



YEARS
EMPOWERING
TECHNOLOGY
1998-2018

Java 11

Programación orientada a objetos con Java

BUILDING INTELLIGENT BUSINESS. BIG DATA, CLOUD, DevOps & NoSQL EXPERTS



SEDE BARCELONA

Av. Diagonal, 98-100
08019 Barcelona
T. 93 206 02 49

SEDE MADRID

c/ Arregui y Aruej, 25-27
28007 Madrid
T. 91 162 06 69

Objetivos

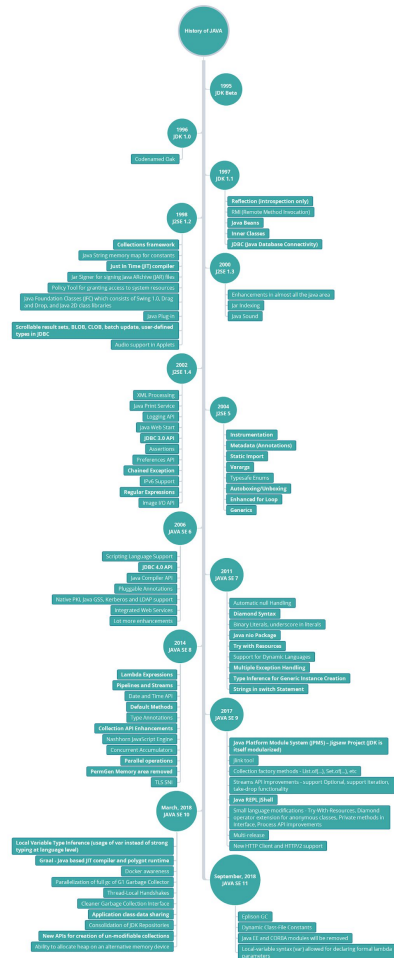
- Creación de aplicaciones multihilo de alto rendimiento
- Creación de aplicaciones de tecnología Java que aprovechan las características orientadas a objetos del lenguaje Java, como la encapsulación, la herencia y el polimorfismo
- Implementación de la funcionalidad de entrada / salida (E / S) para leer y escribir en archivos de datos y texto y comprender flujos de E / S avanzados
- Ejecutar una aplicación de tecnología Java desde la línea de comando
- Manipulación de archivos, directorios y sistemas de archivos utilizando la especificación JDK NIO.2
- Creación de aplicaciones que utilizan el framework Collections
- Realización de operaciones múltiples en tablas de bases de datos, incluida la creación, lectura, actualización y eliminación mediante tecnología JDBC y JPA
- Buscar y filtrar colecciones usando Lambda Expressions
- Implementando técnicas de manejo de errores usando manejo de excepciones
- Uso de las características de concurrencia de Lambda Expression

Objetivos

- Escribir código Java utilizando variables, matrices, construcciones condicionales y de bucle.
- Identificar los principios de la programación modular.
- Acceder y crear campos y métodos estáticos.
- Encapsular una clase utilizando modificadores de acceso y constructores sobrecargados.
- Manipular datos numéricos, de texto y cadenas de datos utilizando los operadores de Java apropiados.
- Establecer variables de entorno para permitir que el compilador de Java y los ejecutables runtime funcionen correctamente.
- Crear clases de Java simples y hacer referencias a objetos para acceder a campos y métodos en una clase.
- Demostrar el polimorfismo mediante la implementación de una interfaz Java.
- Gestionar una checked exception en una aplicación Java.
- Utilizado una Predicate Lambda Expresion como argumento de un método.
- Definir e implementar una jerarquía de clases simple que soporte los requisitos de la aplicación.
- Crear aplicaciones Java que aprovechen las características orientadas a objetos del lenguaje Java, como encapsulación, herencia y polimorfismo.
- Ejecutar una aplicación Java desde la línea de comandos.
- Crear aplicaciones que utilicen el framework Java Collections.
- Buscar y filtrar colecciones utilizando Lambda Expresiones.

Versiones

- Junio de 1991 – Se inició el proyecto de lenguaje Java
- JDK 1.0 – enero de 1996
- JDK 1.1 – febrero de 1997
- J2SE 1.2 – diciembre de 1998
- J2SE 1.3 – mayo de 2000
- J2SE 1.4 – febrero de 2002
- J2SE 5.0 – septiembre de 2004
- Java SE 6 – diciembre de 2006
- Java SE 7 – julio de 2011
- Java SE 8 – 18 de marzo de 2014
- Java SE 9 – julio de 2017
- Java SE 10 – Marzo de 2018
- Java SE 11 – Septiembre de 2018



Historia y versiones



Historia



Historia

En 1991, el equipo “Green Team” dirigido por James Gosling de Sun Microsystems creó un lenguaje de programación para dispositivos digitales de consumo. El lenguaje se llamaba Oak, entonces ¿por qué Oak? Porque había un roble (‘oak’ en inglés) afuera de la oficina de Gosling.

El “Green Team” demostró el uso del lenguaje con una televisión interactiva. Sin embargo, era demasiado avanzado para la televisión digital por cable en ese momento, y era más adecuada una tecnología que comenzaba a despegar, Internet.

Más tarde, el lenguaje pasó a llamarse “Green” y finalmente se le cambió el nombre a “Java” del café precisamente llamado ‘Java’; de ahí el logo de la taza de café.

Como C/C++ era popular en aquel entonces, James Gosling diseñó el lenguaje con la sintaxis de estilo C/C++ y la filosofía “escribe una vez, ejecuta en cualquier lado”. Después de años, Sun Microsystems lanzó la primera implementación pública de Java en 1995. Se anunció que el navegador de Internet Netscape Navigator incorporaría la tecnología Java.

En 2010, Sun Microsystems fue completamente adquirida por Oracle Corporation junto con Java.

Descripción general de la plataforma Java

JRE y JDK

Oracle ofrece dos productos de software principales en la familia de Java TM Platform, Standard Edition (Java TM SE):

Java SE Runtime Environment (JRE)

JRE proporciona las bibliotecas, la máquina virtual Java y otros componentes necesarios para ejecutar applets y aplicaciones escritas en el lenguaje de programación Java. Este entorno de tiempo de ejecución se puede redistribuir con aplicaciones para que sean autónomos.

Java SE Development Kit (JDK)

El JDK incluye las herramientas de desarrollo de línea de comandos JRE y extras como compiladores y depuradores, que son necesarios o útiles para desarrollar aplicaciones y applets.

Lenguaje Java



Características



Un lenguaje orientado a objetos

Hay diferentes estilos de programación. El enfoque orientado a objetos es uno de los estilos de programación más popular. En la programación orientada a objetos, un problema complejo se divide en conjuntos más pequeños mediante la creación de objetos. Esto hace que el código sea reutilizable, tenga beneficios de diseño y haga que el código sea más fácil de mantener.

Java es independiente de la plataforma

Java se creó con la filosofía de “escribe una vez, ejecuta en cualquier lado” (WORA). El código de Java (código Java puro y bibliotecas) que escriba en una plataforma (sistema operativo) se ejecutará en otras plataformas sin modificaciones.

Para ejecutar Java, se utiliza una máquina abstracta llamada Java Virtual Machine (JVM). La JVM ejecuta el bytecode de Java. Entonces, la CPU ejecuta la JVM. Dado que todas las JVM funcionan exactamente igual, el mismo código también funciona en otros sistemas operativos, lo que hace que Java sea independiente de la plataforma.

Sintaxis

La sintaxis de Java se deriva en gran medida de C++. Pero a diferencia de este, que combina la sintaxis para programación genérica, estructurada y orientada a objetos, Java fue construido desde el principio para ser completamente orientado a objetos. Todo en Java es un objeto (salvo algunas excepciones), y todo en Java reside en alguna clase (recordemos que una clase es un molde a partir del cual pueden crearse varios objetos).

A diferencia de C++, Java no tiene sobrecarga de operadores⁷ o herencia múltiple para clases, aunque la herencia múltiple está disponible para interfaces.

Garbage Collector

En Java el problema de fugas de memoria 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 este.

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 (ejemplo: un objeto creado y únicamente usado dentro de un método sólo tiene entidad dentro de este; al salir del método el objeto es eliminado).

Aun así, es posible que se produzcan fugas de memoria si el código almacena referencias a objetos que ya no son necesarios; es decir, pueden aún ocurrir, pero en un nivel conceptual superior. En definitiva, el recolector de basura de Java permite una fácil creación y eliminación de objetos y mayor seguridad.

Java es rápido

El código Java bien optimizado es casi tan rápido como los lenguajes de nivel inferior como C/C ++, y mucho más rápido que Python, PHP, etc.

La compilación JIT y la recompilación dinámica permiten a los programas Java aprovechar la velocidad de ejecución del código nativo sin por ello perder la ventaja de la portabilidad en ambos.

Amplio conjunto de Bibliotecas

<https://docs.oracle.com/en/java/javase/11/core/index.html>

Aplicaciones de Java

- Desarrollo de software – Softwares como Eclipse, OpenOffice, Vuze, MATLAB, etc. usan Java.
- Procesamiento de Big Data: puede utilizar un marco de software popular como Hadoop (que a su vez está escrito en Java) para procesar Big Data. Para usar Hadoop, debe comprender la programación de Java.
- Sistema de negociación: puede crear aplicaciones de negociación que tengan baja latencia utilizando Oracle Extreme Java Trading Platform.
- Dispositivos incorporados: si bien los lenguajes de programación C/C ++ siguen siendo opciones populares para trabajar con sistemas integrados, las tecnologías Java Embedded de Oracle proporcionan plataforma y tiempo de ejecución para miles de millones de dispositivos integrados como: televisores, tarjetas SIM, reproductores de discos Blu-ray, etc.

Lenguaje de programación Java

El lenguaje de programación Java está orientado a objetos de uso general, concurrente, fuertemente tipificado y basado en clases. Normalmente se compila en el conjunto de instrucciones bytecode y en el formato binario definido en la Especificación de máquina virtual de Java.

Máquinas virtuales Java

La máquina virtual Java es una máquina informática abstracta que tiene un conjunto de instrucciones y manipula la memoria en tiempo de ejecución. La máquina virtual Java se transporta a diferentes plataformas para proporcionar independencia de hardware y sistema operativo. La plataforma Java, Standard Edition proporciona dos implementaciones de la máquina virtual Java (VM):

Java HotSpot Client VM

La máquina virtual del cliente en general se emplea en las plataformas que se utilizan para las aplicaciones del cliente. La VM del cliente está sintonizada para reducir el tiempo de inicio y la huella de memoria. Se puede invocar utilizando la opción -client de la línea de comando al iniciar una aplicación.

Más sobre Java

Java HotSpot Server VM

El servidor VM es una implementación diseñada para la máxima velocidad de ejecución del programa, intercambiando tiempo de ejecución y memoria. Se puede invocar mediante el uso de la opción -server de la línea de comando al iniciar una aplicación.

Bibliotecas de base

Clases e interfaces que proporcionan funciones básicas y funcionalidades fundamentales para la plataforma Java.

Paquetes java.lang y java.util

Proporciona las clases fundamentales de objeto y clase, clases contenedoras para tipos primitivos, una clase básica de matemática y más.

Biblioteca Math

La funcionalidad matemática incluye bibliotecas de punto flotante y matemática de precisión arbitraria.

Monitoring and Management

Soporte integral de monitoreo y administración para la plataforma Java, incluida la API virtual Monitoring and Management para Java, la API de monitoreo y gestión para la instalación de registro, jconsole y otras utilidades de monitoreo, monitoreo y administración listos para usar, Java Management Extensions (JMX) y Extensión de plataforma de Oracle.

Package Version Identification

La función de control de versiones del paquete permite el control de versiones a nivel de paquete para que las aplicaciones y los applets puedan identificar, en tiempo de ejecución, la versión de un Java Runtime Environment, VM y un paquete de clase específicos.

Objetos de referencia

Los objetos de referencia admiten un grado limitado de interacción con el recolector de basura. Un programa puede usar un objeto de referencia para mantener una referencia a algún otro objeto de tal manera que el último objeto aún pueda ser recuperado por el recolector. Un programa también puede acordar que se le notifique algún tiempo después de que el compilador haya determinado que la accesibilidad de un objeto dado ha cambiado.

Por lo tanto, los objetos de referencia son útiles para generar cachés simples y cachés que se vacían cuando la memoria se agota, para implementar asignaciones que no impidan que se recuperen sus claves (o valores) y para programar acciones de limpieza premortem de una manera más flexible mucho más de lo que es posible con el mecanismo de finalización de Java. Para obtener más información, consulte la documentación de Objetos de referencia.

Reflection

Reflection permite que el código Java descubra información sobre los campos, métodos y constructores de las clases cargadas, y que use los campos, métodos y constructores reflejados para operar en sus contrapartes subyacentes en los objetos, dentro de las restricciones de seguridad. La API acomoda las aplicaciones que necesitan acceso a los miembros públicos de un objeto de destino (en función de su clase de tiempo de ejecución) o los miembros declarados por una clase determinada. Los programas pueden suprimir el control de acceso reflectante predeterminado.

Framework de colecciones

Una colección es un objeto que representa a un grupo de objetos. El framework de colecciones es una arquitectura unificada para representar colecciones, lo que les permite ser manipuladas independientemente de los detalles de su representación. Reduce el esfuerzo de programación al tiempo que aumenta el rendimiento. Permite la interoperabilidad entre API no relacionadas, reduce el esfuerzo en el diseño y el aprendizaje de nuevas API, y fomenta la reutilización del software.

Utilidades de concurrencia

Las utilidades Concurrency Utilities proporcionan un marco potente y extensible de utilidades de subprocesamiento de alto rendimiento, como grupos de subprocesos y colas de bloqueo. Este paquete libera al programador de la necesidad de crear estas utilidades a mano, de la misma manera que lo hizo el framework de colecciones para las estructuras de datos. Además, estos paquetes proporcionan primitivas de bajo nivel para la programación simultánea avanzada.

Java Archive (JAR) Files

JAR (Java Archive) es un formato de archivo independiente de la plataforma que agrega muchos archivos en uno. Múltiples applets de Java y sus componentes necesarios (archivos, imágenes y sonidos) se pueden agrupar en un archivo JAR y posteriormente descargar a un navegador en una sola transacción HTTP, lo que mejora enormemente la velocidad de descarga.

El formato JAR también admite compresión, lo que reduce el tamaño del archivo y mejora aún más el tiempo de descarga. Además, el autor del applet puede firmar digitalmente entradas individuales en un archivo JAR para autenticar su origen. Es completamente extensible.

Trazas

Las API de trazas o log facilitan el servicio y el mantenimiento del software en los sitios de los clientes produciendo informes de registro adecuados para su análisis por parte de usuarios finales, administradores de sistemas y equipos de desarrollo de software. Las API de registro capturan información como fallas de seguridad, errores de configuración, cuellos de botella de rendimiento y / o errores en la aplicación o plataforma.

Preferencias

La API de Preferencias proporciona una forma para que las aplicaciones almacenen y recuperen los datos de preferencia y configuración del usuario y del sistema. Los datos se almacenan de forma persistente en una tienda de respaldo dependiendo de la implementación. Hay dos árboles separados de nodos de preferencia, uno para las preferencias del usuario y otro para las de sistema.

Otros paquetes E / S

Los paquetes `java.io` y `java.nio` proporcionan un amplio conjunto de API para administrar las E / S de una aplicación. La funcionalidad incluye archivos y dispositivos de E / S, serialización de objetos, gestión de búferes y soporte de conjunto de caracteres. Además, las API admiten funciones para servidores escalables que incluyen E / S multiplexadas y sin bloqueo, asignación de memoria y bloqueos para archivos.

Serialización de objetos

La serialización de objetos amplía las clases principales de entrada / salida de Java con soporte para objetos. La serialización de objetos admite la codificación de objetos y los objetos accesibles desde ellos a una secuencia de bytes; y es compatible con la reconstrucción complementaria del gráfico objeto de la secuencia.

La serialización se utiliza para la persistencia ligera y para la comunicación a través de sockets o invocación de método remoto (RMI).

Networking

Proporciona clases para la funcionalidad de red, incluido el direccionamiento, clases para usar URL y URI, clases de socket para conectarse a servidores, funcionalidad de seguridad de redes y más.

Seguridad

API para funciones relacionadas con la seguridad, como control de acceso configurable, firma digital, autenticación y autorización, criptografía, comunicación segura por Internet y más.

Internacionalización

API que permiten el desarrollo de aplicaciones internacionalizadas. La internacionalización es el proceso de diseño de una aplicación para que pueda adaptarse a varios idiomas y regiones sin cambios de ingeniería.

API de Componente JavaBeans TM

Contiene clases relacionadas con el desarrollo de beans: componentes basados en la arquitectura JavaBeans TM que se pueden combinar como parte del desarrollo de una aplicación.

Java Management Extensions (JMX)

La API Java Management Extensions (JMX) es una API estándar para la gestión y el control de recursos como aplicaciones, dispositivos, servicios y la máquina virtual Java. Los usos típicos incluyen consultar y cambiar la configuración de la aplicación, acumular estadísticas sobre el comportamiento de la aplicación y notificar los cambios de estado y las condiciones erróneas. La API JMX incluye acceso remoto, por lo que un programa de administración remota puede interactuar con una aplicación en ejecución para estos fines.

XML (JAXP)

La plataforma Java proporciona un amplio conjunto de API para procesar documentos y datos XML.

Java Native Interface (JNI)

Java Native Interface (JNI) es una interfaz de programación estándar para escribir métodos nativos de Java e incrustar la máquina virtual Java en aplicaciones nativas. El objetivo principal es la compatibilidad binaria de bibliotecas de métodos nativos en todas las implementaciones de máquinas virtuales Java en una plataforma determinada.

Mecanismo de extensión

Los paquetes opcionales son paquetes de clases Java (y cualquier código nativo asociado) que los desarrolladores de aplicaciones pueden usar para extender la funcionalidad de la plataforma central. El mecanismo de extensión también proporciona una forma para que los paquetes opcionales necesarios se recuperen de las URL especificadas cuando aún no están instalados en el JDK o el entorno de ejecución.

Endorsed Standards Override Mechanism = Mecanismo de invalidación de normas endosadas

Un estándar endosado es una API de Java definida a través de un proceso de estándares que no es Java Community ProcessSM (JCP). Para aprovechar las nuevas revisiones de los estándares aprobados, los desarrolladores y proveedores de software pueden usar el Mecanismo de anulación de estándares endosados para proporcionar versiones más nuevas de un estándar endosado que las incluidas en la plataforma Java publicadas por Oracle.

Bibliotecas de integración

Java Database Connectivity (JDBC) API

La API JDBC [™] proporciona acceso a datos universales desde el lenguaje de programación Java. Utilizando la API JDBC 3.0, los desarrolladores pueden escribir aplicaciones que puedan acceder a prácticamente cualquier fuente de datos, desde bases de datos relacionales hasta hojas de cálculo y archivos planos. La tecnología JDBC también proporciona una base común sobre la que se pueden construir herramientas e interfaces alternativas.

Remote Method Invocation (RMI)

La Invocación de Método Remoto (RMI) permite el desarrollo de aplicaciones distribuidas al proporcionar comunicación remota entre programas escritos en el lenguaje de programación Java. RMI permite que un objeto que se ejecute en una máquina virtual Java, invoque métodos en un objeto que se ejecuta en otra máquina virtual Java, que puede estar en un host diferente.

Java IDL (CORBA) Common Object Request Broker Architecture

La tecnología Java IDL agrega la capacidad CORBA (Common Object Request Broker Architecture) a la plataforma Java, proporcionando interoperabilidad y conectividad basada en estándares.

La tecnología Java IDL agrega la capacidad CORBA (Common Object Request Broker Architecture) a la plataforma Java, proporcionando interoperabilidad y conectividad basada en estándares. Java IDL permite aplicaciones Java distribuidas habilitadas para la Web para invocar operaciones en servicios de red remotos utilizando el IDL estándar de la industria (Lenguaje de Definición de Interfaz de Grupo de Gestión de Objetos) y IIOP (Protocolo Internet Inter-ORB) definido por el Grupo de Gestión de Objetos. Los componentes de tiempo de ejecución incluyen un ORB de Java para computación distribuida utilizando comunicación IIOP.

RMI-IIOP

Invocación de método remoto Java a través de Internet Tecnología de protocolo Inter-ORB El Modelo de programación RMI permite la programación de servidores y aplicaciones CORBA a través de la API de RMI. Puede elegir trabajar completamente dentro del lenguaje de programación Java, o trabajar con otros lenguajes de programación compatibles con CORBA.

Scripting for the Java Platform

Java SE incluye el JSR 223: creación de scripts para la plataforma Java TM API. Este es un framework por el cual las aplicaciones Java pueden "alojar" motores de scripts. Java SE incluye Nashorn Engine, que es una implementación de la especificación del lenguaje ECMAScript Edition 5.1.

Java Naming and Directory Interface™ (JNDI) API

Java Naming and Directory Interface™ (JNDI) proporciona funciones de nomenclatura y directorio para aplicaciones escritas en el lenguaje de programación Java. Está diseñado para ser independiente de cualquier implementación específica de nombres o servicios de directorio.

Bibliotecas de interfaz de usuario

Framework de método de entrada

El framework del método de entrada permite la colaboración entre los componentes de edición de texto y los métodos de entrada al ingresar texto. Los métodos de entrada son componentes de software que permiten al usuario incorporar texto de formas distintas a la simple escritura en un teclado. Se usan comúnmente para ingresar en japonés, chino o coreano, utilizando miles de caracteres diferentes, en teclados con muchas menos teclas. Sin embargo, el framework también admite métodos de entrada para otros idiomas y el uso de mecanismos completamente diferentes, como la escritura a mano o el reconocimiento de voz.

Accesibilidad

Con la API de accesibilidad de Java, los desarrolladores pueden crear fácilmente aplicaciones Java que sean de fácil acceso para personas con discapacidad. Las aplicaciones Java accesibles son compatibles con tecnologías de asistencia, como lectores de pantalla, sistemas de reconocimiento de voz y pantallas Braille actualizables.

Servicio de impresión

La API de Java TM Print Service permite imprimir en todas las plataformas Java, incluidas aquellas que requieren un espacio reducido, como un perfil de Java ME.

Sound

La plataforma Java incluye una potente API para capturar, procesar y reproducir datos de audio y MIDI (interfaz digital de instrumentos musicales). Esta API es compatible con un motor de sonido eficiente que garantiza la mezcla de audio de alta calidad y capacidades de síntesis MIDI para la plataforma.

Drag and Drop Data Transfer

Arrastrar y soltar permite la transferencia de datos tanto en el lenguaje de programación Java como en las aplicaciones nativas.

Image I/O

La API Image de E/S proporciona una arquitectura conectable para trabajar con imágenes almacenadas en archivos y a las que se accede a través de la red. La API proporciona un framework para agregar complementos específicos de formato. Los plug-ins para varios formatos comunes se incluyen con Java Image I/O, pero los terceros pueden usar esta API para crear sus propios complementos para manejar formatos especiales.

Java 2D™ Graphics and Imaging™

La API Java 2D™ es un conjunto de clases para gráficos 2D e imágenes avanzadas. Abarca arte lineal, texto e imágenes en un solo modelo integral. La API proporciona una amplia compatibilidad con la composición de imágenes y las imágenes de canales alfa, un conjunto de clases para proporcionar una definición y conversión precisa del espacio de color y un amplio conjunto de operadores de generación de imágenes orientados a la visualización.

AWT

El juego de herramientas de ventana abstracta (AWT) de la plataforma Java [™] proporciona API para construir componentes de interfaz de usuario como menús, botones, campos de texto, cuadros de diálogo, casillas de verificación y para manejar la entrada del usuario a través de esos componentes. Además, AWT permite renderizar formas simples como óvalos y polígonos y permite a los desarrolladores controlar el diseño de la interfaz del usuario y las fuentes utilizadas por sus aplicaciones.

Swing

Las API de Swing también proporcionan un componente gráfico (GUI) para usar en las interfaces de usuario. Las API de Swing están escritas en el lenguaje de programación de Java sin ninguna dependencia del código que es específico de las facilidades de la GUI provistas por el sistema operativo subyacente.

JavaFX

Java SE 7 Update 2 y posterior incluye JavaFX SDK. La plataforma JavaFX es la evolución de la plataforma de cliente Java que permite a los desarrolladores de aplicaciones crear e implementar fácilmente aplicaciones de Internet (RIA) de manera consistente en múltiples plataformas.

JVM

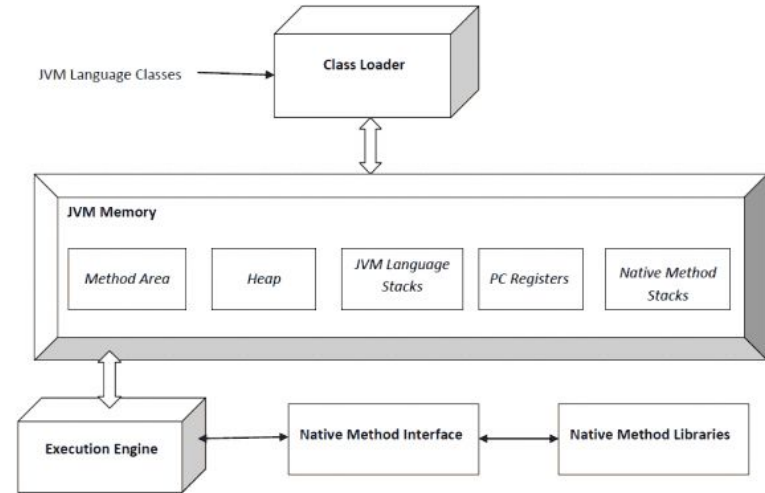


La Máquina Virtual



Que es?

El lenguaje Java es a la vez compilado e interpretado. Con el compilador se convierte el código fuente que reside en archivos cuya extensión es .java, a un conjunto de instrucciones que recibe el nombre de bytecodes que se guardan en un archivo cuya extensión es .class. Estas instrucciones son independientes del tipo de ordenador. El intérprete ejecuta cada una de estas instrucciones en un ordenador específico (Windows, Macintosh, etc). Solamente es necesario, por tanto, compilar una vez el programa, pero se interpreta cada vez que se ejecuta en un ordenador.



Subsistema Class Loader :: Carga

El class loader lee el archivo .class, genera los datos binarios correspondientes y los guarda en el área de métodos.

- Para cada archivo .class, JVM almacena la siguiente información en el área de método.
- Nombre completamente calificado de la clase cargada y su clase primaria inmediata.
- Si el archivo .class está relacionado con Class o Interface o Enum
- Información sobre modificadores, variables, métodos, etc.
- Después de cargar el archivo .class, JVM crea un objeto de tipo Class para representar este archivo en la memoria heap. Tenga en cuenta que este objeto es de tipo Class predefinido en el paquete java.lang. Este objeto Class puede ser utilizado por el programador para obtener información de nivel de clase como nombre de clase, nombre principal, métodos e información de variable, etc. Para obtener esta referencia de objeto, podemos usar el método getClass() de la clase Object.

Vinculación o Enlace

Realiza la verificación, la preparación y (opcionalmente) la resolución.

Verificación: asegura la exactitud del archivo .class, es decir, comprueba si este archivo está formateado correctamente y generado por un compilador válido o no. Si la verificación falla, obtenemos la excepción de tiempo de ejecución `java.lang.VerifyError`.

Preparación: JVM asigna memoria para las variables de clase e inicializa la memoria a los valores predeterminados.

Resolución: es el proceso de reemplazar referencias simbólicas del tipo con referencias directas. Se realiza buscando en el área del método (method area) para localizar la entidad a la que se hace referencia.

Instalando JDK

Descarga

Para obtener una versión de Java podemos utilizar gestores de paquetes como HomeBrew y otras herramientas, o descargando los paquetes de instalación desde la pantalla de

Java SE Development Kit 11 Downloads

Thank you for downloading this release of the Java™ Platform, Standard Edition Development Kit (JDK™). The JDK is a development environment for building applications, and components using the Java programming language.

The JDK includes tools useful for developing and testing programs written in the Java programming language and running on the Java platform.

Important Oracle JDK License Update

The Oracle JDK License has changed for releases starting April 16, 2019.

The new Oracle Technology Network License Agreement for Oracle Java SE is substantially different from prior Oracle JDK licenses. The new license permits certain uses, such as personal use and development use, at no cost – but other uses authorized under prior Oracle JDK licenses may no longer be available. Please review the terms carefully before downloading and using this product. An FAQ is available [here](#).

Commercial license and support is available with a low cost [Java SE Subscription](#).









Oracle also provides the latest OpenJDK release under the open source [GPL License](#) at [jdk.java.net](#).

See also:

- [Java Developer Newsletter](#): From your Oracle account, select **Subscriptions**, expand **Technology**, and subscribe to **Java**.
- [Java Developer Day](#): hands-on workshops (free) and other events
- [Java Magazine](#)
- [JDK 11.0.7 checksum](#)

Java SE Development Kit 11.0.7

This software is licensed under the Oracle Technology Network License Agreement for Oracle Java SE.

Product / File Description	File Size	Download
Linux Debian Package	148.72 MB	 jdk-11.0.7_linux-x64_bin.deb
Linux RPM Package	155.42 MB	 jdk-11.0.7_linux-x64_bin.rpm
Linux Compressed Archive	172.63 MB	 jdk-11.0.7_linux-x64_bin.tar.gz
macOS Installer	168.6 MB	 jdk-11.0.7_osx-x64_bin.dmg
macOS Compressed Archive	168.95 MB	 jdk-11.0.7_osx-x64_bin.tar.gz
Solaris SPARC Compressed Archive	186.43 MB	 jdk-11.0.7_solaris-sparcv9_bin.tar.gz
Windows x64 Installer	152.38 MB	 jdk-11.0.7_windows-x64_bin.exe
Windows x64 Compressed Archive	172.35 MB	 jdk-11.0.7_windows-x64_bin.zip

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, `currentTimeMillis()`, para obtener el tiempo en milisegundos o `exit()` que termina el programa Java.

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/System.html>

Command Line

Comandos

javac: Lee las definiciones de clase e interfaz de Java y compilarlas en código de bytes y archivos de clase.

javap: desmontar uno o más archivos de clase.

javadoc: utiliza la herramienta javadoc para genera páginas HTML de documentación API a partir de archivos fuente Java.

java: Inicia una aplicación Java.

jar: puede usar el comando jar para crear un archivo para clases y recursos, y para manipular o restaurar clases o recursos individuales desde un archivo.

jlink: puede usar la herramienta jlink para ensamblar y optimizar un conjunto de módulos y sus dependencias en una imagen de tiempo de ejecución personalizada.

Comandos

jdeps: analizador de dependencias de clase Java.

jdeprscan: análisis estático que escanea un archivo jar (o alguna otra agregación de archivos de clase) para el uso de elementos API obsoletos.

jconsole: consola gráfica para monitorear y administrar aplicaciones Java.

jps: enumera las JVM instrumentadas en el sistema de destino.

jstat: Visualizamos las estadísticas de JVM.

jstatd: Se utiliza para monitorear la creación y terminación de máquinas virtuales Java HotSpot instrumentadas.

Compilando

```
javac -d build src/*.java
```

Ejecutando

```
java -cp build HelloWorld
```

-cp | -classpath | --class-path son lo mismo

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 la 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.

Sintaxis y keywords

Keywords

Conjunto de palabras clave 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				

Objetos y clases

Clases y Objetos

Objeto

- 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.

Clase

- 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.

```
public class Dog {  
    String breed;  
    int age;  
    String color;  
  
    void barking() {  
    }  
    void hungry() {  
    }  
    void sleeping() {  
    }  
}
```

Una clase puede contener cualquiera de los siguientes tipos de variables.

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

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.

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

Una clase puede tener cualquier cantidad de métodos para acceder al valor de diversos tipos de métodos. En el ejemplo anterior, `barking()`, `hungry()` y `sleep()` son métodos.

Constructores

Al hablar sobre las clases, uno de los subtemas 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.

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) {  
        // This constructor has one parameter, name.  
    }  
}
```

Creando un Objeto

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**.

Hay tres pasos al crear un objeto de una clase:

- **Declaración:** una declaración de variable con un nombre de variable con un tipo de objeto
- **Instanciación:** la palabra clave **new** se usa para crear el objeto.
- **Inicialización:** la palabra clave **new** es seguida por una llamada a un constructor. Esta llamada inicializa el nuevo objeto.

Ejemplo de creación de un objeto:

```
public class Puppy {  
    public Puppy(String name) {  
        // This constructor has one parameter, name.  
        System.out.println("Passed Name is :" + name );  
    }  
  
    public static void main(String []args) {  
        // Following statement would create an object myPuppy  
        Puppy myPuppy = new Puppy( "tommy" );  
    }  
}
```

Si compilamos y ejecutamos el programa anterior, producirá el siguiente resultado:
Salida

```
Passed Name is :tommy
```

Acceso a variables y métodos de instancia

Las variables de instancia y los métodos se acceden a través de objetos creados. Para acceder a una variable de instancia, a continuación se muestra la ruta completa

```
/* First create an object */  
ObjectReference = new Constructor();  
  
/* Now call a variable as follows */  
ObjectReference.variableName;  
  
/* Now you can call a class method as follows */  
ObjectReference.MethodName();
```

Ejemplo de cómo acceder a variables de instancia y métodos de una clase.

```
public class Puppy {  
    int puppyAge;  
  
    public Puppy(String name) {  
        // This constructor has one parameter, name.  
        System.out.println("Name chosen is :" + name );  
    }  
  
    public void setAge( int age ) {  
        puppyAge = age;  
    }  
  
    public int getAge( ) {  
        System.out.println("Puppy's age is :" + puppyAge );  
        return puppyAge;  
    }  
}
```

```
public static void main(String []args) {  
    /* Object creation */  
    Puppy myPuppy = new Puppy( "tommy" );  
  
    /* Call class method to set puppy's age */  
    myPuppy.setAge( 2 );  
  
    /* Call another class method to get puppy's age */  
    myPuppy.getAge( );  
  
    /* You can access instance variable as follows as well */  
    System.out.println("Variable Value :" + myPuppy.puppyAge );  
}  
}
```

Reglas de declaración de archivo de origen

- Solo puede haber una clase pública por archivo fuente.
- Un archivo fuente puede tener múltiples clases no públicas.
- El nombre de la clase pública debe ser también el nombre del archivo fuente, que debe ser anexado por **.java** al final. Por ejemplo: el nombre de la clase es *public class Employee {}*, luego el archivo fuente debe ser como `Employee.java`.
- Si la clase se define dentro de un paquete, la declaración del paquete debe ser la primera declaración en el archivo fuente.
- 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.

Paquete Java

Es una forma de categorizar las clases y las interfaces.

Declaraciones de importación

En Java si se proporciona un nombre completo, que incluye el paquete y el nombre de la clase, entonces el compilador puede ubicar fácilmente el código fuente o las clases. La instrucción de importación es una forma de dar la ubicación adecuada para que el compilador encuentre esa clase en particular.

```
import java.io.*;
```

- El único trabajo de una sentencia import es ahorrar escritura.
- Puede utilizar un asterisco (*) para buscar en el contenido de un solo paquete.
- "static import", se aplica para es de importación estática de identificadores y métodos.
- Puede importar clases de API y / o clases personalizadas.

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.

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”

IS-A, HAS-A

- **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.

Sobreescritura de métodos, polimorfismo y clases estáticas

Overriding

- Si una clase hereda un método de su superclase, existe la posibilidad de sobrescribir 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.

Reglas para Método Overriding

- 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*
- 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.

- Si un método no se puede heredar, no se puede sobrescribir.
- Una subclase dentro del mismo paquete que la superclase de la instancia puede anular cualquier método de superclase que no se declare privado o final.
- Una subclase en un paquete diferente solo puede anular los métodos no finales declarados públicos o protegidos.
- Un método de anulación puede arrojar cualquier excepción sin verificar, independientemente de si el método reemplazado arroja excepciones o no. Sin embargo, el método principal no debe arrojar excepciones comprobadas que sean nuevas o más amplias que las declaradas por el método reemplazado. El método de anulación puede arrojar excepciones más limitadas o menos que el método reemplazado.
- Los constructores no pueden ser sobrescritos.

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.

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.

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 sobrescrito se utiliza en tiempo de ejecución. El tipo de referencia determina qué método sobrecargado se utilizará durante la compilación.

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;
```

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");
```

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 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 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.VariableName`.

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.

Clases abstractas y anidadas

Clase abstracta

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 e interfaces abstractas.

Heredando la clase abstracta

Podemos heredar las propiedades de la clase Employee igual que la clase concreta.

Métodos abstractos

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 abstracta antes del nombre del método en la declaración del método. En lugar de llaves, un método abstracto tendrá un punto y coma (;) al final.

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 otra está permitido en Java. La clase escrita dentro se llama **clase anidada**, y la clase que contiene la clase interna se llama **clase externa**.

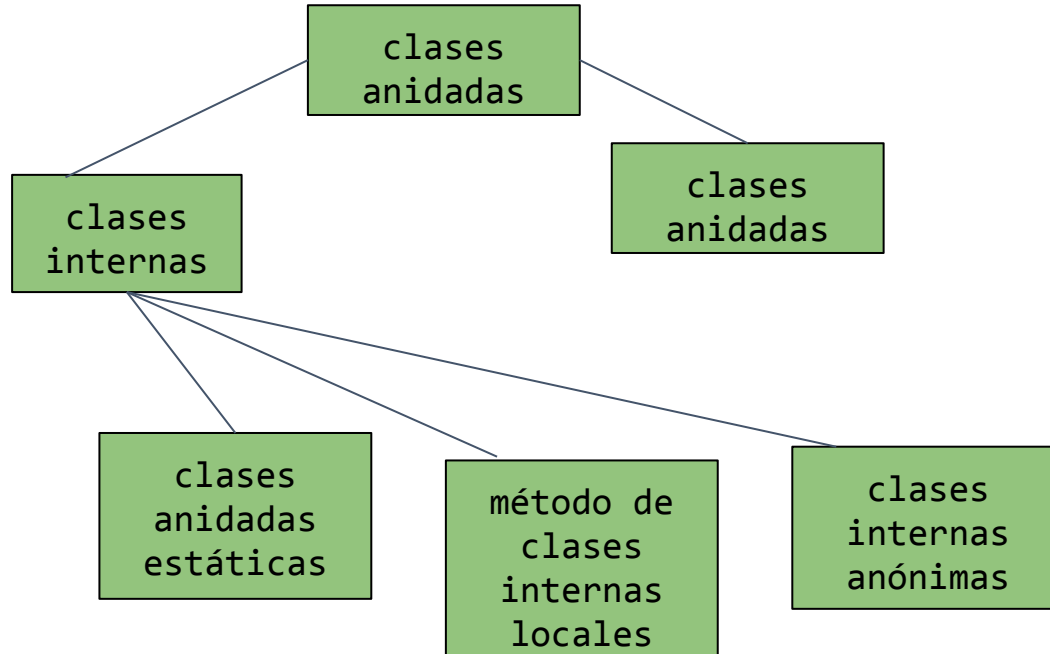
Sintaxis

Sintaxis para escribir una clase anidada. Aquí, la clase **Outer_Demo** es la clase externa y la clase **Inner_Demo** es la clase anidada.

```
class Outer_Demo {  
    class Nested_Demo {  
    }  
}  
class NO_Nested_Demo {  
}
```


Las clases anidadas se dividen en dos tipos

- Clases anidadas no estáticas
- Clases anidadas estáticas



Clases internas (Clases 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.

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

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.

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 el método de una clase o una interfaz.

Clase interna anónima como argumento

Generalmente, si un método acepta un objeto de una interfaz, una clase abstracta o una clase concreta, entonces podemos implementar la interfaz, 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 es una clase anidada que es 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.

Modificadores de acceso

Hay tres modificadores de acceso: public, protected y private, sin embargo existen 4 tipos de niveles de acceso: public, protected, default y private.

Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
<i>no modifier</i>	Y	Y	N	N
private	Y	N	N	N

Modificadores de acceso de miembro

- Los métodos y variables de instancia (no locales) se conocen como "miembros" (member).
- Los miembros pueden utilizar los cuatro niveles de acceso: public, protected, default, y private. 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.
- Si no se puede acceder a una clase, no se puede acceder a sus miembros.
- Determine la visibilidad de la clase antes de determinar la visibilidad del miembro.

Reglas sobre los tipos de acceso de miembro

- 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.

Modificadores de acceso de clase

- Las clases 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 para extender una clase abstracta debe implementar todos sus métodos abstractos.

Excepciones y Afirmaciones

Excepciones

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 ser manejadas.

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.

Categorías de Excepciones

Excepciones comprobada: 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: 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.

try-catch y throw

Un bloque de prueba se utiliza para encerrar el código que podría arrojar una excepción y puede ser seguido por uno o muchos bloques de captura.

Un bloque catch se usa para manejar una excepción. Define el tipo de la excepción y una referencia.

```
try {  
    // Code that may throw an exception  
} catch(Exception e){  
    // Do something with the exception using  
    reference e  
}
```

Si no se maneja la excepción, Java Virtual Machine proporciona un controlador de excepción predeterminado que realiza las siguientes tareas:

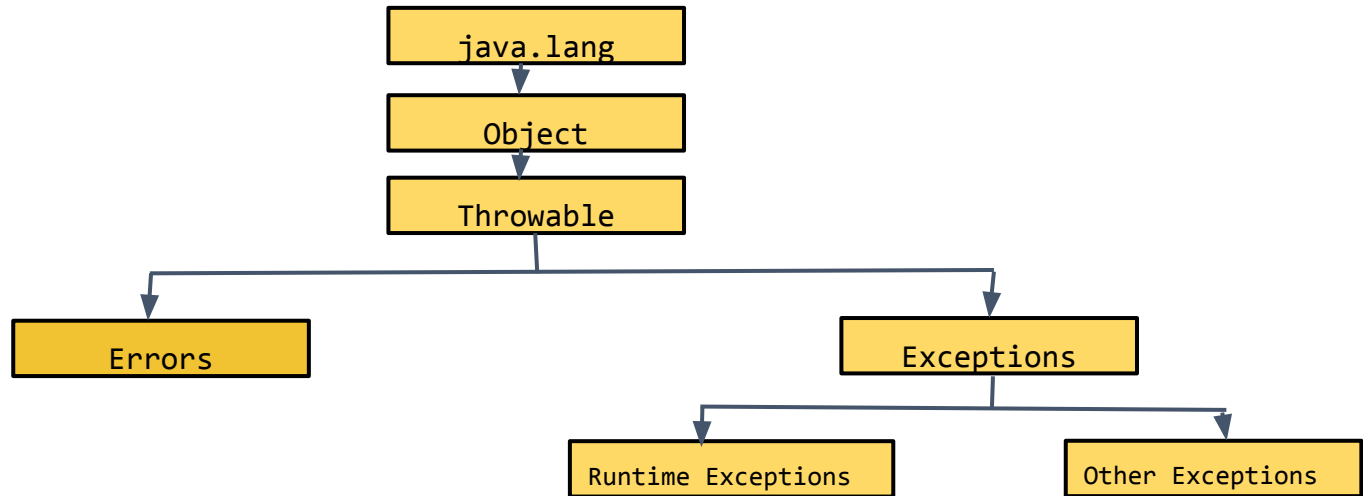
1. Imprime la descripción de la excepción.
2. Imprime el seguimiento de pila (Jerarquía de métodos donde se produjo la excepción).
3. Hace que el programa finalice.

Pero si se maneja una excepción en un bloque try-catch, se mantiene el flujo normal de la aplicación y se ejecuta el resto del código.

Si desea lanzar una excepción manualmente, use la palabra clave throw.

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`.



Try-with-resources

En la versión 7, se incorporó el concepto de try-with-resources, el cual cierra de manera automática los objetos asignados en la sentencia try que implementen `java.lang.AutoCloseable`

```
class TryWithResources1 {  
    public static void main(String [] args) {  
        System.out.println("Type an integer in the console: ");  
        try(Scanner consoleScanner = new Scanner(System.in)) {  
            System.out.println("You typed the integer value: " + consoleScanner.nextInt());  
        } catch(Exception e) { }  
    }  
}
```

catch

Si está manejando múltiples excepciones, los bloques catch deben ordenarse de la más específica a la más general.

Por ejemplo, la captura para `IndexOutOfBoundsException` debe venir antes de la captura de `Exception`, de lo contrario, se genera un error en tiempo de compilación.

```
try {  
    int arr[] = new int[5];  
    arr[10] = 20;  
}  
catch(ArrayIndexOutOfBoundsException e) {}  
catch(ArithmeticException e) {}
```


El problema con este ejemplo es que contiene código duplicado en cada uno de los bloques catch. Desde Java 7, usar un bloque de captura múltiple:

```
try {  
    int arr[] = new int[5];  
    arr[10] = 20;  
}  
catch(ArrayIndexOutOfBoundsException|ArithmeticException e) {}
```

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  
}
```

Assert o aserciones

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  
}
```

En la primera forma, la expresión afirmar debe evaluar a un valor booleano. La otra versión agrega una segunda expresión separada de la primera expresión booleana por dos puntos. Esta expresión se usaría cuando la aserción sea falsa además de arrojar `AssertionError`. Esta segunda expresión debe resolverse en un valor; de lo contrario, se genera un error en tiempo de compilación.

Pero para ejecutar las comprobaciones de aserción, debe habilitarlas con:

```
java -ea com.example.Test  
o  
java -enableassertions com.example.Test
```

Pero no todos los usos de aserciones se consideran apropiados. Estas son las reglas:

- No use aserciones para validar argumentos en un método público
- Usar aserciones para validar argumentos a un método privado
- No use aserciones para validar argumentos de línea de comandos
- Use afirmaciones, incluso en métodos públicos, para buscar casos que nunca se supone que sucedan
- No use expresiones de afirmación que puedan causar efectos secundarios.

Interfaces Java

Interfaces

Las interfaces 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.

Todos sus métodos por defecto son públicos, la declaración explícita de los mismos es opcional.

Una interfaz es como una clase abstracta de 100% y es implícitamente abstracta si escribe el modificador abstracto en la declaración o no.

Reglas de implementación

- 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.
- Una clase de implementación legal no abstracta tiene las siguientes propiedades:
- Debe proporcionar implementaciones concretas para los métodos de la interfaz.
- 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 interfaz.

- Puede declarar excepciones de tiempo de ejecución en cualquier implementación de método de interfaz independientemente de la declaración de interfaz.
- 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 interfaz).

```
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  
}
```


- Una clase que implementa una interfaz puede ser abstract.
- Una clase de implementación abstracta no tiene que implementar los métodos de interfaz (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 interfaz.
- Al realizar el examen, verifique que las declaraciones de interfaz y clase sean legales antes de verificar otra lógica de código.

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.

Ejemplo

```
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 {}
```

Características y uso

- 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 interfaz.
- 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*

Interfaces Funcionales

Interfaces Funcionales

Concepto nuevo en Java SE 8 y que es la base para que podamos escribir expresiones lambda. Una interfaz funcional se define como una interface que tiene uno y solo un método abstracto y que éste sea diferente a los métodos definidos en `java.lang.Object` (a saber: `equals`, `hashCode`, `clone`, etc.).

La interface puede tener métodos por defecto y estáticos sin que esto afecte su condición de ser interfaz funcional.

Existe una nueva anotación denominada `@FunctionalInterface` que permite al compilador realizar la validación de que la interfaz tenga solamente un método abstracto.

Colecciones y genéricos

Características de Generics

Las características de Java Generics se agregaron al lenguaje Java de Java 5, donde se añaden una forma de especificar tipos de concreto a clases de propósito general y métodos que operaban en con Object.

La interfaz de List representa una lista de instancias de objetos. Esto significa que podríamos poner cualquier objeto en una lista.

```
List list = new ArrayList();  
list.add(new Integer(2));  
list.add("a String");  
Integer integer = (Integer) list.get(0);  
String string = (String) list.get(1);
```


Debido a que cualquier objeto podría agregarse, también debería lanzar cualquier objeto obtenido de estos mismos objetos.

Con las características de Java Generics es posible establecer el tipo de colección para limitar los tipos de objetos que se pueden insertar en la colección. Además, no tiene que convertir los valores que obtiene de la colección.

```
List<String> strings = new ArrayList<String>();  
strings.add("a String");  
String aString = strings.get(0);
```

Type Inference (operador diamante)

Las características Generics de Java se actualizaron en Java 7, el compilador de Java puede inferir el tipo de la colección instanciada a partir de la variable a la que está asignada la colección.

```
List<String> strings = new ArrayList<>();
```

En este ejemplo se ha omitido el tipo genérico de `ArrayList`. En su lugar, solo está `<>` escrito. Esto también se conoce como el operador de diamantes, que al escribirlo como tipo genérico, el compilador de Java supondrá que la clase instanciada debe tener el mismo tipo que la variable a la que está asignado.

Genéricos de Java para Loop

Java 5 también obtuvo un nuevo for-loop (también conocido como "for-each") que funciona bien con colecciones genéricas.

Este bucle for-each itera a través de todas las instancias String guardadas en la lista de cadenas. Para cada iteración, la siguiente instancia de String se asigna a la variable aString. Este for-loop es más corto que el original while-loop donde iterarías el recopilador Iterator y llama a Iterator.next() para obtener la siguiente instancia.

Genéricos de Java para otros tipos de colecciones

Por supuesto, es posible usar Generics para otras clases además de las colecciones de Java. También puedes generar tus propias clases.

Java - Generics

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

Métodos genéricos

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

- Todas las declaraciones de métodos genéricos tienen una sección de parámetros de tipo delimitada por corchetes angulares (<y>) que precede al tipo de devolución del método (<E> en el siguiente ejemplo).
- Cada sección de parámetro 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 devolución y actuar como marcadores de posición para los tipos de argumentos pasados al método genérico, que se conocen como argumentos de tipo reales.
- El cuerpo de un método genérico se declara como el de cualquier otro método. Tenga en cuenta que los parámetros de tipo pueden representar únicamente tipos de referencia, no tipos primitivos (como int, double y char).

Java - Collections Framework

Antes de Java 2, Java proporcionaba clases ad hoc como Dictionary, Vector, Stack y Properties para almacenar y manipular grupos de objetos. Aunque estas clases fueron bastante útiles, carecían de un tema central y unificador.

El framework de colecciones se diseñó para cumplir varios objetivos, tales como:

- Alto rendimiento. Las implementaciones para las colecciones fundamentales (matrices dinámicas, listas enlazadas, árboles y hashtables) debían ser altamente eficientes.
- Debía permitir que diferentes tipos de colecciones funcionaran de manera similar y con un alto grado de interoperabilidad.
- Tenía que ampliar y / o adaptar una colección fácilmente.

Con este fin, todo el framework de colecciones está diseñado en torno a un conjunto de interfaces estándar, tales como LinkedList, HashSet y TreeSet, estas interfaces pueden usarse tal como están y también pueden implementar su propia colección. Un framework de colecciones es una arquitectura unificada para representar y manipular colecciones.

Todos los componentes del framework de colecciones contienen lo siguiente:

- Interfaces: son tipos de datos abstractos que representan colecciones.
- Implementaciones, es decir, clases.
- Algoritmos: estos son los métodos que realizan cálculos útiles, como buscar y ordenar, en objetos que implementan interfaces de colección.

Collections Framework

Collections

Antes de Java 2, Java proporciona clases ad hoc como Dictionary, Vector, Stack y Properties para almacenar y manipular grupos de objetos, aunque estas clases fueron bastante útiles, carecían de un tema central y unificador, para ello se diseñó un framework para cumplir varios objetivos, tales como:

- El framework tenía que ser de alto rendimiento. Las implementaciones para las colecciones fundamentales (matrices dinámicas, listas enlazadas, árboles y hashtables) debían ser altamente eficientes.
- Debía permitir que diferentes tipos de colecciones funcionaran de manera similar y con un alto grado de interoperabilidad.
- Tenía que ampliar y / o adaptar una colección fácilmente.

Las colecciones de Java tiene tres componentes principales:

Clases e interfaces abstractas: El marco de las colecciones tiene muchas clases abstractas e interfaces que proporcionan funcionalidad general.

Clases concretas: Estas son las instancias reales de contenedores que usarás en la programas.

Algoritmos: Los implementos `java.util.Collections` normalmente requieren funcionalidad como clasificación, búsqueda, etc. Estos métodos son genéricos: puede utilizar estos métodos en diferentes contenedores.

Collection vs Collections

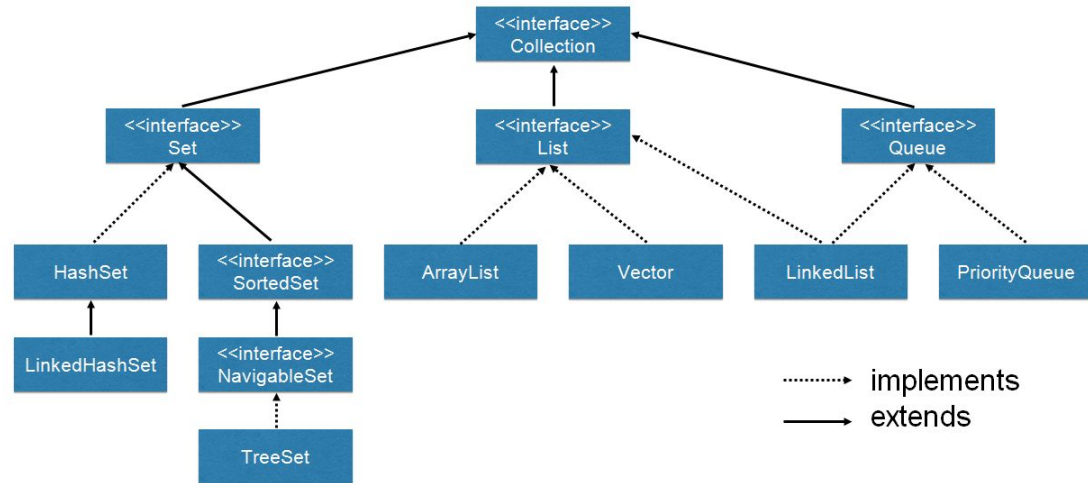
Se debe tener en cuenta que la Collection(s) es un término genérico, mientras que la Collection y las Collections son las API específicas del paquete java.util.

Las Collections (java.util.Collections) es una clase de utilidad que contiene sólo métodos estáticos.

El término general Collection(s) se refiere a un contenedor como mapa, pila, cola, etc. Utilice el término container(s) al referirse a estas colecciones en este capítulo para evitar confusiones.

Biblioteca de colecciones

La biblioteca Java tiene un marco de colecciones que hace uso extensivo de genéricos y proporciona un conjunto de contenedores y algoritmos para su uso y extensión.



Iterable	Una clase que implementa esta interfaz puede usarse para iterar con a para cada declaración.
Collection	Interfaz base común para las clases de la jerarquía de la Collection. Cuando quieres escribir métodos que son muy generales, puede pasar la interfaz de colección.
List	Interfaz base para contenedores que almacenan una secuencia de elementos. Puede acceder al elementos que utilizan un índice y recuperar el mismo elemento posteriormente (random access). Puede almacenar elementos duplicados en una lista.
Set, SortedSet, NavigableSet Queue, Deque	Interfaces para contenedores que no se permite elementos duplicados. SortedSet mantiene los elementos del conjunto en un orden. NavigableSet permite buscar los más cercanos. Queue es una interfaz de base para contenedores que contiene una secuencia de elementos para tratamiento. Por ejemplo, las clases que implementan Queue pueden ser LIFO (last in, first out) o FIFO (first in, first out-as). En un Deque puedes insertar o eliminar elementos de ambos extremos.

Map, SortedMap, NavigableMap	Clase de base para contenedores que asignan claves a valores. En SortedMap, las claves están ordenadas. Un NavigableMap le permite buscar y devolver más cercana para determinados criterios de búsqueda.
Iterator, ListIterator	Puede desplazarse sobre el contenedor en la dirección de avance si una clase implementa la interfaz Iterator. Puede desplazarse en ambas direcciones r implementa la interfaz ListIterator.

Numerosas interfaces y clases abstractas en la jerarquía de Collection proporcionan los métodos comunes que las clases concretas implementan / extienden. Las clases concretas proporcionan la funcionalidad real

ArrayList	Implementado internamente como un array redimensionable. Este es uno de los casos más utilizados clases. Rápido para buscar, pero lento para insertar o eliminar. Permite duplicados.
LinkedList	Implementa internamente una estructura de datos de lista doblemente vinculada. Rápido para insertar o eliminar elementos, pero lento para buscar elementos. Además, LinkedList se puede utilizar cuando se necesita una pila (LIFO) o cola (FIFO). Permite duplicados.
HashSet	Implementado internamente como una estructura de datos de hash-table. No permite almacenar elementos duplicados. Rápido para buscar y recuperar elementos. No mantiene ningún orden para los elementos almacenados.

TreeSet	Implementa internamente una estructura de datos de árbol rojo-negro. Al igual que HashSet, TreeSet no permite almacenar duplicados. Sin embargo, a diferencia de HashSet, almacena los elementos en un orden. Utiliza un árbol de estructura de datos para decidir dónde almacenar o buscar los elementos, y la posición se decide por el orden de clasificación.
HashMap	Implementado internamente como una estructura de datos de tabla hash. Almacena pares de claves y valores. Usos hashing para encontrar un lugar para buscar o almacenar un par. Buscar o insertar es muy rápido. Almacena los elementos en cualquier orden.
TreeMap	Implementado internamente usando una estructura de datos de árbol rojo-negro. A diferencia de HashMap, los elementos están almacenados de manera ordenada.
PriorityQueue	Implementado internamente usando la estructura de datos del pila. Un PriorityQueue es para recuperar elementos basado en la prioridad. Independientemente del orden en que se inserte, al quitar el, el elemento de prioridad más alta se recuperará primero.

Collection

La interfaz Collection proporciona métodos como add() y remove() que son comunes a todos los contenedores.

<code>boolean add(Element elem)</code>	Agrega elem al contenedor subyacente.
<code>void clear()</code>	Elimina todos los elementos del contenedor.
<code>boolean isEmpty()</code>	Comprueba si el contenedor tiene algún elemento o no.
<code>Iterator<Element> iterator()</code>	Devuelve un objeto Iterator <Element> para iterar sobre el contenedor.
<code>boolean remove(Object obj)</code>	Elimina el elemento si obj está presente en el contenedor.
<code>int size()</code>	Devuelve el número de elementos del contenedor.
<code>Object[] toArray()</code>	Devuelve un array que tiene todos los elementos del contenedor.

<code>boolean addAll(Collection<? extends Element> coll)</code>	Agrega todos los elementos dados en la colección pasada como parámetro.
<code>boolean containsAll(Collection<?> coll)</code>	Comprueba si todos los elementos dados en la colección que están presentes en contenedor.
<code>boolean removeAll(Collection<?> coll)</code>	Remueve del contenedor los elementos dados en la colección pasada como parámetro.
<code>boolean retainAll(Collection<?> coll)</code>	Conserva en el contenedor solamente los elementos dados en la colección y elimina todos los demás elementos.

Iterator

La interfaz iterator se encuentra en muchas colecciones y nos permite iterar el contenido de la colección. La misma es muy simple y contiene los siguientes métodos.

<code>boolean hasNext()</code>	Comprueba si el iterador tiene más elementos a recorrer.
<code>E next()</code>	Mueve el iterador al siguiente elemento y devuelve ese elemento (siguiente).
<code>void remove()</code>	Remueve el último elemento.

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 Class

`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 crece según sea necesario.

Por lo tanto, acceder a los elementos del array es muy rápidamente 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 Class

La clase 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.

```
String palStr = "abcba";
List<Character> palindrome = new LinkedList<Character>();
for(char ch : palStr.toCharArray())
    palindrome.add(ch);
System.out.println("Input string is: " + palStr);
ListIterator<Character> iterator = palindrome.listIterator();
ListIterator<Character> revIterator = palindrome.listIterator (palindrome.size());
boolean result = true;
while(revIterator.hasPrevious() && iterator.hasNext()) {
    if(iterator.next() != revIterator.previous()){
        result = false;
        break;
    }
}
if (result)
    System.out.print("Input string is a palindrome");
else
    System.out.print("Input string is not a palindrome");
}
```

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 como este:

```
String tongueTwister = "I feel, a feel, a funny feel, a funny feel I feel,"  
                      +"if you feel the feel I feel, I feel the feel you feel";  
Set<String> words = new HashSet<>();  
  
// split the sentence into words and try putting them in the set  
for(String word : tongueTwister.split("\\W+"))  
    words.add(word);  
  
System.out.println("The tongue twister is: " + tongueTwister);  
System.out.print("The words used were: ");  
System.out.println(words);
```

Dando como respuesta

```
The tongue twister is: I feel, a feel, a funny feel, a funny feel I feel,  
if you feel the feel I feel, I feel the feel you feel  
The words used were: [feel, if, a, funny, you, the, I]
```


TreeSet Class

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

```
String pangram = "the quick brown fox jumps over the lazy dog";  
Set<Character> aToZee = new TreeSet<Character>();  
for(char gram : pangram.toCharArray())  
    aToZee.add(gram);  
System.out.println("The pangram is: " + pangram);  
System.out.print("Sorted pangram characters are: " + aToZee);
```

Dando como respuesta

```
The pangram is: the quick brown fox jumps over the lazy dog  
Sorted pangram characters are: [ , a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u,  
v,w, x, y, z]
```

La interfaz 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.

```
Map<String, String> misspeltWords = new HashMap<String, String>();
misspeltWords.put("calender", "calendar");
misspeltWords.put("tomatos", "tomatoes");
misspeltWords.put("existance", "existence");
misspeltWords.put("acquaintance", "acquaintance");
String sentence = "Buy a calender for the year 2013";
System.out.println("The given sentence is: " + sentence);
for(String word : sentence.split("\\W+")) {
    if(misspeltWords.containsKey(word)) {
        System.out.println("The correct spelling for " + word
            + " is: " + misspeltWords.get(word));
    }
}
```

y obtenemos

```
The given sentence is: Buy a calender for the year 2013
The correct spelling for calender is: calendar
```

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, con NavigableMap, 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.

<code>boolean offer(Element)</code>	Agrega elem en la pila .
<code>Element remove(Element)</code>	Obtiene y remueve el primer elemento (exception cuando no existe)
<code>Element poll()</code>	Obtiene y remueve el primer elemento
<code>Element element()</code>	Remueve el último elemento
<code>Element peek()</code>	Obtiene el primer elemento pero no lo remueve.

Deque (Doubly ended queue)

Es una estructura de datos que le permite insertar y eliminar elementos de ambos extremos, 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`.

<code>void addFirst(Element)</code>	Agrega elem al frente.
<code>void addLast(Element)</code>	Agrega elem al último.
<code>Element removeFirst()</code>	Remueve el primer elemento.
<code>Element removeLast()</code>	Remueve el último elemento
<code>Element getFirst()</code>	Obtiene el primer elemento.
<code>Element getLast()</code>	Obtiene el último elemento.

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)
```

Overriding the hashCode

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;
    }
    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;
    }
    public int hashCode() {
    }
}
```


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.

Comparable y Comparator

Las interfaces Comparable y Comparator se utilizan para comparar objetos similares (por ejemplo, mientras realiza la búsqueda o la clasificación). Suponga que tiene un contenedor que contiene una lista de objeto Persona. Hay cualquier número de atributos comparables, tales como DNI, nombre, número de la licencia de conducir, y así sucesivamente. Dos objetos se pueden comparar en DNI así como el nombre de la persona; esto depende sobre el contexto. Por lo tanto, el criterio para comparar los objetos Persona no puede ser predefinido; un desarrollador tiene que definir este criterio.

Java define Comparable y Comparator interfaces para solucionar esta necesidad.

La clase Comparable tiene sólo un método compareTo(), que se declara de la siguiente manera:

```
int compareTo (Element i)
```

Puesto que está implementando el método `compareTo()`, en una clase, tiene esta referencia disponible. Usted puede comparar el elemento actual con el elemento pasado y devuelva un valor int.

La regla para este valor es:

```
devuelve 1 si el objeto actual > objeto a comparar  
devuelve 0 si el objeto actual == objeto a comparar  
devuelve -1 si el objeto actual < objeto a comparar
```

Resumen

- 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).`

Resumen

- Evite mezclar tipos crudos con tipos genéricos. En otros casos, asegúrese de que el tipo de seguridad a mano.
- Los términos Colección, Colección y Recolección son diferentes.
- Collection - `java.util.Collection <E>` - es la interfaz raíz en la jerarquía de la colección.
- Collections-`java.util.Collections`-es una clase de utilidad que contiene sólo métodos estáticos.
- El término general colección (s) se refiere a contenedores como mapa, pila, cola, etc.
- Las clases contenedor almacenan referencias a objetos, por lo que no puede utilizar tipos primitivos con de las clases de recolección.
- Los métodos `hashCode ()` y `equals ()` deben ser coherentes para una clase. Para la práctica, asegúrese de seguir esta regla: el método `hashCode ()` debe devolver lo mismo, para dos objetos si el método `equals ()` devuelve true para ellos.

Resumen

- Si está utilizando un objeto en contenedores como HashSet o HashMap, asegúrese de anular los métodos hashCode() y equals() correctamente.
- La interfaz Map no extiende la interfaz de Collection.
- No se recomienda almacenar null como argumento, ya que existen métodos en la Deque que devuelve nulo, y sería difícil para usted distinguir entre el éxito o el fracaso de la llamada al método.
- Implementar la interfaz Comparable para sus clases donde un orden natural es posible. Si tu deseas comparar los objetos distintos del orden natural o si no hay un orden natural presente para su tipo de clase, luego cree clases separadas implementando el Comparator.

Expresiones Lambda

Interfaces y expresiones Lambda

Las características más importantes de Java SE 8 son la incorporación de Expresiones Lambda y la API Stream.

```
{args} -> {lambda block}
```

Mediante uso de expresiones lambda podemos crear un código más conciso y significativo, además de abrir la puerta hacia la programación funcional en Java, en donde las funciones juegan un papel fundamental.

Por otro lado, la API Stream nos permite realizar operaciones de tipo filtro/mapeo/reducción sobre colecciones de datos de forma secuencial o paralela y que su implementación sea transparente para el desarrollador. Lambdas y Stream son una combinación muy poderosa que requiere un cambio de paradigma en la forma en la que hemos escrito código Java hasta el momento.

Expresiones Lambda

Existe un problema con las clases anónimas cuando la implementación de su clase anónima es muy simple, como una interfaz que contiene solo un método, entonces la sintaxis de las clases anónimas puede parecer difícil de manejar y poco clara. En estos casos, por lo general, debemos intentar pasar la funcionalidad como argumento a otro método, como qué acción debe realizarse cuando alguien hace clic en un botón.

Por medio de expresiones lambda podemos referenciar métodos anónimos o métodos sin nombre, lo que nos permite escribir código más claro y conciso que cuando usamos clases anónimas. Una expresión lambda se compone de:

- Listado de parámetros separados por comas y encerrados en paréntesis, por ejemplo: (a,b).
- El símbolo de flecha hacia la derecha: ->
- Un cuerpo que puede ser un bloque de código encerrado entre llaves o una sola expresión.

Interfaces funcionales incorporadas

Lambda Expressions

Un problema con las clases anónimas es que si la implementación de su clase anónima es muy simple, como una interfaz que contiene solo un método, entonces la sintaxis de las clases anónimas puede parecer difícil de manejar y poco clara.

En estos casos, por lo general, intenta pasar la funcionalidad como argumento a otro método, como qué acción debe realizarse cuando alguien hace clic en un botón.

Las expresiones de Lambda le permiten hacer esto, para tratar la funcionalidad como argumento de método o código como datos.

Las expresiones Lambda le permiten expresar instancias de clases de método único de forma más compacta.

- Funciones como entidades de primer nivel: Las funciones ahora tienen un rol protagónico cuando de expresiones lambda se trata.
- Métodos por defecto/estáticos en interfaces: Evolución de librerías sin perder compatibilidad gracias a que ahora podemos definir e implementar métodos en las interfaces.
- Interfaces funcionales: Concepto clave para poder escribir expresiones lambda. Interfaces con solo un método abstracto.
- Inferencia de tipos: Revisamos los pasos que realiza el compilador para inferir los tipos de las expresiones lambda en contextos de asignación y de invocación de métodos (parámetros).
- Alcance de las expresiones lambda: Efectivamente Constante y tener en cuenta la diferencia entre clases anónimas y expresiones lambda en cuanto a la palabra reservada `this` se refiere.
- Métodos de referencia: Es otra forma de escribir expresiones lambda de una sola sentencia, con lo cual se logra un código más compacto y fácil de leer.

Las características más importantes de Java SE 8 son la incorporación de Expresiones Lambda y la API Stream. Con la incorporación de expresiones lambda podemos crear código más conciso y significativo, además de abrir la puerta hacia la programación funcional en Java, en donde las funciones juegan un papel fundamental.

Por otro lado, la API Stream nos permite realizar operaciones de tipo filtro/mapeo/reducción sobre colecciones de datos de forma secuencial o paralela y que su implementación sea transparente para el desarrollador.

Lambdas y Stream son una combinación muy poderosa que requiere un cambio de paradigma en la forma en la que hemos escrito código Java hasta el momento.

Consejos en el uso de Lambdas

Utilice las interfaces funcionales estándar

Las interfaces funcionales, que se recopilan en el paquete `java.util.function`, satisfacen las necesidades de la mayoría de los desarrolladores al proporcionar tipos de destino para expresiones lambda y referencias de métodos. Cada una de estas interfaces es general y abstracta, lo que facilita su adaptación a casi cualquier expresión lambda. Los desarrolladores deben explorar este paquete antes de crear nuevas interfaces funcionales.

Dado

```
@FunctionalInterface  
public interface Foo {  
    String method(String string);  
}
```

y un método add () en alguna clase UseFoo , que toma esta interfaz como un parámetro:

```
public String add(String string, Foo foo) {  
    return foo.method(string);  
}
```

Ahora podemos eliminar la interfaz Foo por completo y cambiar nuestro código a:

```
public String add(String string, Function<String, String> fn) {  
    return fn.apply(string);  
}
```

Para ejecutar esto, podemos escribir:

```
Function<String, String> fn = parameter -> parameter + " from  
lambda";  
String result = useFoo.add("Message ", fn);
```

Foo no es más que una función que acepta un argumento y produce un resultado. Java 8 ya proporciona dicha interfaz en la Función `<T, R>` del paquete `java.util.function`

```
Foo foo = parameter -> parameter + " from lambda";  
String result = useFoo.add("Message ", foo);
```

Utilice la anotación `@FunctionalInterface`

Anota tus interfaces funcionales con `@FunctionalInterface`. Al principio, esta anotación parece ser inútil. Incluso sin él, su interfaz se tratará como funcional siempre que solo tenga un método abstracto.

Pero imagine un gran proyecto con varias interfaces: es difícil controlar todo manualmente. Una interfaz, que fue diseñada para ser funcional, podría ser cambiada accidentalmente agregando otros métodos / métodos abstractos, dejándola inutilizable como una interfaz funcional.

Pero al usar la anotación `@FunctionalInterface`, el compilador desencadenará un error en respuesta a cualquier intento de romper la estructura predefinida de una interfaz funcional. También es una herramienta muy útil para hacer que la arquitectura de su aplicación sea más fácil de entender para otros desarrolladores.

No use en exceso los métodos predeterminados en las interfaces funcionales

Al igual que con las interfaces regulares, extender diferentes interfaces funcionales con el mismo método predeterminado puede ser problemático.

Por ejemplo, supongamos que las interfaces Bar y Baz tienen un método predeterminado `defaultCommon`. En este caso, obtendrá un error en tiempo de compilación.

Agregar demasiados métodos predeterminados a la interfaz no es una muy buena decisión arquitectónica. Se debe considerar como un compromiso, solo para ser utilizado cuando sea necesario, para actualizar las interfaces existentes sin romper la compatibilidad con versiones anteriores.

Crear una instancia de interfaces funcionales con expresiones lambda

El compilador le permitirá usar una clase interna para instanciar una interfaz funcional. Sin embargo, esto puede conducir a un código muy detallado.

Deberías preferir las expresiones lambda, por sobre una clase interna.

```
Foo foo = parameter -> parameter + " from lambda";
```

```
Foo fooByIC = new Foo() {  
    @Override  
    public String method(String string) {  
        return string + " from Foo";  
    }  
};
```

El enfoque de expresión lambda se puede usar para cualquier interfaz adecuada de bibliotecas antiguas. Se puede usar para interfaces como `Runnable`, `Comparator`, etc. Sin embargo, esto no significa que deba revisar toda su base de códigos anterior y cambiar todo.

Evite métodos de sobrecarga con interfaces funcionales como parámetros

Use métodos con diferentes nombres para evitar colisiones ya el intento de ejecutarlo nos dará un error de método ambiguo.

```
public interface Adder {  
    String add(Function<String, String> f);  
    void add(Consumer<Integer> f);  
}  
  
public class AdderImpl implements Adder {  
    @Override  
    public String add(Function<String, String> f) {  
        return f.apply("Something ");  
    }  
    @Override  
    public void add(Consumer<Integer> f) {}  
}
```

Pero cualquier intento de ejecutar cualquiera de los métodos de AdderImpl :

```
String r = adderImpl.add(a -> a + " from lambda");
```

Obtendremos el siguiente error

```
reference to add is ambiguous both method  
add(java.util.function.Function<java.lang.String,java.lang.String>)  
in fiandlambdas.AdderImpl and method  
add(java.util.function.Consumer<java.lang.Integer>)  
in fiandlambdas.AdderImpl match
```

Para resolver este problema, tienes dos opciones. El primero es usar métodos con diferentes nombres:

```
String addWithFunction(Function<String, String> f);  
void addWithConsumer(Consumer<Integer> f);
```

El segundo es realizar el lanzamiento de forma manual. Esto no es recomendable:

```
String r = Adder.add((Function) a -> a + " from lambda");
```

No trate las expresiones Lambda como clases internas

A pesar de nuestro ejemplo anterior, donde esencialmente sustituimos la clase interna por una expresión lambda, los dos conceptos son diferentes de una manera importante: alcance.

Cuando usa una clase interna, crea un nuevo alcance. Puede sobrescribir las variables locales del alcance adjunto creando instancias de nuevas variables locales con los mismos nombres. También puede usar la palabra clave `this` dentro de su clase interna como referencia a su instancia.

Sin embargo, las expresiones lambda funcionan con el alcance adjunto. No puede sobrescribir las variables del ámbito adjunto dentro del cuerpo de lambda. En este caso, la palabra clave **this** es una referencia a una instancia que encierra.

Mantenga las expresiones Lambda cortas y autoexplicativas

Si es posible, use construcciones de una línea en lugar de un bloque grande de código. No es un dogma, pero recuerde que lambdas debería ser una expresión, no una narración. A pesar de su sintaxis concisa, lambdas debe expresar con precisión la funcionalidad que proporcionan. Esto es principalmente un consejo de aspecto, ya que el rendimiento no cambiará. En general, sin embargo, es mucho más fácil de entender y trabajar con dicho código.

Evitar bloques de código en el cuerpo de Lambda

En una situación ideal, lambdas debe escribirse en una línea de código. Con este enfoque, la lambda es una construcción autoexplicativa, que declara qué acción se debe ejecutar con qué datos (en el caso de lambdas con parámetros). Si tiene un bloque grande de código, la funcionalidad de lambda no está clara inmediatamente.

Evite especificar tipos de parámetros

Un compilador en la mayoría de los casos puede resolver el tipo de parámetros lambda con la ayuda de la inferencia de tipo . Por lo tanto, agregar un tipo a los parámetros es opcional y se puede omitir.

```
(a, b) -> a.toLowerCase() + b.toLowerCase();  
//en lugar de esto:  
(String a, String b) -> a.toLowerCase() + b.toLowerCase();
```

Evite paréntesis alrededor de un parámetro único

La sintaxis Lambda requiere paréntesis sólo alrededor de más de un parámetro o cuando no hay ningún parámetro. Es por eso que es seguro hacer que su código sea un poco más corto y excluir paréntesis cuando solo hay un parámetro.

Evitar declaración de devolución y llaves

Las llaves y las declaraciones de **return** son opcionales en cuerpos lambda de una sola línea. Esto significa que pueden omitirse por claridad y concisión.

Usar las referencias del método

Muy a menudo, incluso en nuestros ejemplos anteriores, las expresiones lambda simplemente llaman a métodos que ya están implementados en otros lugares. En esta situación, es muy útil usar otra característica de Java 8 como es la referencia a método.

Usar variables "efectivamente finales"

El acceso a una variable no final dentro de las expresiones lambda causará el error en tiempo de compilación. Pero eso no significa que deba marcar todas las variables objetivo como definitivas.

De acuerdo con el concepto "**efectivamente final**" , un compilador trata cada variable como definitiva, siempre que se asigne solo una vez.

El paradigma "efectivamente final" ayuda mucho aquí, pero no en todos los casos. Lambdas no puede cambiar el valor de un objeto del alcance circundante. Pero en el caso de variables de objetos mutables, un estado podría cambiarse dentro de expresiones lambda.

```
int[] total = new int[1];  
Runnable r = () -> total[0]++;  
r.run();
```

Proteja las variables de objeto de la mutación

Uno de los principales propósitos de lambdas es el uso en informática paralela, lo que significa que son realmente útiles cuando se trata de seguridad de hilos.

java.util.function

Consumer<T> Operación que acepta un argumento y no retorna valor.

Function<T,R> Función que acepta un argumento T y produce un resultado.

Supplier<T> Proveedor de objetos del tipo T.

Predicate<T> Representa un predicado de un solo argumento (Boolean-valued function).

BiConsumer<T,U> Operación que acepta dos argumentos, y no retorna valor.

BiFunction<T,U,R> Función que acepta dos argumentos y produce resultado.

BinaryOperator<T> Operación recibe dos operadores del mismo tipo y produce uno del mismo tipo.

BiPredicate<T,U> Predicado (Boolean-valued function) de dos argumentos.

Optional

Antes de Java8

Antes de Java 8, los programadores devolverían nulo en lugar de `Optional`. Hubo algunas deficiencias con este enfoque. Una fue que no había una forma clara de expresar que nulo podría ser un valor especial. Por el contrario, devolver un `Optional` es una declaración clara en la API de que podría no haber un valor allí. Si quisiéramos asegurarnos de que no obtendremos una excepción de puntero nulo, entonces tendríamos que hacer una verificación nula explícita para cada referencia, como se muestra a continuación, y todos estamos de acuerdo en que es un montón de repeticiones.

```
private void getIsoCode( User user){  
    if (user != null) {  
        Address address = user.getAddress();  
        if (address != null) {  
            Country country = address.getCountry();  
            if (country != null) {  
                String isocode = country.getIsocode();  
                if (isocode != null) {  
                    isocode = isocode.toUpperCase();  
                }  
            }  
        }  
    }  
}
```

Para facilitar este proceso, echemos un vistazo a cómo podemos usar la clase `Optional`, desde crear y verificar una instancia hasta usar los diferentes métodos que proporciona y combinarla con otros métodos que devuelven el mismo tipo, siendo este último el verdadero poder de las mentiras opcionales.

Características de opcional

La clase opcional proporciona alrededor de 10 métodos, que podemos usar para crear y usar la clase `Optional` y veremos cómo se usan a continuación.

Crear un `Optional`

Hay tres métodos de creación para crear una instancia opcional.

Devuelve una instancia opcional vacía.

```
// Creating an empty optional  
Optional<String> empty = Optional.empty();
```

Devuelve una instancia `Optional` vacía. No hay valor presente para este `Optional`. Sin embargo, puede ser tentador hacerlo, evite probar si un objeto está vacío comparándolo con `==` contra las instancias devueltas por `Optional.empty()`. No hay garantía de que sea un singleton. En cambio, use `isPresent()`.

Devuelve un `Optional` con el valor presente no nulo actual especificado.

```
// Creating an optional using of  
String name = "java";  
Optional<String> opt = Optional.of(name);
```

El método estático de espera un argumento no nulo; de lo contrario, arrojará una `NullPointerException`. Entonces, ¿`ofNullable` qué pasa si no sabemos si el argumento será nulo o no? Es entonces cuando lo usamos `ofNullable`, que se describe a continuación.

Devuelve un `Optional` que describe el valor especificado, si no es nulo, de lo contrario, devuelve un `Optional` vacío.

```
// Possible null value  
  
Optional<String> optional = Optional.ofNullable(name());  
  
private String name() {  
    String name = "Java";  
    return (name.length() > 5) ? name : null;  
}
```

Al hacer esto, si pasamos una referencia nula, no arroja una excepción, sino que devuelve un objeto opcional vacío:

Por lo tanto, esos son los tres métodos para crear `Optionals`, ya sea de forma dinámica o manual. El siguiente conjunto de métodos es para verificar la presencia de valor.



Buenas Prácticas



Uso y mejores prácticas

Al igual que cualquier otra característica de un lenguaje de programación, se puede usar correctamente o se puede abusar de él. Para conocer la mejor manera de usar la clase `Optional`, uno debe comprender lo siguiente:

Lo que está tratando de resolver

`Optional` es un intento de reducir el número de excepciones del tipo `nullpointer` en los sistemas Java, al agregar la posibilidad de construir API más expresivas que tengan en cuenta la posibilidad de que a veces falten valores de retorno.

Si `Optional` estuvo allí desde el principio, la mayoría de las bibliotecas y aplicaciones probablemente tratarían mejor los valores de retorno faltantes, reduciendo el número de excepciones de puntero nulo y el número general de errores en general.

Lo que NO está tratando de resolver

`Optional` no pretende ser un mecanismo para evitar todo tipo de punteros nulos. Los parámetros de entrada obligatorios de métodos y constructores aún deben probarse, por ejemplo.

Al igual que cuando se usa nulo, `Optional` no ayuda a transmitir el significado de un valor ausente. De manera similar, nulo puede significar muchas cosas diferentes (valor no encontrado, etc.), así como un valor opcional ausente.

La persona que llama del método todavía tendrá que verificar el javadoc del método para comprender el significado de la `Optional` ausente, para tratarlo adecuadamente.

Además, de manera similar a que una excepción marcada puede ser atrapada en un bloque vacío, nada impide que la persona que llama llame `get()` y siga adelante.

Cuándo usarlo

El uso previsto de `Optional` es principalmente como un tipo de retorno . Después de obtener una instancia de este tipo, puede extraer el valor si está presente o proporcionar un comportamiento alternativo si no lo está.

Un caso de uso muy útil de la clase `Optional` es combinarlo con secuencias u otros métodos que devuelvan un valor `Optional` para construir API fluidas.

```
User user = users.stream().findFirst().orElse( new User("default", "1234"));
```

Cuándo no usarlo

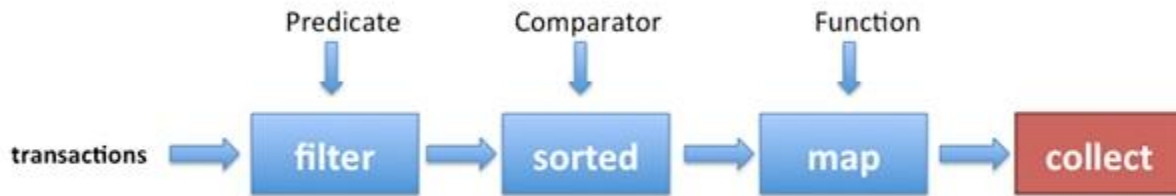
- No lo use como campo en una clase ya que no es serializable. Si necesita serializar un objeto que contiene un valor `Optional`, la biblioteca Jackson brinda soporte para tratar los `Optionals` como objetos ordinarios. Lo que esto significa es que Jackson trata los objetos vacíos como nulos y los objetos con un valor como campos que contienen ese valor.
- No lo use como parámetro para constructores y métodos, ya que conduciría a un código innecesariamente complicado.

```
User user = new User("john@gmail.com", "1234", Optional.empty());
```

Colecciones, streams y filtros

Java Collections - Streams

Java 8 Streams es una nueva adición a la API de colecciones de Java, que brinda una nueva forma de procesar colecciones de objetos. Por lo tanto, los streams en la API de colecciones de Java son un concepto diferente que las secuencias de entrada y salida en la API de IO de Java, incluso si la idea es similar (una secuencia de objetos de una colección, en lugar de una secuencia de bytes o caracteres). Los flujos están diseñados para trabajar con **Java lambda expressions**.



Consignas

Las Streams funcionan perfectamente con lambdas.

Todas las operaciones de flujos toman interfaces funcionales como argumentos, por lo que puede simplificar el código con expresiones lambda (y referencias de métodos).

Las Streams no almacenan sus elementos.

Los elementos se almacenan en una colección o se generan sobre la marcha. Solo se transportan desde la fuente a través de una tubería de operaciones.

Los Streams son inmutables.

Las secuencias no mutan su fuente subyacente de elementos. Si, por ejemplo, un elemento se elimina de la secuencia, se crea una nueva secuencia con este elemento eliminado.

Los Streams no son reutilizables.

Las corrientes se pueden atravesar solo una vez. Después de ejecutar una operación de terminal (veremos lo que esto significa en un momento), debe crear otra secuencia desde la fuente para procesarla aún más.

Los Streams no admiten el acceso indexado a sus elementos.

Nuevamente, las transmisiones no son colecciones o matrices. Lo máximo que puedes hacer es obtener su primer elemento.

Las Streams son fácilmente paralelizables.

Con la llamada de un método (y siguiendo ciertas reglas), puede hacer que una secuencia ejecute sus operaciones simultáneamente, sin tener que escribir ningún código de subprocesamiento múltiple.

Las operaciones de flujo son perezosas cuando es posible.

Las secuencias difieren la ejecución de sus operaciones hasta que se necesiten los resultados o hasta que se sepa cuántos datos se necesitan.

Una cosa que permite esta pereza es la forma en que se diseñan sus operaciones. La mayoría de ellos devuelve una nueva secuencia, lo que permite que las operaciones se encadenen y formen una canalización que permita este tipo de optimizaciones.

Operaciones comunes en los Streams

En Java 8 puede obtener fácilmente una secuencia de cualquier colección llamando al método `stream()`. Después de eso, hay un par de funciones fundamentales que encontrarás todo el tiempo.

El **filtro** devuelve una nueva secuencia que contiene algunos de los elementos del original. Acepta el predicado para calcular qué elementos se deben devolver en la nueva secuencia y elimina el resto. En el código imperativo, emplearíamos la lógica condicional para especificar qué debería suceder si un elemento satisface la condición. En el estilo funcional no nos molestamos con ifs, filtramos los streams y trabajamos solo en los valores que requerimos.

El **map** transforma los elementos de la secuencia en otra cosa, acepta una función para aplicar a todos y cada uno de los elementos de la secuencia y devuelve una secuencia de los valores que produjo la función del parámetro. Este es el pan y la mantequilla del API de secuencias, el mapa le permite realizar un cálculo sobre los datos dentro de una secuencia.

La operación **reduce** realiza una reducción de la secuencia a un solo elemento. Desea sumar todos los valores enteros en la secuencia.

El método **collect** es la manera de salir del pipeline de los streams y obtener una colección concreta de valores, como una lista en el ejemplo anterior.

Fases de procesamiento de Streams

Una vez que haya obtenido una instancia de Stream de una instancia de collection, use esa secuencia para procesar los elementos en la colección.

El procesamiento de los elementos en la secuencia ocurre en dos pasos / fases:

1. Configuración
2. Procesamiento

Primero, el stream está configurada, esta puede consistir en filtros y mapeos.

En segundo lugar, la secuencia se procesa, que consiste en hacer algo con los objetos filtrados y mapeados. Ningún procesamiento tiene lugar durante la configuración de llamadas. No hasta que se invoca un método de procesamiento.

Stream Filter

Diversas operaciones pueden usarse para filtrar elementos de un stream:

- **filter(Predicate)**: Toma un predicado (`java.util.function.Predicate`) como argumento y devuelve un stream que incluye todos los elementos que coinciden con el predicado indicado.
- **distinct**: Devuelve un stream con elementos únicos (según sea la implementación de `equals` para un elemento del stream).
- **limit(n)**: Devuelve un stream cuya máxima longitud es `n`.
- **skip(n)**: Devuelve un stream en el que se han descartado los primeros `n` números.

Stream Map

Los streams admiten el método `map`, que emplea una función (`java.util.function.Function`) como argumento para proyectar los elementos del stream en otro formato. La función se aplica a cada elemento, que se "mapeada" o asocia con un nuevo elemento.

Stream Reduce

Otra posibilidad es combinar todos los elementos de un stream para formular consultas de procesos más complicadas, como "calcular la suma de los valores de todas las transacciones". Para ello, se puede usar la operación reduce con streams; esta operación aplica reiteradamente una operación como la suma a cada elemento hasta que se genera un resultado.

En el ámbito de la programación funcional se la suele llamar operación fold (de pliegue) porque se asimila a la acción de plegar repetidamente un largo trozo de papel (el stream) hasta que queda un pequeño cuadrado, el resultado de la operación de pliegue.

Es útil pensar primero cómo podríamos calcular la suma de los elementos de una lista con un bucle

```
int sum = 0;
for (int x : numbers) { sum += x; }
```

Empleando el método reduce con un stream, podemos sumar todos los elementos de un stream. El método reduce lleva dos argumentos.

```
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
```

O mejor aún, mediante un operador

```
int product = numbers.stream().reduce(1, Integer::max);
```

Stream Collect

El método `collect()` es uno de los métodos de procesamiento de flujo en la interfaz de `Stream`. Cuando se invoca este método, se realizará el filtrado y la asignación, y se recopilará el objeto resultante de esas acciones.

Stream.min() and Stream.max()

Los métodos `min ()` y `max ()` son métodos de procesamiento de flujo. Una vez que se invoquen, la secuencia se repetirá, se aplicará el filtrado y la asignación, y se devolverá el valor mínimo o máximo de la secuencia.

```
String shortest = items.stream()
    .min(Comparator.comparing(item -> item.length()))
    .get();
```

Stream.count()

El método `count ()` simplemente devuelve la cantidad de elementos en la secuencia después de aplicar el filtro.

```
long count = items.stream()
    .filter( item -> item.startsWith("t"))
    .count();
```

Streams Numéricos

Java SE 8 incorpora tres interfaces que transforman streams primitivos en especializados para abordar ese problema: `IntStream`, `DoubleStream` y `LongStream`; cada una de ellas convierte los elementos de un stream de manera especializada para que sean de tipo `int`, `double` o `long`, respectivamente.

```
IntStream oddNumbers = IntStream.rangeClosed(10, 30)
                                .filter(n -> n % 2 == 1);
```

Los métodos más habituales para convertir un stream en una versión especializada son `mapToInt`, `mapToDouble` y `mapToLong`.

Estos métodos funcionan exactamente igual que el método `map` que vimos anteriormente, pero devuelve un stream especializado en lugar de un `Stream<T>`.

Operaciones intermedias y terminales

Una de las virtudes de los streams es que son evaluadas perezosamente, particularmente las funciones que devuelven una instancia de la secuencia:

`filter`, `map`, se llaman intermedias.

Esto significa que no se evaluarán cuando se especifiquen. En cambio, el cálculo ocurrirá cuando el resultado de esa operación sea necesario.

```
Stream <String> names = people.stream ()  
    .filter (p -> p.getGender () == Género.FEMALE)  
    .map (Person :: getName)  
    .map (String :: toUpperCase);
```

Ninguno de los nombres se recopilará inmediatamente y se convertirá en mayúsculas. Cuando ocurre el cálculo, puede preguntar. Cuando se llama a una operación de terminal. Todas las operaciones que devuelven algo que no sea una secuencia son terminales.

Las operaciones como `forEach`, `collect`, `reduce` son terminales. Esto hace que los `streams` sean particularmente eficientes en el manejo de grandes cantidades de datos.

Además de eso, casi siempre se puede intentar paralelizar el procesamiento del flujo convirtiendo el flujo en un flujo paralelo llamando al método `parallel()`, dependiendo de la naturaleza interna del flujo, puede obtener los beneficios de rendimiento.

Hay riesgos de ejecutar cada operación de flujo en paralelo, porque la mayoría de las implementaciones de flujo utilizan `ForkJoinPool` por defecto para realizar las operaciones en segundo plano. Por lo tanto, puede hacer que el flujo de procesamiento en particular sea un poco más rápido, pero sacrifica el rendimiento de toda la JVM sin siquiera darse cuenta.

Construcción de Streams

Hay varias maneras de crear streams, a partir de una colección, streams numericos de números, pero también es posible crear streams a partir de valores, matrices o archivos. Incluso se puede crear un stream a partir de una función para generar streams infinitos, o limitarlos.

```
Stream<Integer> numbersFromValues = Stream.of(1, 2, 3, 4);  
int[] numbers = {1, 2, 3, 4};  
IntStream numbersFromArray = Arrays.stream(numbers);
```

```
Stream<Integer> numbers = Stream.iterate(0, n -> n + 10);
```

```
Stream<Integer> numbers = Stream.iterate(0, n -> n + 10).limit(5).forEach(System.out::println);
```

Streams vs Colecciones

Tanto la noción de colecciones que ya existía en Java como la nueva noción de streams se refieren a interfaces con secuencias de elementos, sin embargo, las colecciones hacen referencia a datos mientras que los streams hacen referencia a cálculos.

En términos simples, la diferencia entre las colecciones y los streams se relaciona con cuándo se hacen los cálculos. Las colecciones son estructuras de datos que se almacenan en la memoria, donde se encuentran todos los valores que tiene la estructura de datos en un momento dado; cada elemento de la colección debe calcularse antes de que se lo pueda agregar a la (cont)...

colección. En cambio, los streams son estructuras de datos fijas conceptualmente cuyos elementos se computan cuando se recibe la solicitud correspondiente.

Cuando se emplea la interfaz `Collection`, es el usuario quien debe ocuparse de la iteración (por ejemplo, mediante `foreach`); ese enfoque se denomina iteración externa.

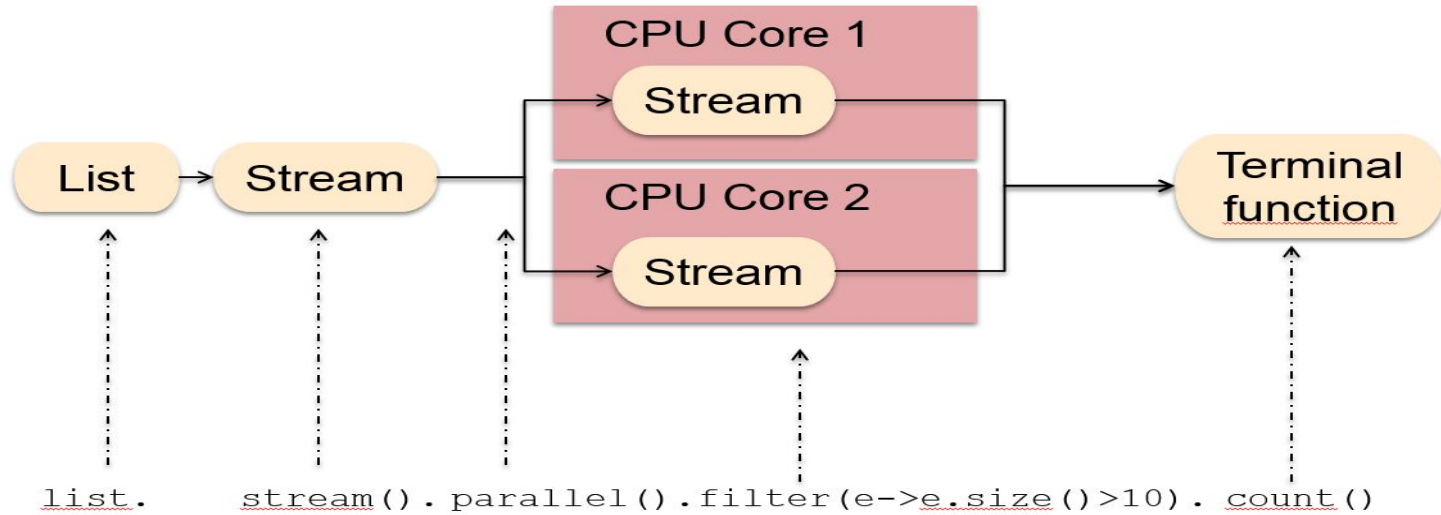
Parallel Streams

Paralelizando

El procesamiento paralelo está por todas partes hoy en día. Debido al aumento de la cantidad de núcleos de CPU y al menor costo de hardware que permite sistemas de clúster más baratos.

Java 8 se preocupa por este hecho con la nueva API de streams y la simplificación de crear un procesamiento paralelo en colecciones y matrices.

Imaginemos que myList es una lista de enteros y que contiene 500,000 valores enteros. La forma de resumir estos valores enteros en la era pre-java 8 se hizo usando un para cada ciclo.



Referencias a Métodos

Referencia de métodos

Es una característica que está relacionada con Lambda Expression. Nos permite hacer referencia a constructores o métodos sin ejecutarlos. Las referencias de métodos y Lambda son similares en que ambos requieren un tipo de destino que consista en una interfaz funcional compatible.

Tipos de Referencia

Tipo	Ejemplo	Sintaxis
1. Referencia a un método estático	<code>ContainingClass::staticMethodName</code>	<code>Class::staticMethodN ame</code>
2. Referencia a un constructor	<code>ClassName::new</code>	<code>ClassName::new</code>
3. Referencia a un método de instancia de un objeto arbitrario de un tipo particular	<code>ContainingType::methodName</code>	<code>Class::instanceMetho dName</code>
4. Referencia a un método de instancia de un objeto particular	<code>containingObject::instanceMethodName</code>	<code>object::instanceMeth odName</code>

Referencia a método estático

```
public class ReferenceToStaticMethodExample {  
    public static void main(String[] args) {  
        List numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 16);  
        List primeNumbers = ReferenceToStaticMethodExample.findPrimeNumbers(numbers,  
            (number) -> ReferenceToStaticMethodExample.isPrime((int) number));  
  
        System.out.println("Prime Numbers are " + primeNumbers);  
    }  
}
```

```
public static boolean isPrime(int number) {  
    if (number == 1) { return false; }  
    for (int i = 2; i < number; i++) { if (number % i == 0) { return false; } }  
    return true;  
}  
  
public static List findPrimeNumbers(List list, Predicate predicate) {  
    List sortedNumbers = new ArrayList(); list.stream().filter((i) ->  
    (predicate.test(i))).forEach((i) -> {  
        sortedNumbers.add(i);  
    });  
    return sortedNumbers;  
}  
}
```


Referencia a un constructor

```
public class ReferenceToConstructor {  
    public static void main(String[] args) {  
        List numbers = Arrays.asList(4,9,16,25,36);  
        List squaredNumbers = ReferenceToConstructor.findSquareRoot(numbers,Integer::new);  
        System.out.println("Square root of numbers = "+squaredNumbers);  
    }  
    private static List findSquareRoot(List list, Function<Integer,Integer> f){  
        List result = new ArrayList();  
        list.forEach(x -> result.add(Math.sqrt(f.apply((Integer) x))));  
        return result;  
    }  
}
```



Referencia a un método de instancia de un objeto arbitrario de un tipo particular

```
public class ReferenceToInstanceMethodAOPT {  
    private static class Person {  
        private final String name;  
        private final int age;  
  
        public Person(String name, int age) {  
            this.name = name;  
            this.age = age;  
        }  
    }  
}
```

```
        public String getName() {  
            return name;  
        }  
  
        public int getAge() {  
            return age;  
        }  
    }  
  
    public static void main(String[] args) {  
        List persons = new ArrayList();  
        persons.add(new Person("Albert", 80));  
        persons.add(new Person("Ben", 15));  
        persons.add(new Person("Charlote", 20));  
        persons.add(new Person("Dean", 6));  
        persons.add(new Person("Elaine", 17));  
    }  
}
```

```
        List allAges = ReferenceToInstanceMethodAOPT.listAllAges(persons, Person::getAge);
        System.out.println("Printing out all ages \n"+allAges);
    }

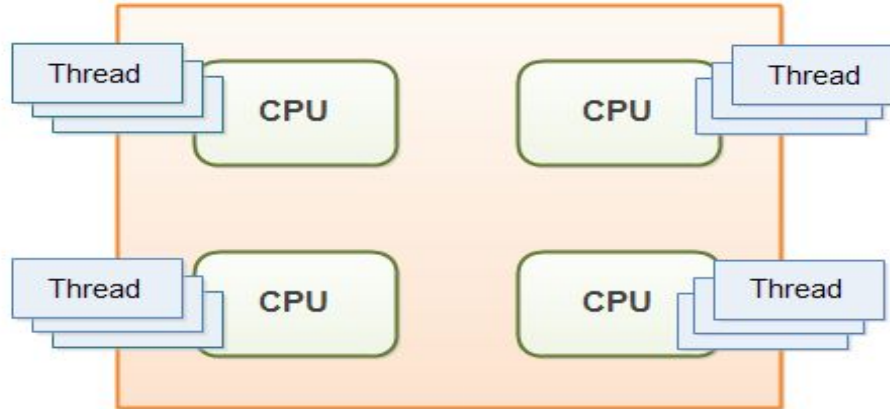
    private static List listAllAges(List person, Function<Person, Integer> f){
        List result = new ArrayList();
        person.forEach(x -> result.add(f.apply((Person)x)));
        return result;
    }
}
```

Referencia a un método de instancia de un objeto particular

```
public class ReferenceToInstanceMethodOAP0 {  
  
    public static void main(String[] args) {  
        List names = new ArrayList();  
        names.add("David");  
        names.add("Richard");  
        names.add("Samuel");  
        names.add("Rose");  
        names.add("John");  
        ReferenceToInstanceMethodOAP0.printNames(names, System.out::println);  
    }  
  
    private static void printNames(List list, Consumer c ){  
        list.forEach(x -> c.accept(x));  
    }  
}
```

Concurrencia

La API de concurrencia se introdujo por primera vez con el lanzamiento de Java 5 y luego se mejoró progresivamente con cada nueva versión de Java. La mayoría de los conceptos que se muestran en este artículo también funcionan en versiones anteriores de Java. Sin embargo, estos ejemplos de código se centran en Java 8 y hacen un uso intensivo de expresiones lambda y otras características nuevas.



Threads y Runnables

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 utilizar hilos para ejecutar código al mismo tiempo, de modo que podamos aprovechar al máximo los núcleos disponibles de la CPU.

Java admite Threads desde JDK 1.0.

Antes de comenzar un nuevo hilo, debe especificar el código que ejecutará este hilo, a menudo denominado tarea. Esto se hace mediante la implementación de Runnable, una interfaz funcional que define un único método void sin argumentos run()

```
Runnable task = () -> {  
    String threadName = Thread.currentThread().getName();  
    System.out.println("Hello " + threadName);  
};  
task.run();  
Thread thread = new Thread(task);  
thread.start();  
System.out.println("Done!");
```

API Concurrent

Los sincronizadores proporcionados en la biblioteca `java.util.concurrent` y sus usos se enumeran aquí:

- Semaphore controla el acceso a uno o más recursos compartidos.
- Phaser se utiliza para soportar una barrera de sincronización.
- CountdownLatch permite que los hilos esperen a que se complete una cuenta regresiva.
- El Exchanger admite el intercambio de datos entre dos hilos.
- CyclicBarrier permite que los hilos esperen en un punto de ejecución predefinido.

Semaphore

Un Semaphore controla el acceso a recursos compartidos. Un semáforo mantiene un contador para especificar el número de recursos que controla el semáforo. Se permite el acceso al recurso si el contador es mayor que cero, mientras que un valor cero del contador indica que no hay recurso disponible en ese momento y, por lo tanto, se deniega el acceso.

CountDownLatch

Este sincronizador permite que uno o más subprocesos esperen a que se complete una cuenta regresiva. Esta cuenta regresiva podría ser para que un conjunto de eventos ocurra o hasta que se complete un conjunto de operaciones que se están realizando en otros subprocesos.

Exchanger

La clase Exchanger es para intercambiar datos entre dos subprocesos. Lo que hace el Intercambiador es algo muy simple: espera hasta que ambos hilos hayan llamado el método `exchange()`.

Cuando ambos hilos han llamado este método, el objeto Intercambiador intercambia realmente los datos compartidos por los hilos entre sí.

Esta clase es útil cuando dos hilos necesitan sincronizarse entre ellos y intercambiar continuamente datos.

CyclicBarrier

Hay muchas situaciones en la programación concurrente donde los hilos pueden necesitar esperar en un punto de ejecución predefinido hasta que todos los otros hilos alcancen ese punto. CyclicBarrier ayuda a proporcionar tal punto de sincronización

Phaser

Phaser es una característica útil cuando pocos hilos independientes tienen que trabajar en fases para completar una tarea.

Por lo tanto, se necesita un punto de sincronización para que los hilos funcionen en una parte de una tarea, espere a que otros completen otra parte de la tarea y realice una sincronización antes de avanzar para completar la siguiente parte de la tarea.

Atomic Variables

Las clases de variables atómicas más utilizadas en Java son `AtomicInteger`, `AtomicLong`, `AtomicBoolean` y `AtomicReference`.

Estas clases representan una referencia `int`, `long`, `boolean` y `object`, respectivamente, que pueden actualizarse atómicamente. Los principales métodos expuestos por estas clases son:

`get ()` - obtiene el valor de la memoria, de modo que los cambios hechos por otros hilos son visibles; equivalente a leer una variable volátil

`set ()` - escribe el valor en la memoria, de modo que el cambio sea visible para otros hilos; equivalente a escribir una variable volátil

`lazySet()`: finalmente escribe el valor en la memoria, puede reordenarse con operaciones de memoria relevantes posteriores. Un caso de uso es anular referencias, por el bien de la recolección de basura, que nunca se volverá a acceder. En este caso, se logra un mejor rendimiento al retrasar la escritura volátil nula

`compareAndSet()`: lo mismo que se describe en la sección 3, devuelve verdadero cuando tiene éxito, de lo contrario es falso

`weakCompareAndSet()` - igual que se describe en la sección 3, pero más débil en el sentido de que no crea lo que sucede antes de los pedidos. Esto significa que no necesariamente ve actualizaciones hechas a otras variables

Locks

El uso de un objeto Lock es similar a la obtención de bloqueos implícitos utilizando la palabra clave synchronized.

El objetivo de ambas construcciones es el mismo: garantizar que sólo un hilo acceda a un recurso compartido a la vez.

Sin embargo, a diferencia de la palabra clave sincronizada, los bloqueos también admiten el mecanismo wait / notify junto con su compatibilidad con objetos Condition.

Condition

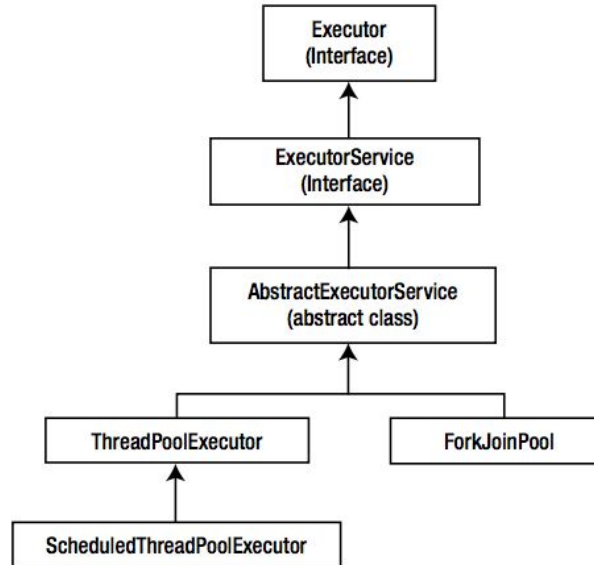
Condition admite el mecanismo de notificación de subprocesos. Cuando una cierta condición no se satisface, un hilo puede esperar para que otro hilo satisfaga esa condición; que otro hilo podrá notificar una vez que se cumple la condición. Una condición está ligada a un bloque. Un objeto Condition ofrece tres métodos para admitir el patrón wait / notify: `await()`, `signal()` y `signalAll()`. que son análogos a los métodos `wait()`, `notify()` y `notifyAll()` soportados por la clase `Object`.

Un hilo puede esperar a que una condición sea verdadera utilizando el método `await()`, que es una llamada de bloqueo interrumpible. Si usted quiere espera no interrumpible, puede llamar `awaitUninterruptibly()`. También puede especificar la duración del `await` utilizando uno de los métodos sobrecargados:

- `long awaitNanos(long nanosTimeout)`
- `boolean await(long time, TimeUnit unit)`
- `boolean awaitUntil(Date deadline)`

Executors and ThreadPools

Executor es una interfaz que declara un solo método: void execute (Runnable). Esto puede no parecer una gran interfaz por sí mismo, pero sus clases derivadas (o interfaces), como ExecutorService, ThreadPoolExecutor y ForkJoinPool, admiten funcionalidad útil.



ExecutorService

API de Concurrencia introduce el concepto de un ExecutorService como un reemplazo de nivel superior para trabajar directamente con los hilos.

Los ejecutores son capaces de ejecutar tareas asíncronas y, por lo general, administrar un conjunto de hilos, por lo que no es necesario crear nuevos hilos manualmente.

En términos generales, ExecutorService proporciona automáticamente un conjunto de hilos y API para asignarle tareas.

Posee los métodos `execute()`, `submit()`, `invokeAny()`, `invokeAll()`, `shutdown()` y `shutdownNow()` que nos permite gestionar el mismo.

```
ExecutorService executorService = new ThreadPoolExecutor(1, 1, 0L,
TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>());
Runnable runnableTask = () -> {
    try {
        TimeUnit.MILLISECONDS.sleep(300);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
};
executorService.execute(runnableTask);
Callable<String> callableTask = () -> {
    TimeUnit.MILLISECONDS.sleep(300);
    return "Task's execution";
};

Future<String> future = executorService.submit(callableTask);
List<Callable<String>> callableTasks = new ArrayList<>();
callableTasks.add(callableTask);
callableTasks.add(callableTask);
callableTasks.add(callableTask);
```

The Fork-Join Framework

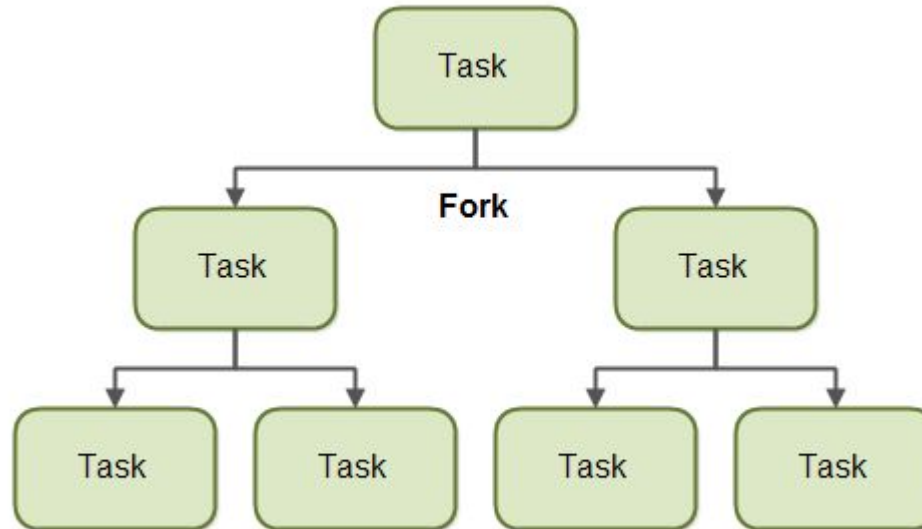
The ForkJoinPool se agregó a Java en Java 7, es similar al Java ExecutorService pero con una diferencia. El ForkJoinPool facilita que las tareas dividan su trabajo en tareas más pequeñas que luego se envían al ForkJoinPool también.

Las tareas pueden seguir dividiendo su trabajo en subtareas más pequeñas durante el tiempo que sea necesario.

El principio de fork and join consta de dos pasos que se realizan recursivamente, que son el paso de fork y el paso de join.

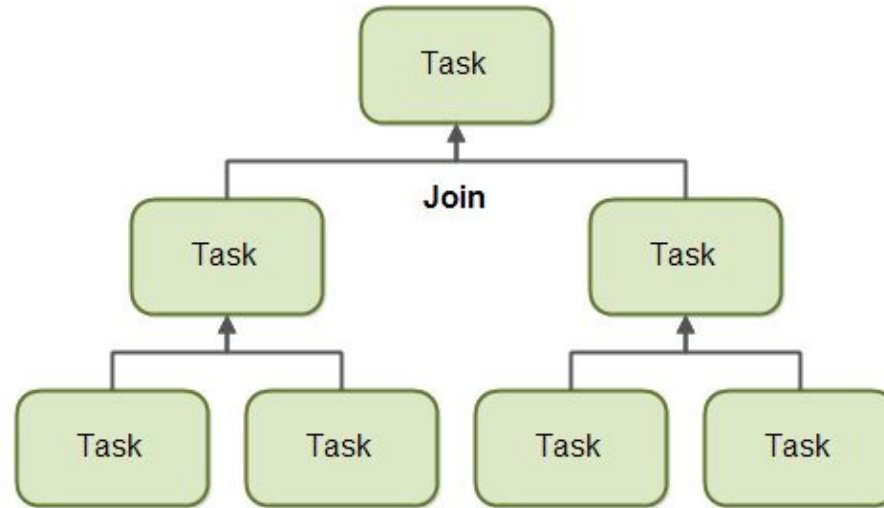
Fork

Una tarea que utiliza el principio de fork y join puede bifurcarse (dividirse) en subtareas más pequeñas que se pueden ejecutar al mismo tiempo. Esto se ilustra en el siguiente diagrama:



Join

Cuando una tarea se ha dividido en subtareas, la tarea espera hasta que las subtareas hayan terminado de ejecutarse, luego la tarea puede unir (fusionar) todos los resultados en un resultado. Esto se ilustra en el siguiente diagrama:



API Java Date/Time

API Java Date/Time

En Java 8, se agregó una nueva API de fecha y hora completa. La nueva API de fecha y hora de Java se encuentra en el paquete `java.time`, que es parte de la biblioteca de clases Java 8 estándar.

El principal cambio en Java 8 date time API es que la fecha y la hora ya no están representadas por una única cantidad de milisegundos desde el 1 de enero de 1970, sino por el número de segundos y nanosegundos desde el 1 de enero de 1970. La cantidad de segundos puede ser positivo y negativo y está representado por un `long`. El número de nanosegundos siempre es positivo y está representado por un `int`. Verá esta nueva representación de fecha y hora en muchas de las clases en la nueva API de fecha y hora de Java.

El paquete `java.time` contiene un conjunto de subpaquetes con más utilidades, etc. Por ejemplo, `java.time.chrono` contiene clases para trabajar con calendarios japoneses, taiwaneses, taiwaneses e islámicos. El paquete `java.time.format` contiene clases utilizadas para analizar y formatear fechas de y para cadenas.

consta de las siguientes clases:

Instant	Representa un instante en el tiempo en la línea de tiempo. En la API de fecha y hora de Java 7, un instante estuvo típicamente representado por una cantidad de milisegundos desde el 1 de enero. 1970. En Java 8, la clase Instant representa un instante en el tiempo representado por una cantidad de segundos y una cantidad de nanosegundos desde el 1 de enero de 1970.
Duration	Representa un período de tiempo, por ejemplo, el tiempo entre dos instantes. Al igual que la clase Instantánea, una Duración representa su tiempo como una cantidad de segundos y nanosegundos.
LocalDate	Representa una fecha sin información de zona horaria, p. un cumpleaños, vacaciones oficiales, etc.
LocalDateTime	Representa una fecha y hora sin información de zona horaria
LocalTime	Representa una hora local del día sin información de zona horaria.

<code>ZonedDateTime</code>	Representa una fecha y hora que incluye información de zona horaria
<code>Period</code>	Tiempo según calendario ISO-8601.
<code>DateTimeFormatter</code>	Formatea objetos de fecha y hora como cadenas. Por ejemplo, un <code>ZonedDateTime</code> o un <code>LocalDateTime</code> .
<code>TemporalAdjuster</code>	Permiten externalizar el proceso de ajuste, lo que permite diferentes enfoques, según el patrón de diseño de la estrategia. Los ejemplos pueden ser un ajustador que establece la fecha evitando los fines de semana, o uno que establece la fecha para el último día del mes.

Instant Calculations

La clase Instant también tiene varios métodos que se pueden usar para hacer cálculos relativos a un Instantáneo. Algunos (no todos) de estos métodos son:

- `plusSeconds()`
- `plusMillis()`
- `plusNanos()`
- `minusSeconds()`
- `minusMillis()`
- `minusNanos()`

```
Instant now      = Instant.now();  
  
Instant later    = now.plusSeconds(3);  
Instant earlier = now.minusSeconds(3);
```

```
Instant now      = Instant.now();  
Instant later    = now.plusSeconds(3);  
Instant earlier = now.minusSeconds(3);
```

Java File I/O (NIO.2)

Java File I/O (NIO.2)

- Use la clase Path para operar en rutas de archivos y directorios.
- Use la clase Files para verificar, eliminar, copiar o mover un archivo o directorio.
- Leer y cambiar los atributos de archivo y directorio.
- Accede recursivamente a un árbol de directorios.
- Encuentre un archivo utilizando la clase PathMatcher.
- Mire un directorio para ver los cambios utilizando WatchService.

Java File I/O (NIO.2) :: Fundamentos

Java ofreció la clase `java.io.File` para acceder a los sistemas de archivos. Esta clase representa un archivo/directorio en el sistema de archivos y le permite realizar operaciones como comprobar la existencia, obtener las propiedades así como eliminarlo. Sin embargo, la primera versión de la API no fue suficiente para satisfacer las necesidades de los desarrolladores, y se tuvo la necesidad de mejorar las API de E/S.

Entre las carencias podemos nombrar:

- La clase `File` carecía de la funcionalidad significativa requerida para implementar incluso la funcionalidad comúnmente utilizada. Por ejemplo, le faltaba un método de copia para copiar un archivo / directorio.
- Muchos métodos solo devuelven un valor booleano, en caso de un error, se devuelve `false`, en lugar de lanzar una excepción, por lo que el desarrollador no tenía forma de saber por qué falló la llamada.
- No proporciona un buen soporte para el manejo de enlaces simbólicos.
- Manejo ineficiente de directorios y rutas.
- Acceso ilimitado e insuficiente de atributos de archivo.

Java File I/O (NIO.2) :: Fundamentos

Java introdujo NIO (New IO) en Java 4 y las principales características fueron:

Channels y selectors:

Un canal es una abstracción sobre las características del sistema de archivos de nivel inferior (como los archivos mapeados en memoria y el bloqueo de archivos) que le permiten transferir datos a una velocidad más rápida. Los canales no bloquean, por lo que Java proporciona otra característica: un selector para seleccionar un canal listo para la transferencia de datos. Un socket es una característica de bloqueo mientras que un canal es una característica no bloqueante.

Buffers:

Java 4 introdujo el almacenamiento en búfer para todas las clases primitivas (excepto Boolean). Proporcionó la clase Buffer que ofrece operaciones como clear, flip, mark, reset y rewind. Las clases concretas (subclases de la clase base Buffer) ofrecen getters y setters para establecer y obtener datos desde y hacia un búfer.

Java File I/O (NIO.2) :: Fundamentos

Charset:

Java 4 también introdujo charset (`java.nio.charset`), codificadores y decodificadores para asignar bytes y símbolos Unicode.

Con la versión SE 7, Java ha introducido un soporte completo para las operaciones de E / S.

Java 7 introduce el paquete `java.nio.file` para un mejor soporte en el manejo de enlaces simbólicos, para proporcionar acceso completo a atributos y para soportar el sistema de archivos extendido a través de interfaces o clases como `Path`, `Paths` y `Files`.

Java File I/O (NIO.2) :: Uso de Path

Java 7 introduce una nueva abstracción de programación para la ruta, es decir, la interfaz de ruta. Esta abstracción Path se utiliza en nuevas características en NIO.2.

Un objeto Path contiene los nombres de los directorios y archivos que hacen que la ruta completa del archivo / directorio sea representada por el objeto Path; la abstracción proporciona métodos para extraer elementos de ruta, manipularlos y anexarlos.

<code>Path getRoot()</code>	Devuelve un objeto Path que representa la raíz de la ruta dada, o null si la ruta no tiene una raíz.
<code>Path getFileName()</code>	Devuelve el nombre de archivo o directorio de la ruta de acceso dada. Tenga en cuenta que el nombre de archivo / directorio es el último elemento o nombre en la ruta de acceso dada.
<code>Path getParent()</code>	Devuelve el objeto Path que representa el padre de la ruta de acceso dada, o null si no existe un componente padre para la ruta de acceso.
<code>int getNameCount()</code>	Devuelve el número de nombres de archivo / directorio en la ruta de acceso dada; devuelve 0 si la ruta dada representa la raíz.

Java File I/O (NIO.2) :: Uso de Path (cont)

<code>Path getName(int i)</code>	Devuelve el i-ésimo archivo / nombre del directorio; el índice 0 comienza desde el nombre más cercano hasta la raíz.
<code>Path subpath(int beginIndex, int endIndex)</code>	Devuelve un objeto Path que forma parte de este objeto Path; el objeto Path devuelto tiene un nombre que comienza en beginIndex hasta el elemento en el índice endIndex - 1. Este método puede arrojar IllegalArgumentException si beginIndex es > número de elementos, o endIndex <= beginIndex, o endIndex es > número de elementos.
<code>Path normalize()</code>	Elimina elementos redundantes en la ruta tal como. (símbolo de punto que indica el directorio actual) y .. (símbolo de doble punto que indica el directorio padre).
<code>Path resolve(Path other)</code> <code>Path resolve(String other)</code>	Resuelve una ruta de acceso en la ruta dada. Por ejemplo, este método podría combinar la ruta dada con la otra ruta y devolver la ruta resultante.
<code>Boolean isAbsolute()</code>	Devuelve true si la ruta dada es una ruta absoluta; devuelve false si no (cuando la ruta dada es una ruta relativa, por ejemplo).
<code>Path startsWith(String path)</code> <code>Path startsWith(Path path)</code>	Devuelve true si este objeto Path comienza con la ruta dada, o bien devuelve false.
<code>Path toAbsolutePath()</code>	Devuelve la ruta absoluta.

Java File I/O (NIO.2) :: Path examples

Ejemplo de uso del Path para comparar 2 destinos

```
class PathCompare1 {  
    public static void main(String[] args) {  
        Path path1 = Paths.get("Test");  
        Path path2 = Paths.get("C:\\\\TEST\\\\NI02\\\\");  
        System.out.println("(path1.compareTo(path2) == 0) is:"  
            + (path1.compareTo(path2) == 0));  
        System.out.println("path1.equals(path2) is: " + path1.equals(path2));  
        System.out.println("path2.equals(path1.toAbsolutePath()) is " +  
            path2.equals(path1.toAbsolutePath()));  
    }  
}
```

Java File I/O (NIO.2) :: Uso del File

Desde java 7, se ofrece una nueva clase `java.nio.file.Files` que puede utilizar para realizar varias operaciones relacionadas con archivos en archivos o directorios. Tenga en cuenta que `Files` es una clase de utilidad, lo que significa que es una clase final con un constructor privado y que consiste sólo en métodos estáticos. Así que puede hacer uso de la clase `Files` llamando a los métodos estáticos que proporciona, como `copy()` para copiar archivos.

Esta clase ofrece una amplia gama de funcionalidades. Con esta clase puede crear directorios, archivos o enlaces simbólicos; crear flujos tales como flujos de directorio, canales de bytes o flujos de entrada / salida; examinar los atributos de los archivos; recorrer el árbol de archivos; o realizar operaciones de archivo como leer, escribir, copiar o eliminar.

```
Path createDirectory(Path dirPath, FileAttribute<?>...  
dirAttrs)  
Path createDirectories(Path dir, FileAttribute<?>...  
attrs)
```

Crea un archivo dado por el `dirPath`, y establece los atributos dados por `dirAttributes`. Puede lanzar excepciones como `FileAlreadyExistsException` o `UnsupportedOperationException` (por ejemplo, cuando el los atributos de archivo no se pueden establecer como dados por `dirAttrs`). La diferencia entre `createDirectory` y `createDirectories` es que `createDirectories` crea directorios intermedios dados por `dirPath` si no están presentes.

```
Path createTempFile(Path dir, String prefix, String  
suffix, FileAttribute<?>... attrs)
```

Crea un archivo temporal con prefijo, sufijo y atributos dados en el directorio dado por `dir`.

Java File I/O (NIO.2) :: Uso del File

<code>Path createTempDirectory(Path dir, String prefix, FileAttribute<?>... attrs)</code>	Crea un directorio temporal con el prefijo dado, atributos de directorio en la ruta especificada por dir.
<code>Path copy(Path source, Path target, CopyOption... options)</code>	Copie el archivo de origen a destino. CopyOption podría ser REPLACE_EXISTING, COPY_ATTRIBUTES o NOFOLLOW_LINKS. Puede lanzar excepciones como FileAlreadyExistsException.
<code>Path move(Path source, Path target, CopyOption... options)</code>	Similar a la operación de copia excepto que se quita el archivo de origen; si el origen y el destino están en el mismo directorio, es una operación de cambio de nombre de archivo.
<code>boolean isSameFile(Path path, Path path2)</code>	Comprueba si los dos objetos Path están ubicados en el mismo archivo o no.
<code>boolean exists(Path path, LinkOption... options)</code>	Comprueba si existe un archivo / directorio en la ruta dada; puede especificar LinkOption.NOFOLLOW_LINKS para no seguir enlaces simbólicos.
<code>Boolean isRegularFile(Path path, LinkOption. . .)</code>	Devuelve true si el archivo representado por path es un archivo regular.
<code>Boolean isSymbolicLink(Path path)</code>	Devuelve true si el archivo presentado por path es un enlace simbólico.

Java File I/O (NIO.2) :: Uso del File

<code>Boolean isHidden(Path path)</code>	Devuelve true si el archivo representado por path es un archivo oculto.
<code>long size(Path path)</code>	Devuelve el tamaño del archivo en bytes representados por path.
<code>UserPrincipal getOwner(Path path, LinkOption. . .),</code> <code>Path setOwner(Path path, UserPrincipal owner)</code>	Obtiene / establece el propietario del archivo.
<code>FileTime getLastModifiedTime(Path path, LinkOption...)</code> <code>Path setLastModifiedTime(Path path, FileTime time)</code>	Obtiene / establece la última hora modificada para la hora especificada.
<code>Object getAttribute(Path path, String attribute,</code> <code>LinkOption...)</code> <code>Path setAttribute(Path path, String attribute, Object</code> <code>value, LinkOption...)</code>	Obtiene / establece el atributo especificado del archivo especificado.

Java File I/O (NIO.2) :: Recorriendo ficheros

El API provee de utilidades para recorrer los árboles de ficheros, para ello la clase Files posee 2 métodos

```
Path walkFileTree(Path start, FileVisitor<? super Path> visitor)
Path walkFileTree(Path start, Set<FileVisitOption> options, int maxDepth, FileVisitor<? super Path> visitor)
```

Ambos métodos poseen una instancia de FileVisitor, el cual controla que hacer mientras recorremos el árbol.

<code>FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)</code>	Se invoca justo antes de acceder a los elementos del directorio.
<code>FileVisitResult visitFile(T file, BasicFileAttributes attrs)</code>	Se invoca cuando se visita un archivo.
<code>FileVisitResult postVisitDirectory(T dir, IOException exc)</code>	Se invoca cuando se accede a todos los elementos del directorio.
<code>FileVisitResult visitFileFailed(T file, IOException exc)</code>	Se invoca cuando no se puede acceder al archivo.

Java File I/O (NIO.2) :: Recorriendo ficheros

La clase `java.io.File` tiene métodos `list()` y `listFiles()`. Ellos pueden ser usados para instancias de `File` que representan una carpeta, y regresan una lista de archivos y carpetas que son “hijos directos” (que están dentro de la carpeta).

Un mecanismo un poco más poderoso es provisto con la clase `java.nio.file.Files`, como lo son los visitantes de archivos y flujos de carpeta, los cuales pueden realizar filtros selectivos y combinaciones (matching) para elementos descubiertos.

Java File I/O (NIO.2) :: Recorriendo ficheros

```
class MyFileVisitor extends SimpleFileVisitor<Path> {
    public FileVisitResult visitFile(Path path, BasicFileAttributes fileAttributes){
        System.out.println("file name:" + path.getFileName());
        return FileVisitResult.CONTINUE;
    }
    public FileVisitResult preVisitDirectory(Path path, BasicFileAttributes fileAttributes){
        System.out.println("-----Directory name:" + path + "-----");
        return FileVisitResult.CONTINUE;
    }
}

public class FileTreeWalk {
    public static void main(String[] args) {
        if(args.length != 1) {
            System.out.println("usage: FileWalkTree <source-path>");
            System.exit(-1);
        }

        Path pathSource = Paths.get(args[0]);

        try {
            Files.walkFileTree(pathSource, new MyFileVisitor());
        } catch (IOException e) { e.printStackTrace(); }
    }
}
```

Java File I/O (NIO.2) :: Revisando ficheros (ej 2)

Ejemplo 2 de uso de la clase FileVisitResult

```
class MyFileCopyVisitor extends SimpleFileVisitor<Path> {
    private Path source, destination;
    public MyFileCopyVisitor(Path s, Path d) {
        source = s;
        destination = d;
    }

    public FileVisitResult visitFile(Path path, BasicFileAttributes fileAttributes) {
        Path newd = destination.resolve(source.relativeTo(path));
        try {
            Files.copy(path, newd, StandardCopyOption.REPLACE_EXISTING);
        } catch (IOException e) {e.printStackTrace(); }

        return FileVisitResult.CONTINUE;
    }

    public FileVisitResult preVisitDirectory(Path path, BasicFileAttributes fileAttributes) {
        Path newd = destination.resolve(source.relativeTo(path));
        try {
            Files.copy(path, newd, StandardCopyOption.REPLACE_EXISTING);
        } catch (IOException e) {e.printStackTrace(); }
        return FileVisitResult.CONTINUE;
    }
}
```

Java File I/O (NIO.2) :: Revisando ficheros (cont)

```
public class FileTreeWalkCopy {  
    public static void main(String[] args) {  
        if(args.length != 2) {  
            System.out.println("usage: FileTreeWalkCopy <source-path> <destination-path>");  
            System.exit(1);  
        }  
        Path pathSource = Paths.get(args[0]);  
        Path pathDestination = Paths.get(args[1]);  
        try {  
            Files.walkFileTree(pathSource, new MyFileCopyVisitor(pathSource,  
pathDestination));  
            System.out.println("Files copied successfully!");  
        } catch (IOException e) { e.printStackTrace();}  
    }  
}
```

Java File I/O (NIO.2) :: Buscando ficheros

- Una vez que entienda cómo recorrer el árbol de archivos, es muy sencillo y fácil encontrar un archivo deseado. Por ejemplo, si está buscando un archivo / directorio en particular, puede intentar coincidir con el nombre del archivo / directorio que está buscando con el método `visitFile()` o `preVisitDirectory()`.
- Sin embargo, si está buscando todos los archivos que coincidan un patrón particular (por ejemplo, todos los archivos fuente de Java o archivos xml) en un árbol de archivos, puede usar glob o regex para nombres de archivos. La interfaz `PathMatcher` es útil en este contexto ya que coincidirá con una ruta de acceso para usted una vez que haya especificado el patrón deseado.
- La interfaz `PathMatcher` se implementa para cada sistema de archivos, y puede obtener una instancia de la clase `FileSystem` utilizando el método `getPathMatcher()`.

Java File I/O (NIO.2) :: Buscando ficheros

Patrones (patterns)

*	Corresponde a cualquier cadena de cualquier longitud, incluso de longitud cero.
**	Similar a "*", pero cruza los límites del directorio.
?	Encuentra cualquier carácter
[xyz]	Encuentra a x, y, o z.
[0-5]	Encuentra cualquier carácter en el rango de 0 a 5.
[a-z]	Encuentra a cualquier letra minúscula.
{xyz,abc}	Encuentra xyz o abc.

Java File I/O (NIO.2) :: Buscando ficheros (ejemplo)

Ejemplo de uso de la clase FileVisitResult con pattern

```
class MyFileFindVisitor extends SimpleFileVisitor<Path> {
    private PathMatcher matcher;
    public MyFileFindVisitor(String pattern){
        try {
            matcher = FileSystems.getDefault().getPathMatcher(pattern);
        } catch (IllegalArgumentException iae) {
            System.err.println("Invalid pattern; did you forget to prefix \"glob:\"?(as in glob:*.java)");
            System.exit(-1);
        }
    }
    public FileVisitResult visitFile(Path path, BasicFileAttributes fileAttributes){
        find(path);
        return FileVisitResult.CONTINUE;
    }
    private void find(Path path) {
        Path name = path.getFileName();
        if(matcher.matches(name))
            System.out.println("Matching file:" + path.getFileName());
    }
    public FileVisitResult preVisitDirectory(Path path, BasicFileAttributes fileAttributes){
        find(path);
        return FileVisitResult.CONTINUE;
    }
}
```

Java File I/O (NIO.2) :: Buscando ficheros (ejemplo)

```
public class FileTreeWalkFind {  
    public static void main(String[] args) {  
        if(args.length != 2){  
            System.out.println("usage: FileTreeWalkFind <start-path> <pattern to search>");  
            System.exit(-1);  
        }  
        Path startPath = Paths.get(args[0]);  
        String pattern = args[1];  
        try {  
            Files.walkFileTree(startPath, new MyFileFindVisitor(pattern));  
            System.out.println("File search completed!");  
        } catch (IOException e) {e.printStackTrace(); }  
    }  
}
```

Java File I/O (NIO.2) :: Observando cambios

- Java ofrece un servicio de visualización de directorios que puede lograr detectar cambios en los ficheros o directorios. Puede registrar un directorio utilizando este servicio para cambiar la notificación de eventos y cada vez que ocurra algún cambio en el directorio (como un nuevo archivo creación, eliminación de archivos y modificación de archivos) obtendrá una notificación de evento sobre el cambio.
- El servicio es conveniente, escalable para realizar de una manera fácil seguimiento de los cambios en un directorio.

Java File I/O (NIO.2) :: Observando cambios

```
Path path = Paths.get("./out");
WatchService watchService = path.getFileSystem().newWatchService();
path.register(watchService, StandardWatchEventKinds.ENTRY_MODIFY,
StandardWatchEventKinds.ENTRY_DELETE, StandardWatchEventKinds.ENTRY_CREATE);
for ( ; ; ) { // bucle infinito
    WatchKey key = watchService.take();
    // iterate for each event
    for (WatchEvent<?> event : key.pollEvents()) {
        switch (event.kind().name()) {
            case "OVERFLOW":
                System.out.println("We lost some events");
                break;
            case "ENTRY_MODIFY":
            case "ENTRY_DELETE":
            case "ENTRY_CREATE":
                System.out.println("Event " +event.kind().name()+" on File " +
event.context());
                break;
            default:
                System.err.println("Unknown event kind "+event.kind().name());
        }
    }
    key.reset();
}
```

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 forma más sencilla de solicitar la recolección de basura es

```
System.gc();
```

No hay garantía de que JVM liberará la memoria.

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.

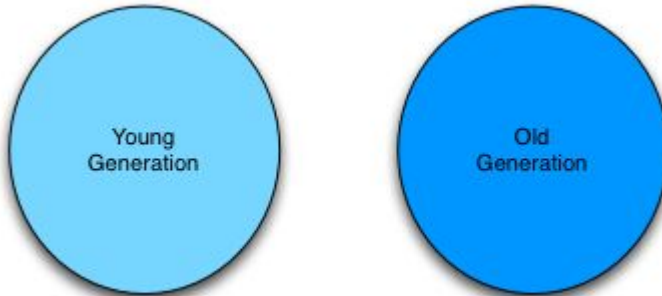
Cada clase hereda el método `finalize()` de `Object`.

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

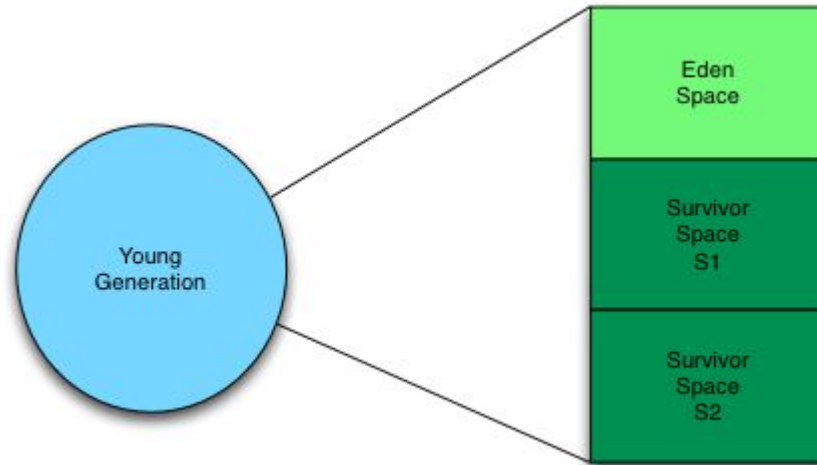
El Java Garbage Collector es uno de los conceptos que más cuesta entender a la gente cuando empieza a programar en Java. ¿Cómo funciona el Java Garbage Collector exactamente?. Java divide la memoria en dos bloques fundamentales , Young generation y Old generation.



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)

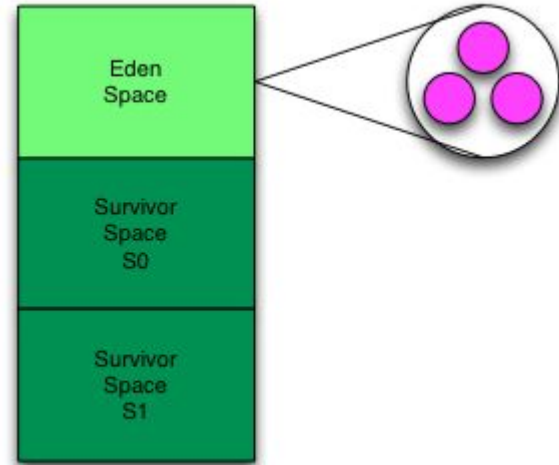
java Garbage Collector Eden



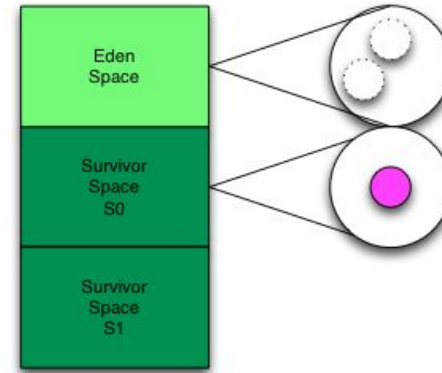
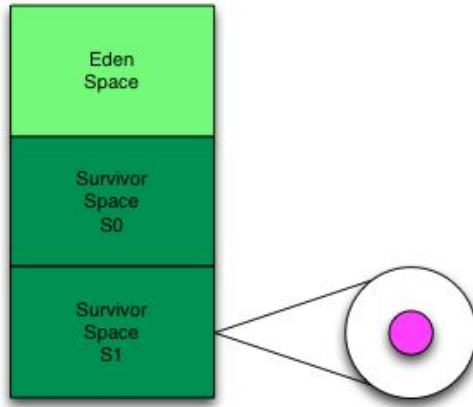
Eden

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.

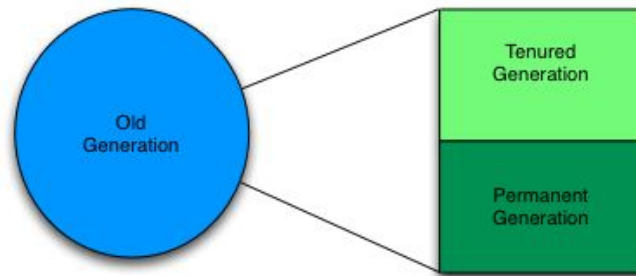


El recolector de basura volverá a pasar otra vez y si queda algún objeto superviviente en el Survivor Space S0 lo moverá al Survivor S1

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 .

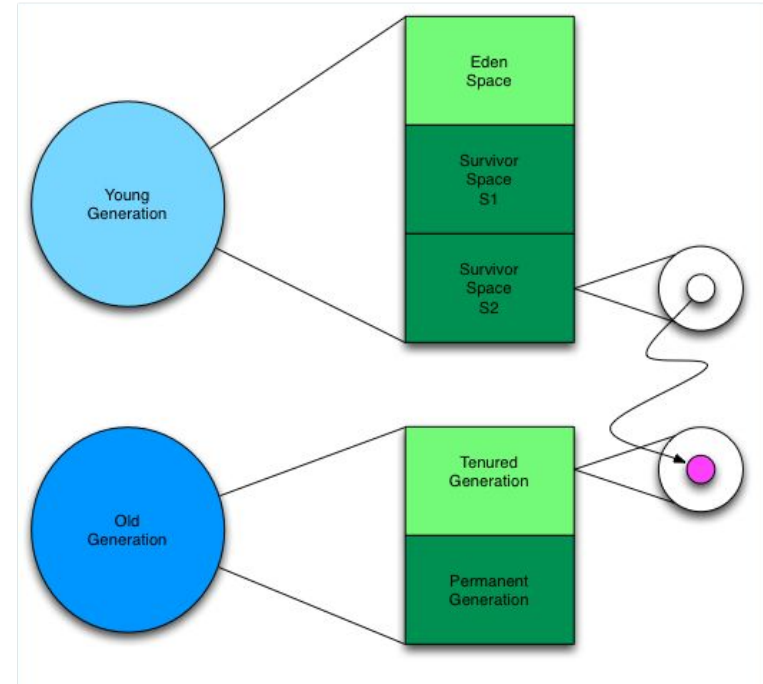
Old Generation

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 ejecute. Se puede hacer no elegible dentro del mismo método.