa que utiliza Java FileInputStream como InputStream (FileInputStream comporta como InputStream).

```
InputStream input = new FileInputStream("input-text.txt");

int data = input.read();
while(data != -1) {
  //do something with data...
  doSomethingWithData(data);

  data = input.read();
}
input.close();
```

Pipes IO Stream

Pipes

Pipes en Java IO proporciona la capacidad de dos subprocesos que se ejecutan en la misma JVM para comunicarse. Por lo tanto, también pueden ser fuentes o destinos de datos.

No puede usar una Pipe para comunicarse con un hilo en una JVM diferente (proceso diferente), las partes que se comunican deben estar ejecutándose en el mismo proceso y deben ser hilos diferentes.

La creación de una tubería utilizando Java IO se realiza a través de las clases `PipedOutputStream` y `PipedInputStream` . Un `PipedInputStream` debe estar conectado a un `PipedOutputStream` . Los datos escritos en el `PipedOutputStream` por un hilo pueden leerse desde el `PipedInputStream` conectado por otro hilo.

```
final PipedOutputStream output = new PipedOutputStream();

final PipedInputStream  input  = new PipedInputStream(output);

Thread thread1 = new Thread(new Runnable() {
        @Override
        public void run() {
        try {
          output.write("Hello world, pipe!".getBytes());
                } catch (IOException e) {
                }
            }
        });
```

```
Thread thread2 = new Thread(new Runnable() {
    @Override
    public void run() {
        try {
            int data = input.read();
            while(data != -1){
                System.out.print((char) data);
                data = input.read();
            }
        } catch (IOException e) {
        }
    }
});

thread1.start();
thread2.start();
```

Readers y Writers

Reader & Writer

La clase Java Reader es la clase base para todas las subclases de Reader en la API IO de Java. Un Reader es como un InputStream excepto que se basa en caracteres en lugar de en bytes. En otras palabras, un Java Reader está destinado a leer texto, mientras que un InputStream está diseñado para leer bytes en bruto.

La clase de Java Writer es la clase base para todas las subclases de Writer en la API de IO de Java. Un Writer es como un OutputStream excepto que se basa en caracteres en lugar de en bytes. En otras palabras, un Writer está diseñado para escribir texto, mientras que un OutputStream está diseñado para escribir bytes sin procesar.

Ambas poseen enconding.

```
Reader reader = new BufferedReader(new FileReader(...));

Writer writer = new BufferedWriter(new FileWriter(...));
```

PrintWriter

La clase Java `PrintWriter` permite escribir datos formateados en un Writer subyacente. Por ejemplo, escribir int , long y otros datos primitivos formateados como texto, en lugar de como sus valores de byte.

El Java `PrintWriter` es útil si está generando informes (o similares) donde tiene que mezclar texto y números. La clase PrintWriter tiene todos los mismos métodos que PrintStream excepto los métodos para escribir bytes en bruto. Al ser una subclase de Writer PrintWriter está diseñado para escribir texto.

```
FileWriter writer = new FileWriter ("data / report.txt");

 try (PrintWriter printWriter =
     nuevo PrintWriter (escritor)) {

     printWriter.write ("Hello World");
     printWriter.write ((int) 123);

 }
```

FileReader

La clase Java FileReader hace posible leer el contenido de un archivo como una secuencia de caracteres. Funciona de manera muy similar a FileInputStream excepto que FileInputStream lee bytes, mientras que FileReader lee caracteres. El FileReader está destinado a leer texto, en otras palabras. Un carácter puede corresponder a uno o más bytes dependiendo del esquema de codificación de caracteres.

```
Reader fileReader = new FileReader("input-text.txt");

int data = fileReader.read();
while(data != -1) {
  //do something with data...
  doSomethingWithData(data);
  data = fileReader.read();
}
fileReader.close();
```

StringReader

La clase StringReader Java StringReader permite convertir una String normal en un Reader . Esto es útil si tiene datos como una cadena pero necesita pasar esa cadena a un componente que sólo acepta un Reader .

```
String input = "Hola Mundo String... ";
StringReader stringReader = new StringReader(input);

int data = stringReader.read();
while(data != -1) {
  //do something with data...
  doSomethingWithData(data);

  data = stringReader.read();
}
stringReader.close();
```

System.in, System.out, y System.error

Son fuentes o destinos comunes de datos. El más utilizado es probablemente System.out para escribir resultados en la consola desde los programas de la consola, que son iniciadas por el tiempo de ejecución de Java cuando se inicia una JVM, por lo que no tiene que crear una instancia de las transmisiones usted mismo (aunque puede intercambiarlas en el tiempo de ejecución).

System.in es un InputStream que normalmente está conectado a la entrada del teclado de los programas de la consola. System.in no se usa con tanta frecuencia, ya que los datos se pasan comúnmente a una aplicación Java de línea de comandos a través de argumentos de línea de comandos o archivos de configuración.

System.out es un PrintStream . System.out normalmente genera los datos que escribe en la consola. Esto se usa a menudo desde programas de consola solamente como herramientas de línea de comandos. Esto también se usa a menudo para imprimir declaraciones de depuración de un programa (aunque podría decirse que no es la mejor manera de obtener información de depuración de un programa).

Serialización de Objetos

Serializable

La interfaz Java Serializable (`java.io.Serializable`) es una interfaz de marcador que sus clases deben implementar si se van a serializar y deserializar. La serialización (escritura) de objetos Java se realiza con ObjectOutputStream y la deserialización (lectura) se realiza con ObjectInputStream . La misma no contiene métodos. Por lo tanto, una clase que implementa Serializable no tiene que implementar ningún método específico. La implementación de Serializable tanto simplemente le dice a las clases de serialización de Java que esta clase está diseñada para la serialización de objetos.

```
import java.io.Serializable;

public class Person implements Serializable {
    public String name = null;
    public int age = 0;
}
```

serialVersionUID

La variable serialVersionUID es utilizada por la API de serialización de objetos de Java para determinar si un objeto deserializado fue serializado (escrito) con la misma versión de la clase, ya que ahora está intentando deserializarlo.

Imagina que un objeto Person está serializado en el disco. Entonces se hace un cambio a la clase Person . A continuación, intenta deserializar el objeto Person almacenado. Ahora el objeto Person serializado puede no corresponder a la nueva versión de la clase Person .Para detectar tales problemas, una clase que implementa Serializable debe contener un campo serialVersionUID . Si realiza grandes cambios en la clase, también debe cambiar su valor serialVersionUID .

El SDK de Java y muchos IDE de Java contienen herramientas para generar el serialVersionUID para que no tenga que hacerlo.

https://www.oracle.com/technetwork/articles/java/javaserial-1536170.html

Realidad

En el mundo actual, muchos proyectos Java serializan objetos Java utilizando diferentes mecanismos que el mecanismo de serialización Java, como por ejemplo, los objetos Java se serializan en JSON, BSON u otros formatos binarios más optimizados. Esto tiene la ventaja de que los objetos también son legibles por aplicaciones que no son Java. Por ejemplo, JSON.

Estos otros mecanismos de serialización de objetos normalmente no requieren que sus clases Java implementen Serializable , por cierto. Normalmente están utilizando Java Reflection para inspeccionar su clase, por lo que la implementación de la interfaz Serializable sería superflua, no agregaría ninguna información útil.

ObjectInputStream /ObjectInputStream

La clase Java `ObjectInputStream` le permite leer objetos Java desde un InputStream lugar de solo bytes sin procesar. Envuelve un InputStream en un `ObjectInputStream` y luego puede leer los objetos desde él. Por supuesto, los bytes

```
ObjectOutputStream objectOutputStream =
    new ObjectOutputStream(new FileOutputStream("data/person.bin"));
Person person = new Person();
person.name = "Francisco Philip";
person.age  = 25;
objectOutputStream.writeObject(person);
objectOutputStream.close();

ObjectInputStream objectInputStream =
            new ObjectInputStream(new FileInputStream("data/person.bin"));
Person personReaded = (Person) objectInputStream.readObject();
objectInputStream.close();
System.out.println(personRead.name);
System.out.println(personRead.age);
```

La clase Scanner

La clase Scanner

`Scanner` es una clase en el paquete `java.util` utilizada para obtener la entrada de los tipos primitivos como int, double etc. y también String. Es la forma más fácil de leer datos en un programa Java, aunque no es muy eficiente si se quiere un método de entrada para escenarios donde el tiempo es una restricción, como en la programación competitiva.

Para crear un objeto de clase `Scanner`, normalmente pasamos el objeto predefinido `System.in`, que representa el flujo de entrada estándar. Podemos pasar un objeto de clase `File` si queremos leer la entrada de un archivo.

Para leer valores numéricos de un determinado tipo de datos XYZ, la función que se utilizará es nextXYZ(). Por ejemplo, para leer un valor de tipo short, podemos usar nextShort().

Para leer cadenas (strings), usamos `nextLine()`.

Para leer un solo carácter, se usa next().charAt(0).

La función `next()`devuelve el siguiente token/palabra en la entrada como cadena y la función charAt (0) devuelve el primer carácter de esa cadena.

Un ejemplo de uso simple, donde el origen de los datos es el System.in

```
import java.util.Scanner;

public class Demo {

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println(“ingrese su nombre”);
    String name = sc.nextLine();
    System.out.println(“ingrese su edad”);
    int age = sc.nextInt();
    System.out.println("Nombre: "+name);
    System.out.println("Edad: "+age);
  }

}
```

Por otro lado, la clase nos permite evaluar si existe otra entrada pendiente.

```
import java.util.Scanner;

public class Demo {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int sum = 0, count = 0;
    while (sc.hasNextInt()) {
      int num = sc.nextInt();
      sum += num;
      count++;
    }
    int mean = sum / count;
    System.out.println("La media es: " + mean);
   }
}
```

Input Parsing

Parsing

Algunas de las clases en la API de IO de Java están diseñadas para ayudarlo a analizar la entrada. Estas clases son:

```
PusbackInputStream
PusbackReader
StreamTokenizer
PushbackReader
LineNumberReader
```

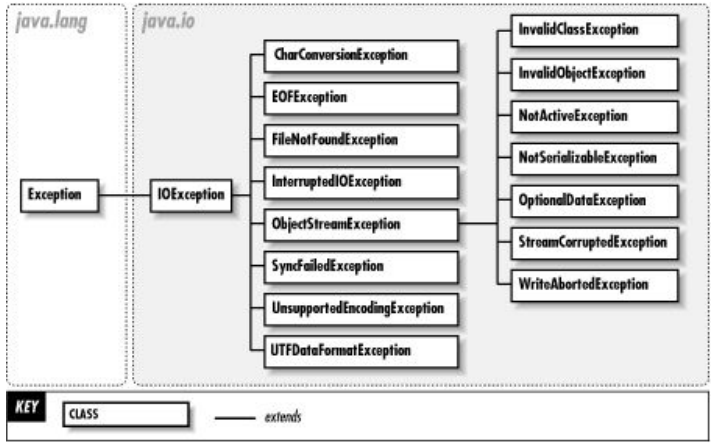
Nos permiten dar más funcionalidades que nos simplificarán la implementación.

IOException

Manejo de errores

Todo error producido por el uso del API IO de Java genera excepciones.

Como vemos, todas heredan de IOException, y pueden ser procesadas y tratadas de manera discriminada.



autoclosable

Como ya comentamos, Java 7 nos proporciona una solución a la tediosa gestión de las excepciones de este tipo. Para ello el uso de try-with-resources es un recurso ideal y muy utilizado.

All Known Subinterfaces:

AsynchronousByteChannel, AsynchronousChannel, BaseStream<T,S>, ByteChannel, CachedRowSet, CallableStatement, Channel, Clip, Closeable, Connection, DataLine, DirectoryStream<T>, DoubleStream, FilteredRowSet, GatheringByteChannel, ImageInputStream, ImageOutputStream, InterruptibleChannel, IntStream, JavaFileManager, JdbcRowSet, JMXConnector, JoinRowSet, Line, LongStream, MidiDevice, MidiDeviceReceiver, MidiDeviceTransmitter, Mixer, MulticastChannel, NetworkChannel, ObjectInput, ObjectOutput, Port, PreparedStatement, ReadableByteChannel, Receiver, ResultSet, RMIConnection, RowSet, ScatteringByteChannel, SecureDirectoryStream<T>, SeekableByteChannel, Sequencer, SourceDataLine, StandardJavaFileManager, Statement, Stream<T>, SyncResolver, Synthesizer, TargetDataLine, Transmitter, WatchService, WebRowSet, WritableByteChannel

All Known Implementing Classes:

AbstractInterruptibleChannel, AbstractSelectableChannel, AbstractSelector, AsynchronousFileChannel, AsynchronousServerSocketChannel, AsynchronousSocketChannel, AudioInputStream, BufferedInputStream, BufferedOutputStream, BufferedReader, BufferedWriter, ByteArrayInputStream, ByteArrayOutputStream, CharArrayReader, CharArrayWriter, CheckedInputStream, CheckedOutputStream, CipherInputStream, CipherOutputStream, DatagramChannel, DatagramSocket, DataInputStream, DataOutputStream, DeflaterInputStream, DeflaterOutputStream, DigestInputStream, DigestOutputStream, FileCacheImageInputStream, FileCacheImageOutputStream, FileChannel, FileImageInputStream, FileImageOutputStream, FileInputStream, FileLock, FileOutputStream, FileReader, FileSystem, FileWriter, FilterInputStream, FilterOutputStream, FilterReader, FilterWriter, Formatter, ForwardingJavaFileManager, GZIPInputStream, GZIPOutputStream, ImageInputStreamImpl, ImageOutputStreamImpl, InflaterInputStream, InflaterOutputStream, InputStream, InputStream, InputStream, InputStreamReader, JarFile, JarInputStream, JarOutputStream, LineNumberInputStream, LineNumberReader, LogStream, MemoryCacheImageInputStream, MemoryCacheImageOutputStream, MLet, MulticastSocket, ObjectInputStream, ObjectOutputStream, OutputStream, OutputStream, OutputStream, OutputStreamWriter, Pipe.SinkChannel, Pipe.SourceChannel, PipedInputStream, PipedOutputStream, PipedReader, PipedWriter, PrintStream, PrintWriter, PrivateMLet, ProgressMonitorInputStream, PushbackInputStream, PushbackReader, RandomAccessFile, Reader, RMIConnectionImpl, RMIConnectionImpl_Stub, RMIConnector, RMIIIOPServerImpl, RMIJRMPServerImpl, RMIServerImpl, Scanner, SelectableChannel, Selector, SequenceInputStream, ServerSocket, ServerSocketChannel, Socket, SocketChannel, SSLServerSocket, SSLSocket, StringBufferInputStream, StringReader, StringWriter, URLClassLoader, Writer, XMLDecoder, XMLEncoder, ZipFile, ZipInputStream, ZipOutputStream

https://docs.oracle.com/javase/8/docs/api/java/lang/AutoCloseable.html

Garbage Collector

Garbage Collector

La recolección automática de basura es el proceso de mirar la memoria del heap, identificar qué objetos están en uso y cuáles no, y eliminar los objetos no utilizados. Un objeto en uso, o un objeto referenciado, significa que alguna parte de su programa aún mantiene un puntero a ese objeto. Cualquier parte de su programa ya no hace referencia a un objeto no utilizado, u objeto no referenciado. Por lo tanto, la memoria utilizada por un objeto sin referencia puede ser reclamada.

En un lenguaje de programación como C, asignar y desasignar memoria es un proceso manual. En Java, el proceso de desasignar la memoria se maneja automáticamente por el recolector de basura. El proceso básico se puede describir a continuación.

Como funciona?

El recolector de basura está bajo el control de JVM. JVM ejecutará el recolector de basura cuando detecte que la memoria se está agotando. Puede solicitar a la JVM que ejecute el recolector de basura, pero no hay garantía de que conceda su solicitud.

Cuando se ejecuta el recolector de basura, su propósito es encontrar y eliminar los objetos que no se pueden alcanzar.

El objeto se vuelve elegible para la recolección de elementos no utilizados cuando no se puede acceder a él por ningún hilo en vivo.

Si en nuestro programa java, hay una variable de referencia que se refiere a un objeto y esa variable de referencia está disponible para los subprocesos en vivo, entonces el objeto es accesible.

```
Dog d = new Dog();
```

Cuando decimos new en la clase Dog, el objeto Dog se crea en el heap y se refiere a la variable de referencia d. Cuando la variable de referencia muere o no más en el alcance, el objeto se vuelve elegible para la recolección de basura.

La recolección de basura no puede ser forzada, solo podemos solicitar a JVM que ejecute la recolección de basura para liberar la memoria. La

Haciendo objetos elegibles explícitamente

Asignando un nulo a una variable de referencia

Un objeto se convierte en elegible para la recolección de basura cuando no hay referencias accesibles a él. Eliminar una referencia a un objeto es establecer la variable de referencia en nulo.

Objetos de las islas.

Una clase que tiene una variable de instancia que es una variable de referencia para otra instancia de la misma clase. Si se eliminan todas las demás referencias a estos dos objetos, aunque tengan una referencia válida entre sí, pero ningún hilo en vivo tendrá acceso a ninguno de los objetos. Cuando el recolector de basura se ejecuta, descubre tales islas de objetos y los elimina.

El método finalize ()

A veces, un objeto deberá realizar ciertas tareas antes de que se destruya. Ejemplo: si un objeto contiene algún recurso que no sea Java, tenemos que asegurarnos de que estos recursos se congelan antes de que el objeto se destruya.

Para manejar este tipo de situaciones, java proporciona un mecanismo llamado finalizar, mediante el cual podemos definir la acción que debe realizar el recolector de basura.

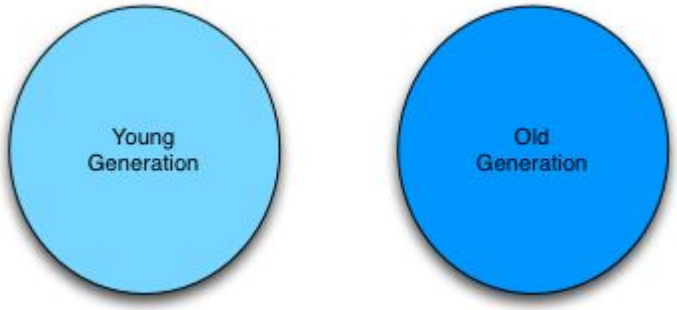
Cada clase hereda el método `finalize()` de `Object`.

El recolector de basura llama al método finalize cuando no hay más referencias al objeto existente.

El método de finalización nunca se ejecutará más de una vez en ningún objeto.

Tipos de memoria

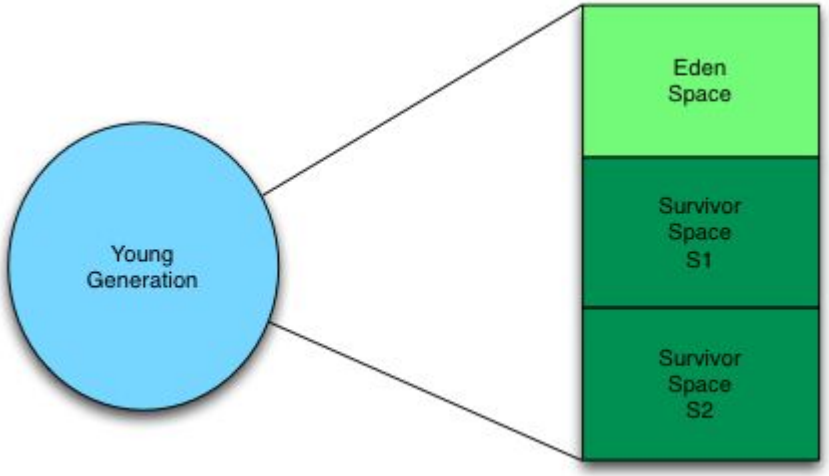
El Java Garbage Collector es uno de los conceptos que más cuesta entender a la gente cuando empieza a programar en Java. ¿Cómo funciona el Java Garbage Collector exactamente?. Java divide la memoria en dos bloques fundamentales , Young generation y Old generation.



Young Generation

En la zona de Young Generation se almacenan los objetos que se acaban de construir en el programa . Esta zona de memoria se divide en Eden Space ,Survivor Space (S0 y S1)

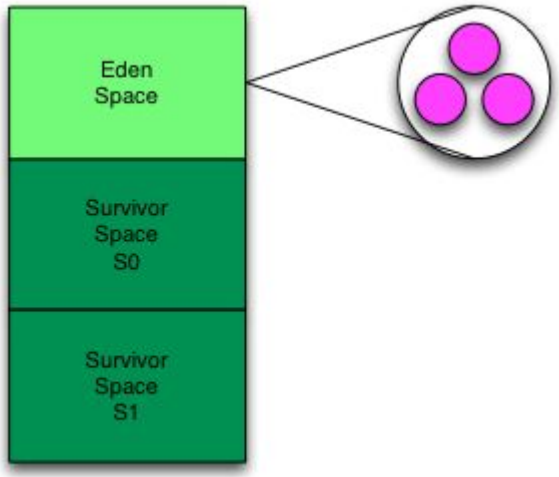
java Garbage Collector Eden



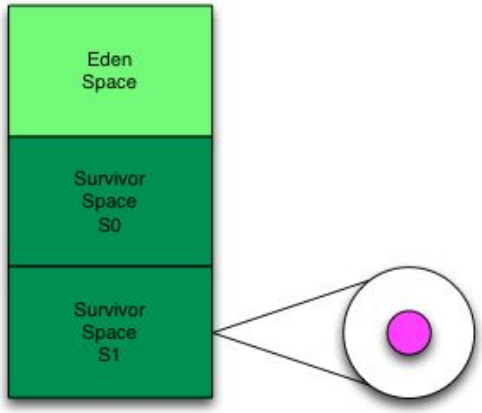
Eden

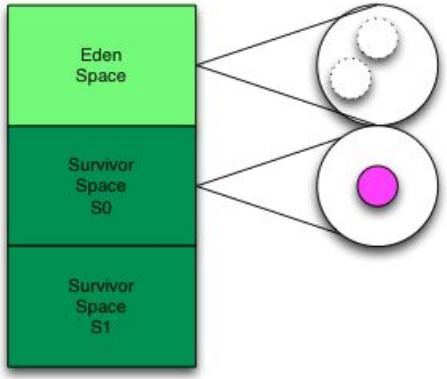
La zona de Eden es la zona en la que los objetos que acabamos de construir se almacenan

Cuando el recolector de basura pasa , elimina todos los objetos que ya no dispongan de referencias en el Eden Space y los supervivientes los mueve al Survivor Space.



Cuando el recolector de basura pasa , elimina todos los objetos que ya no dispongan de referencias en el Eden Space y los supervivientes los mueve al Survivor Space.



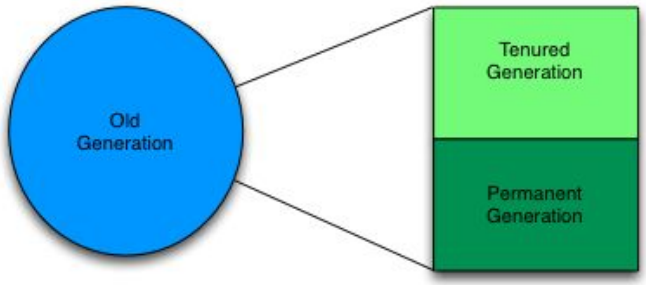


El recolector de basura volverá a pasar otra vez y si queda algún objeto superviviente en el Survivor Space S0 lo moverá al Survivor S1

Después de varias pasadas del recolector de basura los objetos que todavía están vivos en el Survivor Space dejan de ser considerados Young Generation y pasan a ser considerados Old Generation .

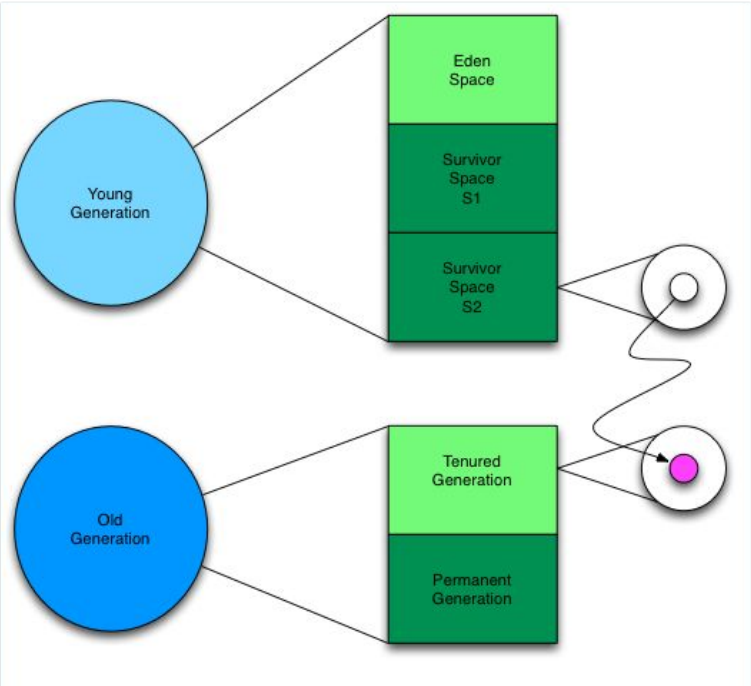
Old Generation

Esta zona está dividida en dos partes , la primera se denomina Tenured Generation y es donde los objetos que tienen un ciclo de vida largo se almacenan. La segunda zona se denomina Permanent Generation y es donde están cargadas las clases Java que la JVM necesita.



Así pues cuando nosotros tenemos objetos que han sobrevivido a varios garbage collectors pasan de el Survivor Generation Space a Tenured Generation Space

Finalmente el recolector de basura también pasará por el Tenured Space para liberar objetos. Esta operación se realizará de una forma mucho más espaciada y cuando el espacio se encuentra prácticamente lleno.



Links

https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

Consideraciones

- En Java, el garbage collection (GC) provee la gestión automática de memoria. El propósito de GC es eliminar los objetos que no se pueden alcanzar.
- Sólo la JVM decide cuándo ejecutar el GC; Sólo puede sugerirlo.
- No se puede saber el algoritmo del GC con seguridad.
- Los objetos deben ser considerados elegibles antes de que puedan ser recogidos basura.
- Un objeto es elegible cuando ningún hilo en directo puede alcanzarlo.
- Para llegar a un objeto, debe tener una referencia real y accesible a ese objeto. Las aplicaciones Java pueden quedarse sin memoria.
- “Islands of objects” Pueden ser recolectados de basura, aunque se refieran entre sí.

- Se puede solicitar la recolección por parte del garbage collector con `System.gc();`
- La clase Object tiene un método finalize().
- El método finalize () está garantizado para ejecutarse una vez y sólo una vez antes de que el garbage collector elimine un objeto.
- El garbage collector no da garantías; finalize () puede que nunca se ejecutarse. Se puede hacer no elegible dentro del mismo método.

Hilos

Hilos

- Formas de crear un hilo
- Arrancar un hilo
- Ciclo de vida de un hilo
- Planificar hilos
- Sincronización
- Iteración entre hilos

Introducción

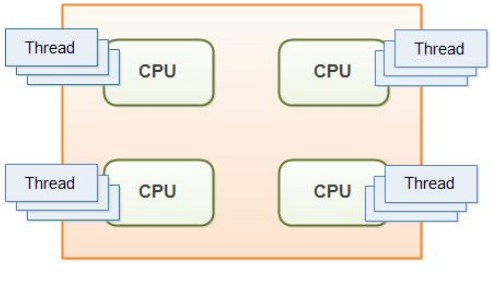
En los viejos tiempos una computadora tenía una sola CPU, y sólo era capaz de ejecutar un solo programa a la vez. Más tarde llegó la multitarea, lo que significó que las computadoras podrían ejecutar múltiples programas (tareas o procesos) al mismo tiempo. No era realmente "al mismo tiempo", ya que la única CPU era compartida entre los programas, en la que el sistema operativo distribuía los tiempos entre los programas en ejecución.

Junto con la multitarea surgieron nuevos desafíos para los desarrolladores de software. Los programas ya no pueden suponer que tienen todo el tiempo de CPU disponible, ni toda la memoria o cualquier otro recurso de la computadora. Un programa debería liberar todos los recursos que ya no utiliza, por lo que otros programas pueden usarlos.

Más tarde, sin embargo, llegó multithreading lo que significa que usted podría tener múltiples hilos de ejecución dentro del mismo programa. Un hilo de ejecución se puede considerar como una CPU que ejecuta el programa. Cuando tiene varios subprocesos ejecutando el mismo programa, es como tener múltiples CPUs ejecutándose dentro del mismo programa.

Todos los sistemas operativos modernos admiten simultaneidad a través de procesos y subprocesos. Los procesos son instancias de programas que normalmente se ejecutan de forma independiente entre sí iniciando un programa java, el sistema operativo genera un nuevo proceso que se ejecuta en paralelo a otros programas. Dentro de esos procesos, podemos

Multithreading



Multithreading puede ser una gran manera de aumentar el tipos de programas, sin embargo, es aún más difícil que están ejecutando dentro del mismo programa y por lo escribiendo la misma memoria de manera simultánea. Esto puede resultar en errores no vistos en un programa con un solo hilo. Algunos de estos errores no se pueden ver en máquinas de una sola CPU, porque dos subprocesos nunca realmente ejecutar "manera simultánea". Las computadoras modernas, sin embargo, vienen con CPUs de varios núcleos, e incluso con múltiples CPUs también. Esto significa que los hilos independientes pueden ejecutarse simultáneamente mediante núcleos o CPU separados.

Bloqueos

La clase Object tiene métodos como wait(), notify() / notifyAll(), etc., que son útiles para multi-threading. Dado que cada clase en Java deriva de la clase Object, todos los objetos tienen algunas capacidades básicas de multi-subprocesos, es decir que podemos adquirir un bloqueo en cualquier objeto en Java .

Sin embargo, para crear un hilo, este soporte básico de Object no es útil. Para ello, una clase debe extender la clase Thread o implementar la interfaz Runnable. Tanto Thread como Runnable están en la biblioteca java.lang, por lo que no tiene que importar estas clases explícitamente para escribir programas multihilo.

Thread

`static Thread currentThread()`	Devuelve la referencia a la hebra “current” thread.
`String getName()`	Devuelve el nombre de la “current” thread.
`int getPriority()`	Devuelve la prioridad de la “current” thread.

`void join(),`
`void join(long),`
`void join(long, int)`
	El subproceso actual que invoca la unión en otra hebra espera hasta que la otra muere. Opcionalmente puede dar el tiempo de espera en milisegundos (dado en tiempo) o el tiempo de espera en milisegundos, así como nanosegundos (dado en largo e int).

`void run()`	Una vez que inicie un hilo (utilizando el método start ()), el método run () será llamado cuando el hilo esté listo para ejecutarse.

`void setName(String)`	Cambia el nombre del subproceso al nombre dado en el argumento.
`void setPriority(int)`	Establece la prioridad del subproceso en el valor del argumento dado.
`void sleep(long)` `void sleep(long, int)`	Hace que el hilo actual duerma por milisegundos dados (dados en tiempo) o por milisegundos y nanoseconds dados (dados en long e int).
`void start()`	Inicia el hilo; JVM llama al método run () de la hebra.

Object

`void wait(),`
`void wait(long),`
`void wait(long, int)`
	El subproceso actual debería haber adquirido un bloqueo en este objeto antes de llamar a cualquiera de los métodos de espera.
	Si se llama wait (), el hilo espera infinitamente hasta que algún otro hilo lo notifique (llamando al método notify () / notifyAll ()) para este bloqueo.
	El método wait (long) toma milisegundos como argumento. El hilo espera hasta que se notifique o se produzca el tiempo de espera.
	El método wait (long, int) es similar a wait (long) y además toma nanoseconds como argumento.

`void notify()`	El subproceso actual debería haber adquirido un bloqueo en este objeto antes de llamar a notify (). La JVM elige un hilo único que está esperando en el bloqueo y lo despierta.
`void notifyAll()`	El subproceso actual debería haber adquirido un bloqueo antes de llamar a notifyAll (). La JVM despierta todos los hilos que esperan en un bloqueo.

Formas de crear un hilo

Al extender la clase Thread se debe reemplazar el método run(), si no sobreescribe el método run(), se llamará el método por defecto de la clase Thread, que no hace nada.

```
class MyThread extends Thread {
	public void run() {
		try {
		sleep(1000);
		} catch (InterruptedException ex) {}
		System.out.println("In run method; thread name is:
"+getName());
	}
}
```

Arrancar un hilo

Luego para comenzar o crear un subproceso debemos invocar el método start () en el objeto de la clase Thread (o su clase derivada), entonces la JVM programa el hilo, lo moverá a un estado ejecutable y luego ejecutará el método run ().

```
Thread myThread=new MyThread();
       myThread.start();

}
```

La interface Runnable

La clase Thread implementa la interfaz Runnable. En lugar de extender la clase Thread, puede implementar la interfaz Runnable.

```
class MyThread implements Runnable {
	public void run() {
		System.out.println("Name : "+ Thread.currentThread().getName());
	}
}
```

Ejecucción con Runnable

Cuando implemente la interfaz Runnable, debe definir el método run(). Recuerde que Runnable no declara el método start(), es Thread el que la tiene, por los cual mediante el constructor sobrecargado

```
Thread(Runnable target).
Thread myThread=new Thread(new MyThread());
myThread.start();
```

Runnable

La clase Thread implementa la interfaz Runnable. En lugar de extender la clase Thread, puede implementar la interfaz Runnable.

```
class MyThread implements Runnable {
   public void run() {
     System.out.println("Name : "+ Thread.currentThread().getName());
   }
}
```

Runnable

Cuando implemente la interfaz Runnable, debe definir el método run(). Recuerde que Runnable no declara el método start(), es Thread el que la tiene, por los cual mediante el constructor sobrecargado Thread(Runnable target).

```
Thread myThread=new Thread(new MyThread());
myThread.start();
```

Runnable vs Thread override

Puede ampliar la clase Thread o implementar la interfaz Runnable para crear un subproceso ¿cual usar?. Entonces, ¿cuál eliges?

- Dado que Java admite sólo una herencia, si se extiende desde Thread, no se puede extender desde ninguna otra clase (no existe herencia múltiple), así como separar los recursos (hebras) de la funcionalidad pudiendo, por ejemplo crear pools o ejecuciones mediante factorías.
- Dado que la herencia es una relación is-a, rara vez se necesita que la clase tenga una relación is-a con la clase Thread. Desde el enfoque OOP no convendría extender la clase Thread, conviene orientarse a interfaces y que podamos extender alguna otra clase.
- Por lo tanto, se sugiere que es mejor implementar la interfaz Runnable a menos que haya algunas razones fuertes para extender la clase Thread.
- Sin embargo, extender la clase Thread es más conveniente en muchos casos en donde es necesario acceder al objeto mediante Thread.currentThread().

start() & run()

Debemos sobreescribir el método run() e invocar el método start(). Nunca llame al método run() directamente para invocar un subproceso. Utilice el método start() y déjelo a la JVM para invocar implícitamente el método run(). Llamar el método run ()

```
class MyThread implements Runnable {
   public void run() {
       System.out.println("In run method; thread name is: " +
       Thread.currentThread().getName());
   }
   public static void main(String args[]) throws Exception {
       Thread myThread = new Thread(new MyThread());
       myThread.run(); // note run() instead of start() here
       System.out.println("In main method; thread name is : " +
       Thread.currentThread().getName());
       }
   }
}
```

Thread Name, Priority, y Group

Es necesario comprender tres aspectos principales asociados con cada hilo de Java: su nombre, prioridad y el grupo de hilos al que pertenece.

Cada subproceso tiene un nombre, que se puede utilizar para identificar el subproceso. Si no da un nombre explícitamente, un hilo obtendrá un nombre predeterminado.

La prioridad puede variar de 1, la más baja, a 10, la más alta, el Thread poseen unas constantes que determinan estas Thread.MIN_PRIORITY, Thread.NORM_PRIORITY, Thread.MAX_PRIORITY.

La prioridad del subproceso normal es por defecto 5 y puede cambiar este valor de prioridad predeterminado proporcionando explícitamente un valor de prioridad.

Cada subproceso fo

```
Thread t=new Thread();
t.setName("SimpleThread");
t.setPriority(9);
System.out.println(t);
```

sleep()

Thread.sleep() interactúa con el programador de subprocesos para poner el subproceso actual en estado de espera durante un período de tiempo especificado. Una vez que el tiempo de espera ha terminado, el estado del hilo se cambia a estado ejecutable y espera a la CPU para su ejecución posterior. Por lo tanto, el tiempo real que el tiempo de espera de hilo actual depende del programador de subprocesos que forma parte del sistema operativo.

```
public static void sleep(long millis) throws InterruptedException
```

La interrupción InterruptedException Es lanzada si algún hilo ha interrumpido el subproceso actual. El estado interrumpido del subproceso actual se borra cuando se genera esta excepción.

```
public class TimeBomb extends Thread {
	String[] timeStr = { "Zero", "One", "Two", "Three", "Four",
			     "Five", "Six", "Seven", "Eight","Nine"};
	void run() {
		for(int i = 9; i > = 0; i--) {
		try {
			System.out.println(timeStr[i]);
			Thread.sleep(1000);
		} catch(InterruptedException ie) {}
```

join()

La clase Thread tiene el método de instancia join () para esperar que un hilo muera. En el programa TimeBomb, desea el hilo main () para esperar a que el hilo del temporizador complete su ejecución. Puede utilizar el método de instancia join () en la clase

```
public static void main(String []s) {
	TimeBomb timer = new TimeBomb();
	System.out.println("Starting 10 second count down. . . ");
	timer.start();
	try {
		timer.join();
	} catch(InterruptedException ie) {}
	System.out.println("Boom!!!");
	}
}
```

La clase Thread tiene el método de instancia join () para esperar que un hilo muera. En el programa TimeBomb, desea el hilo main () para esperar a que el hilo del temporizador complete su ejecución. Puede utilizar el método
versión
método

```
public static void main(String []s) {
	TimeBomb timer = new TimeBomb();
	System.out.println("Starting 10 second count down. . . ");
	timer.start();
	try {
		timer.join();
	} catch(InterruptedException ie) {}
	System.out.println("Boom!!!");
	}
}
```

interrupt( ), isInterrupted() e interrupted()

Una interrupción es una indicación a un hilo que debe detener lo que está haciendo y hacer otra cosa. Depende del programador decidir exactamente cómo un hilo responde a una interrupción, pero es muy común que el hilo termine.

Un subproceso envía una interrupción invocando interrupt() en el objeto Thread para que el subproceso se interrumpa. Para que el mecanismo de interrupción funcione correctamente, el hilo interrumpido debe soportar su propia interrupción.

El mecanismo de interrupción se implementa utilizando un indicador interno conocido como el estado de interrupción. Invocar interrupt() establece este indicador. Cuando un subproceso comprueba una interrupción invocando el método estático Thread.interrupted, el estado de interrupción se borra. El método isInterrupted() no estático, que es utilizado por un subproceso para consultar el estado de interrupción de otro, no cambia el indicador de estado de interrupción.

Por convención, cualquier método que salga lanzando una InterruptedException borra el estado de interrupción cuando lo hace. Sin embargo, es siempre posible que el estado de la interrupción sea fijado otra vez inmediatamente, por otro hilo que invoca la interrupción.

El siguiente ejemplo muestra cómo interrumpir un thread en ejecución mediante interrupted() y comprobar si fue interrumpido isInterrupted()).

```
public class TestingInterrupt  implements Runnable {
   public void run() {
        try {
        work2();
        } catch (InterruptedException x) {
        System.out.println("in run() - interrupted in work2()");
        return;
        }
        System.out.println("in  run()  -  doing  stuff  after  nap,  leaving
normally");
   }
   public void work2() throws InterruptedException {
        while (true) {
                if (Thread.currentThread().isInterrupted()) {
                        Thread.sleep(2000);
                        }
                }
        }
   public void work() throws InterruptedException {
```

```
  public void work() throws InterruptedException {
   while (true) {
        for (int i = 0; i < 100000; i++) {
        int j = i * 2;
        }
        System.out.println("AisInterrupted()="+
Thread.currentThread().isInterrupted());
        if (Thread.interrupted()) {
        System.out.println("BisInterrupted()="+
Thread.currentThread().isInterrupted());
        throw new InterruptedException();
        }
        }
   }
```

```
   public static void main(String[] args) {
        TestingInterrupt si = new TestingInterrupt();
        Thread t = new Thread(si);
        t.start();
        try {
        Thread.sleep(2000);
        } catch (InterruptedException x) { }

        System.out.println("in main() - interrupting other thread");
        t.interrupt();
        System.out.println("in main() - leaving");
   }
}
```

Asynchronous

Como sabemos, los Threads se ejecutan asincrónicamente, sin embargo, no se ejecutan secuencialmente (como las llamadas de función), por lo que el orden de la ejecución de los hilos no es predecible; en otras palabras, el comportamiento del hilo no es de naturaleza determinista.

```
class AsyncThread extends Thread {
        public void run() {
        System.out.println("Starting the thread " + getName());
        for(int i = 0; i < 3; i++) {
        System.out.println("In thread " + getName() + "; iteration " + i);
        try {
                // sleep for sometime before the next iteration
                Thread.sleep(10);
        } catch(InterruptedException ie) {}
        }
        }
public static void main(String args[]) {
AsyncThread asyncThread1 = new AsyncThread();
AsyncThread asyncThread2 = new AsyncThread();
        asyncThread1.start();
        asyncThread2.start();
}
}
```

Los dos Estados en "Runnable"

Una vez que un subproceso hace que la transición del estado new al estado runnable, se puede pensar en el hilo que tiene dos estados en el nivel OS: el estado ready y el estado running.

Un hilo está en ready cuando está esperando al sistema operativo para ejecutarlo en el procesador y cuando el sistema operativo realmente lo procesa, el estado es running. Puede haber muchos hilos esperando el tiempo del procesador. El hilo current puede terminar tomando mucho tiempo y finalmente puede renunciar a la CPU voluntariamente y n este caso, el hilo vuelve de nuevo al estado ready.

Sincronización

No es posible invocar dos metodos sincronizados del mismo objeto a la vez. Si un metodo se esta ejecutando en un hilo, todos los hilos que invoquen a otro metodo se bloquean hasta que el primero termina.

Cuando se llama a un método sincronizado, se establece automaticamente una relacion happens-before con cualquier llamada a un método sincronizado de este objeto. Esto garantiza que cualquier cambio sobre el objeto es visible para cualquier hilo.

Problemas de acceso concurrentes

La programación simultánea está plagada de problemas y trampas. Discutiremos dos accesos concurrentes principales problemas-carreras de datos y bloqueos-en esta sección.

Los hilos comparten memoria y pueden modificar datos de forma simultánea. Dado que la modificación puede realizarse al mismo tiempo sin salvaguardas, esto puede conducir a resultados no intuitivos. Cuando dos o más subprocesos intentan acceder a una variable y uno de ellos desea modificarla, se obtiene un problema conocida como data race, race condition o race hazard

```
class Counter {
public static long count = 0;
}
```

```
class UseCounter implements Runnable {
      public void increment() {
            Counter.count++;
            System.out.print(Counter.count + " ");
      }
      public void run() {
            increment();
            increment();
            increment();
      }
}
```

Sincronización de hebras - Bloques

En bloques sincronizados, se utiliza la palabra clave synchronized para una variable de referencia y se sigue por un bloque de código. Un hilo tiene que adquirir un bloqueo en la variable sincronizada para entrar en el bloque; cuando la ejecución del bloque se completa, el hilo suelta el bloqueo. Por ejemplo, puede obtener un bloqueo en esta referencia si el bloque de código es dentro de un método no estático (bloquea this):

```
synchronized (this) {
    // segmento de código protegido por el bloqueo de mutex
}
```

Sincronización de hebras - Método

Se puede declarar un método completo sincronizado. En ese caso, cuando se llama al método declarado como sincronizado, se obtiene un bloqueo en el objeto en el que se llama el método y se libera cuando el método vuelve al llamador.

```
public synchronized void assign(int i) {
    val=i;
}
```

Un método sincronizado es equivalente a un bloque sincronizado si encierra todo el cuerpo del método en un bloque (this) sincronizado. Por lo tanto, el equivalente asign () método utilizando bloques sincronizados es

Un método sincronizado es equivalente a un bloque sincronizado si encierra todo el cuerpo del método en un bloque (this) sincronizado. Por lo tanto, el equivalente asign () método utilizando bloques sincronizados es

```
public void assign() {
          synchronized(this) {
              val=i;
    }
}
```

Deadlocks

Deadlock describe una situación en la que dos o más subprocesos están bloqueados para siempre, esperando uno al otro.

```
public class Deadlock {
        static class Friend {
        private final String name;
        public Friend(String name) {
        this.name = name;
        }
        public String getName() {
        return this.name;
        }
        public synchronized void bow(Friend bower) {
        System.out.format("%s: %s"
                + "  has bowed to me!%n",
                this.name, bower.getName());
        bower.bowBack(this);
        }
        public synchronized void bowBack(Friend bower) {
        System.out.format("%s: %s"
                + " has bowed back to me!%n",
                this.name, bower.getName());
        }
        }
```

```
public static void main(String[] args) {
        final Friend alphonse =
        new Friend("Alphonse");
        final Friend gaston =
        new Friend("Gaston");
        new Thread(new Runnable() {
        public void run() { alphonse.bow(gaston); }
        }).start();
        new Thread(new Runnable() {
        public void run() { gaston.bow(alphonse); }
        }).start();
        }
}
```

Livelocks

Para ayudar a entender los "livelocks", consideremos una analogía. Suponga que hay dos automóviles robotizados que están programados para conducir automáticamente en la carretera. Hay una situación en la que dos coches robotizados llegan a los dos extremos opuestos de un estrecho puente. El puente es tan estrecho que sólo un coche puede pasar a través de una vez. Los coches robotizados están programados que esperen a que el otro coche pase primero. Cuando ambos coches intentan entrar al puente al mismo tiempo, la siguiente situación podría ocurrir: cada coche comienza a entrar en el puente, nota que el otro coche está tratando de hacer el mismo, e invierte! Tenga en cuenta que los coches siguen moviéndose hacia delante y hacia atrás y, por lo tanto, aparecen como si estuvieran haciendo mucho de trabajo, pero no hay progreso por ninguno de los coches. Esta situación se llama un "livelock".

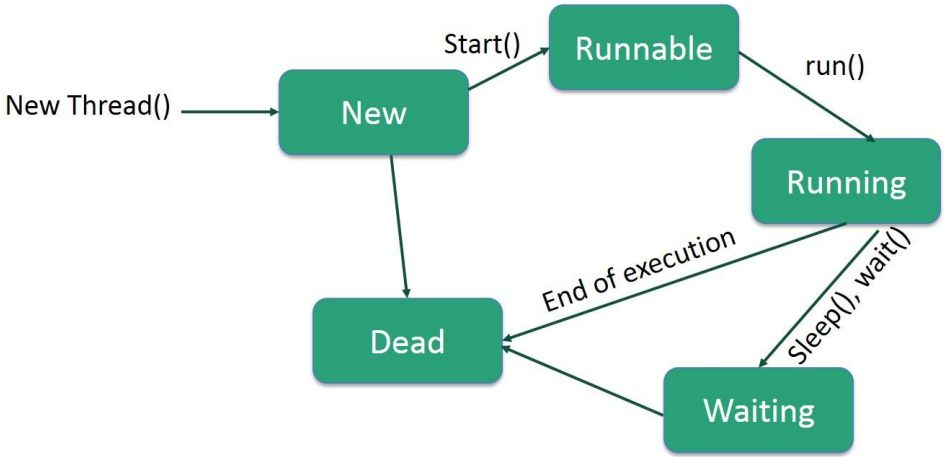
Consideremos dos hilos t1 y t2. Suponga que el hilo t1 hace un cambio y el hilo t2 deshace ese cambio. Cuando tanto los hilos t1 como t2 funcionan, parecerá que se está haciendo mucho trabajo, pero no se hace ningún progreso. La similitud entre livelocks y deadlocks es que el proceso "se bloquea" y el programa nunca termina. Sin embargo, en un deadlock, los subprocesos se atascan en el mismo estado esperando a que otros subprocesos liberen un sistema compartido recurso; en un livelock, los hilos siguen ejecutando una tarea, y hay un cambio continuo en los estados del proceso, pero el aplicación en su conjunto no avanza.

Lock salvation

Considere la situación en la que numerosos hilos tienen diferentes prioridades asignadas (en el rango de menor prioridad, 1, a la prioridad más alta, 10, que es el rango permitido para la prioridad de los subprocesos en Java). Cuando un bloqueo mutex es disponible, el programador de subprocesos dará prioridad a los subprocesos con alta sobre baja. Si hay muchos hilos de alta prioridad que quieren obtener el bloqueo y también mantener el bloqueo durante períodos de tiempo largos, lo hilos de baja prioridad nunca tendrán la prioridad, y en una situación "mueren de tiempo tratando de obtener la bloqueo se conoce como bloqueo de hambre.

Ciclo de vida de un hilo

Un hilo pasa por varias etapas e su ciclo de vida. Por ejemplo, u hilo nace, se inicia, se ejecuta luego muere. El siguiente diagram muestra el ciclo de vida complet de un hilo.



Las siguientes son las etapas del ciclo de vida:

New: un nuevo hilo comienza su ciclo de vida en el nuevo estado. Permanece en este estado hasta que el programa inicie el hilo. También se conoce como un hilo nacido.

Runnable: después de que se inicie un subproceso recién nacido, el subproceso se convierte en ejecutable. Se considera que un hilo en este estado está ejecutando su tarea.

Waiting: en ocasiones, un hilo pasa al estado de espera mientras el hilo espera a que otro hilo realice una tarea. Un hilo vuelve al estado ejecutable solo cuando otro hilo señala al hilo en espera para que continúe ejecutándose.

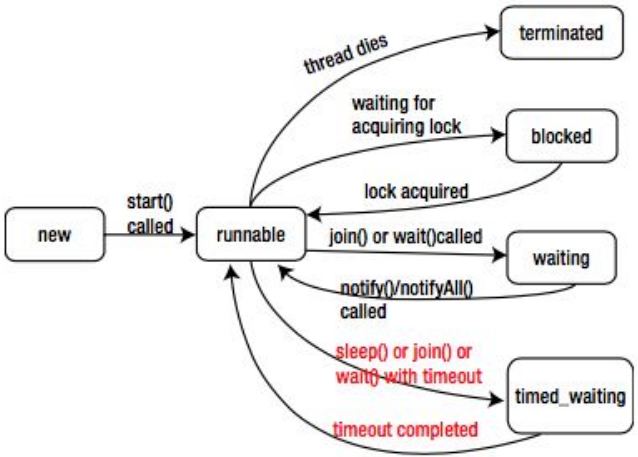
Timed Waiting: un hilo ejecutable puede ingresar al estado de espera

Iteración entre hilos :: Wait / notify

En los programas de múltiples hilos, a menudo hay una necesidad de un hilo para comunicarse con otro hilo. Wait / notify es útil cuando los hilos deben comunicarse para proporcionar una funcionalidad.

Si desea hacer un hilo de espera para otro hilo, puede pedirle que espere el objeto de espera utilizando el comando wait () método. Un hilo permanece en el estado de espera hasta que otro subproceso llama al método notify() o notifyAll() en el objeto de espera. Para entender el mecanismo de espera / notificación, usted puede simular esta situación cafetería. Puede implementar la máquina de café como un hilo y el camarero como otro hilo en dos diferentes clases La máquina de café puede notificar al camarero para tomar el café, y puede esperar hasta que el camarero ha tomado la café de la bandeja. Del mismo modo, el camarero puede tomar el café si está disponible y notificar a la máquina de café para hacer otra taza

Estados de la hebra




```
class SleepyThread extends Thread {
        public void run() {
                synchronized(SleepyThread.class) {
                        try {
                        Thread.sleep(1000);
                }  catch(InterruptedException ie) {}
        }
        }
}

class MoreThreadStates {
        public static void main(String []s) {
Thread t1=new SleepyThread();
Thread t2=new SleepyThread();
t1.start();
t2.start();
System.out.println(t1.getName()+": I'm in state " +
 t1.getState());
System.out.println(t2.getName()+": I'm in state " + t2.getState());
}
}
```

IllegalMonitorStateException

Debe tener cuidado al escribir código para las hebras, teniendo siempre en cuenta los estados de los hilos. Cuando llama a start (), el subproceso se desplaza al nuevo estado. No hay una transición de estado adecuada desde el nuevo estado si vuelve a llamar start (), por lo que la JVM lanza una

```
class ThreadStateProblem {
        public static void main(String []s) {
        Thread thread=new Thread();
        thread.start();
        thread.start();
        }
}
```

Planificar hilos

Iteración entre hilos

Resumen

Programación Concurrente

- Puede crear clases que sean capaces de multi-threading implementando el Runnable o extendiendo la clase Thread.
- Siempre sobreescriba el método run() de la clase Thread ya que el predeterminado no hace nada.
- Llame al método start () y no al método run () directamente en el código. (Déjelo a la JVM para llame al método run ().)
- Cada subproceso tiene un nombre de subproceso, prioridad y grupo de subprocesos asociados con él; el valor por defecto toString () en Thread los imprime.
- Si llama al método sleep() de un subproceso, el subproceso no suelta el bloqueo y se mantiene en bloqueo.

- Puede utilizar el método join() para esperar a que otro hilo termine.
- En general, si no está utilizando la función "interrupt" en los subprocesos, es seguro ignorar InterruptedException; sin embargo, es mejor que registre o imprima el seguimiento de la pila si se produce esa excepción.
- Los subprocesos se ejecutan de forma asincrónica; no puede predecir el orden en que se ejecutan.
- Los hilos son también no deterministas: en muchos casos, no se pueden reproducir problemas como deadlocks o data race cada vez.

Estados del hilo

- Hay tres estados básicos de subproceso: NEW, RUNNABLE y

- Un hilo estará en el estado BLOCKED cuando se espera para adquirir un bloqueo. El hilo estará en el TIMED_WAITING cuando se da un tiempo de espera para llamadas como wait. El hilo estará en espera estado cuando, por ejemplo, wait() se llama (sin un valor de tiempo de espera).
- Obtendrá una IllegalThreadStateException si sus operaciones resultan en estado de Threads no válido.

Problemas de acceso concurrentes

- Las lecturas y escrituras concurrentes a los recursos pueden conducir al problema de la race data (cuando no hay mecanismos de synch que cuiden el valor del dato, no es ThreadSafe).

- Un bloqueo suele ocurrir cuando dos hilos adquieren bloqueos en orden opuesto. Cuando uno hila ha adquirido un bloqueo y espera otro bloquo, otro hilo ha adquirido que otro bloqueo y espera que se libere el primer bloqueo. Por lo tanto, no se hace ningún progreso y el programa interbloquea la ejecucción.
- Para evitar bloqueos, es mejor evitar la adquisición de bloqueos múltiples. Cuando usted tiene que adquirir tales bloqueos múltiples, asegurarse de que se adquieren en el mismo orden en todos los programa.

El mecanismo de wait / notify

- Cuando un hilo tiene que esperar a que una condición o evento particular sea satisfecho por otro hilo, puede utilizar un mecanismo de espera / notificación como un mecanismo de comunicación entre subprocesos.
- Cuando un hilo necesita esperar una condición / evento en particular, puede llamar a wait () con o sin un valor de tiempo de espera especificado.
- Para evitar que se pierdan las notificaciones, es mejor utilizar siempre notifyAll () en lugar de notify().

Links

https://www.tutorialspoint.com/java/

https://docs.oracle.com/javase/8/docs/api/