Java8 Programming

Introducción

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

Temario

- Descripción general de la plataforma Java
- Sintaxis de Java y revisión de clase
- Encapsulación y Subclases
- Sobrescribir métodos, polimorfismo y clases estáticas
- Clases abstractas y anidadas
- Interfaces y expresiones Lambda
- Colecciones y genéricos
- Colecciones, streams y filtros
- Interfaces funcionales Lambda
- Operaciones Lambda
- Excepciones y Afirmaciones
- API Java Date/Time
- Fundamentos de E / S
- Archivo de E / S (NIO.2)
- Concurrencia
- El Framework Fork-Join
- Streams paralelas
- Aplicaciones de base de datos con JDBC
- Localización

Descripción general de la plataforma Java

JRE y JDK

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

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 recopilador 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 ™

Contiene clases relacionadas con el desarrollo de beans: componentes basados en la arquitectura JavaBeans ™ 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 ™ 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 EMCAScript 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 ™ 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.

Sintaxis de Java y revisión de clase

Conceptos Fundamentales

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

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

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				

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 sobreescribir el método siempre que no esté marcado como final.
- El beneficio de sobrescribir es la capacidad de definir un comportamiento que sea específico para el tipo de subclase, lo que significa que una subclase puede implementar un método de clase padre en función de sus requisitos.
- En términos orientados a objetos, sobrescribir significa anular la funcionalidad de un método existente.

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

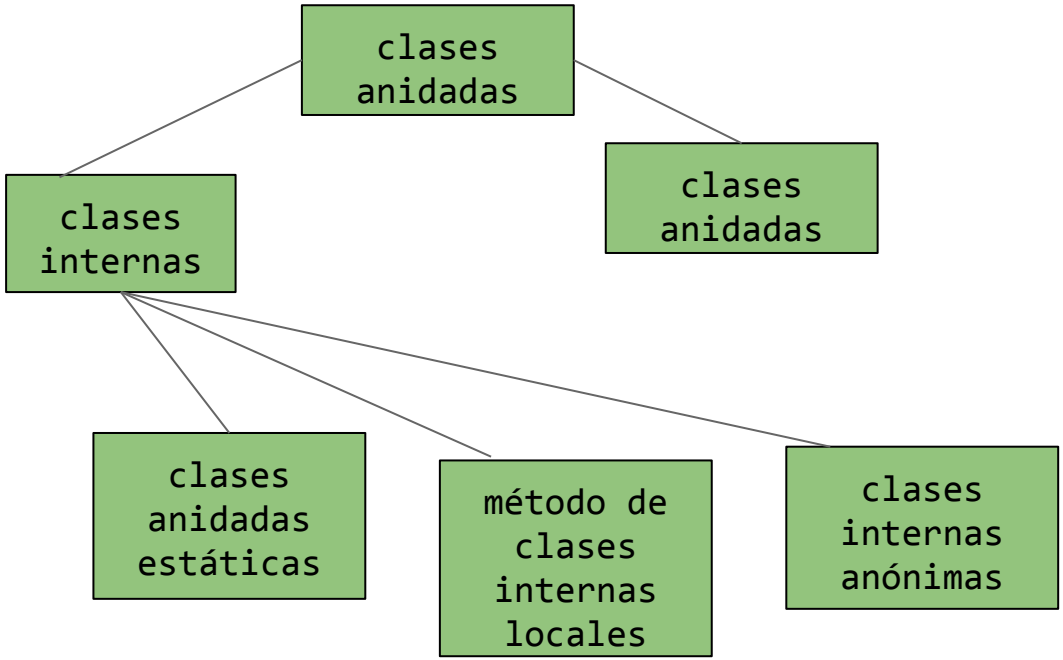
Sintaxis

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 {
   }
```

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
no modifier	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.

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 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

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.

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	`ContainingClass::staticMethodName`	`Class::staticMethodName`
2. Referencia a un constructor	`ClassName::new`	`ClassName::new`
3. Referencia a un método de instancia de un objeto arbitrario de un tipo particular	`ContainingType::methodName`	`Class::instanceMethodName`
4. Referencia a un método de instancia de un objeto particular	`containingObject::instanceMethodName`	`object::instanceMethodName`

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 ReferenceToInstanceMethodOAPO {

    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");
            ReferenceToInstanceMethodOAPO.printNames(names,System.out::println);
    }
    private static void printNames(List list, Consumer c ){
        list.forEach(x -> c.accept(x));
    }
}
```

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 bloque 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:

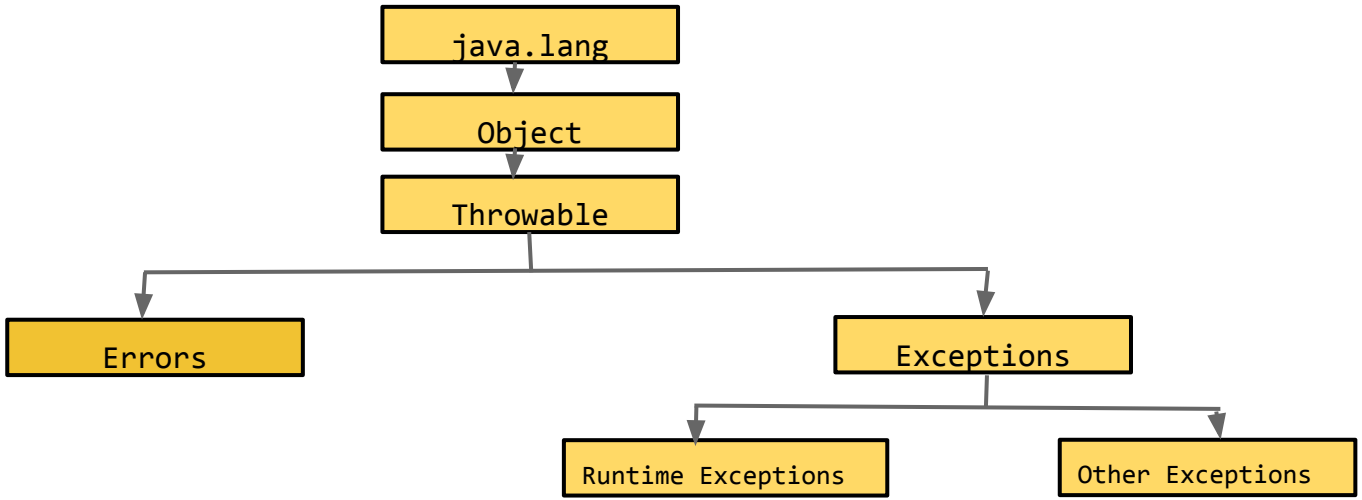
1. Imprime la descripción de la excepción.
2. Imprime el seguimiento de pila (Jerarquía de métodos donde se produjo la excepción).
3. Hace que el programa finalice.

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(ArrayIndexOutOfBoundsExceptio|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.

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 dara 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 sobreescribir 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 solo 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.

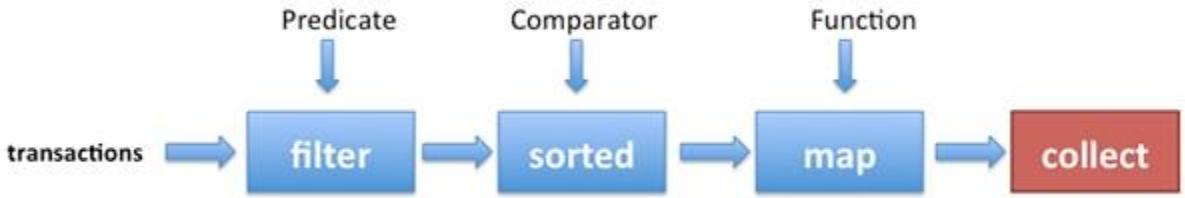
BinaryOperator<T> Operación recibe dos operadores del mismo tipo y produce uno del mismo tipo.

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

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.**



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 cómputos.

En términos simples, la diferencia entre las colecciones y los streams se relaciona con cuándo se hacen los cómputos. 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.

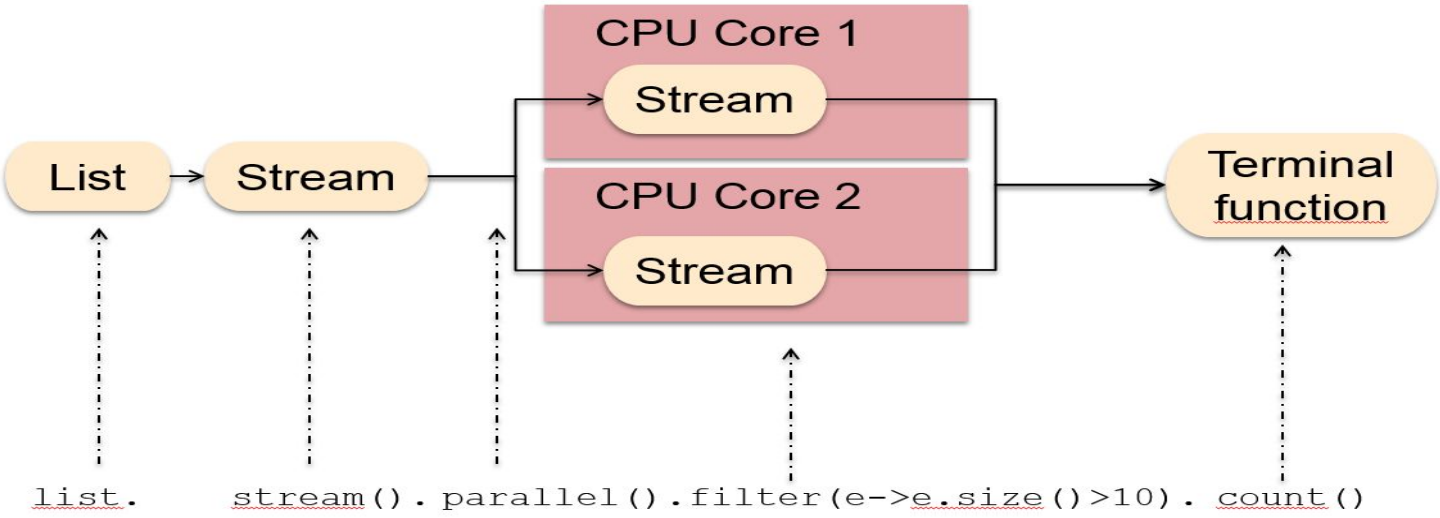
Parallel Streams

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.


CPU Core 1
Stream
List
Stream
CPU Core 2
Stream
Terminal function
list.
stream().parallel().filter(e->e.size()>10). count()

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 tipo 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 cambio, 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

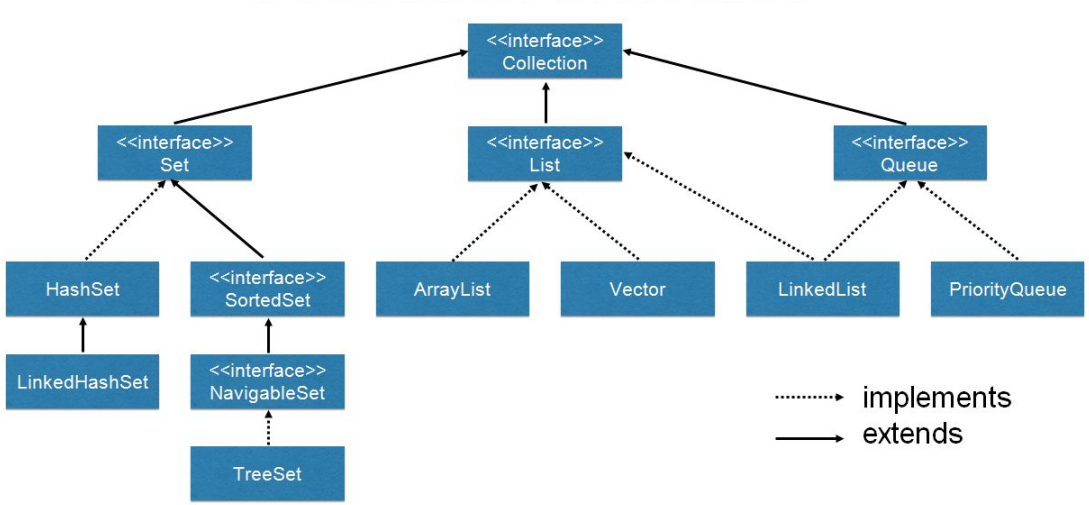
Se debe tener en cuenta que la Collection(s) es un término genérico, mientras que la Collection y las Collections son las API específicas del paquete java.util.

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. Cuándo 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 hacia delante e0class 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 arreglo 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 montón. 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.

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

`boolean addAll(Collection<? extends Element> coll)`	Agrega todos los elementos dados en la colección pasada como parámetro.
`boolean containsAll(Collection<?> coll)`	Comprueba si todos los elementos dados en la colección que están presentes en contenedor.
`boolean removeAll(Collection<?> coll)`	Remueve del contenedor los elementos dados en la colección pasada como parámetro.
`boolean retainAll(Collection<?> coll)`	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 no permite iterar el contenido de la colección. La misma es muy simple y contiene los siguientes métodos.

`boolean hasNext()`	Comprueba si el iterador tiene más elementos a recorrer.
`E next()`	Mueve el iterador al siguiente elemento y devuelve ese elemento (siguiente).
`void remove()`	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 lo 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 wordsused 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("aquaintance", "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.

`boolean offer(Element)`	Agrega elem en la pila .
`Element remove(Element)`	Obtiene y remueve el primer elemento (exception cuando no existe)
`Element poll()`	Obtiene y remueve el primer elemento
`Element element()`	Remueve el último elemento
`Element peek()`	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 extremo, la misma se introdujo en Java 6 en el paquete java.util.collection y extiende de la Interfaz Queue. Por lo tanto, todos los métodos proporcionados por Queue también están disponibles en la interfaz Deque.

`void addFirst(Element)`	Agrega elem al frente.
`void addLast(Element)`	Agrega elem al último.
`Element removeFirst()`	Remueve el primer elemento.
`Element removeLast()`	Remueve el último elemento
`Element getFirst()`	Obtiene el primer elemento.
`Element getLast()`	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;
       }
}
```

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.

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 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 largo. 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 com 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.
El núcleo de Java 8 date time API 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.
TemporalAdjuster	

`ZonedDateTime`	Representa una fecha y hora que incluye información de zona horaria
Period	
`DateTimeFormetter`	Formatea objetos de fecha y hora como cadenas. Por ejemplo, un ZonedDateTime o un LocalDateTime.

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);
```

I/O Fundamentals

I/O Fundamentals

El lenguaje Java proporciona un modelo simple para entrada y salida (E / S). Todo I / O es realizado escribiendo y leyendo de secuencias de datos. Los datos pueden existir en un archivo o una matriz, se canaliza desde otra secuencia, o incluso proviene de un puerto en otra computadora. La flexibilidad de este modelo lo convierte en una poderosa abstracción de cualquier entrada y salida requerida.
El paquete `java.io` contiene casi todas las clases que pueda necesitar para realizar entradas y salidas (E / S) en Java. Todas estas secuencias representan una fuente de entrada y un destino de salida. La secuencia en el paquete `java.io` admite muchos datos, como primitivas, objetos, caracteres localizados, etc.

Streaming

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

Streaming

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

- InputStream: se utiliza para leer datos de una fuente.
- OutputStream: se utiliza para escribir datos en un destino.

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

Byte Streams

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

Character Streams

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

Standard Streams

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

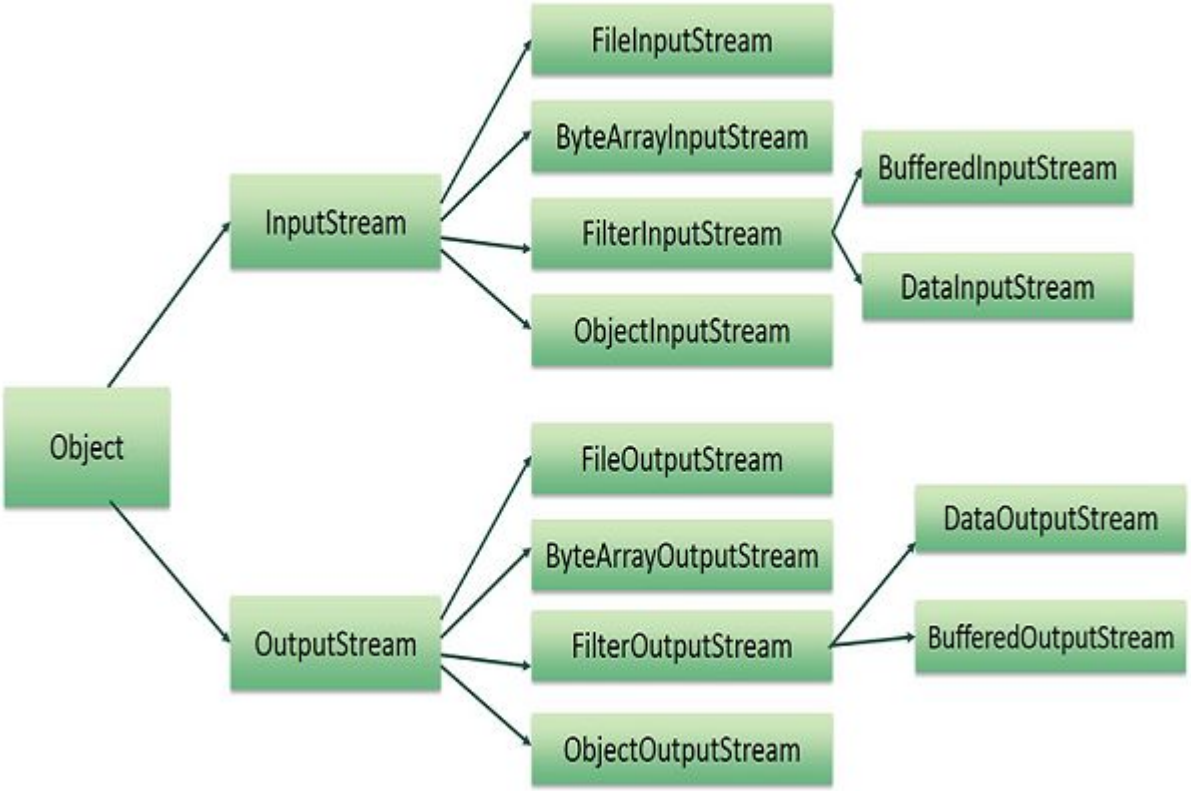
- Entrada estándar: se usa para alimentar los datos al programa del usuario y generalmente se usa un teclado como flujo de entrada estándar y se representa como `System.in.`

- Salida estándar: se utiliza para generar los datos producidos por el programa del usuario y, por lo general, se utiliza una pantalla de computadora para el flujo de salida estándar y se representa como `System.out.`
- Error estándar: se utiliza para generar los datos de error producidos por el programa del usuario y, por lo general, se utiliza una pantalla de computadora para el flujo de error estándar y se representa como System.err.

Reading and Writing Files

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

Jerarquía de clases para tratar las transmisiones de entrada y salida.



FileOutputStream

FileOutputStream se usa para crear un archivo y escribir datos en él. La transmisión crearía un archivo, si no existe, antes de abrirlo para su salida.

File Navigation and I/O

Hay varias otras clases que nos gustaría conocer para conocer los conceptos básicos de navegación de archivos y E / S.

- Clase de archivo
- Clase FileReader
- Clase FileWriter

Directorios en Java

Un directorio es un archivo que puede contener una lista de otros archivos y directorios.

Utiliza el objeto File para crear directorios, para listar los archivos disponibles en un directorio.

Para obtener detalles completos, consulte una lista de todos los métodos a los que puede llamar en el objeto Archivo y qué están relacionados con los directorios.

Crear directorios

Hay dos métodos útiles de utilidad de archivos, que se pueden usar para crear directorios:

- El método `mkdir()` crea un directorio, devuelve `true` en caso de éxito y `false` en caso de error. El error indica que la ruta especificada en el objeto Archivo ya existe o que el directorio no se puede crear porque toda la ruta aún no existe.
- El método `mkdirs()` crea un directorio y todos los padres del directorio.

Listando directorios

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

File I/O (NIO.2)

Introducción

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.

Java NIO consta de los siguientes componentes principales:

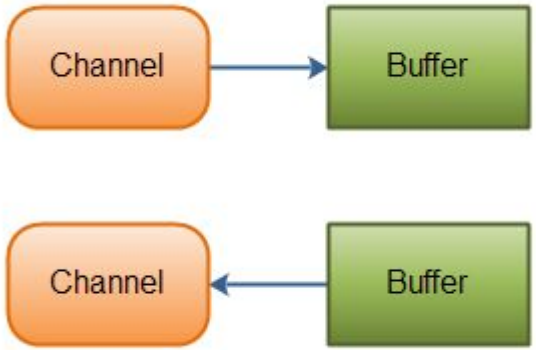
- Canales
- Buffers
- Selectores

Java NIO tiene más clases y componentes que estos, pero el `Channel`, `Buffer` y `Selector` forman el núcleo de la API. El resto de los componentes, como `Pipe` y `FileLock`, son simplemente clases de utilidad que se usarán junto con los tres componentes principales.

Java NIO: Channels and Buffers

En la API de IO estándar, trabajas con secuencias de bytes y secuencias de caracteres. En NIO, trabajas con canales y búferes. Los datos siempre se leen de un canal a un búfer, o se escriben desde un búfer a un canal.

Normalmente, todo IO en NIO comienza con un canal. Un canal es un poco como una secuencia. Desde el Canal, los datos se pueden leer en un Buffer. Los datos también se pueden escribir desde un Buffer en un Channel. Aquí hay una ilustración de eso:



Java NIO: canales de lectura de datos en Buffers, y Buffers escriben datos en Canales

Implementaciones principales de Channel en Java NIO:

- FileChannel
- DatagramChannel
- SocketChannel
- ServerSocketChannel
- Como puede ver, estos canales cubren la red UDP + TCP IO y el archivo IO.

Implementaciones básicas de Buffer en Java NIO:

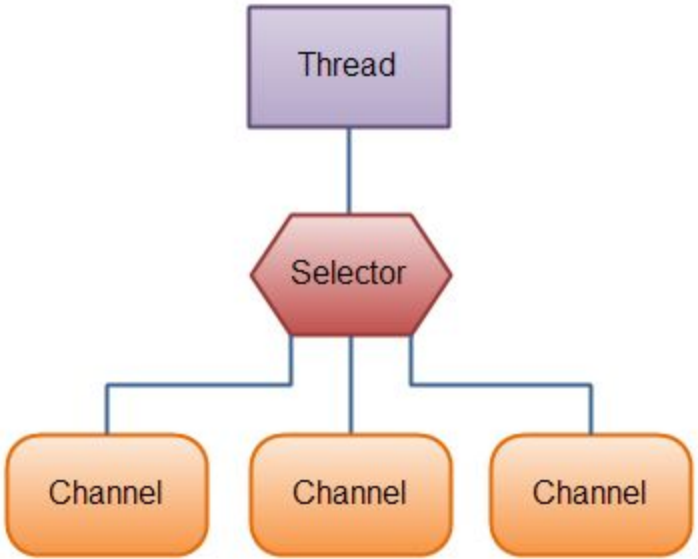
- ByteBuffer
- CharBuffer
- DoubleBuffer
- FloatBuffer
- IntBuffer
- LongBuffer
- ShortBuffer

Java NIO le permite hacer IO sin bloqueo. Por ejemplo, un hilo puede pedirle a un canal que lea datos en un búfer. Mientras el canal lee datos en el búfer, el hilo puede hacer otra cosa. Una vez que se leen los datos en el búfer, el hilo puede continuar procesando. Lo mismo es válido para escribir datos en canales.

Java NIO: Selectors

Java NIO contiene el concepto de "selectores". Un selector es un objeto que puede monitorear múltiples canales para eventos (como: conexión abierta, datos recibidos, etc.). Por lo tanto, un solo hilo puede monitorear múltiples canales de datos.
Un selector permite que un solo hilo maneje múltiples canales. Esto es útil si su aplicación tiene muchas conexiones (Canales) abiertas, pero solo tiene poco tráfico en cada conexión. Por ejemplo, en un servidor de chat.

Aquí hay una ilustración de un hilo usando un selector para manejar 3 canales:

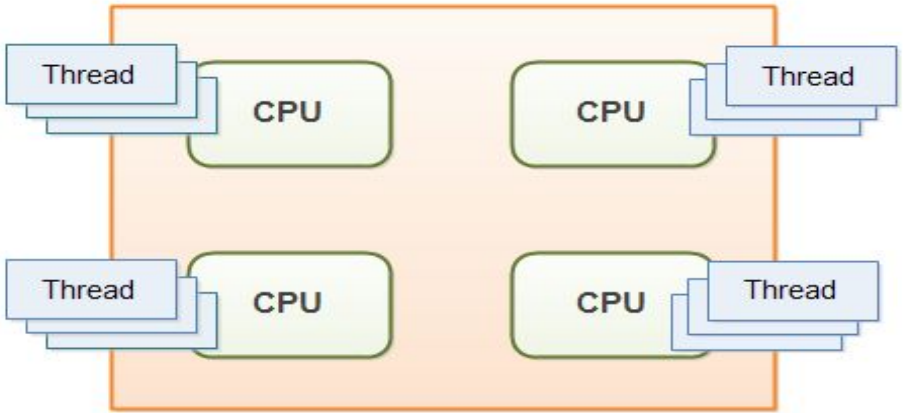


Java NIO: un subproceso usa un selector para manejar 3 canales

Para usar un selector, registra el canal con él. Luego lo llamas método select (). Este método se bloqueará hasta que haya un evento listo para uno de los canales registrados. Una vez que el método retorna, el hilo puede procesar estos eventos. Ejemplos de eventos son conexión entrante, datos recibidos, etc.

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

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

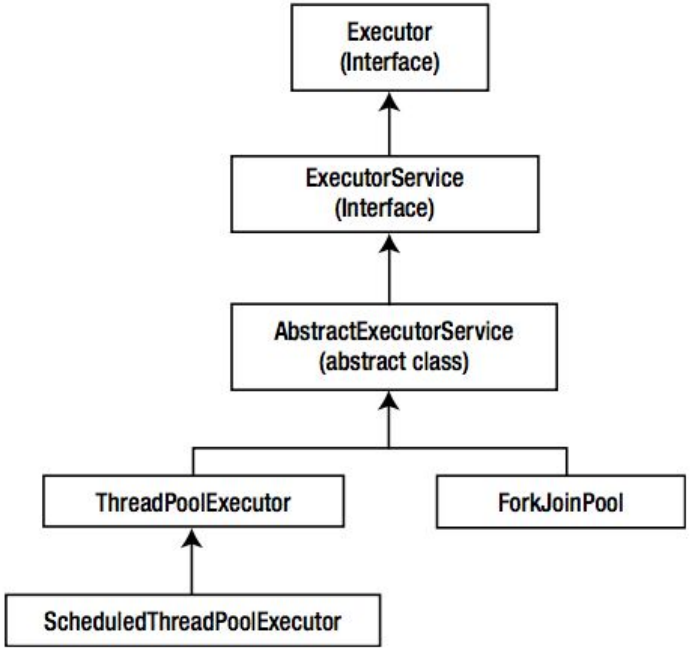
`Condition` admite el mecanismo de notificación de subprocesos. Cuando una cierta condición no se satisface, un hilo puede esperar para que otro hilo satisfaga esa condición; que otro hilo podrá notificar una vez que se cumple la condición. Una condición está ligada a un bloque. Un objeto `Condition` 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 a 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 interruptible, 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()` , `nvokeAll()` `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";
};

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

The Fork-Join Framework

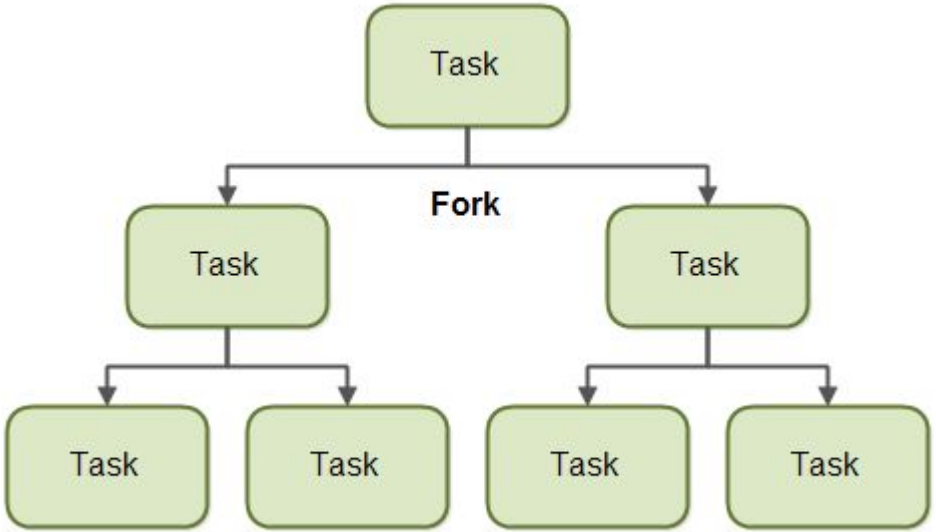
The `ForkJoinPool` se agregó a Java en Java 7, es similar al Java ExecutorService pero con una diferencia. El `ForkJoinPool` facilita que las tareas dividan su trabajo en tareas más pequeñas que luego se envían al `ForkJoinPool` también.

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.

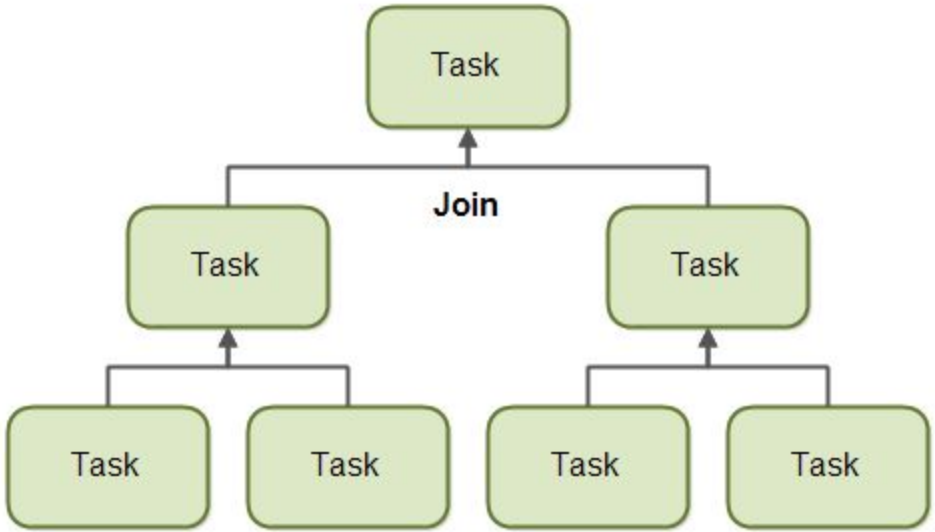
Fork

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

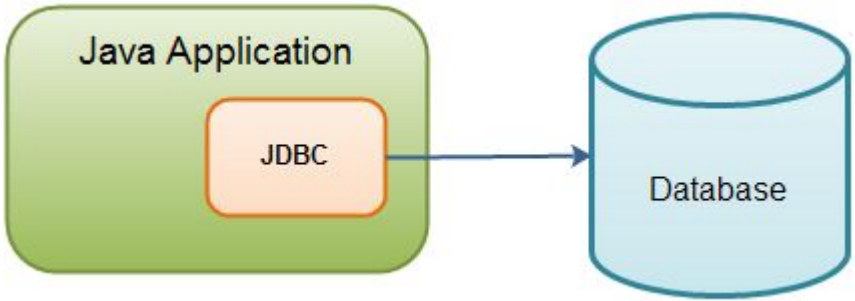
Cuando una tarea se ha dividido en subtareas, la tarea espera hasta que las subtareas hayan terminado de ejecutarse, luego la tarea puede unir (fusionar) todos los resultados en un resultado. Esto se ilustra en el siguiente diagrama:



Database Applications with JDBC

Introducción

La API Java JDBC permite que las aplicaciones Java se conecten a bases de datos relacionales a través de una API estándar, por lo que sus aplicaciones Java se vuelven independientes (casi) de la base de datos que utiliza la aplicación.



JDBC estandariza cómo conectarse a una base de datos, cómo ejecutar consultas en su contra, cómo navegar el resultado de dicha consulta y cómo ejecutar las actualizaciones en la base de datos. JDBC no estandariza el SQL enviado a la base de datos. Esto aún puede variar de una base de datos a otra.

Este tutorial de JDBC cubre la versión de JDBC disponible en Java 6. El tutorial no cubrirá todos los detalles de la API de JDBC, sino que se enfocará en las características más comúnmente utilizadas. El resto se puede leer en el JavaDoc después. Una vez que tenga una buena comprensión de JDBC, leer los últimos detalles en JavaDoc o en otro lugar, no será tan difícil.

JDBC Overview

La API JDBC consta de las siguientes partes principales:

- Controladores JDBC (Drivers)
- Conexiones (Connections)
- Declaraciones (Statements)
- Conjuntos de resultados (ResultSets)

Hay cuatro casos de uso de JDBC:

- Consulta la base de datos (lee datos de ella).
- Consulta los metadatos de la base de datos.
- Actualiza la base de datos.
- Realiza transacciones.

Componentes básicos de JDBC

Controladores JDBC

Un controlador JDBC es una colección de clases de Java que le permite conectarse a una determinada base de datos. Por ejemplo, MySQL tendrá su propio controlador JDBC. Un controlador JDBC implementa muchas de las interfaces JDBC. Cuando su código usa un controlador JDBC dado, realmente solo usa las interfaces JDBC estándar. El controlador concreto JDBC utilizado está oculto detrás de las interfaces JDBC. Por lo tanto, puede agregar un nuevo controlador JDBC sin que su código lo note.

Por supuesto, los controladores JDBC pueden variar un poco en las características que admiten.

Conexiones (Connections)

Una vez que se carga e inicializa un controlador JDBC, debe conectarse a la base de datos. Lo hace al obtener una conexión a la base de datos a través de la API JDBC y el controlador cargado. Toda comunicación con la base de datos ocurre a través de una conexión. Una aplicación puede tener más de una conexión abierta a una base de datos a la vez. Esto es realmente muy común.

Declaraciones (Statements)

Una declaración es lo que utiliza para ejecutar consultas y actualizaciones en la base de datos. Hay algunos tipos diferentes de declaraciones que puede usar. Cada declaración corresponde a una única consulta o actualización.

Conjuntos de resultados (ResultSets)

Cuando realiza una consulta en la base de datos, obtiene un ResultSet. A continuación, puede recorrer este ResultSet para leer el resultado de la consulta.

Casos de uso comunes de JDBC

Consulta la base de datos

Uno de los casos de uso más comunes es leer datos de una base de datos. Leer datos de una base de datos se llama consultar la base de datos.

Consultar los metadatos de la base de datos

Otro caso de uso común es consultar los metadatos de la base de datos. Los metadatos de la base de datos contienen información sobre la base de datos en sí. Por ejemplo, información sobre las tablas definidas, las columnas en cada tabla, los tipos de datos, etc.

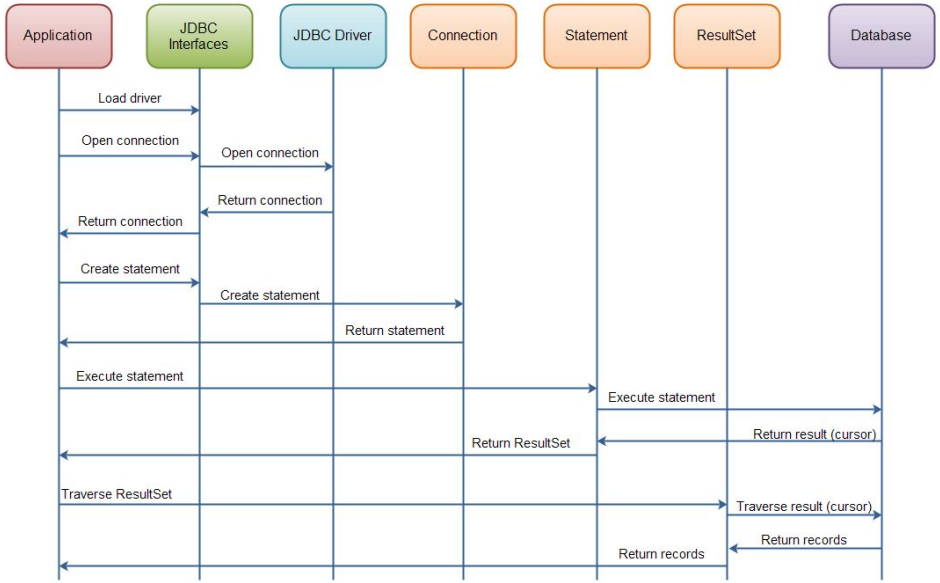
Actualiza la base de datos

Otro caso de uso común de JDBC es actualizar la base de datos. Actualizar la base de datos significa escribirle datos. En otras palabras, agregar nuevos registros o modificar (actualizar) registros existentes.

Realizar transacciones

Transacciones es otro caso de uso común. Una transacción agrupa múltiples actualizaciones y posiblemente consultas en una sola acción. O bien todas las acciones se ejecutan, o ninguna de ellas.

Diagrama de interacción de componentes JDBC



JDBC Driver Types

Un controlador JDBC es un conjunto de clases de Java que implementan las interfaces JDBC, dirigidas a una base de datos específica. Las interfaces JDBC vienen con Java estándar, pero la implementación de estas interfaces es específica de la base de datos a la que necesita conectarse. Tal implementación se llama un controlador JDBC.

Hay 4 tipos diferentes de controladores JDBC:

- Tipo 1: controlador de puente JDBC-ODBC
- Tipo 2: controlador de código Java + Native
- Tipo 3: Todos los controladores de traducción Java + Middleware
- Tipo 4: Todo el controlador de Java.

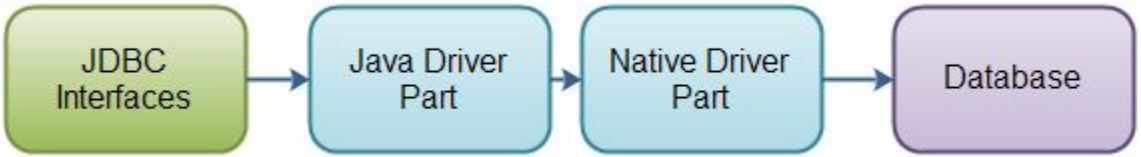
Hoy en día, la mayoría de los controladores son de tipo 4. Sin embargo, voy a discutir los 4 tipos de controladores en breve.

JDBC Driver Tipo 1

Un controlador JDBC de tipo 1 consiste en una parte de Java que traduce las llamadas de la interfaz JDBC a las llamadas ODBC. Un puente ODBC luego llama al controlador ODBC de la base de datos dada. Los controladores tipo 1 están destinados (principalmente) a ser utilizados al principio, cuando no había controladores tipo 4 (todos los controladores Java). Aquí hay una ilustración de cómo se organiza un controlador JDBC de tipo 1:

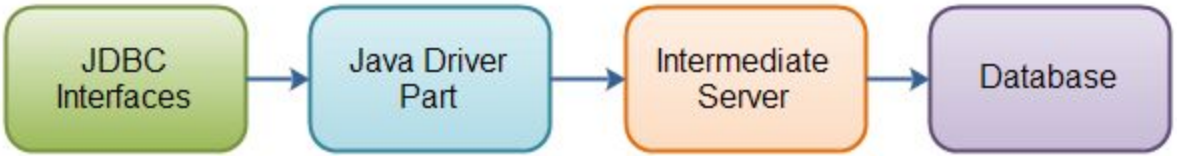
JDBC Driver Tipo 2

Un controlador JDBC tipo 2 es como un controlador tipo 1, excepto que la parte ODBC se reemplaza por una parte de código nativo. La parte del código nativo está dirigida a un producto de base de datos específico. Aquí hay una ilustración de un controlador JDBC de tipo 2:



JDBC Driver Tipo 3

Un controlador tipo 3 JDBC es un controlador de Java que envía las llamadas de la interfaz JDBC a un servidor intermedio. El servidor intermedio se conecta a la base de datos en nombre del controlador JDBC. Aquí hay una ilustración de un controlador JDBC de tipo 3:



JDBC Driver Tipo 4

Un controlador JDBC de tipo 4 es un controlador de Java que se conecta directamente a la base de datos. Se implementa para un producto de base de datos específico. Hoy en día, la mayoría de los controladores JDBC son controladores tipo 4. Aquí hay una ilustración de cómo se organiza un controlador JDBC de tipo 4:

JDBC: Database Connections

Antes de poder leer o escribir datos en una base de datos a través de JDBC, debe abrir una conexión a la base de datos.

Cargando el controlador JDBC

Lo primero que debe hacer antes de poder abrir una conexión de base de datos es cargar el controlador JDBC para la base de datos. En realidad, desde Java 6 esto ya no es necesario, pero al hacerlo no fallará. Usted carga el controlador JDBC de esta manera:

```
Class.forName ("driverClassName");
```

Cada controlador JDBC tiene una clase de controlador principal que inicializa el controlador cuando se carga. Por ejemplo, para cargar el controlador `H2Database`, escriba esto:

```
Class.forName("org.h2.Driver");
```

Solo debe cargar el controlador una vez. No necesita cargarlo antes de que se abra cada conexión. Solo antes de que se abriera la primera conexión.

Apertura de la conexión

Para abrir una conexión de base de datos, utiliza la clase `java.sql.DriverManager` . Usted llama a su método getConnection (), así:

```
String url      = "jdbc:h2:~/test";   //database specific url.
String user     = "sa";
String password = "";

Connection connection =  DriverManager.getConnection(url, user, password);
```

La url es la url de su base de datos. Debe consultar la documentación de su base de datos y el controlador JDBC para ver cuál es el formato para su base de datos específica. La URL que se muestra arriba es para una base de datos H2.

Los parámetros de `user` y `password` son el nombre de usuario y la contraseña de su base de datos.

Cerrando la conexión

Una vez que haya terminado de usar la conexión de la base de datos, debe cerrarla. Esto se hace llamando al método `Connection.close()`, así:

```
connection.close ();
```

JDBC: Consultando la base de datos

Consultar una base de datos significa buscar a través de sus datos. Usted debe enviar declaraciones SQL a la base de datos. Para hacerlo, primero necesita una conexión de base de datos abierta. Una vez que tiene una conexión abierta, necesita crear un objeto `Statement`, como este:

```
Statement statement = connection.createStatement ();
```

Una vez que haya creado la Declaración, puede usarla para ejecutar consultas SQL, como esta:

```
String sql = "select * from people";
ResultSet result = statement.executeQuery (sql);
```

Cuando ejecuta una consulta SQL, obtiene un ResultSet. El ResultSet contiene el resultado de su consulta SQL. El resultado se devuelve en filas con columnas de datos. Usted itera las filas del ResultSet así:

```
while (result.next ()) {
      String name = result.getString ("nombre");
      long age = result.getLong ("edad");
}
```

El método `ResultSet.next()` pasa a la siguiente fila en `ResultSet`, si ya hay filas. Si hay más filas, devuelve verdadero. Si no hubiera más filas, devolverá falso.

Debe llamar a `next()` al menos una vez antes de poder leer cualquier dato. Antes de la primera llamada siguiente (), ResultSet se posiciona antes de la primera fila.

Puede obtener datos de columna para la fila actual llamando a algunos de los métodos `getXXX()`, donde XXX es un tipo de datos primitivo. Por ejemplo:

```
result.getString ("columnName");
result.getLong ("columnName");
result.getInt ("columnName");
result.getDouble ("columnName");
result.getBigDecimal ("columnName");
```

El nombre de la columna para obtener el valor de se pasa como parámetro a cualquiera de estas llamadas al método `getXXX()`.

También puede pasar un índice de la columna, como este:

```
result.getString (1);
result.getLong (2);
result.getInt (3);
result.getDouble (4);
result.getBigDecimal (5);
```

Para que funcione, necesita saber qué índice tiene una columna determinada en ResultSet. Puede obtener el índice de una columna determinada llamando al método `ResultSet.findColumn()`, como este:

```
int columnIndex = result.findColumn ("columnName");
```

Si itera grandes cantidades de filas, hacer referencia a las columnas por su índice podría ser más rápido que por su nombre.

Cuando termine de iterar el `ResultSet`, debe cerrar tanto el `ResultSet` como el objeto Statement que lo creó (si lo hizo). Lo haces llamando a sus métodos close (), como este:

```
result.close ();
statement.close ();
```

Por supuesto, debe llamar a estos métodos dentro de un bloque finally para asegurarse de que se invoquen incluso si se produce una excepción durante la iteración `ResultSet`.

JDBC: Actualizando la base de datos

Para actualizar la base de datos necesita usar un Statement. Pero, en lugar de llamar al método executeQuery (), llama al método executeUpdate ().

Hay dos tipos de actualizaciones que puede realizar en una base de datos:

- Actualizar valores de registro
- Eliminar registros

El método executeUpdate () se usa para ambos tipos de actualizaciones.

Actualización de registros

Aquí hay un ejemplo de valor de registro de actualización:

```
Statement statement = connection.createStatement();
String    sql       = "update people set name='John' where id=123";
int rowsAffected    = statement.executeUpdate(sql)
```

Las filas afectadas por la llamada a `statement.executeUpdate(sql)` indican cuántos registros de la base de datos se vieron afectados por la instrucción SQL.

Eliminar registros

Aquí hay un ejemplo de borrar registro:

```
Statement statement = connection.createStatement();
String    sql       = "delete from people where id=123";
int rowsAffected    = statement.executeUpdate(sql);
```

De nuevo, las filas afectadas por la llamada a `statement.executeUpdate(sql)` indican cuántos registros de la base de datos se vieron afectados por la sentencia de SQL.

JDBC: ResultSet

Un `ResultSet` consiste en registros, cada registro contiene un conjunto de columnas y cada registro contiene la misma cantidad de columnas, aunque no todas las columnas pueden tener un valor, pueden tener un valor nulo.

Name	Age	Gender
John	27	Male
Jane	21	Female
Jeanie	31	Female

Este ResultSet tiene 3 columnas diferentes (Name, Age, Gender) y 3 registros con diferentes valores para cada columna.

Creando un ResultSet

Podemos crear un `ResultSet` mediante la ejecucción de un `Statement` o `PreparedStatement` de esta manera:

```
Statement statement = connection.createStatement();
ResultSet result = statement.executeQuery("select * from people");
```

O de la siguiente:

```
String sql = "select * from people";
PreparedStatement statement = connection.prepareStatement(sql);
ResultSet result = statement.executeQuery();
```

ResultSet Type, Concurrency and Holdability

Iterando un ResultSet

Para iterar el ResultSet, usa su método next (), que nos devolverá verdadero si el ResultSet tiene un próximo registro, y mueve el ResultSet para apuntar al siguiente registro. Si no hubo más registros, next() devuelve falso. Una vez que el método next() haya devuelto false, no deberías volver a llamarlo. Hacerlo puede resultar en una excepción.

```
while (result.next ()) {
      // ... obtener valores de columna de este registro
}
```

Como puede ver, el método next () se llama realmente antes de acceder al primer registro. Eso significa que el ResultSet comienza señalando antes del primer registro. y na vez que next () ha sido invocado apuntara al primer registro.

De forma similar, cuando se llama a next () y devuelve false, el ResultSet en realidad apunta después del último registro.

Iterando un ResultSet

De forma similar, cuando se llama a `next()` y devuelve `false`, el `ResultSet` en realidad apunta después del último registro.

No puede obtener el número de filas en un `ResultSet`, excepto si itera hasta el final y cuenta las filas. Sin embargo, si el ResultSet es forward-only, no puede retroceder mediante prvl. Incluso si pudieras retroceder, sería una manera lenta de contar las filas en el ResultSet. Le conviene estructurar su código para que no necesite saber la cantidad de registros con anticipación.

Accediendo a los valores de un ResultSet

Al iterar el ResultSet, desea acceder a los valores de columna de cada registro. Lo haces llamando a uno o más de los muchos métodos getXXX (). Usted pasa el nombre de la columna para obtener el valor de, a los muchos métodos getXXX (). Por ejemplo:

```
while (result.next ()) {
      result.getString ("nombre");
      result.getInt ("edad");
      result.getBigDecimal ("coeficiente");
}
```

Hay muchos métodos getXXX () a los que puede llamar, que devuelven el valor de la columna como un tipo de datos determinado, p. String, int, long, double, BigDecimal, etc. Todos toman el nombre de la columna para obtener el valor de columna para, como parámetro.

Los métodos getXXX () también vienen en versiones que toman un índice de columna en

Los métodos getXXX () también vienen en versiones que toman un índice de columna en lugar de un nombre de columna. Por ejemplo:

```
while (result.next ()) {
      result.getString (1);
      result.getInt (2);
      result.getBigDecimal (3);
}
```

Si no conoce el índice de una determinada columna, puede encontrar el índice de esa columna utilizando el método ResultSet.findColumn (String columnName).

```
int nameIndex = result.findColumn ("nombre");
int ageIndex = result.findColumn ("edad");
int coeffIndex = result.findColumn ("coeficiente");
```

Tipos de ResultSet

Un ResultSet puede ser de un cierto tipo. El tipo determina algunas características y habilidades del ResultSet.

No todos los tipos son compatibles con todas las bases de datos y controladores JDBC. Deberá verificar su base de datos y el controlador JDBC para ver si es compatible con el tipo que desea utilizar. El método DatabaseMetaData.supportsResultSetType (int type) devuelve verdadero o falso dependiendo de si el tipo dado es compatible o no. La clase DatabaseMetaData se trata en un texto posterior.

En el momento de escribir, hay tres tipos de ResultSet:

- ResultSet.TYPE_FORWARD_ONLY
- ResultSet.TYPE_SCROLL_INSENSITIVE
- ResultSet.TYPE_SCROLL_SENSITIVE

El tipo predeterminado es TYPE_FORWARD_ONLY

TYPE_FORWARD_ONLY significa que el ResultSet solo se puede navegar hacia adelante. Es decir, solo puede pasar de la fila 1, a la fila 2, a la fila 3, etc. No puede retroceder en el ResultSet.

TYPE_SCROLL_INSENSITIVE significa que el ResultSet se puede navegar (desplazarse) tanto hacia adelante como hacia atrás. También puede saltar a una posición relativa a la posición actual, o saltar a una posición absoluta. ResultSet es insensible a los cambios en la fuente de datos subyacente mientras el ResultSet está abierto. Es decir, si un registro en ResultSet se cambia en la base de datos por otro hilo o proceso, no se reflejará en los ResulsSet ya abiertos de este tipo.

TYPE_SCROLL_SENSITIVE significa que el ResultSet puede navegarse (desplazarse) tanto hacia adelante como hacia atrás. También puede saltar a una posición relativa a la posición actual, o saltar a una posición absoluta. ResultSet es sensible a los cambios en la fuente de datos subyacente mientras el ResultSet está abierto. Es decir, si un registro en ResultSet se cambia en la base de datos por otro hilo o proceso, se reflejará en los ResulsSet ya abiertos de este tipo.

Concurrencia con ResultSet

Actualizando ResultSets

Concurrencia con ResultSet

Concurrencia con ResultSet

ResultSet Holdability

La capacidad de mantenimiento de ResultSet determina si un ResultSet se cierra cuando se llama al método commit () de la conexión subyacente.

No todos los modos de holdability son compatibles con todas las bases de datos y controladores JDBC. DatabaseMetaData.supportsResultSetHoldability (int holdability) devuelve verdadero o falso dependiendo de si el modo de holdability dado es compatible o no. La clase DatabaseMetaData se trata en un texto posterior.

Hay dos tipos de holgabilidad:

Java File I/O (NIO.2)

Java File I/O (NIO.2)

- Use the Path class to operate on file and directory paths.
- Use the Files class to check, delete, copy, or move a file or directory.
- Read and change file and directory attributes.
- Recursively access a directory tree.
- Find a file by using the PathMatcher class.
- Watch a directory for changes by using 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.

`Path getRoot()`	Devuelve un objeto Path que representa la raíz de la ruta dada, o null si la ruta no tiene una raíz.
`Path getFileName()`	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.
`Path getParent()`	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.
`int getNameCount()`	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)

`Path getName(int i)`	Devuelve el i-ésimo archivo / nombre del directorio; el índice 0 comienza desde el nombre más cercano hasta la raíz.
`Path subpath(int beginIndex, int endIndex)`	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.
`Path normalize()`	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).
`Path resolve(Path other)` `Path resolve(String other)`	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.
`Boolean isAbsolute()`	Devuelve true si la ruta dada es una ruta absoluta; devuelve false si no (cuando la ruta dada es una ruta relativa, por ejemplo).
`Path startsWith(String path)` `Path startsWith(Path path)`	Devuelve true si este objeto Path comienza con la ruta dada, o bien devuelve false.
`Path toAbsolutePath()`	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\\NIO2\\");

            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

Java 7 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

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

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

`Boolean isHidden(Path path)`	Devuelve true si el archivo representado por path es un archivo oculto.
`long size(Path path)`	Devuelve el tamaño del archivo en bytes representados por path.
`UserPrincipal getOwner(Path path, LinkOption. . .),` `Path setOwner(Path path, UserPrincipal owner)`	Obtiene / establece el propietario del archivo.

`FileTime getLastModifiedTime(Path path, LinkOption...)`
 	
`Path setLastModifiedTime(Path path, FileTime time)`
	Obtiene / establece la última hora modificada para la hora especificada.

`Object getAttribute(Path path, String attribute, LinkOption...)`
 	
`Path setAttribute(Path path, String attribute, Object value, LinkOption...)`
	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.

`FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)`	Se invoca justo antes de acceder a los elementos del directorio.
`FileVisitResult visitFile(T file, BasicFileAttributes attrs)`	Se invoca cuando se visita un archivo.
`FileVisitResult postVisitDirectory(T dir, IOException exc)`	Se invoca cuando se accede a todos los elementos del directorio.
`FileVisitResult visitFileFailed(T file, IOException exc)`	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 lo visitores 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

Ejemplo

```
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.relativize(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.relativize(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 (ejemplo 2 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 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 (ej)

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 (ej)

```
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 7 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 el conveniente, escalable para realizar de una manera fácil seguimiento de los cambios en un directorio.

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

```
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();
        }
```

Localization

Localization

- Read and set the locale by using the Locale object.
- Build a resource bundle for each locale.
- Load a resource bundle in an application.
- Format text for localization by using NumberFormat and DateFormat.

Localization

La localización es el proceso de adaptar el contenido de un producto a una región o idioma específico. Traducir palabras es la parte más conocida de este proceso, pero no es la única parte. La localización generalmente incluye:

- Números
- Fechas
- Monedas
- Imágenes y sonidos
- Diseños

Incluso en países con los mismos idiomas, puede encontrar diferencias, por ejemplo, inglés americano vs. inglés británico.

Entonces los beneficios de la localización son:

- Con la adición de datos localizados, el mismo ejecutable se puede ejecutar en todo el mundo.
- Los elementos textuales, como los mensajes de estado y las etiquetas de los componentes de la GUI, no están codificados en el programa. En cambio, se almacenan fuera del código fuente y se recuperan dinámicamente.
- El soporte para nuevos lenguajes no requiere recompilación.
- Los datos dependientes de la cultura, como las fechas y las monedas, aparecen en formatos que se ajustan a la región y el idioma del usuario final.

Localization :: Introducción

La localización tiene que ver con hacer que el software sea relevante y útil para los usuarios de diferentes culturas, es decir, personalización de software para personas de diferentes países o lenguas.

¿Cómo localizar una aplicación de software?
Se deben tener en cuenta dos directrices importantes cuando se localiza una aplicación de software:

- No codifique el texto (como mensajes a los usuarios, elementos de texto en las GUI, etc.) y separarlos en archivos externos o clases dedicadas. Con esto logrado suele haber esfuerzo mínimo para agregar soporte para un nuevo entorno local en su software.
- Manejar aspectos específicos de la cultura tales como fecha, hora, moneda y números de formato con la localización en mente. En lugar de asumir un entorno local predeterminado, diseñe de tal se recupera la configuración actual y se personaliza.

Localization :: La clase Locale

`static Locale[] getAvailableLocales()`	Retorna la lista de Locales disponibles en la JVM.
`static Locale getDefault()`	Retorna el Locale por defecto en la JVM.
`static void setDefault(Locale newLocale)`	Asigna el Locale por defecto a la JVM.
`String getCountry()`	Retorna the country code for the locale object.
`String getDisplayCountry()`	Retorna el nombre del pais de lenguaje del objeto Locale
`String getLanguage()`	Retorna el cofigo de lenguaje del objeto Locale
`String getDisplayLanguage().`	Retorna el nombre de lenguaje del objeto Locale
`String getVariant()`	Retorna el variant del objeto Locale
`String getDisplayVariant()`	Retorna el nombre del variant del objeto Locale
`String toString()`	

Localization :: Resource Bundles

```
Locale.setDefault(Locale.CANADA_FRENCH);
Locale defaultLocale = Locale.getDefault();
System.out.printf("The default locale is %s %n", defaultLocale);
System.out.printf("The default language code is %s and the name is %s %n",
defaultLocale.getLanguage(), defaultLocale.getDisplayLanguage());
System.out.printf("The default country code is %s and the name is %s %n",
defaultLocale.getCountry(), defaultLocale.getDisplayCountry());
System.out.printf("The default variant code is %s and the name is %s %n",
defaultLocale.getVariant(), defaultLocale.getDisplayVariant());
```

Obtenemos

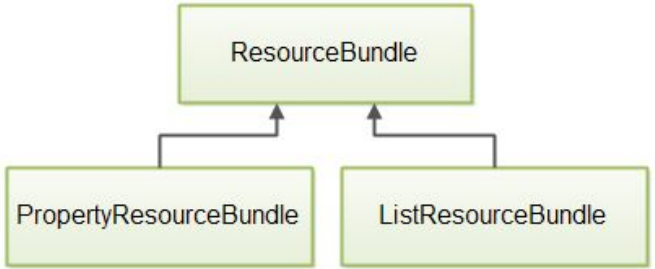
```
The default locale is fr_CA
The default language code is fr and the name is français
The default country code is CA and the name is Canada
The default variant code is and the name is Canada
```

Localization :: Loading a Resource Bundle

La clase `java.util.ResourceBundle` se utiliza para almacenar textos y componentes que son locales sensibles. Este texto examina de cerca la clase ResourceBundle y sus subclases.

La jerarquía de clases `ResourceBundle`

La clase `ResourceBundle` tiene dos subclases denominadas `PropertyResourceBundle` y `ListResourceBundle`.



Localization :: ResourceBundle

La clase `PropertyResourceBundle` es concreta y proporciona soporte para varios entornos locales en forma de archivos de propiedad. Para cada `Locale`, específica las claves y los valores en un archivo de propiedades para ese `Locale`. Mediante el método `ResourceBundle.getBundle()`, el archivo de propiedad relevante se cargará automáticamente. Para ello se; tienen que seguir ciertas convenciones de nomenclatura planteadas más adelante. Sólo puede utilizar String como claves y valores cuando utiliza archivos de propiedades.

La clase `ListResourceBundle`: Para agregar soporte a una configuración regional, puede ampliar este resumen clase. En su clase derivada, tiene que anular el método `getContents()`, que devuelve un `Objeto[][]`. Esta matriz debe tener la lista de claves y valores. Las claves deben ser `String`. Normalmente, los valores también son cadenas, pero los valores pueden ser cualquier cosa: clips de sonido, clips de vídeo, URL o imágenes.

Localization :: Resource Bundles

Java aplica una convención de nomenclatura predefinida que se debe seguir para crear grupos de recursos. Sólo a través de los nombres de los bundles de propiedades, la biblioteca Java carga los locales relevantes.

```
packagequalifier.bundlename + "_" + language + "_" + country + "_" + (variant + "_ #" "#") + script + "-" + extensiones
```

Aquí está la descripción de los elementos de este nombre completo:

- **packagequalifier:** El nombre del paquete (o los subpaquetes) en el que el recurso paquete.
- **bundlename**: El nombre del paquete de recursos que utilizará en el programa para referirse y cárgalo.
- **lenguage**: una abreviatura de dos letras típicamente dada en minúsculas para el idioma de la localidad (en casos raros, podría ser tres letras también).
- **country**: Una abreviatura de dos letras típicamente dada en mayúsculas para el país del locale (en casos, podría ser tres letras también).
- **variante**: una lista arbitraria de variantes (en minúsculas o mayúsculas) para diferenciar localizaciones cuando necesita más de una configuración regional para una combinación de idioma y país.

Localization :: ResourceBundle

La clase ResourceBundle

`Object getObject(String key)`	Devuelve el valor asignado a la clave dada. Lanza un `MissingResourceException` si no hay objeto para una clave determinada es encontrado.

`static ResourceBundle getBundle(String baseName)`
 	
`static final ResourceBundle getBundle(String baseName, Locale locale)`
 	
`final ResourceBundle getBundle(String baseName, Locale targetLocale, Control control)`
	Devuelve el `ResourceBundle` para el dado baseName, locale y control; lanza un `MissingResourceException` si no se encuentra ningún paquete de recursos coincidente. La instancia de Control se utiliza para controlar u obtener información sobre el proceso de carga del paquete de recursos.

`String getString(String key)`	Devuelve el valor asignado a la clave dada; equivalente a emitir el valor de retorno de `getObject()` a `String`. Muestra una excepción `MissingResourceException` si no se encuentra ningún objeto para una clave determinada. Lanza ClassCastException si el objeto devuelto no es una cadena.

Localization :: PropertyResourceBundle

Si diseña su aplicación con la localización en mente usando archivos de propiedades, puede agregar soporte para nuevos Locale a la aplicación sin cambiar código.

En el caso de la localización mediante `PropertyResourceBundle`, se utilizan archivos de propiedades para asignar las mismas cadenas de claves y valores. En él, se referirá a las cadenas de claves y al cargar el archivo de propiedades coincidentes para la configuración regional, los valores correspondientes para las claves se obtendrá de los archivos de propiedad para su uso en el programa.

Dados los siguientes ficheros de recursos

`ResourceBundle.properties`	`ResourceBundle_es.properties`	`ResourceBundle_it.properties`
`Greeting = Hello`	`Greeting = Hola`	`Greeting = Ciao`

Localization :: PropertyResourceBundle

Mediante el API podemos acceder definiendo el Locale a utilizar

```
Locale currentLocale = Locale.getDefault();
ResourceBundle resBundle = ResourceBundle.getBundle("ResourceBundle", currentLocale);
System.out.printf(resBundle.getString("Greeting"));
```

Luego, podemos cambiar el Locale de la JVM mediante el mismo API

```
Locale currentLocale = Locale.setDefault(Locale.ITALY);
```

o mediante propiedades del sistema

```
-Duser.language = it -Duser.region = IT
```

Localization :: Loading a Resource Bundle Control

El método `getBundle()` toma un objeto ResourceBundle.Control como un parámetro adicional. Al extender esta clase ResourceBundle.Control y pasar la instancia de esa clase extendida a el método `getBundle()`, puede cambiar el proceso predeterminado de búsqueda de paquetes de recursos o leerlo formatos de paquetes de recursos no estándar (como archivos XML).

```
class TalkativeResourceBundleControl extends ResourceBundle.Control {
	public List <Locale> getCandidateLocales(String baseName, Locale locale) {
		List <Locale> candidateLocales = super.getCandidateLocales(baseName, locale);
		System.out.printf("Candidate locales for base bundle name %s and locale %s %n",
			baseName, locale.getDisplayName());
		for(Locale candidateLocale : candidateLocales) {
			System.out.println(candidateLocale);
		}
		return candidateLocales;
	}
}
```

```
ResourceBundle.properties -- Global bundle
ResourceBundle_ar.properties -- Arabic language bundle
ResourceBundle_en.properties -- English bundle
ResourceBundle_it.properties -- Italian language bundle
ResourceBundle_it_IT_Rome.properties -- Italian (Italy, Rome, Vatican) bundle
```

Localization :: Loading a Resource Bundle Control

```
class CandidateLocales {
	public static void loadResourceBundle(String resourceBundleName, Locale locale) {
		// Pass an instance of TalkativeResourceBundleControl
		// to print candidate locales
		ResourceBundle resourceBundle = ResourceBundle.getBundle(resourceBundleName, locale,
			new TalkativeResourceBundleControl());
		String rbLocaleName = resourceBundle.getLocale().toString();
		// if the resource bundle locale name is empty,
		// it means default property file
		if(rbLocaleName.equals("")) {
			System.out.println("Loaded the default property file with name: " +
				resourceBundleName);
		} else {
			System.out.println("Loaded the resource bundle for the locale: " +
				resourceBundleName + "." + rbLocaleName);
		}
	}
	public static void main(String[] args) {
		// trace how ResourceBundle_it_IT_Rome.properties is resolved
		loadResourceBundle("ResourceBundle", new Locale("it", "IT", "Rome"));
	}
}
```

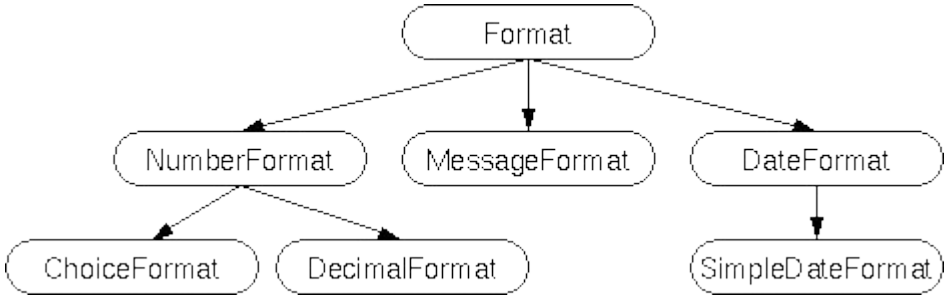
Localization :: Locale Resumen

Leer y configurar la configuración regional mediante el objeto Locale
• Una localidad representa un idioma, cultura o país; la clase Locale en Java proporciona una abstracción para este concepto.
• Cada localidad puede tener tres entradas: el idioma, el país y la variante. Puede utilizar códigos disponibles para el idioma y el país para formar etiquetas de configuración regional. No hay etiquetas estándar para
variantes; usted puede proporcionar las secuencias variantes basadas en su necesidad.
• Los métodos getter de la clase Locale -como getLanguage (), getCountry () y getVariant () - códigos de retorno; mientras que los métodos similares de getDisplayCountry (), getDisplayLanguage () y getDisplayVariant () devuelven nombres.
• El método getDefault () en Locale devuelve la configuración regional predeterminada en la JVM. Tu puedes cambiar
esta configuración predeterminada a otra configuración regional mediante el método setDefault ().
• Hay muchas maneras de crear o obtener un objeto Locale correspondiente a una configuración regional:
? Utilice el constructor de la clase Locale.
• Utilice el método forLanguageTag (String languageTag) en la clase Locale.
• Construya un objeto Locale instanciando Locale.Builder y luego llamando a setLanguageTag () de ese objeto.
• Utilice las constantes estáticas predefinidas para locales en la clase Locale.

Localization :: Formateando I18N

El texto es obviamente el aspecto principal a ser internacionalizado. Sin embargo, hay muchos aspectos que se manejan de forma diferente en la configuración regional: fecha y hora, números y monedas.

La clase `Format` es una clase base abstracta que define la API para formatear y analizar datos relacionados con la configuración regional. `Format` declara dos métodos importantes: `format` que da formatos de objetos sensibles a la configuración regional en `Strings` y `parseObject` que analiza `Strings` devolviendo un objeto. Todo lo que fue formateado por el método `format` puede ser parseado por metodo `parseObject`.



Localization :: Formateando I18N

NumberFormat

La clase `NumberFormat` proporciona soporte para procesar números de una manera sensible a la configuración regional (separador de miles, los caracteres de puntuación, valor de moneda, etc.) son diferentes y la clase `NumberFormat` proporciona esta funcionalidad. Como sabemos, en la clase `Format`, "formatear" significa convertir un valor numérico a un valor de cadena de una manera sensible a la cultura; similarmente, "parsear" significa convertir un número a la forma numérica.

```
long tenMillion = 10_000_000L;
// first print ten million in German locale
NumberFormat germanFormat = NumberFormat.getInstance(Locale.GERMANY);
String localizedTenMillion = germanFormat.format(tenMillion);
System.out.println("Ten million in German locale is " + localizedTenMillion)
Number parsedAmount = germanFormat.parse(localizedTenMillion);
if(tenMillion == parsedAmount.longValue()) {
      System.out.println("Successfully parsed the number for the locale");
}
```

Localization :: Formateando I18N

NumberFormat

```
Locale locale = Locale.getDefault();
Currency currencyInstance = Currency.getInstance(locale);
System.out.println(" The currency code for locale " + locale
     + " is: " + currencyInstance.getCurrencyCode()
     + " \n The currency symbol is " + currencyInstance.getSymbol()
     + " \n The currency name is " + currencyInstance.getDisplayName());
```

Localization :: Formateando I18N

DateFormat

La clase DateFormat provee soporte para procesar fecha y hora en formato locale-sensitive. La misma tiene tres métodos de construccion sobrecargados - getDateInstance(), getTimeInstance () y getDateTimeInstance () - que devuelven instancias de DateFormat para fecha de procesamiento, hora y fecha/hora, respectivamente.

```
Date today = new Date();
Locale [] locales = { Locale.CANADA, Locale.FRANCE, Locale.GERMANY, Locale.ITALY };
int [] dateStyleFormats = { DateFormat.SHORT, DateFormat.MEDIUM, DateFormat.LONG,
DateFormat.FULL, DateFormat.DEFAULT};
for(Locale locale : locales) {
      // DateFormat.FULL refers to the full details of the date
      DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.FULL, locale);
      System.out.println("Date in locale " + locale + " is: " + dateFormat.format(today));
      for(int dateStyleFormat : dateStyleFormats) {
            DateFormat dateFormat2 = DateFormat.getDateInstance(dateStyleFormat);
            System.out.println(dateFormat2.format(now));
      }
}
```

Localization :: Formateando I18N

SimpleDateFormat

`SimpleDateFormat` extiende la clase `DateFormat` y la misma utiliza el concepto de una cadena de patrón para dar formato la fecha y la hora. Se deben codificar el formato de la fecha o la hora usando letras mayúsculas y minúsculas para formar una fecha o hora, tal como el pattern lo determina.

```
String pattern = "dd-MM-yy"; /* d for day, M for month, y for year */
SimpleDateFormat formatter = new SimpleDateFormat(pattern);
// the default Date constructor initializes to current date/time
System.out.println(formatter.format(new Date()));
```

Localization :: Formateando I18N

SimpleDateFormat (date)

```
Date Pattern
G             Era             (BC/AD)
y                                Year
Y              Week              year
M Month (in year)
w        Week        (in        year)
W Week (in month)
D Day (in year)
d Day (in month)
F Day of week in month
E Day name in week
u Day number of week (1-7)
```

```
String [] dateFormats = {
            "dd-MM-yyyy", "d '('E')' MMM, YYYY", "w'th week of' YYYY",
      "EEEE, dd'th' MMMM, YYYY" };

Date today = new Date();
System.out.println("Default format for the date is " +
            DateFormat.getDateInstance().format(today));

for(String dateFormat : dateFormats) {
      System.out.printf("Date in pattern \"%s\" is %s %n", dateFormat,
      new SimpleDateFormat(dateFormat).format(today));
}
```

Localization :: Formateando I18N

SimpleDateFormat (time)

```
Time pattern
a Marker for the text am/pm marker
H    Hour    (value    range    0-23)
k    Hour    (value    range    1-24)K
K Hour in am/pm (value range 0-11)
h Hour in am/pm (value range 1-12)
m                              Minute
s Second
S                          Millisecond
Z  Time  zone  (general  time  zone
format)
```

```
String              []              timeFormats                =              {
      "h:mm",     "hh  'o''clock'",  "H:mm  a",  "hh:mm:ss:SS",  "K:mm:ss  a,
zzzz"                                                                        };
Date              today              =              new              Date();
System.out.println("Default    format    for    the    time    is    "    +
DateFormat.getTimeInstance().format(today));
for(String          timeFormat          :          timeFormats)          {
      System.out.printf("Time  in  pattern  \"%s\"  is  %s  %n",  timeFormat,
            new                SimpleDateFormat(timeFormat).format(today));
}
```