00.02 Java General

# 00.01 Ventanas de entrada de datos, mensajes y archivos

* **[Ventanas de diálogo con Swing](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/00dialogosenswing/" \l "ventanas-de-diálogo-con-swing)**
  + **[1. Paquetes principales del API](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/00dialogosenswing/" \l "1-paquetes-principales-del-api)**
  + **[2. Componentes de Swing](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/00dialogosenswing/" \l "2-componentes-de-swing)**
  + **[3. JOptionPane](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/00dialogosenswing/" \l "3-joptionpane)**
    - **[showMessageDialog](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/00dialogosenswing/" \l "showmessagedialog)**
    - **[showInputDialog](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/00dialogosenswing/" \l "showinputdialog)**
  + **[4. JFileChooser](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/00dialogosenswing/" \l "4-jfilechooser)**
    - **[Directorio de trabajo](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/00dialogosenswing/" \l "directorio-de-trabajo)**
    - **[Selección de archivos y/o directorios](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/00dialogosenswing/" \l "selección-de-archivos-yo-directorios)**
    - **[Filtros de archivos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/00dialogosenswing/" \l "filtros-de-archivos)**

## Ventanas de diálogo con Swing

**JFC (Java Foundation Classes)** es un conjunto de funciones/clases para **crear interfaces gráficas de usuario (GUI)** y añadir funcionalidad gráfica e interactividad a las aplicaciones Java que estudiaréis en la materia de ***Desenvolvemento de Interfaces***.

Incluye: **componentes gráficos de Swing** (botones, paneles, tablas, ventanas, etc.), **configuración de apariencia, API Java 2D, API de accesibilidad** (lectores de pantalla, Braille,…); **internacionalización** (gestión de idiomas del mundo,…)

En la materia de **Acceso a Datos** vamos a dar una **pequeña introducción a dos componentes de diálogo** que nos facilitarán la introducción de datos en la primera parte, mostrar mensajes o selección de ficheros hasta que lo estudiéis en otras materias:

* [***JOptionPane***](https://docs.oracle.com/en/java/javase/22/docs/api/java.desktop/javax/swing/JOptionPane.html)***:***
  + <https://docs.oracle.com/en/java/javase/22/docs/api/java.desktop/javax/swing/JOptionPane.html>
* [***JFileChooser***](https://docs.oracle.com/en/java/javase/22/docs/api/java.desktop/javax/swing/JFileChooser.html)
  + <https://docs.oracle.com/en/java/javase/22/docs/api/java.desktop/javax/swing/JFileChooser.html>

Una ventana de diálogo es una **subventana independiente** destinada a llevar un aviso temporal además de la ventana principal de la aplicación Swing.

Suelen usarse para **mostrar mensajes de error o una advertencia**, presentar imágenes, árboles de directorios, etc.

Para facilitar el trabajo, existen las clases de utilidad:

* [JOptionPane](https://docs.oracle.com/en/java/javase/22/docs/api/java.desktop/javax/swing/JOptionPane.html) para crear ventanas de **diálogos estándar y sencillos**.
* Barras de progreso: [JProgressBar](https://docs.oracle.com/en/java/javase/22/docs/api/java.desktop/javax/swing/JProgressBar.html) (ver también [ProgressMonitor](https://docs.oracle.com/en/java/javase/22/docs/api/java.desktop/javax/swing/ProgressMonitor.html))
* Elección de color: [JColorChooser](https://docs.oracle.com/en/java/javase/22/docs/api/java.desktop/javax/swing/JColorChooser.html).
* Selección de archivos: [JFileChooser](https://docs.oracle.com/javase/tutorial/uiswing/components/filechooser.html).
* Diálogos de impresión: [API de impresión](https://docs.oracle.com/javase/tutorial/2d/printing/index.html).

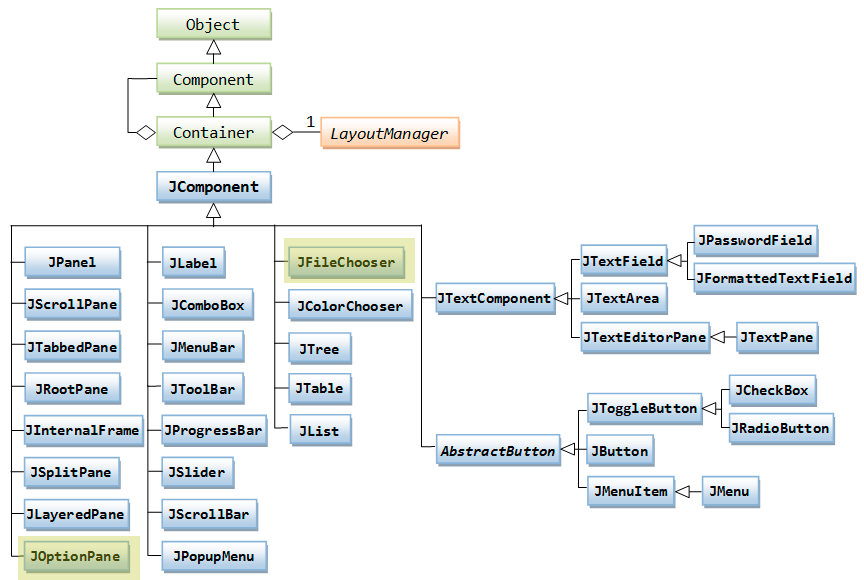
### 1. Paquetes principales del API

* javax.swing
* javax.swing.event

Además de:

|  |  |  |
| --- | --- | --- |
| javax.accessibility | javax.swing.plaf | javax.swing.text |
| **javax.swing** | javax.swing.plaf.basic | javax.swing.text.html |
| **javax.swing.colorchooser** | javax.swing.plaf.multi | javax.swing.text.rtf |
| **javax.swing.border** | javax.swing.plaf.metal | javax.swing.text.html.parser |
| **javax.swing.event** | javax.swing.plaf.synth | javax.swing.tree |
| **javax.swing.filechooser** | javax.swing.table | javax.swing.undo |

### 2. Componentes de Swing



### 3. JOptionPane

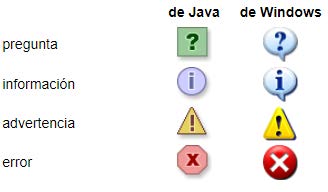
[**JoptionPane**](ttps://docs.oracle.com/en/java/javase/22/docs/api/java.desktop/javax/swing/JOptionPane.html)permite crear y personalizar rápidamente varios tipos diferentes de **cuadros de diálogo**.

Proporciona soporte para diseñar **cuadros de diálogo estándar**, con iconos, especificar el título y el texto del cuadro de diálogo y personalizar el texto del botón.

La compatibilidad con iconos de le **permite especificar fácilmente qué icono muestra el cuadro de diálogo.**

**Son modales**, esto es, bloquea el acceso a la ventana padre.

Puede utilizar un **icono personalizado**, ningún icono o cualquiera de los **cuatro iconos estándar**:

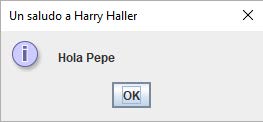
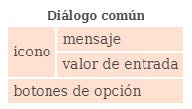


Cada apariencia tiene sus propias versiones de los cuatro íconos estándar.

La forma más sencilla de crear y mostrar diálogos con ***JoptionPane*** es por medio de los métodos ***showXxxDialog***:

|  |  |
| --- | --- |
| **Método** | **Descripción** |
| **showConfirmDialog** | Pregunta de confirmación, como sí/no/cancelar. |
| **showInputDialog** | Solicita entrada de datos. |
| **showMessageDialog** | Muestra un mensaje informativo |
| **showOptionDialog** | Permite personalizar el *JOptionPane*. |

*Existe una versión para marcos internos,* ***JinternalFrame*** *con métodos de la forma:* ***showInternalXxxx***

El formato del mensaje tiene una apariencia siguiendo la siguiente estructura:

Los parámetros de los métodos showXxxDialog son los siguientes:

* **parentComponent**: componente **padre**, el JFrame. Si el valor es **null**, se usa el JFrame por defecto y se centra en la pantalla.
* **Mensaje**: **mensaje** de la ventana de diálogo. **Normalmente String**, pero puede ser cualquier objeto:
  + **Object []**: será interpretado como una serie de mensajes (uno por objeto) situados en vertical.
  + **Component**: mostrará el componente en la ventana de diálogo.
  + **Icon**: el icono será mostrado dentro de un JLabel.
  + **Otros**: se convierten a String llamando al método toString() y mostrándolo dentro de un JLabel.
* **Título**: título de la ventana.
* **messageType**: define el estilo del mensaje. Los posibles valores son:
  + ERROR\_MESSAGE
  + INFORMATION\_MESSAGE
  + WARNING\_MESSAGE
  + QUESTION\_MESSAGE
  + PLAIN\_MESSAGE
* **optionType**: conjunto **botones** de opción que aparen debajo de la ventana. Se pueden proporcionar otro botones usando el parámetro options.
  + DEFAULT\_OPTION
  + YES\_NO\_OPTION
  + YES\_NO\_CANCEL\_OPTION
  + OK\_CANCEL\_OPTION
* **options**: es una descripción más detallada del **conjunto de botones** que aparecen en la parte inferior de la ventana. Lo usual es un array de String, pero puede ser un array de Object:
  + **Component**: el componente se añade a la lista de botones.
  + **Icon:** se crea un JButton con este icono.
  + **Otros**: el objeto se convierte en String (toString()) y se emplea como etiqueta del botón.
* **Icono**: **icono** de la ventana de diálogo. El valor por defecto está determinado por el tipo de mensaje.
* **InitialValue**: **valor por defecto** para ventanas de tipo input.

Los métodos **showXxxDialog** devuelven un **entero**, cuyos valores posibles son las constantes que referencian al botón pulsado:

* YES\_OPTION
* NO\_OPTION
* CANCEL\_OPTION
* OK\_OPTION
* CLOSED\_OPTION

Object[] opcionesBoton = {"Sí, por supuesto", "No, gracias", "No estoy loco!"};

int resultado = JOptionPane.showOptionDialog(this, "¿Te has vacunado de COVID?", // mensaje

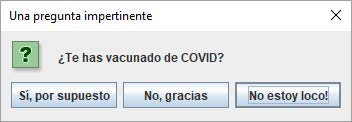
"Una pregunta impertinente", // título

JOptionPane.YES\_NO\_CANCEL\_OPTION, // OptionType

JOptionPane.QUESTION\_MESSAGE, // Tipo de mensaje null, // icono

opcionesBoton,

opcionesBoton[2]); // valor por defecto



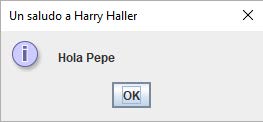
#### showMessageDialog

showMessageDialog : muestra un mensaje simple con un botón. (this es la referencia al formulario)

JOptionPane.showMessageDialog(this, "Hola Pepe",

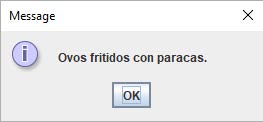
"Un saludo a Harry Haller",

JOptionPane.INFORMATION\_MESSAGE);



Con título e icono por defecto:

JOptionPane.showMessageDialog(this, "Ovos fritidos con paracas.");



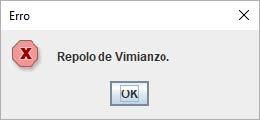
Con título, icono de aviso:

JOptionPane.showMessageDialog(this, "Arroz con chícharos.", "Aviso", JOptionPane.WARNING\_MESSAGE);

[](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/media/joptionpane4.jpg)

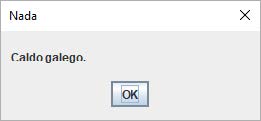
Con título, icono de error:

JOptionPane.showMessageDialog(this, "Repolo de Vimianzo.", "Erro", JOptionPane.ERROR\_MESSAGE);

[](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/media/joptionpane5.jpg)

Con título, sin icono:

JOptionPane.showMessageDialog(this, "Caldo galego.", "Nada", JOptionPane.PLAIN\_MESSAGE);

[](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/media/joptionpane6.jpg)

Con título, icono personalizado:

ImageIcon icoSalada = new ImageIcon( getClass().getResource("/images/ensalada.png"));

JOptionPane.showMessageDialog(this, "Salada con verde.", "Primeiro prato", JOptionPane.INFORMATION\_MESSAGE, iconaSalada);



#### showInputDialog

**showInputDialog** : es el único método **showXxxDialog** que no devuelve un entero.

**Devuelve un objeto**, normalmente un String:

Object[] platoFavorito = {"chícharos", "doce", "churros"};

String s = (String)JOptionPane.showInputDialog(this, "o meu prato favorito é: ",

"Introduce tu plato favorito", JOptionPane.PLAIN\_MESSAGE, iconaSalada, platoFavorito, "churros");

[](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/media/joptionpane8.jpg)

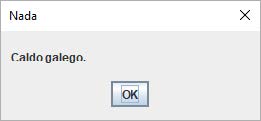
Si ponemos null en el array de opciones aparecerá una caja de texto:

**Devuelve un objeto**, normalmente un String:

String s = (String)JOptionPane.showInputDialog(this,

"o meu prato favorito é: ", "Introduce tu plato favorito", JOptionPane.PLAIN\_MESSAGE, iconaSalada,

null, "churros");



### 4. JFileChooser

Los **selectores de archivos proporcionan una GUI para navegar por el sistema de archivos y luego elegir un archivo o directorio de una lista**, o introducir el nombre de un archivo o directorio.  Normalmente usa la **clase JfileChooser para mostrar un cuadro de diálogo modal que contiene el selector de archivos**. Otra forma de presentar un selector de archivos es agregar una instancia de JFileChooser a un contenedor (ventana etc)

Ejemplo:

JFileChooser fc = new JFileChooser();

int returnVal = fc.showOpenDialog(this);

if (returnVal == JFileChooser.APPROVE\_OPTION) {

File file = fc.getSelectedFile();

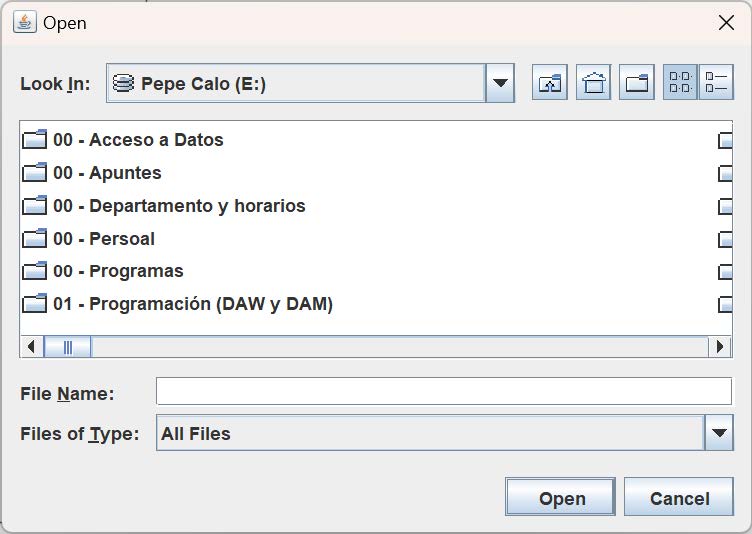
// Esto es lo que se hace con el archivo seleccionado

} else {

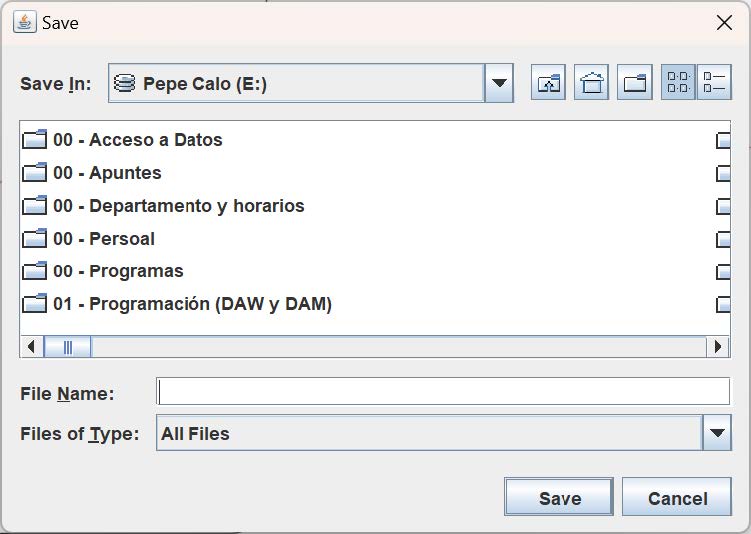
// Esto es lo que se hace si se cancela la selección

}

El método **showOpenDialog** muestra un cuadro de diálogo para abrir un archivo.



El método **showSaveDialog** muestra un cuadro de diálogo para guardar un archivo.



JFileChooser fcImaxe = new JFileChooser();

FileNameExtensionFilter filtro

= new FileNameExtensionFilter("Imágenes JPG y PNG", "jpg", "png");

fcImaxe.setFileFilter(filtro);

int valorSel = fcImaxe.showOpenDialog(null);

if (valorSel == JFileChooser.APPROVE\_OPTION){

System.out.println("Has seleccionado la imagen:" + fcImaxe.getSelectedFile().getName());

}

#### Directorio de trabajo

**ShowOpenDialog** recoge el **componente padre de la ventana de diálogo** y afecta a la posición de la ventana de diálogo.

Por **defecto muestra los archivos del directorio de trabajo del usuario**, pero puede especificarse el directorio inicial de varios modos:

* En el constructor:

JFileChooser fc = new JFileChooser("e:\\");

* Por medio del método:

fc.setCurrentDirectory(new File("e:\\"));

Selección de archivos y/o directorios

**showOpenDialog / showSaveDialog** recoge **devuelven un entero** que indica si se ha seleccionado un archivo: APPROVE\_OPTION o CANCEL\_OPTION

Una vez seleccionado un archivo o directorio (en ese caso debe indicarse que se permite selección de directorios) **puede invocarse al método getSelectedFile() para recuperar el archivo (File)**:

File arquivo = fcImaxe.getSelectedFile();

Una vez recuperado el archivo podemos obtener muchos datos del mismo (lo veremos en la unidad de archivos):

File arquivo = fcImaxe.getSelectedFile();

arquivo.getPath();

arquivo.getName();

arquivo.isDirectory();

arquivo.exists();

arquivo.delete();

// ...

Se puede utilizar **la misma instancia de la JFileChooser para mostrar un cuadro de diálogo estándar para guardar**.

int valor = fc.showSaveDialog(null);

Al utilizar la misma instancia del JFileChooser:

* **Recuerda el directorio actual entre usos**, por lo que las versiones para abrir y guardar comparten automáticamente el mismo directorio actual.
* **Sólo se personaliza un selector de archivos**, y las personalizaciones se aplican tanto a la versión para abrir como para guardar.

Se puede c**ambiar el modo de selección de archivos**, por ejemplo para seleccionar directorios:

fc.setFileSelectionMode(JFileChooser.DIRECTORIES\_ONLY);

Además, existen otros modos de selección:

* FILES\_AND\_DIRECTORIES.
* FILES\_ONLY (por defecto).
* DIRECTORIES\_ONLY.

#### Filtros de archivos

Por defecto, el **JfileChooser muestra todos los archivos y directorios, excepto los ocultos**.

Pueden programarse filtros de archivos para escoger algún tipo de archivo o directorio.

El JfileChooser **llama al método accept (de FileFilter) determina qué se mostrará**.

Existen varios tipos de filtros:

* **setFileHidingEnabled(false): para mostrar los archivos ocultos**.
* Subclases de la clase abstracta FileFilter: **FileNameExtensionFilter**: <https://docs.oracle.com/en/java/javase/22/docs/api/java.desktop/javax/swing/filechooser/FileNameExtensionFilter.html>

FileFilter filtro = new FileNameExtensionFilter("archivo JPEG", "jpg", "jpeg");

JFileChooser fc; // = ...;

fc.setFileFilter(filtro);

// fc.addChoosableFileFilter(filtro); // agrega a los seleccionables.

// fc. setAcceptAllFileFilterUsed(false);

# 00.02 Java Stream API

* **[Introducción](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/01streamapi/" \l "introducción)**
* **[1. Definición de Java Stream](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/01streamapi/" \l "1-definición-de-java-stream)**
* **[2. Procesamiento de Streams](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/01streamapi/" \l "2-procesamiento-de-streams)**
* **[3. Crear un Stream](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/01streamapi/" \l "3-crear-un-stream)**
* **[4. Operaciones Terminales y No Terminales](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/01streamapi/" \l "4-operaciones-terminales-y-no-terminales)**
* **[5. Operaciones No Terminales](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/01streamapi/" \l "5-operaciones-no-terminales)**
  + **[filter()](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/01streamapi/" \l "filter)**
  + **[map()](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/01streamapi/" \l "map)**
  + **[flatMap()](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/01streamapi/" \l "flatmap)**
  + **[distinct()](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/01streamapi/" \l "distinct)**
  + **[limit()](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/01streamapi/" \l "limit)**
  + **[peek()](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/01streamapi/" \l "peek)**
* **[6. Operaciones Terminales](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/01streamapi/" \l "6-operaciones-terminales)**
  + **[anyMatch()](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/01streamapi/" \l "anymatch)**
  + **[allMatch()](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/01streamapi/" \l "allmatch)**
  + **[noneMatch()](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/01streamapi/" \l "nonematch)**
  + **[findFirst()](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/01streamapi/" \l "findfirst)**
  + **[findAny()](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/01streamapi/" \l "findany)**
  + **[forEach()](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/01streamapi/" \l "foreach)**
  + **[collect()](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/01streamapi/" \l "collect)**
  + **[count()](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/01streamapi/" \l "count)**
  + **[reduce()](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/01streamapi/" \l "reduce)**
  + **[min() y max()](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/01streamapi/" \l "min-y-max)**
  + **[forEachOrdered()](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/01streamapi/" \l "foreachordered)**
* **[Resumen](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/01streamapi/" \l "resumen)**

## Introducción

La API de Streams de Java ofrece un enfoque funcional para el procesamiento de colecciones de objetos. Se introdujo en Java 8 junto con varias otras características de programación funcional. Este tutorial de Java Stream explicará cómo funcionan estos streams funcionales y cómo usarlos.

La API de Java Stream no está relacionada con [Java InputStream](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/InputStream.html) y [Java OutputStream](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/OutputStream.html)  de [Java IO](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/package-summary.html). InputStream y OutputStream se relacionan con flujos de bytes, mientras que la **API de Stream de Java se utiliza para procesar flujos de objetos**.

## 1. Definición de Java Stream

Un Stream en Java es un componente capaz de realizar una iteración interna de sus elementos, lo que significa que puede iterar sobre sus elementos por sí mismo. En contraste, al usar las características de iteración de [Java Collections](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/java-collections/index.html) (por ejemplo, un [Java Iterator](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/java-collections/iterator.html) o el bucle for-each de Java utilizado con un [Java Iterable](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/java-collections/iterable.html)), debes implementar la iteración de los elementos tú mismo.

## 2. Procesamiento de Streams

Puedes adjuntar oyentes a un Stream. Estos oyentes se llaman cuando el Stream itera internamente los elementos. Los oyentes se llaman una vez por cada elemento en el stream. De esta manera, cada oyente procesa cada elemento en el stream. Esto se denomina procesamiento de stream.

Los oyentes de un stream forman una cadena. El primer oyente en la cadena puede procesar el elemento en el stream y luego devolver un nuevo elemento para que el siguiente oyente en la cadena lo procese. Un oyente puede devolver el mismo elemento o uno nuevo, dependiendo del propósito de ese oyente (procesador).

## 3. Crear un Stream

Hay muchas formas de obtener un Stream en Java. Una de las formas más comunes de obtener un Stream es desde una [Java Collection](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Collection.html" \l "stream()). Un ejemplo de cómo obtener un Stream desde una [Java List](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/List.html):

List<String> items = new ArrayList<String>();

items.add("uno");

items.add("dos");

items.add("tres");

Stream<String> stream = items.stream();

Este ejemplo crea primero una lista de Java, luego agrega tres [Java Strings](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/java/strings.html)y, finalmente, llama al método stream() para obtener una instancia de Stream.

## 4. Operaciones Terminales y No Terminales

La interfaz Stream tiene una selección de operaciones terminales y no terminales. Una operación no terminal de stream es una operación que agrega un oyente al stream sin hacer nada más. Una operación terminal de stream es una operación que inicia la iteración interna de los elementos, llama a todos los oyentes y devuelve un resultado.

Un ejemplo de Java Stream que contiene tanto una operación no terminal como una terminal:

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Stream;

public class EjemplosDeStream {

public static void main(String[] args) {

List<String> listaDeCadenas = new

ArrayList<>();

listaDeCadenas.add("Uno");

listaDeCadenas.add("Dos");

listaDeCadenas.add("Tres");

// Operación no terminal (filter)

Stream<String> streamFiltrado = listaDeCadenas.stream().filter(s -> s.startsWith("T"));

// Operación terminal (forEach)

streamFiltrado.forEach(System.out::println);

}

}

En este ejemplo, filter es una operación no terminal que filtra elementos basándose en el predicado proporcionado. Luego, forEach es una operación terminal que itera sobre los elementos restantes y aplica la función de impresión.

## 5. Operaciones No Terminales

Las operaciones no terminales de stream devuelven un nuevo Stream. Estas operaciones no realizan ninguna iteración interna. Se ejecutan “perezosamente”, lo que significa que no realizan ninguna acción hasta que se activa una operación terminal.

### filter()

La operación filter es una **operación no terminal** que acepta un [Java Predicate](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/function/Predicate.html))como argumento y devuelve un nuevo Stream que contiene solo los elementos que cumplen con el predicado.

List<String> listaDeCadenas = Arrays.asList("Uno", "Dos", "Tres", "Cuatro", "Cinco");

// Filtrar elementos que comienzan con "C"

Stream<String> streamFiltrado = listaDeCadenas.stream().filter(s -> s.startsWith("C"));

// Imprimir elementos

streamFiltrado.forEach(System.out::println);

En este ejemplo, filter se utiliza para seleccionar solo las cadenas que comienzan con “C”.

### map()

La operación **map** es una **operación no termina** que transforma cada elemento del **Stream** utilizando la función proporcionada como argumento. Devuelve un nuevo **Stream** que contiene los elementos transformados:

<R> Stream<R> map(Function<? super T,? extends R> mapper)  
R apply(T t) // "método" de la interface Function

List<String> listaDeCadenas = Arrays.asList("apple", "banana", "orange");  
  
// Convertir a mayúsculas  
Stream<String> streamMayusculas = listaDeCadenas.stream().map(String::toUpperCase);  
  
// Imprimir elementos  
streamMayusculas.forEach(System.out::println);

En este ejemplo, map se utiliza para convertir cada cadena a mayúsculas.

### flatMap()

La operación flatMap es una operación no terminal que transforma cada elemento del Stream en cero o más elementos según la función proporcionada como argumento. Luego, combina los elementos resultantes en un solo Stream.

<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)  
R apply(T t)

List<List<Integer>> numeros = Arrays.asList(  
 Arrays.asList(1, 2, 3),  
 Arrays.asList(4, 5, 6),  
 Arrays.asList(7, 8, 9)  
);  
  
// Obtener un solo Stream de todos los números  
Stream<Integer> numerosStream = numeros.stream().flatMap(List::stream);  
  
// Imprimir elementos  
numerosStream.forEach(System.out::println);

En este ejemplo, **flatMap** se utiliza para obtener un solo **Stream** de todos los números en las listas anidadas.

### distinct()

La operación distinct es una operación no terminal que elimina los elementos duplicados del Stream, basándose en su implementación del método equals().

List<String> listaDeCadenas = Arrays.asList("apple", "banana", "orange", "banana", "apple");  
  
// Eliminar duplicados  
Stream<String> streamSinDuplicados = listaDeCadenas.stream().distinct();  
  
// Imprimir elementos  
streamSinDuplicados.forEach(System.out::println);

En este ejemplo, distinct se utiliza para eliminar las cadenas duplicadas.

### limit()

La operación limit es una operación no terminal que reduce la longitud del Stream a la cantidad especificada.

List<String> listaDeCadenas = Arrays.asList("apple", "banana", "orange", "grape");  
  
// Limitar a los primeros dos elementos  
Stream<String> streamLimitado = listaDeCadenas.stream().limit(2);  
  
// Imprimir elementos  
streamLimitado.forEach(System.out::println);

En este ejemplo, limit se utiliza para limitar el Stream a los primeros dos elementos.

### peek()

La operación peek es una operación no terminal que permite realizar un *side effect* en cada elemento del Stream, como imprimir el elemento antes de que se pase a la siguiente operación.

List<String> listaDeCadenas = Arrays.asList("apple", "banana", "orange");  
  
// Imprimir cada elemento antes de la transformación  
List<String> resultado = listaDeCadenas.stream()  
 .peek(System.out::println)  
 .map(String::toUpperCase)  
 .collect(Collectors.toList());

En este ejemplo, peek se utiliza para imprimir cada elemento antes de que se transforme a mayúsculas.

## 6. Operaciones Terminales

Las operaciones terminales de stream inician la iteración interna de los elementos y devuelven un resultado final. Después de que se realiza una operación terminal, un Stream no puede ser utilizado de nuevo.

### anyMatch()

La operación anyMatch es una operación terminal que devuelve true si al menos un elemento del Stream cumple con la condición dada, de lo contrario, devuelve false.

List<String> listaDeCadenas = Arrays.asList("apple", "banana", "orange");  
  
// Verificar si alguna cadena contiene "nan"  
boolean contieneNan = listaDeCadenas.stream().anyMatch(s -> s.contains("nan"));  
  
System.out.println(contieneNan); // Salida: true

En este ejemplo, anyMatch se utiliza para verificar si alguna cadena contiene la subcadena “nan”.

### allMatch()

La operación allMatch es una operación terminal que devuelve true si todos los elementos del Stream cumplen con la condición dada, de lo contrario, devuelve false.

List<String> listaDeCadenas = Arrays.asList("apple", "banana", "orange");  
  
// Verificar si todas las cadenas tienen longitud mayor que 3  
boolean todasLargas = listaDeCadenas.stream().allMatch(s -> s.length() > 3);  
  
System.out.println(todasLargas); // Salida: true

En este ejemplo, allMatch se utiliza para verificar si todas las cadenas tienen una longitud mayor que 3.

### noneMatch()

La operación noneMatch es una operación terminal que devuelve true si ninguno de los elementos del Stream cumple con la condición dada, de lo contrario, devuelve false.

List<String> listaDeCadenas = Arrays.asList("apple", "banana", "orange");  
  
// Verificar si ninguna cadena contiene "grape"  
boolean ningunaContieneUva = listaDeCadenas.stream().noneMatch(s -> s.contains("gr  
  
ape"));  
  
System.out.println(ningunaContieneUva); // Salida: true

En este ejemplo, noneMatch se utiliza para verificar si ninguna cadena contiene la subcadena “grape”.

### findFirst()

La operación findFirst es una operación terminal que devuelve el primer elemento del Stream en un Optional. Si el Stream está vacío, devuelve un Optional vacío.

List<String> listaDeCadenas = Arrays.asList("apple", "banana", "orange");  
  
// Obtener el primer elemento  
Optional<String> primerElemento = listaDeCadenas.stream().findFirst();  
  
primerElemento.ifPresent(System.out::println); // Imprimir el primer elemento si está presente

En este ejemplo, findFirst se utiliza para obtener el primer elemento de la lista.

### findAny()

La operación findAny es una operación terminal que devuelve cualquier elemento del Stream en un Optional. Si el Stream está vacío, devuelve un Optional vacío.

List<String> listaDeCadenas = Arrays.asList("apple", "banana", "orange");  
  
// Obtener cualquier elemento  
Optional<String> cualquierElemento = listaDeCadenas.stream().findAny();  
  
cualquierElemento.ifPresent(System.out::println); // Imprimir cualquier elemento si está presente

En este ejemplo, findAny se utiliza para obtener cualquier elemento de la lista.

### forEach()

La operación forEach es una operación terminal que ejecuta una acción para cada elemento del Stream.

List<String> listaDeCadenas = Arrays.asList("apple", "banana", "orange");  
  
// Imprimir cada elemento  
listaDeCadenas.stream().forEach(System.out::println);

En este ejemplo, forEach se utiliza para imprimir cada elemento de la lista.

### collect()

La operación collect es una operación terminal que transforma los elementos del Stream en una estructura de datos diferente, como una List, Set, o Map. Se utiliza junto con la interfaz Collector.

List<String> listaDeCadenas = Arrays.asList("apple", "banana", "orange");  
  
// Recoger elementos en una List  
List<String> listaRecolectada = listaDeCadenas.stream().collect(Collectors.toList());  
  
// Imprimir elementos  
listaRecolectada.forEach(System.out::println);

En este ejemplo, collect se utiliza para recolectar los elementos en una List.

### count()

La operación count es una operación terminal que devuelve el número de elementos en el Stream.

List<String> listaDeCadenas = Arrays.asList("apple", "banana", "orange");  
  
// Contar elementos  
long cantidad = listaDeCadenas.stream().count();  
  
System.out.println(cantidad); // Salida: 3

En este ejemplo, count se utiliza para contar el número de elementos en la lista.

### reduce()

La operación reduce es una operación terminal que combina los elementos del Stream en un solo resultado mediante una función asociativa y un valor identidad. Puede devolver un Optional si el Stream está vacío.

List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);  
  
// Sumar todos los elementos  
Optional<Integer> suma = numeros.stream().reduce(Integer::sum);  
  
suma.ifPresent(System.out::println); // Imprimir la suma si está presente

En este ejemplo, reduce se utiliza para sumar todos los elementos de la lista.

### min() y max()

Las operaciones min y max son operaciones terminales que devuelven el elemento mínimo y máximo del Stream, respectivamente, basándose en el orden natural o un comparador proporcionado.

List<Integer> numeros = Arrays.asList(3, 1, 4, 1, 5, 9, 2, 6, 5);  
  
// Encontrar el mínimo y el máximo  
Optional<Integer> minimo = numeros.stream().min(Comparator.naturalOrder());  
Optional<Integer> maximo = numeros.stream().max(Comparator.naturalOrder());  
  
minimo.ifPresent(System.out::println); // Imprimir el mínimo si está presente  
maximo.ifPresent(System.out::println); // Imprimir el máximo si está presente

En este ejemplo, min y max se utilizan para encontrar el elemento mínimo y máximo de la lista.

### forEachOrdered()

La operación forEachOrdered es similar a forEach, pero garantiza que los elementos se procesen en orden en un Stream paralelo.

List<String> listaDeCadenas = Arrays.asList("apple", "banana", "orange");  
  
// Imprimir cada elemento en orden en un Stream paralelo  
listaDeCadenas.parallelStream().forEachOrdered(System.out::println);

En este ejemplo, forEachOrdered se utiliza para imprimir cada elemento en orden en un Stream paralelo.

## Resumen

En Java, los Streams proporcionan una forma declarativa y funcional de procesar colecciones de datos. Las operaciones de stream se dividen en dos categorías: operaciones no terminales y operaciones terminales. Las operaciones no terminales se componen para formar una cadena de operaciones que se ejecutan perezosamente. La ejecución se inicia solo cuando se encuentra una operación terminal.

Las operaciones no terminales, como filter, map, flatMap, distinct, limit, y peek, permiten filtrar, transformar y manipular los elementos del Stream. Las operaciones terminales, como anyMatch, allMatch, noneMatch, findFirst, findAny, forEach, collect, count, reduce, min, max, y forEachOrdered, producen un resultado final o realizan una acción final en los elementos del Stream.

Al aprovechar las operaciones de stream, puedes escribir código más conciso, legible y eficiente cuando trabajas con colecciones de datos en Java.

# 00.03 DESCARGA DE ARCHIVOS DE INTERNET

* [1. Introducción](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/02urlfiles/#1-introducci%C3%B3n)
* [2. Java IO](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/02urlfiles/#2-java-io)
* [3. Java NIO](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/02urlfiles/#3-java-nio)

## 1. Introducción

Muchos de los siguientes ejercicios trabajan con archivos JSON de APIs JSON públicas o abiertas de Internet.

Existen varios modos de acceder a recursos, archivos, de Internet:

* Java IO.
* Java NIO.
* Bibliotecas externas como AsyncHttpClient y Apache Commons IO.

## 2. Java IO

La API más básica que podemos usar para descargar un archivo es Java IO. Podemos utilizar la clase URL para abrir una conexión al archivo que queremos descargar.

Para leer de manera eficaz el archivo, podemos utilizar el método openStream() para obtener un InputStream:

BufferedInputStream in = new BufferedInputStream(  
 new URL(FILE\_URL).openStream())

o:

URL url = new URL(FILE\_URL);  
URLConnection conn = url.openConnection();  
  
BufferedReaderBufferedReader br = new BufferedReader(  
 new InputStreamReader(conn.getInputStream()));

La ventaja de este método es que podemos pasarle parámetros (cookies, tokens, identificadores de sesión…) a la cabecera HTTP:

URL url = new URL(FILE\_URL);  
HttpURLConnection urlc = (HttpURLConnection) url.openConnection();   
urlc.setInstanceFollowRedirects(true);  
urlc.setRequestProperty("User-Agent", "");  
urlc.connect();  
  
BufferedReader in =   
new BufferedReader(new InputStreamReader(urlc.getInputStream()));

Al leer desde un InputStream, se recomienda encapsularlo en un BufferedInputStream para aumentar el rendimiento.

Como hemos visto, la mejora de rendimiento proviene del almacenamiento en búfer. Al leer un byte a la vez mediante el método read(), cada llamada al método implica una llamada al sistema al sistema de archivos subyacente. Cuando la JVM invoca la llamada al sistema read(), el contexto de ejecución del programa cambia de modo usuario a modo kernel y viceversa. Este cambio de contexto es costoso desde una perspectiva de rendimiento. Al leer un gran número de bytes, el rendimiento de la aplicación será deficiente debido al gran número de cambios de contexto involucrados.

Para escribir los bytes leídos desde la URL en nuestro archivo local, utilizaremos el método write() de la clase FileOutputStream:

try (BufferedInputStream in = new BufferedInputStream(  
 new URL(FILE\_URL).openStream());  
 FileOutputStream fileOutputStream =   
 new FileOutputStream(FILE\_NAME)) {  
 byte dataBuffer[] = new byte[1024];  
 int bytesRead;  
 while ((bytesRead = in.read(dataBuffer, 0, 1024)) != -1) {  
 fileOutputStream.write(dataBuffer, 0, bytesRead);  
 }  
} catch (IOException e) {  
 // gestión de la excepción  
}

Cuando se utiliza un BufferedInputStream, el método read() leerá tantos bytes como hayamos establecido para el tamaño del búfer. En nuestro ejemplo, ya estamos haciendo esto al leer bloques de 1024 bytes a la vez, por lo que BufferedInputStream no es necesario. Para archivos JSON, como son archivos de texto, precisamos convertir el flujo entrada (InputStream) en un Reader por medio de la clase InputStreamReader, pasando a leer línea a línea.

Muchos métodos de procesado de archivos JSON (fromJson, parse,…) tienen una versión sobrecargada que recoge un Reader además de un String.

## 3. Java NIO

En el caso anterior, bajamos a nivel de flujo pero, como hemos estudiado, a partir de Java 7, se dispone de la clase Files que contiene métodos auxiliares para manejar operaciones de entrada/salida (IO).

Se puede utilizar el método Files.copy() para leer todos los bytes de un InputStream y copiarlos a un archivo local:

InputStream in = new URL(FILE\_URL).openStream();  
Files.copy(in, Paths.get(FILE\_NAME),   
 StandardCopyOption.REPLACE\_EXISTING);

El ejemplo anterior funciona bien, pero puede mejorarse. El principal ¿inconveniente? es que los bytes se almacenan en búfer en la memoria.

Java NIO tiene métodos para transferir bytes directamente entre dos canales sin almacenamiento en búfer.

El paquete Java NIO ofrece la posibilidad de transferir bytes entre dos canales sin almacenarlos en el espacio de memoria de la aplicación.

Para leer el archivo desde nuestra URL, crearemos un ReadableByteChannel a partir del flujo de URL:

ReadableByteChannel readableByteChannel =  
 Channels.newChannel(url.openStream());

Los bytes leídos del ReadableByteChannel se transferirán a un FileChannel correspondiente al archivo que se va a descargar:

FileOutputStream fileOutputStream = new FileOutputStream(FILE\_NAME);  
FileChannel fileChannel = fileOutputStream.getChannel();

Puede usarse el método transferFrom() de la clase ReadableByteChannel para descargar los bytes desde la URL dada a nuestro FileChannel:

fileChannel.transferFrom(readableByteChannel,   
 0, Long.MAX\_VALUE);

Los métodos transferTo() y transferFrom() son más eficientes que simplemente leer desde un flujo utilizando un búfer. Dependiendo del sistema operativo subyacente, los datos pueden transferirse directamente desde la caché del sistema de archivos a nuestro archivo sin copiar ningún byte en el espacio de memoria de la aplicación.

En sistemas Linux y UNIX, estos métodos utilizan la técnica de copia cero que reduce el número de cambios de contexto entre el modo kernel y el modo usuario.

# 00.04 IMPLEMENTACIÓN DE HASHCODE()

* [Sobreescribe hashCode cuando sobreescribes equals](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/02javageneral/03hashcode/#sobreescribe-hashcode-cuando-sobreescribes-equals)

## Sobreescribe hashCode cuando sobreescribes equals

Debes seimpre sobreescribir hashCode en cada clase que sobrescriba equals. Si no lo haces se violará el contrato general de hashCode, lo que **hará que NO funcione correctamente en colecciones como HashMap y HashSet**. Aquí está el contrato, adaptado de la especificación de Object:

* Cuando se invoca el método hashCode en un objeto repetidamente durante la ejecución de una aplicación, **debe devolver consistentemente el mismo valor**, siempre que la información utilizada en las comparaciones de equals no se modifique. Este valor no necesita ser consistente de una ejecución de una aplicación a otra.
* Si **dos objetos son iguales según el método equals(Object), llamar a hashCode en los dos objetos debe producir el mismo resultado entero**.
* Si dos objetos son diferentes según el método equals(Object), no es necesario que llamar a hashCode en cada uno de los objetos produzca resultados distintos. Sin embargo, el programador **debe ser consciente de que producir resultados distintos para objetos diferentes puede mejorar el rendimiento de las tablas hash**.

La disposición clave que se viola cuando no se sobrescribe hashCode es la segunda: **los objetos iguales deben tener códigos hash iguales**.  
Dos instancias distintas pueden ser lógicamente iguales según el método equals de una clase, pero el método hashCode de Object, son sólo dos objetos con poco en común. Por lo tanto, el método hashCode de Object devuelve dos números aparentemente aleatorios en lugar de dos números iguales, como exige el contrato.

Por ejemplo, añadimos una instancia de la clase NumeroTelefono como clave en un HashMap:

Map<NumeroTelefono, String> m = new HashMap<>();  
m.put(new NumeroTelefono(707, 867, 5309), "Otto");

En este punto, podrías esperar que m.get(new NumeroTelefono(707, 867, 5309)) devuelva “Otto”, pero en cambio, devuelve null. Observa que hay dos instancias de NumeroTelefono involucradas: una se utiliza para la inserción en el HashMap, y una segunda instancia igual se utiliza para la recuperación (intentada). La falta de sobrescritura de hashCode en la clase NumeroTelefono hace que las dos instancias iguales tengan códigos hash diferentes, violando el contrato de hashCode. Por lo tanto, es probable que el método get busque el número de teléfono en una tabla hash diferente al que fue almacenado por el método put. Incluso si las dos instancias resultan en el mismo cubo hash, es muy probable que el método get devuelva null, porque HashMap tiene una optimización que almacena en caché el código hash asociado con cada entrada y no se molesta en verificar la igualdad de objetos si los códigos hash no coinciden.

Para solucionar este problema, es tan sencillo como escribir un método hashCode adecuado para NumeroTelefono. Entonces, ¿cómo debería implementarse un método hashCode? Es fácil escribir uno malo. Este, por ejemplo, siempre es legal pero nunca debería ser usado:

// La peor implementación legal posible de hashCode: ¡nunca uses esto!  
@Override  
public int hashCode() {  
 return 42;  
}

Es **legal porque asegura que los objetos iguales tengan el mismo código hash**. Es muy nmalo porque asegura que cada objeto tenga el mismo código hash. Por lo tanto, cada objeto se asigna a la misma tabla hash, y las tablas hash degeneran en listas enlazadas. Los programas que deberían ejecutarse en tiempo lineal en su lugar se ejecutan en tiempo cuadrático. **Para tablas hash grandes, esta es la diferencia entre funcionar y no funcionar**.

**Una buena función de hash tiende a producir códigos hash diferentes para instancias diferentes**. Esto es exactamente lo que se entiende por la tercera parte del contrato de hashCode. Idealmente, una función de hash debería distribuir uniformemente cualquier colección razonable de instancias diferentes en todos los valores int. Lograr este ideal puede ser difícil, pero afortunadamente, no es demasiado difícil lograr una aproximación justa. Aquí tienes una receta sencilla:

1. Declara una variable int llamada **resultado y inicialízala con el código hash c para el primer campo significativo en tu objeto**, calculado en el paso 2.1. (Un campo significativo es un campo que afecta las comparaciones de equals).
2. Para cada campo significativo restante f en tu objeto, realiza lo siguiente:
   1. 2.1 Calcula un **código hash c para el campo**:
      1. i. Si el campo es de un tipo primitivo, calcula Type.hashCode(f), donde Type es la clase primitiva encapsulada correspondiente al tipo de f.
      2. Si el campo es una referencia a un objeto y el método equals de esta clase compara el campo mediante la invocación recursiva de equals, invoca recursivamente hashCode en el campo. Si se requiere una comparación más compleja, calcula una “representación canónica” para este campo e invoca hashCode en la representación canónica. Si el valor del campo es null, utiliza 0 (u otra constante, pero 0 es tradicional).
      3. Si el campo es un array, trátalo como si cada elemento significativo fuera un campo separado. Es decir, calcula un código hash para cada elemento significativo aplicando estas reglas recursivamente y combina los valores según el paso 2.b. Si el array no tiene elementos significativos, utiliza una constante, preferiblemente no 0. Si todos los elementos son significativos, utiliza Arrays.hashCode.
   2. Combina el código hash c calculado en el paso 2.a en resultado de la siguiente manera:

resultado = 31 \* resultado + c;

1. Devuelve resultado.

Cuando hayas terminado de escribir el método hashCode, pregúntate a ti mismo si las instancias iguales tienen códigos hash iguales. Escribe pruebas unitarias para verificar tu intuición (a menos que hayas utilizado AutoValue para generar tus métodos equals y hashCode, en cuyo caso puedes omitir estas pruebas de manera segura). Si las instancias iguales tienen códigos hash diferentes, averigua por qué y soluciona el problema.

Puedes excluir campos derivados del cálculo del código hash. En otras palabras, puedes ignorar cualquier campo cuyo valor pueda calcularse a partir de campos incluidos en el cálculo. Debes excluir cualquier campo que no se utilice en comparaciones de equals, o corres el riesgo de violar la segunda disposición del contrato de hashCode.

La multiplicación en el paso 2.b hace que el resultado dependa del orden de los campos, lo que produce una función de hash mucho mejor si la clase tiene múltiples campos similares. Por ejemplo, si se omitiera la multiplicación de una función de hash de String, todos los anagramas tendrían códigos hash idénticos. Se eligió el valor 31 porque es un número primo impar. Si fuera par y la multiplicación desbordara, se perdería información, ya que la multiplicación por 2 es equivalente a un desplazamiento. La ventaja de usar un número primo es menos clara, pero es tradicional. Una propiedad agradable del 31 es que la multiplicación se puede reemplazar por un desplazamiento y una resta para obtener un mejor rendimiento en algunas arquitecturas: 31 \* i == (i << 5) - i. Las VM modernas realizan este tipo de optimización automáticamente.

Aplicaremos la receta anterior a la clase NumeroTelefono:

// Método típico de hashCode  
@Override  
public int hashCode() {  
 int resultado = Short.hashCode(codigoArea);  
 resultado = 31 \* resultado + Short.hashCode(prefijo);  
 resultado = 31 \* resultado + Short.hashCode(numeroT);  
 return resultado;  
}

Dado que este método devuelve el resultado de un cálculo determinista simple cuyas únicas entradas son los tres campos significativos en una instancia de NumeroTelefono, es evidente que las instancias iguales de NumeroTelefono tienen códigos hash iguales. Este método es, de hecho, una implementación de hashCode perfectamente buena para NumeroTelefono, al nivel de las bibliotecas de plataformas Java. Es simple, bastante rápido y hace un trabajo razonable al dispersar números de teléfono diferentes en diferentes cubos hash.

Si bien esta estrategia produce funciones de hash bastante buenas, no son de última generación. Son comparables en calidad a las funciones de hash que se encuentran en los tipos de valor de las bibliotecas de plataformas Java y son adecuadas para la mayoría de los usos. Si realmente necesitas funciones de hash menos propensas a producir colisiones, consulta la clase com.google.common.hash.Hashing de Guava [Guava].

**La clase Objects tiene un método estático que recoge un número arbitrario de objetos y devuelve un código hash para ellos**. Este método, llamado **hash, te permite escribir métodos hashCode de una línea** cuya calidad es comparable a los declarados anteriormente. Desafortunadamente, se ejecutan más lentamente porque implican la creación de un array para pasar un número variable de argumentos, así como el boxing y unboxing si alguno de los argumentos es de tipo primitivo. Este estilo de función de hash se recomienda **sólo en situaciones donde el rendimiento no es crítico**. Aquí tienes una función de hash para NumeroTelefono escrita utilizando esta técnica:

// Método hashCode de una línea - rendimiento mediocre  
@Override  
public int hashCode() {  
 return Objects.hash(numeroT, prefijo, codigoArea);  
}

Si una clase es inmutable y el costo de calcular el código hash es significativo, podrías considerar almacenar en caché el código hash en el objeto en lugar de recalcularlo cada vez que se solicita. Si crees que la mayoría de los objetos de este tipo se utilizarán como claves hash, deberías calcular el código hash cuando se crea la instancia. De lo contrario, podrías optar por inicializar perezosamente el código hash la primera vez que se invoca hashCode. Se requiere cierto cuidado para asegurar que la clase siga siendo segura para subprocesos en presencia de un campo inicializado de forma perezosa. Nuestra clase NumeroTelefono no merece este tratamiento, pero solo para mostrarte cómo se hace, aquí está. Ten en cuenta que el valor inicial para el campo hashCode (en este caso, 0) no debería ser el código hash de una instancia comúnmente creada:

// Método hashCode con caché de código hash inicializado perezosamente  
private int hashCode; // Inicializado automáticamente a 0  
@Override  
public int hashCode() {  
 int resultado = hashCode;  
 if (resultado == 0) {  
 resultado = Short.hashCode(codigoArea);  
 resultado = 31 \* resultado + Short.hashCode(prefijo);  
 resultado = 31 \* resultado + Short.hashCode(numeroT);  
 hashCode = resultado;  
 }  
 return resultado;  
}

No te dejes tentar a excluir campos significativos del cálculo del código hash para mejorar el rendimiento. Aunque la función de hash resultante puede ejecutarse más rápido, su baja calidad puede degradar el rendimiento de las tablas hash hasta el punto en que se vuelven inutilizables. En particular, la función de hash puede enfrentarse a una gran colección de instancias que difieren principalmente en las regiones que has elegido ignorar. Si esto sucede, la función de hash asignará todas estas instancias a unos pocos códigos hash, y los programas que deberían ejecutarse en tiempo lineal en su lugar se ejecutarán en tiempo cuadrático.

Esto no es solo un problema teórico. Antes de Java 2, la función de hash de String utilizaba como máximo dieciséis caracteres distribuidos uniformemente en toda la cadena, comenzando por el primer carácter. Para grandes colecciones de nombres jerárquicos, como las URL, esta función mostraba exactamente el comportamiento patológico descrito anteriormente.

No proporciones una especificación detallada para el valor devuelto por hashCode, de modo que los clientes no puedan depender razonablemente de él; esto te da la flexibilidad para cambiarlo. Muchas clases en las bibliotecas de Java, como String e Integer, especifican el valor exacto devuelto por su método hashCode como una función del valor de la instancia. Esto no es una buena idea, sino un error con el que nos vemos obligados a vivir: obstaculiza la capacidad de mejorar la función de hash en futuras versiones. Si dejas los detalles sin especificar y se encuentra un defecto en la función de hash o se descubre una función de hash mejor, puedes cambiarla en una versión posterior.

En resumen, debes sobrescribir hashCode cada vez que sobres equals, o tu programa no funcionará correctamente. Tu método hashCode debe obedecer el contrato general especificado en Object y debe hacer un trabajo razonable asignando códigos hash diferentes a instancias diferentes. Esto es fácil de lograr, aunque ligeramente tedioso, utilizando la receta en la página 51. Como se menciona en el Ítem 10, el framework AutoValue proporciona una excelente alternativa para escribir manualmente los métodos equals y hashCode, y los IDE también ofrecen parte de esta funcionalidad.