# Programación Java SE 7

**Volumen II • Guía del alumno**

D67238CS20
Edición 2.0
Noviembre de 2011
D81764

ORACLE

## Authors

Michael Williams

Tom McGinn

Matt Heimer

## Technical Contributors and Reviewers

Lee Klement

Steve Watts

Brian Earl

Vasily Strelnikov

Andy Smith

Nancy K.A.N

Chris Lamb

Todd Lowry

Ionut Radu

Joe Darcy

Brian Goetz

Alan Bateman

David Holmes

## Editors

Richard Wallis

Daniel Milne

Vijayalakshmi Narasimhan

## Graphic Designer

James Hans

## Publishers

Syed Imtiaz Ali

Sumesh Koshy

# Contenido

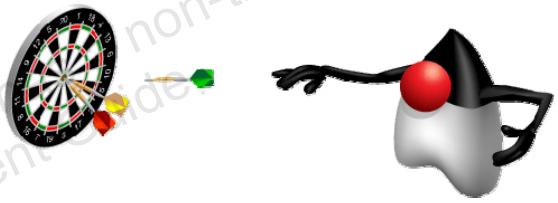
# 9

# Excepciones y afirmaciones

# Objetivos

Al finalizar esta lección, debería estar capacitado para:

- Definir el objetivo de las excepciones de Java
- Utilizar las sentencias `try` y `throw`
- Utilizar las cláusulas `catch`, multi-`catch` y `finally`
- Cerrar automáticamente recursos con una sentencia `try`-with-resources
- Reconocer categorías y clases de excepciones comunes
- Crear excepciones personalizadas y recursos que se puedan cerrar automáticamente
- Probar invariantes a través de afirmaciones

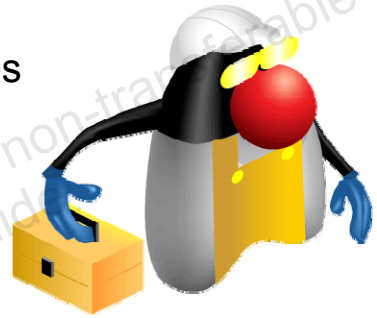


ORACLE

# Manejo de errores

Las aplicaciones encontrarán errores durante su ejecución.
Una aplicación fiable debe manejar los errores lo mejor posible.
Los errores:

- Deben ser la “excepción” y no el comportamiento esperado
- Deben poder manejarse para crear aplicaciones fiables
- Se pueden producir como resultado de bugs en las aplicaciones
- Se pueden producir debido a factores más allá del control de la aplicación
  - Bases de datos inaccesibles
  - Fallo del disco duro



ORACLE







### Devolución de un resultado de fallo

Algunos lenguajes de programación utilizan el valor de retorno de un método para indicar si el método se ha completado correctamente o no. En el ejemplo C `int x = printf("hi");`, un valor negativo en `x` indicaría un fallo. Muchas de las funciones de la biblioteca estándar de C devuelven un valor negativo en caso de fallo. El problema es que el ejemplo anterior se podría escribir también como `printf("hi");`, donde se ignora el valor de retorno. En Java también existe el mismo problema: los valores de retorno se pueden ignorar.

Cuando esté creando un método en lenguaje Java y no consiga ejecutarlo correctamente, sepa que puede recurrir a las funciones de generación y manejo de excepciones disponibles en el lenguaje, en lugar de usar valores de retorno.

## Manejo de excepciones en Java

Cuando se usan bibliotecas de Java que se basan en recursos externos, el compilador le exigirá que “maneje o declare” las excepciones que se puedan producir.

- Manejar una excepción significa que hay que agregar un bloque de código para manejar el error.
- Declarar una excepción significa que se declara que un método puede fallar y no ejecutarse correctamente.

ORACLE







### La regla de manejo o de declaración

Muchas bibliotecas exigirán conocer el proceso de manejo de excepciones. Estas incluyen:

- E/S de archivos (NIO: `java.nio`)
- Acceso a base de datos (JDBC: `java.sql`)

Manejar una excepción significa utilizar una sentencia `try-catch` para transferir el control a un bloque de manejo de excepciones cuando se produzca una excepción. Declarar una excepción significa agregar una cláusula `throws` a una declaración de método para indicar que el método puede fallar cuando se ejecute de un modo concreto. En otras palabras, "manejar" implica que la responsabilidad es suya, mientras que "declarar" implica que la responsabilidad es de otra persona.

# La sentencia `try-catch`

La sentencia `try-catch` se utiliza para manejar excepciones.

```
try {
    System.out.println("About to open a file");
    InputStream in =
        new FileInputStream("missingfile.txt");
    System.out.println("File open");
} catch (Exception e) {
    System.out.println("Something went wrong!");
}
```

Se omite la línea si la línea anterior no pudo abrir el archivo.

Esta línea solo se ejecuta si se produjo algún error en el bloque `try`.







## La cláusula `catch`

Si se produce una excepción dentro de un bloque `try`, la ejecución se transferirá al bloque `catch` asociado. Se omitirán todas las líneas dentro del bloque `try` que aparezcan detrás de la excepción, y no se ejecutarán. La cláusula `catch` debe usarse para:

- Volver a intentar la operación
- Probar una operación alternativa
- Cerrar o volver sin generar un error

Evite dejar bloques `catch` vacíos. Ocultar sin más las excepciones no es buena práctica.

# Objetos Exception

A las cláusulas `catch` se les pasan referencias a objetos `java.lang.Exception`. La clase `java.lang.Throwable` es la clase principal para `Exception`, y describe varios métodos utilizables.

```
try{
    //...
} catch (Exception e) {
    System.out.println(e.getMessage());
    e.printStackTrace();
}
```

ORACLE







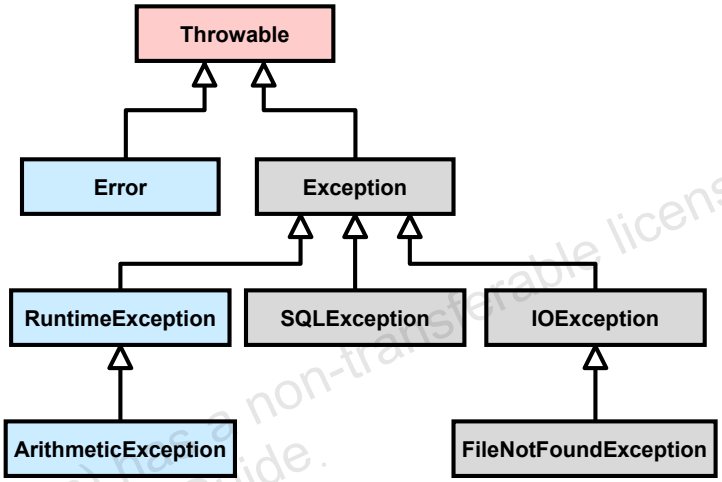
### Registro de excepciones

Cuando se produzcan errores en su aplicación, a menudo le vendrá bien registrar qué ha pasado. Los desarrolladores de Java tienen a su disposición varias bibliotecas de registro, incluida Log4j de Apache y el marco de registro incorporado `java.util`. Si bien estas bibliotecas de registro no forman parte del objeto de estudio de este curso, verá que algunos IDE, como NetBeans, recomiendan eliminar todas las llamadas a `printStackTrace()`. Esto es así porque las aplicaciones con calidad de producción deben emplear una biblioteca de registro, en lugar de generar mensajes de depuración en la pantalla.

# Categorías de excepciones

La clase `java.lang.Throwable` conforma la base de la jerarquía de clases de excepciones. Hay dos categorías de excepciones principales:

- Excepciones comprobadas, las cuales se deben “manejar o declarar”.
- Excepciones no comprobadas, que normalmente no se “manejan o declaran”.




ORACLE







## Gestión de excepciones

Cuando se genera un objeto `Exception` y se pasa a una cláusula `catch`, se instancia desde una clase que representa el tipo concreto de problema que se ha producido. Estas clases relacionadas con las excepciones se pueden dividir en dos categorías: comprobadas y no comprobadas.

### Excepciones no comprobadas

`java.lang.RuntimeException`, `java.lang.Error` y sus subclases se categorizan como excepciones no comprobadas. Estos tipos de excepciones no deberían producirse normalmente al ejecutarse la aplicación. Se puede usar una sentencia `try-catch` para detectar más fácilmente el origen de estas excepciones, pero cuando una aplicación está lista para producción, deberá quedar poco código para gestionar `RuntimeException` y sus subclases. Las subclases Error representan errores que no puede corregir, como que la máquina JVM se esté quedando sin memoria. Entre las excepciones `RuntimeException` con las que habitualmente tendrá que solucionar problemas se encuentran:

- `ArrayIndexOutOfBoundsException`: acceso a un elemento de matriz que no existe.
- `NullPointerException`: uso de una referencia que no apunta a ningún objeto.
- `ArithmeticException`: división entre cero.

# Prueba

Las excepciones `NullPointerException` se deben capturar mediante una sentencia `try-catch`.

a. Verdadero
b. Falso

# Prueba

¿Cuál de los siguientes tipos son todas excepciones probadas `instancesof`?

a. `Error`
b. `Throwable`
c. `RuntimeException`
d. `Exception`

# Manejo de excepciones

Debe siempre capturar el tipo de excepción más específico. Se pueden asociar varios bloques catch con una única sentencia try.

```
try {
    System.out.println("About to open a file");
    InputStream in = new FileInputStream("missingfile.txt");
    System.out.println("File open");
    int data = in.read();
    in.close();
} catch (FileNotFoundException e) {
    System.out.println(e.getClass().getName());
    System.out.println("Quitting");
} catch (IOException e) {
    System.out.println(e.getClass().getName());
    System.out.println("Quitting");
}
```

El orden es importante. Primero es necesario capturar las excepciones más específicas (es decir, las clases secundarias antes que las clases principales).

ORACLE







### Excepciones comprobadas

Todas las clases que son subclase de `Exception`, excepto `RuntimeException` y sus subclases, entran dentro de la categoría de excepciones comprobadas. Estas excepciones se deben “manejar o declarar” con una sentencia `try` o `throws`. La documentación HTML de una API Java (Javadoc) describirá qué excepciones comprobadas se pueden generar mediante un método o un constructor y por qué.

Capturar el tipo más específico de excepción le permite escribir bloques `catch` destinados a manejar tipos de errores muy específicos. Debe evitar capturar el tipo base de `Exception`, ya que es difícil crear un bloque `catch` con finalidad general que pueda gestionar todos los errores posibles.

**Nota:** las excepciones devueltas por Java Persistence API (JPA) amplían `RuntimeException`, y por ello se categorizan como excepciones sin comprobar. Estas excepciones necesitan “manejarse o declararse” en código listo para producción, incluso aunque no sea necesario hacerlo mediante el compilador.

# La cláusula `finally`

```
InputStream in = null;
try {
    System.out.println("About to open a file");
    in = new FileInputStream("missingfile.txt");
    System.out.println("File open");
    int data = in.read();
} catch (IOException e) {
    System.out.println(e.getMessage());
} finally {
    try {
        if(in != null) in.close();
    } catch (IOException e) {
        System.out.println("Failed to close file");
    }
}
```

Las cláusulas `finally` se ejecutan con independencia de si se ha generado o no un objeto `Exception`.

Siempre hay que cerrar los recursos abiertos.

ORACLE







### Cierre de recursos

Cuando se abren recursos, como archivos o conexiones a bases de datos, siempre deben cerrarse una vez dejan de hacer falta. Intentar cerrar estos recursos dentro del bloque `try` puede ser problemático, ya que se puede terminar omitiendo la operación de cierre. Los bloques `finally` se ejecutan siempre, independientemente de si se ha producido o no un error al ejecutarse el bloque `try`. Si el control salta a un bloque `catch`, el bloque `finally` se ejecutará después del bloque `catch`.

En ocasiones, la operación que desea realizar en el bloque `finally` puede provocar ella misma la generación del objeto `Exception`. En ese caso, tal vez sea necesario anidar una sentencia `try-catch` dentro de un bloque `finally`. También se puede anidar una sentencia `try-catch` dentro de bloques `try` y `catch`.

# La sentencia `try`-with-resources

Java SE 7 proporciona una nueva sentencia `try`-with-resources para cerrar automáticamente recursos.

```
System.out.println("About to open a file");
try (InputStream in =
      new FileInputStream("missingfile.txt")) {
    System.out.println("File open");
    int data = in.read();
} catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```





### Recursos que se pueden cerrar

La sentencia `try-with-resources` puede evitar tener que usar un largo bloque `finally`. Los recursos abiertos con la sentencia `try`-with-resources se cierran siempre. Las clases que implantan `java.lang.AutoCloseable` se pueden utilizar como recurso. Si un recurso se tiene que cerrar automáticamente, su referencia se deberá declarar dentro de los paréntesis de la sentencia `try`.

Si van separados por punto y coma, se pueden abrir varios recursos. Si abre varios recursos, se cerrarán en el orden inverso a su apertura.

# Excepciones suprimidas

Si se produce una excepción en el bloque `try` de una sentencia `try`-with-resources *y* se produce una excepción mientras se cierran los recursos, las excepciones resultantes se suprimirán.

```
} catch(Exception e) {
    System.out.println(e.getMessage());
    for(Throwable t : e.getSuppressed()) {
        System.out.println(t.getMessage());
    }
}
```

ORACLE







### Excepciones en recursos

Si se produce una excepción al crear el recurso `AutoCloseable`, el control saltará inmediatamente a un bloque `catch`.

Si se produce una excepción en el cuerpo del bloque `try`, se cerrarán todos los recursos *antes de* que se ejecute el bloque `catch`. Si se genera una excepción mientras se cierran los recursos, se suprimirá.

Si el bloque `try` se ejecuta sin excepciones, pero se genera una excepción al cerrar un recurso, el control saltará a un bloque `catch`.

# La interfaz de `AutoCloseable`

Un recurso en una sentencia `try`-with-resources debe implantar una de las siguientes opciones:

- `java.lang.AutoCloseable`
  - Nueva en JDK 7
  - Puede devolver un objeto `Exception`
- `java.io.Closeable`
  - Refactorizada en JDK7 para ampliar `AutoCloseable`
  - Puede devolver un objeto `IOException`

```
public interface AutoCloseable {
    void close() throws Exception;
}
```

ORACLE







**Cierre frente a cierre automático**

La documentación de la API de Java dice lo siguiente sobre `AutoCloseable`: "Tenga en cuenta que, al contrario que el método de cierre `Closeable`, este método de cierre *no* tiene que ser idempotente. En otras palabras, llamar a este método de cierre más de una vez puede tener un efecto secundario visible, a diferencia de `Closeable.close`, que no puede tener ningún efecto si se le llama más de una vez. No obstante, se anima encarecidamente a los implantadores de esta interfaz a hacer sus métodos de cierre idempotentes."

# Captura de varias excepciones

Java SE 7 proporciona una nueva cláusula multi-`catch`.

```
ShoppingCart cart = null;
try (InputStream is = new FileInputStream(cartFile);
     ObjectInputStream in = new ObjectInputStream(is)) {
     cart = (ShoppingCart)in.readObject();
} catch (ClassNotFoundException | IOException e) {
    System.out.println("Exception deserializing " + cartFile);
    System.out.println(e);
    System.exit(-1);
}
```

Cuando hay varios tipos de excepciones, se separan con una barra vertical.

ORACLE







### Las ventajas de multi-`catch`

A veces interesa realizar la misma acción, independientemente de la excepción que se esté generando. La nueva cláusula multi-`catch` reduce la cantidad de código que hay que escribir gracias a que elimina la necesidad de incluir varias cláusulas `catch` con el mismo comportamiento.

Otra ventaja de la cláusula multi-`catch` es que reduce la posibilidad de intentar capturar una excepción genérica. La captura de objetos `Exception` impide ver otros tipos de excepciones que se pueden generar mediante código agregado más tarde a un bloque `try`.

Los diferentes tipos separados por barras verticales no pueden tener una relación de herencia. No se pueden incluir ambas `FileNotFoundException` y `IOException` dentro de una cláusula multi-`catch`.

La E/S de archivos y la serialización de objetos se tratan en la lección sobre los "conceptos fundamentales de E/S de Java".

# Declaración de excepciones

Puede declarar que un método devuelve una excepción en lugar de manejarla.

```
public static int readByteFromFile() throws IOException {
    try (InputStream in = new FileInputStream("a.txt")) {
        System.out.println("File open");
        return in.read();
    }
}
```

Observe la falta de cláusulas `catch`. La sentencia `try`-with-resources se está utilizando solo para cerrar recursos.



Mediante la cláusula `throws`, un método puede declarar que devuelve una o más excepciones durante su ejecución. Si se genera una excepción mientras se ejecuta el método, este dejará de ejecutarse y se devolverá la excepción al emisor. Los métodos sustituidos pueden declarar las mismas excepciones, menos excepciones o excepciones más específicas, pero no más excepciones ni más genéricas. Un método puede declarar varias excepciones mediante una lista separada por comas.

```
public static int readByteFromFile() throws FileNotFoundException,
IOException {
    try (InputStream in = new FileInputStream("a.txt")) {
        System.out.println("File open");
        return in.read();
    }
}
```

Si bien técnicamente no necesita declarar `FileNotFoundException`, ya que es una subclase de `IOException`, es buena práctica hacerlo.

# Manejo de excepciones declaradas

Las excepciones que pueden devolver los métodos deben manejarse. Al declarar una excepción simplemente se especifica que es otra persona la que tiene que manejarla.

```
public static void main(String[] args) {
   try {
        int data = readByteFromFile();
    } catch (IOException e) {
        System.out.println(e.getMessage());
    }
}
```

Método que ha declarado una excepción





### Manejo de excepciones

Su aplicación debe siempre manejar sus excepciones. Cuando se agrega una cláusula `throws` a un método, simplemente se retrasa el manejo de la excepción. De hecho, las excepciones se pueden devolver repetidamente a la pila de llamadas. Las aplicaciones Java SE estándar deben manejar las excepciones antes de sacarlas del método `main`; de no hacerlo, se corre el riesgo de que el programa termine anormalmente. Se puede declarar que `main` devuelva una excepción, pero, a menos que esté diseñando programas que quiera que terminen de forma abrupta, deberá evitar hacerlo.

# Devolución de excepciones

Puede volver a emitir una excepción capturada previamente. Tenga en cuenta que hay una cláusula `throws` y una sentencia `throw`.

```
public static int readByteFromFile() throws IOException {
    try (InputStream in = new FileInputStream("a.txt")) {
        System.out.println("File open");
        return in.read();
    } catch (IOException e) {
        e.printStackTrace();
        throw e;
    }
}
```

ORACLE







### Reemisión precisa

Java SE 7 permite volver a emitir el tipo de excepción precisa. El ejemplo que sigue no se compilaría con Java SE 6, ya que la cláusula `catch` recibe un objeto `Exception`, pero el método devuelve un objeto `IOException`. Para obtener más información acerca de la nueva función de reemisión precisa, consulte http://download.oracle.com/javase/7/docs/technotes/guides/language/catch-multiple.html#rethrow.

```
public static int readByteFromFile() throws IOException {
    try {
        InputStream in = new FileInputStream("a.txt");
        System.out.println("File open");
        return in.read();
    } catch (Exception e) {
        e.printStackTrace();
        throw e;
    }
}
```

# Excepciones personalizadas

Puede crear clases de excepciones personalizadas extendiendo `Exception` o una de sus subclases.

```
public class DAOException extends Exception {

    public DAOException() {
        super();
    }

    public DAOException(String message) {
        super(message);
    }
}
```





Las bibliotecas de clases de Java estándar nunca devuelven excepciones personalizadas. Para hacer uso de una clase de excepción personalizada, deberá devolverla usted mismo. Por ejemplo:

```
throw new DAOException();
```

Una clase de excepción personalizada puede sustituir métodos o agregar nuevas funcionalidades. Las reglas de herencia son las mismas, incluso aunque el tipo de clase principal sea una excepción.

Dado que las excepciones capturan información sobre un problema que se ha producido, es posible que necesite agregar campos y métodos dependiendo del tipo de información que sea necesario capturar. Si una cadena puede capturar toda la información necesaria, puede usar el método `getMessage()` que todas las clases `Exception` heredan de `Throwable`. Los constructores de `Exception` que reciban una cadena la almacenarán para que `getMessage()` la devuelva.

# Prueba

¿Qué palabra clave usaría para agregar una cláusula a un método que indique que el método puede generar una excepción?

a. `throw`
b. `thrown`
c. `throws`
d. `assert`

# Excepciones de envoltorio

Utilice una excepción de envoltorio para ocultar el tipo de excepción que se esté generando sin simplemente ocultar la excepción.

```
public class DAOException extends Exception {
    public DAOException(Throwable cause) {
        super(cause);
    }

    public DAOException(String message, Throwable cause)
    {
        super(message, cause);
    }
}
```

ORACLE







### Obtención de la causa

La clase `Throwable` contiene un método `getCause()` que se puede usar para recuperar una excepción de envoltorio.

```
try {
    //…
} catch (DAOException e) {
    Throwable t = e.getCause();
}
```

# Revisión del patrón DAO

El patrón DAO utiliza la abstracción (una interfaz) para permitir sustituir la implantación. Un DAO de base de datos o archivo debe gestionar excepciones. Una implantación de DAO puede usar una excepción de envoltorio para preservar la abstracción y evitar ocultar excepciones.

```
public Employee findById(int id) throws DAOException {
    try {
        return employeeArray[id];
    } catch (ArrayIndexOutOfBoundsException e) {
     throw new DAOException("Error finding employee in DAO", e);
    }
}
```

ORACLE







### Excepciones DAO

Un DAO basado en archivo debe gestionar excepciones `IOException` y un DAO basado en JDBC debe gestionar excepciones `SQLException`. Si estos tipos de excepciones los devolvió un DAO, los clientes se vincularán con una implantación en lugar de con una abstracción. Al modificar la interfaz de DAO e implantar clases para devolver una excepción de envoltorio (`DAOException`), podrá conservar la abstracción y los clientes podrán saber cuándo encuentra un problema la implantación de DAO.

# Afirmaciones

- Utilice afirmaciones para documentar y verificar las suposiciones y la lógica interna de un único método:
  - Invariantes internas
  - Invariantes de flujo de control
  - Condiciones posteriores e invariantes de clases
- Usos de afirmaciones no adecuados

  Las afirmaciones se pueden desactivar en el tiempo de ejecución; por ello:
  - No utilice afirmaciones para comprobar los parámetros de un método público.
  - No utilice métodos que puedan generar efectos secundarios en la comprobación de la afirmación.

ORACLE







**Por qué utilizar afirmaciones**

Puede utilizar afirmaciones para agregar código a sus aplicaciones que garantice que la aplicación se está ejecutando según lo esperado. El uso de afirmaciones le permite probar diferentes fallos en las condiciones. En caso de fallo, se termina la aplicación y se muestra información de depuración. No deben usarse afirmaciones si las comprobaciones tienen que ejecutarse siempre, puesto que la comprobación de las afirmaciones se puede desactivar.

## Sintaxis de las afirmaciones

- La sintaxis de una afirmación es la siguiente:

```
assert <expresión_booleana> ;
assert <expresión_booleana> : <expresión_detalle> ;
```

- Si *<expresión_booleana>* se evalúa en *false*, entonces se devuelve `AssertionError`.
- El segundo argumento se convierte a una cadena y se utiliza como texto descriptivo en el mensaje `AssertionError`.



### La sentencia `assert`

Las afirmaciones combinan el mecanismo de manejo de excepciones de Java con código que se ejecuta condicionalmente. A continuación se incluye un ejemplo de pseudocódigo del comportamiento de afirmaciones:

```
if (AssertionsAreEnabled) {
      if (condition == false) throw new AssertionError();
}
```

`AssertionError` es una subclase de `Error` y, por lo tanto, entra en la categoría de excepciones sin comprobar.

# Invariantes internas

- El problema es:

```
1   if (x > 0) {
2     // hacer esto
3   } else {
4     // hacer eso
5   }
```

- La solución es:

```
1   if (x > 0) {
2     // hacer esto
3   } else {
4     assert ( x == 0 );
5     // hacer eso, salvo que x sea negativo
6   }
```

# Invariantes de flujo de control

Ejemplo:

```
1 switch (suit) {
2       case Suit.CLUBS: // ...
3         break;
4       case Suit.DIAMONDS: // ...
5         break;
6       case Suit.HEARTS: // ...
7         break;
8       case Suit.SPADES: // ...
9         break;
10    default: assert false : "Palo de cartas desconocido";
11        break;
12    }
```

# Condiciones posteriores e invariantes de clases

Ejemplo:

```
1    public Object pop() {
2      int size = this.getElementCount();
3      if (size == 0) {
4        throw new RuntimeException("Intento de extracción de montón vacío");
5      }
6
7      Object result = /* código para recuperar elemento extraído */ ;
8
9      // prueba de la condición posterior
10     assert (this.getElementCount() == size - 1);
11
12   return result;
13   }
```

ORACLE

# Control de evaluación de tiempo de ejecución de afirmaciones

- Si se ha desactivado la comprobación de afirmaciones, el código se ejecuta igual de rápido que si las comprobaciones no hubieran estado allí nunca.
- La comprobación de afirmaciones está desactivada por defecto. Active las afirmaciones con cualquiera de los siguientes comandos:

```
java -enableassertions MyProgram
```

```
java -ea MyProgram
```

- La comprobación de las afirmaciones se puede controlar por clases, paquetes y jerarquías de paquetes. Consulte: http://download.oracle.com/javase/7/docs/technotes/guides/language/assert.html

ORACLE

# Prueba

¿Deben utilizarse afirmaciones para validar entradas del usuario?

a. Verdadero
b. Falso

# Resumen

En esta lección, debe haber aprendido a hacer lo siguiente:

- Definir el objetivo de las excepciones de Java
- Utilizar las sentencias `try` y `throw`
- Utilizar las cláusulas `catch`, multi-`catch` y `finally`
- Cerrar automáticamente recursos con una sentencia `try`-with-resources
- Reconocer categorías y clases de excepciones comunes
- Crear excepciones personalizadas y recursos que se puedan cerrar automáticamente
- Probar invariantes a través de afirmaciones

# Visión general de la práctica 9-1: Captura de excepciones

En esta práctica, se abordan los siguientes temas:

- Adición de sentencias `try-catch` a una clase
- Manejo de excepciones

ORACLE



En esta práctica, escribirá código para gestionar excepciones comprobadas y no comprobadas.

# Visión general de la práctica 9-2: Ampliación del objeto `Exception`

En esta práctica, se abordan los siguientes temas:

- Adición de sentencias `try-catch` a una clase
- Manejo de excepciones
- Ampliación de la clase `Exception`
- Creación de un recurso personalizado con cierre automático
- Uso de una sentencia `try`-with-resources
- Devolución de excepciones con `throw` y `throws`



En esta práctica, actualizará una implantación de patrón DAO para usar una excepción de envoltorio personalizada.

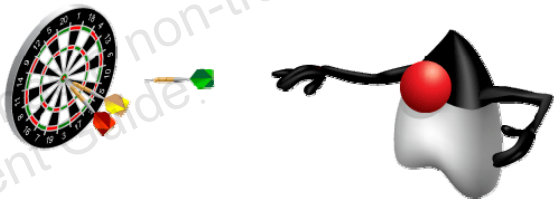
# 10

# Conceptos fundamentales de E/S en Java

# Objetivos

Al finalizar esta lección, debería estar capacitado para:

- Describir los aspectos básicos de entrada y salida en Java
- Leer datos de la consola y escribir datos en ella
- Utilizar flujos para leer y escribir datos
- Leer y escribir objetos mediante serialización



ORACLE

# Conceptos básicos de E/S en Java

El lenguaje de programación Java proporciona un completo juego de bibliotecas para realizar funciones de entrada/salida (E/S).

- Java define los canales de E/S como flujos.
- Un flujo de E/S representa un origen de entrada o un destino de salida.
- Los flujos pueden representar muchos tipos de orígenes y destinos diferentes, como archivos de disco, dispositivos, otros programas y matrices de memoria.
- Los flujos admiten muchos tipos de datos diferentes, como bytes simples, tipos de datos primitivos, caracteres localizados y objetos.

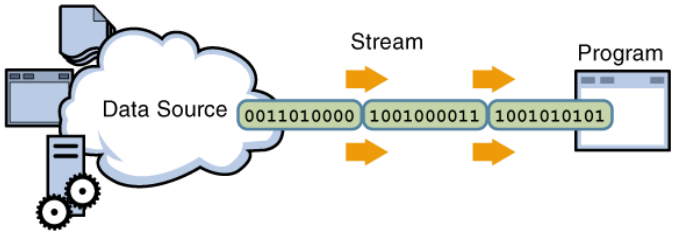
ORACLE



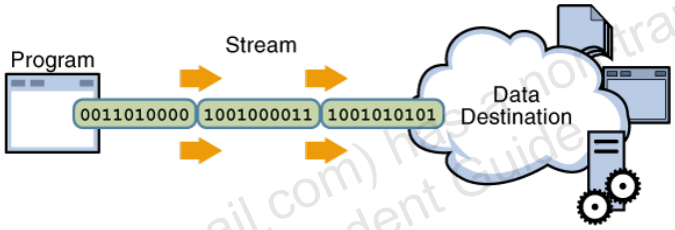
Algunos flujos simplemente pasan datos, mientras que otros manipulan y transforman los datos de formas útiles.

# Flujos de E/S

- Los programas utilizan flujos de entrada para leer datos desde un origen un elemento cada vez.




- Los programas utilizan flujos de salida para escribir datos en un destino (receptor) un elemento cada vez.




ORACLE



Independientemente de cómo trabajen internamente, todos los flujos presentan el mismo modelo simple a los programas que los usan: un flujo es una secuencia de datos.

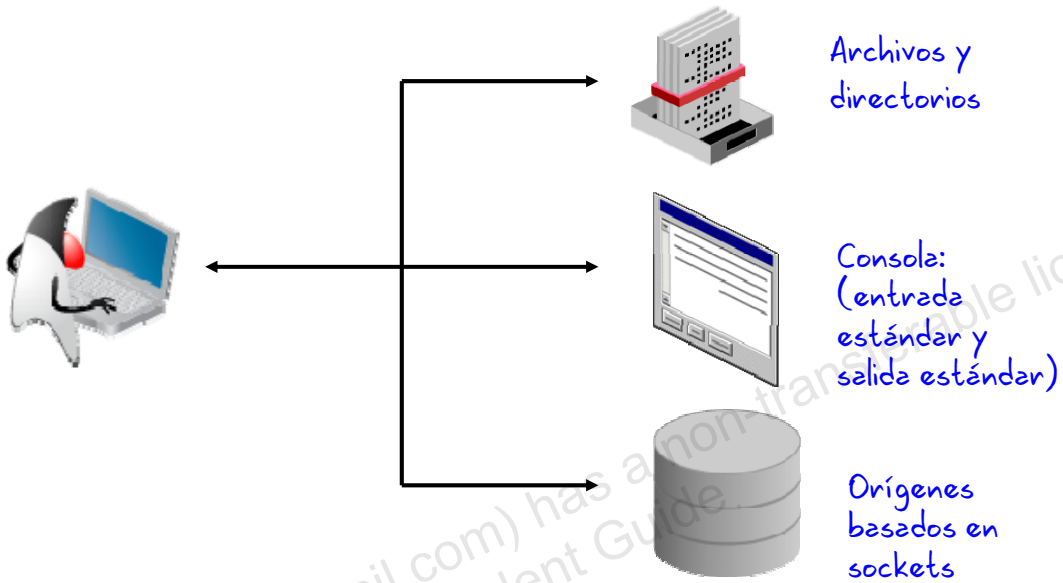
Un flujo es un flujo de datos. Un flujo puede proceder de un origen o se puede generar en un receptor.

- Un flujo de origen inicia el flujo de datos, llamado también flujo de entrada.
- Un flujo de receptor termina el flujo de datos, llamado también flujo de salida.

Los orígenes y los receptores son ambos flujos de nodo. Los tipos de flujos de nodo son archivos, memoria y canales entre threads o procesos.

# Aplicación de E/S

Generalmente existen tres maneras en las que un desarrollador puede usar la entrada y la salida:




ORACLE



Los desarrolladores de aplicaciones generalmente utilizan flujos de E/S para leer y escribir archivos, para leer y escribir información en y desde algunos dispositivos de salida, como el teclado (entrada estándar) y la consola (salida estándar). Por último, es posible que una aplicación necesite utilizar un socket para comunicarse con otra aplicación en un sistema remoto.

# Datos dentro de flujos

- La tecnología Java soporta dos tipos de flujos: de caracteres y de bytes.
- La entrada y la salida de datos de caracteres se maneja a través de lectores y escritores.
- La entrada y la salida de datos de bytes se maneja a través de flujos de entrada y flujos de salida:
  - Normalmente, el término *flujo* hace referencia a un flujo de bytes.
  - Los términos *lector* y *escritor* hacen referencia a flujos de caracteres.

| Flujo | Flujos de bytes | Flujos de caracteres |
|---|---|---|
| Flujos de origen | `InputStream` | `Reader` |
| Flujos de receptor | `OutputStream` | `Writer` |

ORACLE



La tecnología Java soporta dos tipos de flujos de entrada de datos: bytes no procesados y caracteres Unicode. Generalmente, el término *flujo* hace referencia a flujos de bytes, y los términos *lector* y *escritor* hacen referencia a flujos de caracteres.

Más concretamente, los flujos de entrada de bytes se implantan mediante subclases de la clase `InputStream`, y los flujos de salida de bytes se implantan mediante subclases de la clase `OutputStream`. Los flujos de entrada de caracteres se implantan mediante subclases de la clase `Reader`, y los flujos de salida de caracteres se implantan mediante subclases de la clase `Writer`.

Los flujos de bytes se aplican mejor a la lectura y escritura de bytes no procesados (como archivos de imágenes, archivos de audio y objetos). Las diferentes subclases ofrecen métodos para proporcionar soporte específico para uno de estos tipos de flujos.

Los flujos de caracteres se diseñan para leer caracteres (como los incluidos en archivos y otros flujos basados en caracteres).

# Métodos `InputStream` de flujos de bytes

- Los tres métodos básicos `read` son:

```
int read()
int read(byte[] buffer)
int read(byte[] buffer, int offset, int length)
```

- Otros métodos incluyen:

```
void close();              // Close an open stream
int available();           // Number of bytes available
long skip(long n);         // Discard n bytes from stream

boolean markSupported();   //
void mark(int readlimit);  // Push-back operations
void reset();              //
```

ORACLE







### Métodos `InputStream`

El método `read()` devuelve un valor `int` que contiene un byte leído del flujo, o un valor `-1` que indica la condición de fin de archivo. Los otros dos métodos de lectura leen el flujo en una matriz de bytes y devuelven el número de bytes leídos. Los dos argumentos `int` del tercer método indican un subrango en la matriz de destino que debe rellenarse.

**Nota:** para una mayor eficiencia, lea siempre los datos en el bloque práctico más grande posible, o use flujos en buffer.

Cuando haya terminado con un flujo, ciérrelo. Si tiene una pila de flujos, utilice flujos de filtro para cerrar el flujo en la parte superior de la pila. Esta operación también cierra los flujos inferiores.

**Nota:** en Java SE 7, `InputStream` implanta `AutoCloseable`, lo cual significa que, si usa una clase `InputStream` (o una de sus subclases) en un bloque `try`-with-resources, el flujo se cierra automáticamente al final de la sentencia try.

El método `available` informa sobre el número de bytes inmediatamente disponibles para leerse en el flujo. La operación de lectura real que siga a esta llamada puede devolver más bytes.

El método `skip` desecha el número específico de bytes del flujo.

Los métodos `markSupported()`, `mark()` y `reset()` realizan operaciones de retroceso en un flujo, si el flujo en cuestión lo soporta. El método `markSupported()` devuelve true si los métodos `mark()` y `reset()` están operativos para ese flujo concreto. El método `mark(int)` indica que debe anotarse el punto actual en el flujo y se debe asignar un buffer lo suficientemente grande como para soportar, como mínimo, el número de bytes del argumento especificado. El parámetro del método `mark(int)` especifica el número de bytes que se pueden volver a leer llamando a `reset()`. Tras las operaciones `read()` posteriores, cuando se llama al método `reset()`, se devuelve el flujo de entrada al punto marcado. Si se lee más allá del buffer marcado, `reset()` no tendrá ningún significado.

## Métodos `OutputStream` de flujos de bytes

- Los tres métodos básicos `write` son:

```
void write(int c)
void write(byte[] buffer)
void write(byte[] buffer, int offset, int length)
```

- Otros métodos incluyen:

```
void close(); // Automatically closed in try-with-resources
void flush(); // Force a write to the stream
```



Métodos **`OutputStream`**

Como con entradas, procure siempre escribir los datos en el bloque práctico más grande posible.

# Ejemplo de flujo de bytes

```
1 import java.io.FileInputStream; import java.io.FileOutputStream;
2 import java.io.FileNotFoundException; import java.io.IOException;
3
4 public class ByteStreamCopyTest {
5     public static void main(String[] args){
6         byte[] b = new byte[128]; int bLen = b.length;
7         // Example use of InputStream methods
8         try (FileInputStream fis = new FileInputStream (args[0]);
9              FileOutputStream fos = new FileOutputStream (args[1])) {
10             System.out.println ("Bytes available: " + fis.available());
11            int count = 0; int read = 0;
12            while (fis.read(b) != -1) {
13                if (read < bLen) fos.write(b, 0, read);
14                else fos.write(b);
15                count += read;
16             }
17             System.out.println ("Wrote: " + count);
18         } catch (FileNotFoundException f) {
19             System.out.println ("File not found: " + f);
20         } catch (IOException e) {
21             System.out.println ("IOException: " + e);
22         }
23     }
24 }
```

Tenga en cuenta que es necesario saber cuántos bytes se leen cada vez en la matriz de bytes.

ORACLE



Este ejemplo copia un archivo en otro a través de una matriz de bytes. Tenga en cuenta que `FileInputStream` y `FileOutputStream` se usan en un principio para flujos de bytes no procesados, como archivos de imágenes.

**Nota:** el método `available()`, según la documentación de Java, informa sobre "una estimación del número de bytes restantes que pueden leerse (u omitirse) desde este flujo de entrada sin bloques".

# Métodos `Reader` de flujos de caracteres

- Los tres métodos básicos `read` son:

```
int read()
int read(char[] cbuf)
int read(char[] cbuf, int offset, int length)
```

- Otros métodos incluyen:

```
void close()
boolean ready()
long skip(long n)
boolean markSupported()
void mark(int readAheadLimit)
void reset()
```

ORACLE







### Métodos `Reader`

El primer método devuelve un valor `int` que contiene un carácter Unicode leído del flujo, o un valor `-1` que indica la condición de fin de archivo. Los otros dos métodos leen de una matriz de caracteres y devuelven el número de bytes leídos. Los dos argumentos `int` del tercer método indican un subrango en la matriz de destino que debe rellenarse.

# Métodos `Writer` de flujos de caracteres

- Los métodos básicos `write` son:

```
void write(int c)
void write(char[] cbuf)
void write(char[] cbuf, int offset, int length)
void write(String string)
void write(String string, int offset, int length)
```

- Otros métodos incluyen:

```
void close()
void flush()
```





### Métodos `Writer`

Estos métodos son análogos a los métodos `OutputStream`.

# Ejemplo de flujo de caracteres

```
1 import java.io.FileReader; import java.io.FileWriter;
2 import java.io.IOException; import java.io.FileNotFoundException;
3
4 public class CharStreamCopyTest {
5     public static void main(String[] args){
6         char[] c = new char[128]; int cLen = c.length;
7         // Example use of InputStream methods
8         try (FileReader fr = new FileReader(args[0]);
9              FileWriter fw = new FileWriter(args[1])) {
10             int count = 0;
11             int read = 0;
12             while ((read = fr.read(c)) != -1) {
13                 if (read < cLen) fw.write(c, 0, read);
14                 else fw.write(c);
15                 count += read;
16             }
17             System.out.println("Wrote: " + count + " characters.");
18         } catch (FileNotFoundException f) {
19             System.out.println("File " + args[0] + " not found.");
20         } catch (IOException e) {
21             System.out.println("IOException: " + e);
22         }
23     }
24 }
```

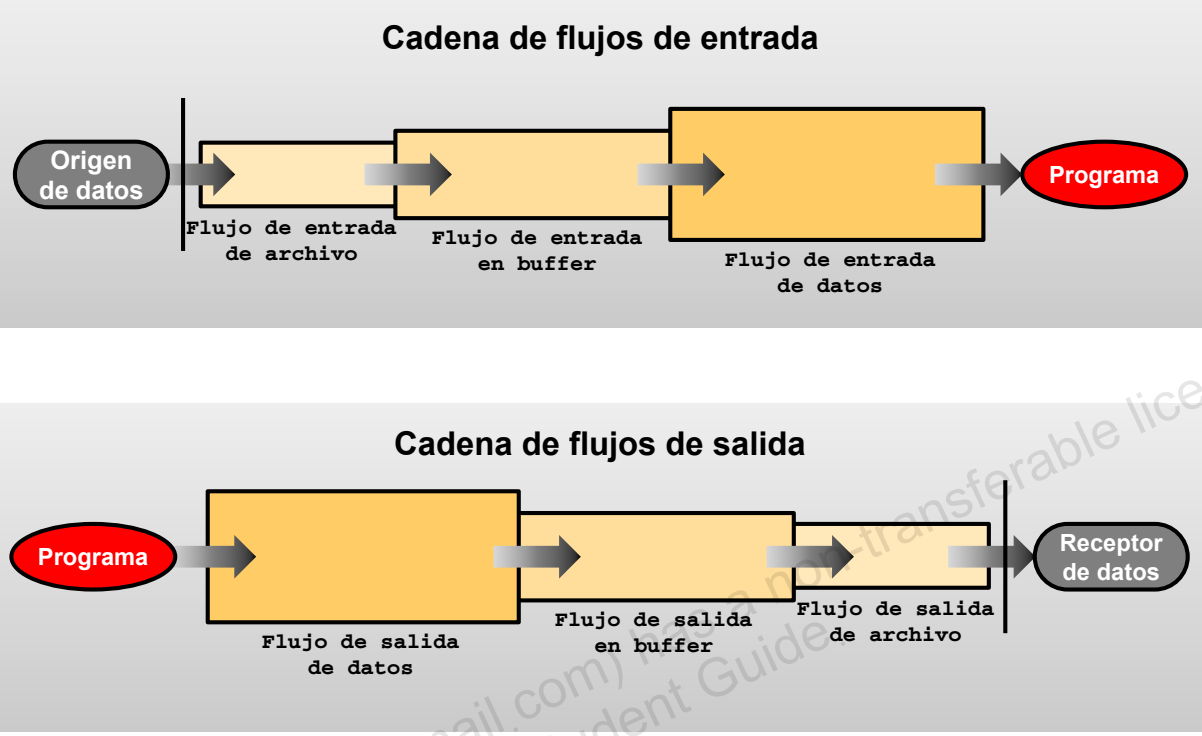
Ahora, en lugar de una matriz de bytes, esta versión utiliza una matriz de caracteres.

ORACLE



De forma parecida al ejemplo del flujo de bytes, esta aplicación copia un archivo en otro mediante una matriz de caracteres en lugar de una matriz de bytes. `FileReader` y `FileWriter` son clases diseñadas para leer y escribir flujos de caracteres, como por ejemplo archivos de texto.

# Cadenas de flujos de E/S






Los programas muy pocas veces utilizan un único objeto de flujo. En lugar de ello, encadenan una serie de flujos para procesar los datos. La primera imagen en la diapositiva muestra un ejemplo de un flujo de entrada. En este caso, para que la operación resulte más eficiente, se coloca en el buffer un flujo de archivo que después se convierte a elementos de datos (datos primitivos de Java). La segunda imagen muestra un ejemplo de flujo de salida; en este caso, primero se escriben los datos, luego se colocan en el buffer y, por último, se escriben en un archivo.

# Ejemplo de flujos en cadena

```
1 import java.io.BufferedReader; import java.io.BufferedWriter;
2 import java.io.FileReader; import java.io.FileWriter;
3 import java.io.FileNotFoundException; import java.io.IOException;
4
5 public class BufferedStreamCopyTest {
6    public static void main(String[] args) {
7        try (BufferedReader bufInput
8                 = new BufferedReader(new FileReader(args[0]));
9             BufferedWriter bufOutput
10                 = new BufferedWriter(new FileWriter(args[1]))) {
11            String line = "";
12            while ((line = bufInput.readLine()) != null) {
13                bufOutput.write(line);
14                bufOutput.newLine();
15            }
16        } catch (FileNotFoundException f) {
17            System.out.println("File not found: ");
18        } catch (IOException e) {
19            System.out.println("Exception: " + e);
20        }
21    }
22}
```

Una clase `FileReader` encadenada a una clase `BufferedFileReader`: esto permite usar un método que lea una cadena.

El buffer de caracteres se sustituye por `String`. Observe cómo `readLine()` utiliza el carácter de nueva línea como terminador. Así pues, deberá volver a agregarlo al archivo de salida.

ORACLE



Aquí aparece una vez más la aplicación de copia. Esta versión muestra el uso de un objeto `BufferedReader` encadenado a la clase `FileReader` que vio antes.

El flujo de este programa es el mismo que en el caso anterior. En lugar de leer un buffer de caracteres, este programa lee también una línea cada vez a través de la variable de línea para mantener el valor `String` que devuelve el método `readLine`, lo cual resulta mucho más eficiente. El motivo de ello es que cada solicitud de lectura realizada desde un método `Reader` hace que se genere una solicitud de lectura para el carácter subyacente o el flujo de bytes. Una clase `BufferedReader` lee caracteres desde el flujo a un buffer (aunque el tamaño del buffer se puede establecer, el valor por defecto es generalmente suficiente).

# Flujos de procesamiento

| Funcionalidad | Flujos de caracteres | Flujos de bytes |
|---|---|---|
| Almacenamiento en buffer (cadenas) | BufferedReader<br>BufferedWriter | BufferedInputStream<br>BufferedOutputStream |
| Filtrado | FilterReader<br>FilterWriter | FilterInputStream<br>FilterOutputStream |
| Conversión (de bytes a caracteres) | InputStreamReader<br>OutputStreamWriter | |
| Serialización de objetos | | ObjectInputStream<br>ObjectOutputStream |
| Conversión de datos | | DataInputStream<br>DataOutputStream |
| Recuento | LineNumberReader | LineNumberInputStream |
| Consulta hacia | PushbackReader | PushbackInputStream |
| Impresión | PrintWriter | PrintStream |

ORACLE



Los flujos de procesamiento realizan conversiones en otros flujos. El tipo de flujo se elige según la funcionalidad que se necesite obtener para el flujo final.

# E/S de la consola

La clase `System` del paquete `java.lang` tiene tres campos de instancias estáticas: `out`, `in` y `err`.

- El campo `System.out` es una instancia estática de un objeto `PrintStream` que le permite escribir en una *salida estándar.*
- El campo `System.in` es una instancia estática de un objeto `InputStream` que le permite leer a partir de una *entrada estándar.*
- El campo `System.err` es una instancia estática de un objeto `PrintStream` que le permite escribir en un *error estándar.*

ORACLE







**Sistema de uso de E/S de la consola**

- **`System.out`** es el flujo de salida “estándar”. Este flujo ya está abierto y listo para aceptar datos de salida. Generalmente, este flujo corresponde a salida en pantalla o a otro destino de salida especificado por el entorno host o el usuario.
- **`System.in`** es el flujo de entrada “estándar”. Este flujo ya está abierto y listo para suministrar datos de entrada. Generalmente, este flujo corresponde a entrada del teclado o a otro origen de entrada especificado por el entorno host o el usuario.
- **`System.err`** es el flujo de salida de errores “estándar”. Este flujo ya está abierto y listo para aceptar datos de salida.

Generalmente, este flujo corresponde a salida en pantalla o a otro destino de salida especificado por el entorno host o el usuario. La convención establece que este flujo de salida se utilice para mostrar mensajes de error u otra información que el usuario debe poder ver de manera inmediata, incluso si el flujo de salida principal, el valor de la variable `out`, se ha redirigido a un archivo o a otro destino que generalmente no se supervisa de manera continua.

# java.io.Console

Además de los objetos `PrintStream`, `System` también puede acceder a una instancia del objeto `java.io.Console`:

```
1 Console cons = System.console();
2 if (cons != null) {
3     String userTyped; String pwdTyped;
4     do {
5         userTyped = cons.readLine("%s", "User name: ");
6         pwdTyped = new String(cons.readPassword("%s", "Password: "));
7         if (userTyped.equals("oracle") && pwdTyped.equals("tiger")) {
8            userValid = true;
9         } else {
10           System.out.println("Wrong user name/password. Try again.\n");
11        }
12    } while (!userValid);
13 }
```

`readPassword` no hace eco de los caracteres introducidos en la consola.

- Tenga en cuenta que deberá pasar el nombre de usuario y la contraseña para el proceso de autenticación.

ORACLE



El objeto `Console` representa la consola basada en caracteres asociada con la máquina JVM actual. Que una máquina virtual tenga o no una consola dependerá de la plataforma subyacente y también de la forma en la que se invoque a la máquina virtual.

NetBeans, por ejemplo, no utiliza una consola. Para ejecutar el ejemplo en el proyecto `SystemConsoleExample`, utilice la línea de comandos.

**Nota:** este ejemplo solo pretende ilustrar los métodos de la clase Console. Deberá asegurarse de que la duración de los campos `userTyped` y `pwdTyped` es lo más corta posible, así como de que pasen las credenciales recibidas a algún tipo de servicio de autenticación. Consulte la API de Java Authentication and Authorization Service (JAAS) para obtener más información: http://download.oracle.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html

# Escritura en una salida estándar

- Los métodos `println` y `print` forman parte de la clase `java.io.PrintStream`.
- Los métodos `println` imprimen el argumento y un carácter de línea nueva (`\n`).
- Los métodos `print` imprimen el argumento sin un carácter de línea nueva.
- Los métodos `print` y `println` se sobrecargan para la mayor parte de los tipos primitivos (`boolean`, `char`, `int`, `long`, `float` y `double`) y para `char[]`, `Object` y `String`.
- Los métodos `print(Object)` y `println(Object)` llaman al método `toString` en el argumento.

ORACLE







**Métodos de impresión**

Tenga en cuenta que existe también un método de impresión con formato, `printf`. Este método lo ha visto anteriormente en la lección titulada "Procesamiento de cadenas".

# Lectura a partir de una entrada estándar

```
1 import java.io.BufferedReader;
2 import java.io.IOException;
3 import java.io.InputStreamReader;

4 public class KeyboardInput {
5     public static void main(String[] args) {
6     try (BufferedReader in =
7          new BufferedReader (new InputStreamReader (System.in))) {
8        String s = "";
9        // Read each input line and echo it to the screen.
10        while (s != null) {
11            System.out.print("Type xyz to exit: ");
12            s = in.readLine().trim();
13            System.out.println("Read: " + s);
14            if (s.equals ("xyz")) System.exit(0);
15        }
16    } catch (IOException e) {
17        System.out.println ("Exception: " + e);
18    }
19}
```

Encadenar un lector en buffer a un flujo de entrada que toma la entrada de la consola.

ORACLE



La sentencia `try`-with-resources en la línea 6 abre `BufferedReader`, el cual está encadenado a un `InputStreamReader`, que a su vez está encadenado a la entrada de la consola estándar estática `System.in`.

Si la lectura de la cadena es igual a “`xyz`”, entonces el programa se cerrará. El propósito del método `trim()` en la cadena devuelta por `in.readLine` es eliminar todos los caracteres de espacio en blanco.

# E/S de canal

Introducidos en JDK 1.4, los canales leen bytes y caracteres en bloques, en lugar de un byte o un carácter cada vez.

```
1 import java.io.FileInputStream; import java.io.FileOutputStream;
2 import java.nio.channels.FileChannel; import java.nio.ByteBuffer;
3 import java.io.FileNotFoundException; import java.io.IOException;
4
5 public class ByteChannelCopyTest {
6     public static void main(String[] args) {
7         try (FileChannel fcIn = new FileInputStream(args[0]).getChannel();
8              FileChannel fcOut = new FileOutputStream(args[1]).getChannel()) {
9             ByteBuffer buff = ByteBuffer.allocate((int) fcIn.size());
10             fcIn.read(buff);
11            buff.position(0);
12            fcOut.write(buff);
13        } catch (FileNotFoundException f) {
14            System.out.println("File not found: " + f);
15        } catch (IOException e) {
16            System.out.println("IOException: " + e);
17        }
18    }
19 }
```

Crear un buffer con el mismo tamaño que el tamaño del archivo y después leer y escribir el archivo en una sola operación.

ORACLE



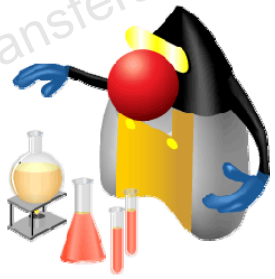
En este ejemplo, se puede leer un archivo completo en un buffer y después escribirlo en una sola operación.

La E/S de canal se introdujo en el paquete `java.nio` en JDK 1.4.

# Visión general de la práctica 10-1: Escritura de una aplicación simple de E/S de la consola

En esta práctica, se abordan los siguientes temas:

- Escritura de una clase principal que acepte un nombre de archivo como argumento.
- Uso de la E/S de la consola `System` para leer una cadena de búsqueda.
- Uso del encadenamiento de flujos para utilizar el método adecuado para buscar la cadena en el archivo e informar sobre el número de incidencias.
- Continuación de la lectura desde la consola hasta que se introduzca una secuencia de cierre.



ORACLE



En esta práctica, escribirá el código necesario para leer un nombre de archivo como un argumento de aplicación y utilizará la consola `System` para leer a partir de una entrada estándar hasta que se introduzca un carácter de terminación.

# Persistencia

Guardar datos en algún tipo de almacenamiento permanente se conoce como "persistencia". Los objetos que pueden ser persistentes se pueden almacenar en un disco (o en otro dispositivo de almacenamiento), o se pueden enviar a otra máquina para almacenarse allí.

- Los objetos no persistentes solo existen mientras se esté ejecutando la máquina Java Virtual Machine.
- La serialización de Java es el mecanismo estándar para guardar un objeto como secuencia de bytes que después se puede reconstruir en una copia del objeto.
- Para serializar un objeto de una clase específica, la clase debe implantar la interfaz `java.io.Serializable`.

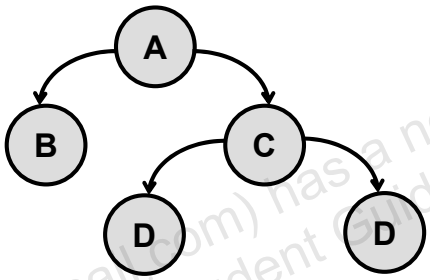
ORACLE



La interfaz `java.io.Serializable` no define ningún método, y sirve solo como marcador para indicar que la clase ha de tenerse en cuenta para una posible serialización.

# Serialización y gráficos de objetos

- Cuando se serializa un objeto, solo se conservan los campos del objeto.
- Cuando un campo hace referencia a un objeto, los campos del objeto al que se hace referencia se serializan también si la clase del objeto es igualmente serializable.
- El árbol de los campos de un objeto constituye el *gráfico del objeto*.




ORACLE





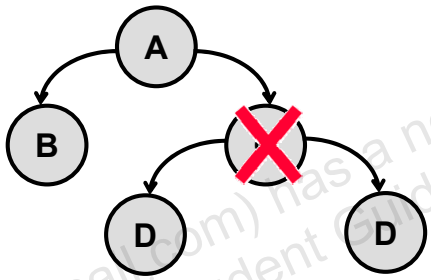


**Gráficos de objetos**

La serialización recorre el gráfico del objeto y escribe esos datos en el archivo (o en otro flujo de salida) para cada nodo del gráfico.

# Campos y objetos transitorios

- Algunas clases de objetos no son serializables, ya que representan información transitoria específica del sistema operativo.
- Si el gráfico del objeto contiene una referencia no serializable, se devuelve `NotSerializableException` y la operación de serialización falla.
- Los campos que no deben serializarse o que no necesitan hacerlo, se pueden marcar con la palabra clave `transient`.






**Transient**

Si se encuentra un campo que contienen una referencia a un objeto y no se ha marcado como serializable (se implanta `java.io.Serializable`), se devuelve `NotSerializableException` y la operación de serialización falla por completo. Para serializar un gráfico con campos que hacen referencia a objetos que no son serializables, estos campos deben marcarse con la palabra clave `transient`.

# Transient: ejemplo

```
public class Portfolio implements Serializable {
    public transient FileInputStream inputFile;
    public static int BASE = 100;
    private transient int totalValue = 10;
    protected Stock[] stocks;
}
```

Los campos `static` no se serializan.

La serialización incluirá todos los miembros de la matriz `stocks`.

- El modificador de acceso del campo no afecta a los datos que se vayan a serializar.
- Los valores almacenados en campos estáticos no se serializan.
- Cuando se anula la serialización del objeto, los valores de los campos estáticos se establecen en los valores declarados en la clase. El valor de los campos transitorios no estáticos se establece en el valor por defecto para el tipo.

ORACLE



Cuando se anula la serialización de un objeto, los valores de los campos transitorios estáticos se establecen en los valores definidos en la declaración de la clase. Los valores de los campos no estáticos se establecen en el valor por defecto para su tipo. Así, en el ejemplo de la diapositiva, el valor de BASE será 100, por la declaración de la clase. Los campos transitorios no estáticos, `inputFile` y `totalValue`, se definen en su valor por defecto, `null` y `0` respectivamente.

# UID de versión de serialización

- Durante el proceso de serialización se usa un número de versión, serialVersionUID, para asociar la salida serializada con la clase empleada en el proceso de serialización.
- Al anular la serialización, se comprueba el valor de serialVersionUID para verificar que las clases cargadas son compatibles con el objeto cuya serialización se está anulando.
- Si el receptor de un objeto serializado ha cargado clases para ese objeto con serialVersionUID diferentes, la anulación de la serialización resultará en `InvalidClassException`.
- Una clase serializable puede declarar su propio serialVersionUID declarando de forma explícita un campo llamado `serialVersionUID` como final estático y tipo long:

```
private static long serialVersionUID = 42L;
```

ORACLE



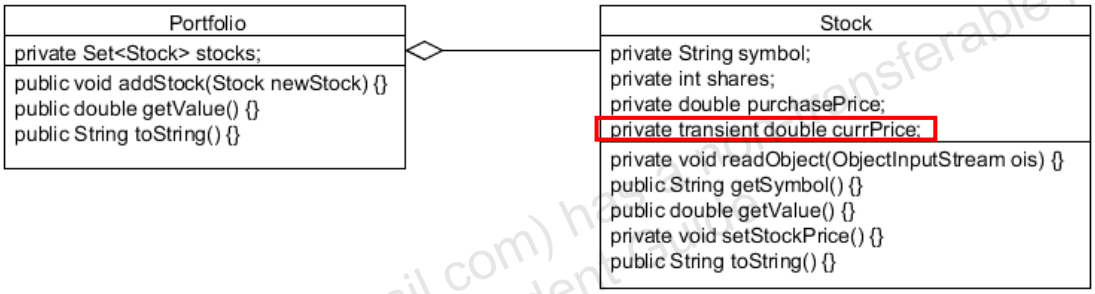
**Nota:** la documentación de `java.io.Serializable` indica lo siguiente:

*Si una clase serializable no declara explícitamente un valor serialVersionUID, entonces el tiempo de ejecución de la serialización calculará el valor serialVersionUID por defecto para esa clase de acuerdo con distintos aspectos de la clase, tal y como se describe en la especificación de serialización de objetos de Java(TM). No obstante, se recomienda encarecidamente que todas las clases serializables declaren explícitamente valores serialVersionUID, ya que el cálculo por defecto de serialVersionUID es muy sensible a detalles de clase que pueden variar dependiendo de las implantaciones del compilador, y por lo tanto pueden resultar en excepciones InvalidClassException no esperadas durante la anulación de la serialización. Así pues, para garantizar un valor serialVersionUID uniforme en diferentes implantaciones del compilador de Java, es necesario que una clase serializable declare un valor serialVersionUID explícito. También se recomienda encarecidamente utilizar el modificador private en las declaraciones serialVersionUID explícitas siempre que sea posible, ya que este tipo de declaraciones se aplica solo a la clase que declara de forma inmediata. Los campos serialVersionUID no son útiles como miembros heredados. Las clases de matriz no pueden declarar un valor serialVersionUID explícito, por lo que siempre llevan el valor calculado por defecto; el requisito de coincidencia entre los valores serialVersionUID no se aplica en las clases de matriz.*

# Ejemplo de serialización

En este ejemplo hay una cartera de valores constituida por un juego de acciones.

- Durante la serialización, el precio actual no se serializa, por lo que se marca como `transient`.
- No obstante, queremos que el valor actual de las acciones se establezca en el valor de mercado actual al anular la serialización.




ORACLE

# Escritura y lectura de un flujo de objetos

```
1 public static void main(String[] args) {
2     Stock s1 = new Stock("ORCL", 100, 32.50);
3     Stock s2 = new Stock("APPL", 100, 245);
4     Stock s3 = new Stock("GOGL", 100, 54.67);
5     Portfolio p = new   Portfolio(s1, s2, s3);
6     try (FileOutputStream fos = new FileOutputStream(args[0]);
7          ObjectOutputStream out = new ObjectOutputStream(fos)) {
8         out.writeObject(p);
9     } catch (IOException i) {
10         System.out.println("Exception writing out Portfolio: " + i);
11    }
12    try (FileInputStream fis = new FileInputStream(args[0]);
13         ObjectInputStream in = new ObjectInputStream(fis)) {
14        Portfolio newP = (Portfolio)in.readObject();
15    } catch (ClassNotFoundException | IOException i) {
16        System.out.println("Exception reading in Portfolio: " + i);
17 }
```

Portfolio es el objeto raíz.

El método writeObject escribe el gráfico de objeto de p en el flujo de archivo.

El método readObject restaura el objeto desde el flujo de archivo.

ORACLE



La clase `SerializeStock`.

- **Línea 6 – 8:** una clase `FileOutputStream` aparece encadenada a una clase `ObjectOutputStream`. Esto permite escribir los bytes no procesados generados por `ObjectOutputStream` en un archivo a través del método `writeObject`. Este método recorre el gráfico del objeto y escribe los datos de los campos no transitorios y no estáticos como bytes no procesados.
- **Línea 12 – 14:** para restaurar un objeto desde un archivo, hay una clase `FileInputStream` encadenada con una clase `ObjectInputStream`. Los bytes no procesados que lee el método `readObject` restauran un `Object` con los campos de datos no estáticos y transitorios. Este valor `Object` debe convertirse al tipo esperado.

# Métodos de serialización

Los objetos que se están serializando (o se está anulando su serialización) pueden controlar la serialización de sus propios campos.

```
public class MyClass implements Serializable {
    // Fields
    private void writeObject(ObjectOutputStream oos) throws IOException {
        oos.defaultWriteObject();
        // Write/save additional fields
        oos.writeObject(new java.util.Date());
    }
}
```

Llamada a `defaultWriteObject` para serializar los campos de estas clases.

- Por ejemplo, en esta clase la hora actual se escribe en el gráfico del objeto.
- Durante la anulación de la serialización se invoca a un método similar:

```
private void readObject(ObjectInputStream ois) throws
ClassNotFoundException, IOException {}
```

ORACLE



Se invoca al método `writeObject` en el objeto que se está serializando. Si el objeto no contiene este método, se invoca en su lugar al método `defaultWriteObject`.

- También se puede llamar a este método solo una vez desde el método del objeto `writeObject`.

Durante la anulación de la serialización, se invoca al método `readObject` en el objeto cuya serialización se está anulando (si está presente el archivo de clase del objeto). La firma del método es importante.

```
private void readObject(ObjectInputStream ois) throws
                                ClassNotFoundException, IOException {
    ois.defaultReadObject();
    // Print the date this object was serialized
    System.out.println ("Restored from date: " +
                                  (java.util.Date)ois.readObject()));
}
```

# Ejemplo de `readObject`

```
1 public class Stock implements Serializable {
2     private static final long serialVersionUID = 100L;
3     private String symbol;
4     private int shares;
5     private double purchasePrice;
6     private transient double currPrice;
7
8     public Stock(String symbol, int shares, double purchasePrice) {
9         this.symbol = symbol;
10         this.shares = shares;
11         this.purchasePrice = purchasePrice;
12         setStockPrice();
13     }
14
15     // This method is called post-serialization
16     private void readObject(ObjectInputStream ois)
17                             throws IOException, ClassNotFoundException {
18         ois.defaultReadObject();
19         // perform other initialization
20         setStockPrice();
21     }
22 }
```

El valor `currPrice` de las acciones lo establece el método `setStockPrice` al crear el objeto Stock, pero no se llama al constructor durante la anulación de la serialización.

El valor `currPrice` de las acciones se establece después de que se anule la serialización de los otros campos.

ORACLE



En la clase Stock se proporciona el método `readObject` para garantizar que el valor `currPrice` de las acciones se establezca (mediante el método `setStockPrice`) después de anular la serialización del objeto Stock.

**Nota:** la firma del método `readObject` es fundamental para llamar a este método durante la anulación de la serialización.

# Resumen

En esta lección, debe haber aprendido a hacer lo siguiente:

- Describir los aspectos básicos de entrada y salida en Java
- Leer datos de la consola y escribir datos en ella
- Utilizar flujos para leer y escribir datos
- Leer y escribir objetos mediante serialización

ORACLE

# Prueba

El objetivo de encadenar flujos es:

a. Permitir a los flujos agregar funcionalidad
b. Cambiar la dirección del flujo
c. Modificar el acceso del flujo
d. Cumplir con los requisitos de JDK 7

# Prueba

Para evitar que se serialicen campos específicos del sistema operativo, deberá marcar el campo como:

a. `private`
b. `static`
c. `transient`
d. `final`

# Prueba

Dados los siguientes fragmentos:

```
public MyClass implements Serializable {
    private String name;
    private static int id = 10;
     private transient String keyword;
    public MyClass(String name, String keyword) {
        this.name = name; this.keyword = keyword;
    }
}
```

```
MyClass mc = new MyClass ("Zim", "xyzzy");
```

Suponiendo que no se hacen más cambios en los datos, ¿cuál es el valor de los campos `name` y `keyword` después de anular la serialización de la instancia de objeto `mc`?

a. `Zim, ""`
b. `Zim, null`
c. `Zim, xyzzy`
d. `"", null`

ORACLE

# Prueba

Dados los siguientes fragmentos:

```
1 public class MyClass implements Serializable {
2     private transient String keyword;
3     public void readObject(ObjectInputStream ois)
4                            throws IOException, ClassNotFoundException {
5         ois.defaultReadObject();
6         String this.keyword = (String)ois.readObject();
7     }
8 }
```

¿Qué hace falta para anular correctamente la serialización de un flujo que contenga este objeto?

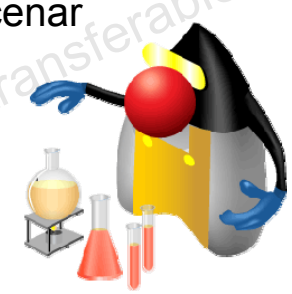
a. Hacer el campo `keyword` `static`
b. Cambiar el modificador de acceso de campo a `public`
c. Hacer el método `readObject` `private` (línea 3)
d. Usar `readString` en lugar de `readObject` (línea 6)

ORACLE

# Visión general de la práctica 10-2: Serialización y anulación de la serialización de `ShoppingCart`

En esta práctica, se abordan los siguientes temas:

- Creación de una aplicación que serialice un objeto `ShoppingCart` compuesto de una `ArrayList` de objetos `Item`.
- Uso de la palabra clave `transient` para evitar la serialización del total de `ShoppingCart`. Esto permitirá que los elementos puedan variar de precio.
- Uso del método `writeObject` para almacenar la fecha actual en el flujo serializado.
- Uso del método `readObject` para volver a calcular el coste del carro tras anular la serialización e imprimir la fecha en la que el objeto se serializó.



ORACLE

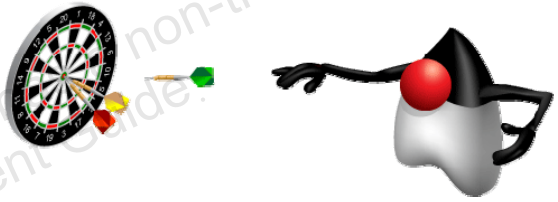
# 11

# E/S de archivos Java (NIO.2)

# Objetivos

Al finalizar esta lección, debería estar capacitado para:

- Utilizar la interfaz `Path` para realizar operaciones en archivos y en rutas de acceso a directorios
- Utilizar la clase `Files` para comprobar, suprimir, copiar o mover un archivo o un directorio
- Utilizar métodos de la clase `Files` para leer y escribir archivos mediante E/S de canales o E/S de flujos
- Leer y cambiar atributos de archivos y de directorios
- Acceder de forma recurrente al árbol de un directorio
- Localizar un archivo a través de la clase `PathMatcher`



ORACLE

# Nueva API de E/S de archivos Java (NIO.2)


ORACLE



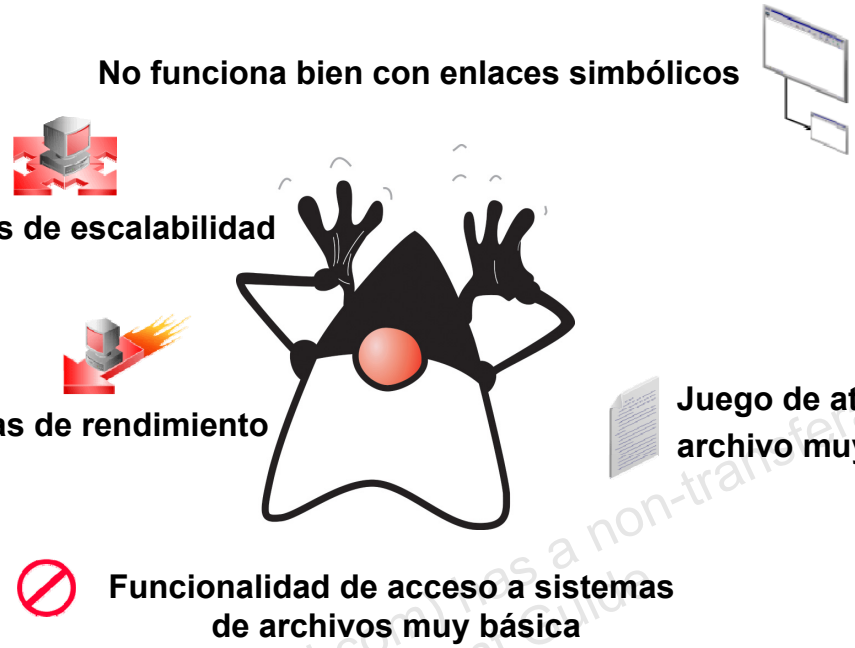
La API NIO en JSR 51 instauró la base para usar NIO en Java, centrándose en buffers, canales y juegos de caracteres. JSR 51 proporcionó la primera opción de E/S de socket escalable en la plataforma con una API de E/S no bloqueante y multiplexada que permitió desarrollar servidores muy escalables sin tener que recurrir a código nativo.

Para muchos desarrolladores, el objetivo más importante de JSR 203 es abordar problemas con `java.io.File` a través del desarrollo de una nueva interfaz de sistema de archivos.

La nueva API:

- Funciona de manera más consistente entre una plataforma y otra.
- Facilita la escritura de programas que manejan fluidamente los fallos de las operaciones del sistema de archivos.
- Proporciona un acceso más eficiente a un juego más amplio de atributos.
- Permite a los desarrolladores de aplicaciones sofisticadas aprovechar funciones propias de la plataforma cuando sea indispensable.
- Permite soportar sistemas de archivos no nativos para conectarlos a la plataforma.

# Limitaciones de `java.io.File`





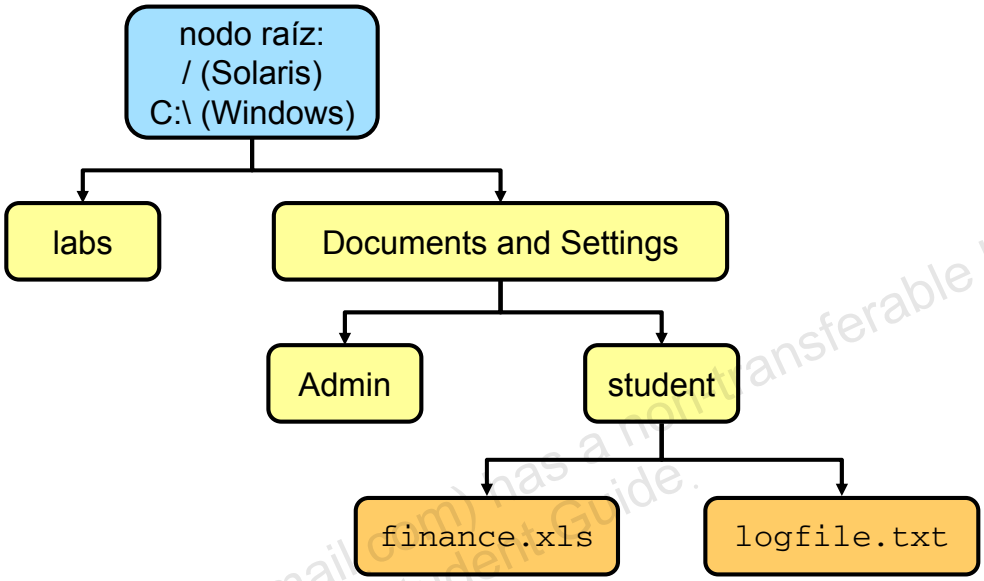

La API de archivos de E/S Java (`java.io.File`) presentaba bastantes retos a los desarrolladores.

- Muchos métodos no devolvían excepciones al fallar, por lo que resultaba imposible obtener un mensaje de error útil.
- Faltaban varias operaciones (copia de archivo, desplazamiento, etc.).
- El método de cambio de nombre no funcionaba consistentemente de una plataforma a otra.
- No existía soporte real para enlaces simbólicos.
- Interesaba contar con más soporte para metadatos, como permisos de archivo, propietario de archivo y otros atributos de seguridad.
- El acceso a los metadatos no funcionaba de forma eficaz: cada llamada a metadatos resultaba en una llamada del sistema, lo cual hacía que el funcionamiento de las operaciones fuera muy ineficaz.
- Muchos de los métodos de archivo no escalaban. Si se solicitaba una lista de directorios de gran tamaño en un servidor, podía colgarse el sistema.
- No era posible escribir código fiable que recorriera un árbol de archivos de forma recurrente y respondiera correctamente si había enlaces simbólicos circulares.

Además, la E/S global no se escribía de manera que pudiera ampliarse. Los desarrolladores pidieron poder desarrollar sus propias implantaciones de sistemas de archivos. Por ejemplo, guardando un pseudoarchivo en memoria o aplicando formato zip a archivos.

## Sistemas de archivos, rutas y archivos

En NIO.2, los archivos y los directorios se representan a través de una ruta, que es la ubicación relativa o absoluta del archivo o del directorio.








### Sistemas de archivos

Antes de implantarse NIO.2 en JDK 7, los archivos se representaban mediante la clase `java.io.File`.

En NIO.2, las instancias de los objetos `java.nio.file.Path` se utilizan para representar la ubicación relativa o absoluta de un archivo o un directorio.

Los sistemas de archivos son estructuras jerárquicas (de árbol). Los sistemas de archivos pueden tener uno o más directorios raíz. Por ejemplo, una máquina típica Windows tendrá dos nodos raíz de disco como mínimo: C:\ y D:\.

Tenga en cuenta que los sistemas de archivos tienen también sus propias características en cuanto a separadores de rutas, tal y como se ve en la diapositiva.

# Ruta de acceso relativa frente a ruta de acceso absoluta

- Las rutas de acceso pueden ser *relativas* o *absolutas*.
- Las rutas de acceso absolutas contienen el elemento raíz y la lista completa del directorio para localizar el archivo.
- Ejemplo:

```
...
/home/peter/statusReport
...
```

- Una ruta de acceso relativa debe combinarse con otra ruta para poder acceder a un archivo.
- Ejemplo:

```
...
clarence/foo
...
```

ORACLE



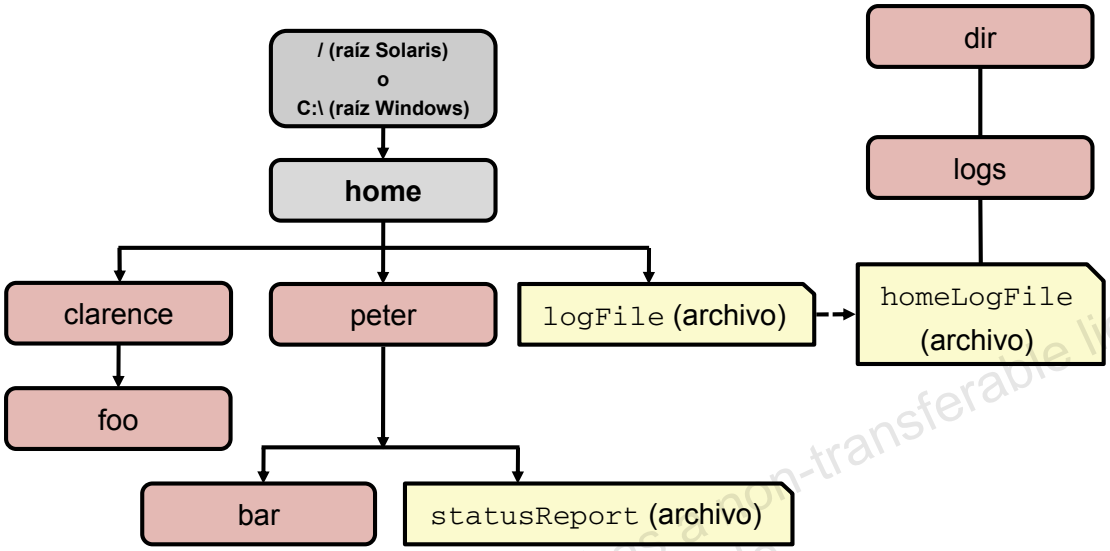




Las rutas pueden ser relativas o absolutas. Las rutas de acceso absolutas contienen el elemento raíz y la lista completa del directorio para localizar el archivo. Por ejemplo, `/home/peter/statusReport` es una ruta absoluta. Toda la información necesaria para localizar el archivo se encuentra en la cadena de la ruta.

Una ruta de acceso relativa debe combinarse con otra ruta para poder acceder a un archivo. Por ejemplo, `clarence/foo` es una ruta relativa. Sin más información, los programas no podrán localizar con fiabilidad el directorio `clarence/foo` dentro del sistema de archivos.

# Enlaces simbólicos






Los objetos del sistema de archivos son generalmente directorios o archivos. Todos los usuarios están familiarizados con estos objetos. Pero algunos sistemas de archivos soportan la noción de enlaces simbólicos. Los enlaces simbólicos se conocen también como “symlinks” o “enlaces lógicos”.

El enlace simbólico es un archivo especial que sirve como referencia a otro archivo. Los enlaces simbólicos generalmente son transparentes para el usuario. Leer o escribir un enlace simbólico es lo mismo que leer o escribir en otro archivo o directorio.

En el diagrama de la diapositiva, `logFile` aparece ante el usuario como un archivo normal, pero en realidad es un enlace simbólico a `dir/logs/HomeLogFile`. `HomeLogFile` es el destino del enlace.

# Conceptos de Java NIO.2

Antes de JDK 7, la clase `java.io.File` era el punto de entrada para todas las operaciones de archivo o de directorio. NIO.2 introduce un nuevo paquete y nuevas clases:

- `java.nio.file.Path`: localiza un archivo o un directorio mediante una ruta de acceso dependiente del sistema.
- `java.nio.file.Files`: realiza operaciones en archivos y en directorios a través de una interfaz `Path`.
- `java.nio.file.FileSystem`: proporciona una interfaz hacia un sistema de archivos y una fábrica para crear un objeto `Path` y otros objetos que acceden a un sistema de archivos.
- Todos los métodos que acceden al sistema de archivos devuelven `IOException` o una subclase.

ORACLE







### Java NIO.2

Una diferencia significativa entre NIO.2 y `java.io.File` es la arquitectura de acceso al sistema de archivos. Con la clase `java.io.File`, los métodos utilizados para manipular la información de ruta están en la misma clase que los métodos utilizados para leer y escribir archivos y directorios.

En NIO.2, los dos métodos van aparte. Las rutas se crean y manipulan mediante la interfaz `Path`, mientras que las operaciones en los archivos y los directorios recaen en la clase `Files`, la cual trabaja solo con objetos `Path`.

Por último, al contrario que `java.io.File`, los métodos de clase de archivos que operan directamente en el sistema de archivos devuelven `IOException` (o una subclase). Las subclases proporcionan detalles sobre la causa de la excepción.

# Interfaz `Path`

La interfaz `java.nio.file.Path` proporciona el punto de entrada para manipular archivos y directorios en NIO.2.

- Para obtener un objeto `Path`, deberá obtener una instancia del sistema de archivos por defecto y después invocar al método `getPath`:

```
FileSystem fs = FileSystems.getDefault();
Path p1 = fs.getPath ("D:\\labs\\resources\\myFile.txt");
```

Barra diagonal inversa con escape

- El paquete `java.nio.file` proporciona también una clase helper final estática llamada `Paths` para realizar la acción `getDefault`:

```
Path p1 = Paths.get ("D:\\labs\\resources\\myFile.txt");
Path p2 = Paths.get ("D:", "labs", "resources", "myFile.txt");
Path p3 = Paths.get ("/temp/foo");
Path p4 = Paths.get (URI.create ("file:///~/somefile");
```

ORACLE



El punto de entrada para directorios y archivos de NIO.2 es una instancia de la interfaz `Path`. El proveedor (en este caso el proveedor por defecto) crea un objeto que implanta esta clase y maneja todas las operaciones realizadas en un archivo o un directorio dentro de un sistema de archivos.

Los objetos `Path` son inmutables. Una vez se crean, ya no se pueden cambiar.

Tenga en cuenta que, si planea utilizar el sistema de archivos por defecto, es decir, el sistema de archivos en el que se está ejecutando JVM para las operaciones `Path`, la utilidad `Paths` es el método más corto. No obstante, si quisiera realizar operaciones `Path` en un sistema de archivos diferente al sistema por defecto, obtendría una instancia del sistema de archivos deseado y usaría el primer modo de creación de objetos `Path`.

**Nota:** el sistema de archivos de Windows utiliza una barra diagonal inversa por defecto. No obstante, Windows acepta tanto barras diagonales normales como inversas en las aplicaciones (excepto en el shell de comandos). Las barras diagonales inversas en Java deben llevar carácter de escape. Para poder representar una barra diagonal inversa en una cadena, es necesario introducirla dos veces. Como se ve feo y los usuarios de Windows usan barras diagonales tanto normales como inversas, los ejemplos que se incluyen en este curso usarán la barra diagonal normal en las cadenas.

## Características de la interfaz `Path`

La interfaz `Path` define los métodos utilizados para localizar un archivo o un directorio en un sistema de archivos. Estos métodos incluyen:

- Para acceder a los componentes de una ruta:
  - `getFileName`, `getParent`, `getRoot` y `getNameCount`
- Para realizar operaciones en una ruta:
  - `normalize`, `toUri`, `toAbsolutePath`, `subpath`, `resolve` y `relativize`
- Para comparar rutas:
  - `startsWith`, `endsWith` y `equals`

ORACLE







### Los objetos `Path` son como objetos `String`

Lo más práctico es pensar en los objetos `Path` como si fueran objetos `String`. Los objetos `Path` se pueden crear a partir de una sola cadena de texto o a partir de un juego de componentes:

- Un *componente raíz*, el cual identifica la jerarquía del sistema de archivos.
- Un *elemento de nombre*, el más alejado del elemento raíz, que define el archivo o el directorio al que apunta la ruta.
- También puede haber otros elementos separados por un carácter especial o un delimitador que identifiquen los nombres de los directorios que forman parte de la jerarquía.

Los objetos `Path` son inmutables. Una vez creados, las operaciones que se realicen en los objetos `Path` devolverán nuevos objetos `Path`.

# Path: ejemplo

```
1 public class PathTest
2     public static void main(String[] args) {
3          Path p1 = Paths.get(args[0]);
4          System.out.format("getFileName: %s%n", p1.getFileName());
5          System.out.format("getParent: %s%n", p1.getParent());
6          System.out.format("getNameCount: %d%n", p1.getNameCount());
7          System.out.format("getRoot: %s%n", p1.getRoot());
8          System.out.format("isAbsolute: %b%n", p1.isAbsolute());
9          System.out.format("toAbsolutePath: %s%n", p1.toAbsolutePath());
10         System.out.format("toURI: %s%n", p1.toUri());
11     }
12 }
```

```
java PathTest D:/Temp/Foo/file1.txt
getFileName: file1.txt
getParent: D:\Temp\Foo
getNameCount: 3
getRoot: D:\
isAbsolute: false
toAbsolutePath: D:\Temp\Foo\file1.txt
toURI: file:///D:/Temp/Foo/file1.txt
```

Ejecutado en un equipo Windows. Observe que, excepto en un shell `cmd`, se pueden usar barras diagonales normales e inversas.

ORACLE



Al contrario de lo que ocurre con las clases `java.io.File`, los archivos y los directorios se representan a través de instancias de objetos `Path` de manera *dependiente del sistema*.

La interfaz `Path` ofrece varios métodos para proporcionar información sobre la ruta:

- `Path getFileName`: el punto final de esta interfaz `Path`, devuelto como un objeto `Path`.
- `Path getParent`: la ruta principal o una nula. Todo lo incluido en la ruta hasta el nombre del archivo (archivo o directorio).
- `int getNameCount`: el número de elementos de nombre que conforman esta ruta.
- `Path getRoot`: el componente raíz de esta interfaz `Path`.
- `boolean isAbsolute`: `true` si la ruta contiene un elemento raíz dependiente del sistema. **Nota:** puesto que este ejemplo se ejecuta en una máquina Windows, el elemento raíz *dependiente del sistema* contiene la letra de la unidad y un signo de dos puntos. En sistemas operativos basados en UNIX, `isAbsolute` devuelve true en cualquier ruta que empiece por una barra diagonal.
- `Path toAbsolutePath`: devuelve una ruta que representa la ruta absoluta para esta ruta de acceso.
- `java.net.URI toUri`: devuelve una URI absoluta.

**Nota:** los objetos `Path` pueden crearse para cualquier ruta. No es necesario que exista el archivo o el directorio real.

# Eliminación de redundancias de `Path`

- Muchos sistemas de archivos utilizan una notación "`.`" para denotar el directorio actual y "`..`" para denotar el directorio principal.
- Los dos ejemplos siguientes incluyen redundancias:

```
/home/./clarence/foo
/home/peter/../clarence/foo
```

- El método `normalize` elimina los elementos redundantes, entre ellos todas las incidencias de "`.`" o de "`directory/..`".
- Ejemplo:

```
Path p = Paths.get("/home/peter/../clarence/foo");
Path normalizedPath = p.normalize();
```

```
/home/clarence/foo
```

ORACLE







Muchos sistemas de archivos utilizan una notación "`.`" para denotar el directorio actual y "`..`" para denotar el directorio principal. Es posible que se produzca una situación en la que `Path` contenga información redundante sobre el directorio. Puede que el servidor esté configurado para guardar sus archivos log en el directorio "`/dir/logs/.`", y que desee eliminar la notación "`/.`" final de la ruta.

El método `normalize` elimina todos los elementos redundantes, entre ellos todas las incidencias de "`.`" o de "`directory/..`". Los ejemplos de la diapositiva se normalizarían en `/home/clarence/foo`.

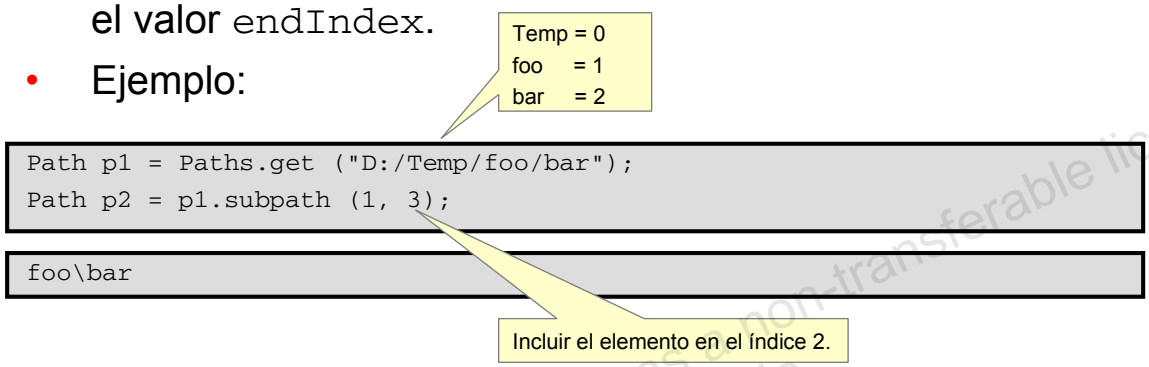
Es importante tener presente que `normalize` no comprueba el sistema de archivos cuando limpia una ruta. Se trata de una operación puramente sintáctica. En el segundo ejemplo, si `peter` fuera un enlace simbólico, al eliminar `peter/..` podría ocurrir que una ruta dejara de poder localizar el archivo buscado.

# Creación de una subruta

- Se puede obtener una parte de una ruta creando una subruta con el método `subpath`:

```
Path subpath(int beginIndex, int endIndex);
```

- El elemento que devuelve `endIndex` es uno menos que el valor `endIndex`.
- Ejemplo:




ORACLE



El nombre de elemento más próximo a la raíz tiene el índice 0.

El elemento más lejos de la raíz tiene el índice `count-1`.

**Nota:** el objeto `Path` devuelto tiene los elementos que empiezan por `beginIndex` y se amplía al elemento en el índice `endIndex-1`.

# Unión de dos rutas

- El método `resolve` se utiliza para combinar dos rutas.
- Ejemplo:

```
Path p1 = Paths.get("/home/clarence/foo");
p1.resolve("bar");     // Returns /home/clarence/foo/bar
```

- Al pasar una ruta absoluta al método `resolve` se devuelve la ruta de acceso transferida.

```
Paths.get("foo").resolve("/home/clarence"); // Returns /home/clarence
```



El método `resolve` se utiliza para combinar rutas. Este método acepta una ruta de acceso parcial, que es una ruta que no incluye un elemento raíz, y esa ruta parcial se anexa a la ruta original.

# Creación de una ruta entre dos rutas

- El método `relativize` le permite construir una ruta desde una ubicación del sistema de archivos a otra ubicación.
- El método construye una ruta que empieza en la ruta original y termina en la ubicación especificada por la ruta transferida.
- La nueva ruta es relativa a la ruta original.
- Ejemplo:

```
Path p1 = Paths.get("peter");
Path p2 = Paths.get("clarence");

Path p1Top2 = p1.relativize(p2);   // Result is ../clarence
Path p2Top1 = p2.relativize(p1);   // Result is ../peter
```

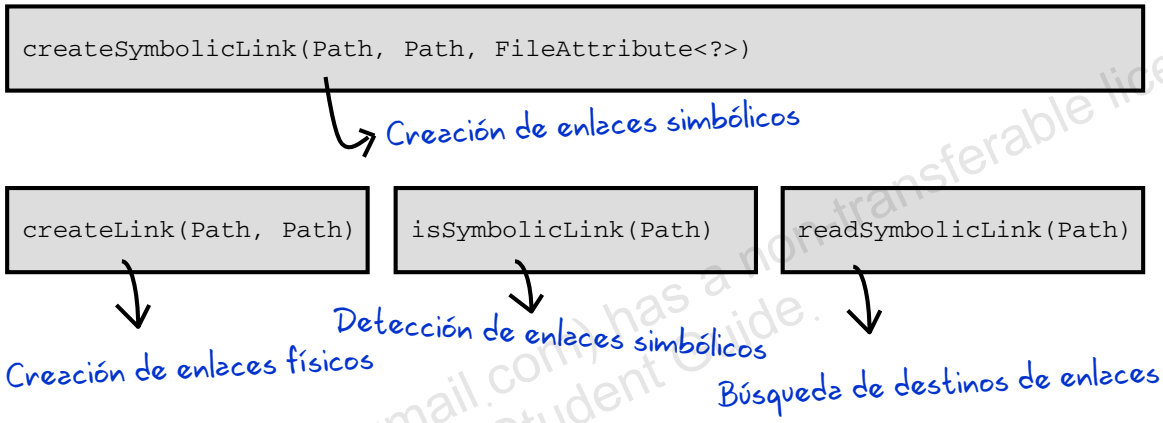
ORACLE



Un requisito común al escribir código de E/S de archivo es la capacidad de construir una ruta desde una ubicación en el sistema de archivos a otra ubicación. Esto se consigue con el método `relativize`. Este método construye una ruta que empieza en la ruta original y termina en la ubicación especificada por la ruta transferida. La nueva ruta es relativa a la ruta original.

# Trabajo con enlaces

- La interfaz `Path` reconoce enlaces.
- Todos los métodos `Path`:
  - detectan qué hacer al encontrarse un enlace simbólico; o
  - proporcionan una opción que permite configurar el comportamiento en caso de encontrarse un enlace simbólico.




ORACLE



El paquete `java.nio.file` y la interfaz `Path` en particular reconocen enlaces. Todos los métodos `Path` detectan qué hacer en caso de encontrarse un enlace simbólico, o proporcionan una opción para configurar el comportamiento en caso de encontrarse un enlace simbólico.

Algunos sistemas de archivos soportan también enlaces físicos. Los enlaces físicos son más restrictivos que los enlaces simbólicos:

- El destino del enlace debe existir.
- Generalmente no se pueden usar enlaces físicos para los directorios.
- Los enlaces físicos no pueden cruzar particiones ni volúmenes. Por lo tanto, no pueden existir en distintos sistemas de archivos al mismo tiempo.
- Los enlaces físicos parecen, y se comportan como, archivos normales, por lo que pueden resultar difíciles de encontrar.
- Los enlaces físicos son, a todos los efectos, la misma entidad que los archivos originales. Tienen los mismos permisos de archivo, los mismos registros de hora, etc. Todos los atributos son idénticos.

Debido a estas restricciones, los enlaces físicos no se usan con la misma frecuencia que los enlaces simbólicos, aunque los métodos `Path` funcionan sin problema con los enlaces físicos.

# Prueba

Dado un objeto `Path` con la siguiente ruta de acceso:

`/export/home/heimer/../williams/./documents`

¿Qué método `Path` eliminaría los elementos redundantes?

a. `normalize`
b. `relativize`
c. `resolve`
d. `toAbsolutePath`

ORACLE

# Prueba

Dada la siguiente ruta:

```
Path p = Paths.get
("/home/export/tom/documents/coursefiles/JDK7");
```

y la sentencia:

```
Path sub = p.subPath (x, y);
```

¿Qué valores para x e y producirán un objeto `Path` que contenga `documents/coursefiles`?

a. x = 3, y = 4
b. x = 3, y = 5
c. x = 4, y = 5
d. x = 4, y = 6

ORACLE

# Prueba

Dado el siguiente fragmento de código:

```
Path p1 = Paths.get("D:/temp/foo/");
Path p2 = Paths.get("../bar/documents");
Path p3 = p1.resolve(p2).normalize();
System.out.println(p3);
```

¿Cuál es el resultado?

a. Error del compilador
b. Excepción de E/S
c. `D:\temp\foo\documents`
d. `D:\temp\bar\documents`
e. `D:\temp\foo\..\bar\documents`

ORACLE

# Operaciones `File`

- Comprobación de un directorio o un archivo
- Supresión de un directorio o un archivo
- Copia de un directorio o un archivo
- Desplazamiento de un directorio o un archivo
- Gestión de metadatos
- Lectura, escritura y creación de archivos
- Archivos de acceso aleatorio
- Creación y lectura de directorios

ORACLE







La clase `java.nio.file.Files` está en el punto de entrada principal para operaciones con los objetos `Path`.

Los métodos estáticos de esta clase leen, escriben y manipulan archivos y directorios representados por objetos `Path`.

La clase `Files` también reconoce enlaces. Sus métodos detectan los enlaces simbólicos en objetos `Path` y proporcionan o gestionan automáticamente opciones para tratar los enlaces.

## Comprobación de un directorio o un archivo

Los objetos `Path` representan el concepto de una ubicación de archivo o de directorio. Para poder acceder al archivo o al directorio es necesario acceder primero al sistema de archivos y determinar si existe o no con los siguientes métodos `Files`:

- `exists(Path p, LinkOption... option)`
  Realiza pruebas para ver si existe un archivo. Por defecto, le siguen enlaces simbólicos.
- `notExists(Path p, LinkOption... option)`
  Realiza pruebas para ver si no existe un archivo. Por defecto, le siguen enlaces simbólicos.
- Ejemplo:

```
Path p = Paths.get(args[0]);
System.out.format("Path %s exists: %b%n", p,
                  Files.exists(p, LinkOption.NOFOLLOW_LINKS));
```

Argumento opcional

ORACLE







Recuerde que los objetos `Path` pueden apuntar a archivos o a directorios que no existen. Los métodos `exists()` y `notExists()` se utilizan para determinar si `Path` apunta a un directorio o un archivo legítimo, y los detalles de este archivo o directorio.

Cuando se comprueba la existencia de un archivo se pueden obtener tres resultados:

- Se puede verificar que existe el archivo.
- Se puede verificar que no existe el archivo.
- Que el estado del archivo sea desconocido. Este resultado se puede producir cuando el programa no tiene acceso al archivo.

**Nota:** `!Files.exists(path)` no es equivalente a `Files.notExists(path)`. Si ambos, `exists` y `notExists`, devuelven false, no se puede determinar la existencia del archivo o del directorio. Por ejemplo, en Windows se puede conseguir solicitando el estado de una unidad fuera de línea, como una unidad de CD-ROM.

## Comprobación de un directorio o un archivo

Para verificar que se puede acceder a un archivo, la clase `Files` proporciona los siguientes métodos `boolean`.

- `isReadable (Path)`
- `isWriteable (Path)`
- `isExecutable (Path)`

Tenga en cuenta que estas pruebas no son atómicas con respecto al resto de las operaciones del sistema de archivos. Por lo tanto, es posible que los resultados de estas pruebas no resulten fiables una vez terminen los métodos.

- El método `isSameFile (Path, Path)` realiza pruebas para comprobar si hay dos rutas que apuntan al mismo archivo. Esto resulta especialmente útil en sistemas de archivos que soportan enlaces simbólicos.

ORACLE



El resultado de cualquiera de estas pruebas queda inmediatamente obsoleto una vez finalizada la operación. Según se explica la documentación: “Tenga en cuenta que el resultado de este método queda inmediatamente obsoleto. No existe garantía de que un intento posterior de abrir el archivo para escribir termine con éxito (ni incluso de que acceda al mismo archivo). Preste atención a la hora de usar este método en aplicaciones especialmente sensibles a la seguridad”.

# Creación de archivos y directorios

Se pueden crear archivos y directorios con uno de los siguientes métodos:

```
Files.createFile (Path dir);
Files.createDirectory (Path dir);
```

- El método `createDirectories` se puede utilizar para crear directorios que no existen, de arriba abajo:

```
Files.createDirectories(Paths.get("D:/Temp/foo/bar/example"));
```



La clase `Files` también incluye métodos para crear archivos y directorios temporales, discos duros y enlaces simbólicos.

## Supresión de un directorio o un archivo

Puede suprimir archivos, directorios o enlaces. La clase `Files` proporciona dos métodos:

- `delete(Path)`
- `deleteIfExists(Path)`

```
//...
Files.delete(path);
//...
```

Devuelve NoSuchFileException, DirectoryNotEmptyException o IOException

```
//...
Files.deleteIfExists(Path)
//...
```

No devuelve ninguna excepción



El método `delete(Path)` suprime el archivo o devuelve una excepción si falla la operación de supresión. Por ejemplo, si no existe el archivo, se devuelve `NoSuchFileException`.

El método `deleteIfExists(Path)` suprime también el archivo, pero si el archivo no existe, no se devuelve ninguna excepción. Generar un fallo en modo silencioso resulta útil cuando se tienen varios threads para suprimir archivos y no se desea devolver una excepción solo porque un thread lo hizo primero.

# Copia de un directorio o un archivo

- Puede copiar un archivo o un directorio mediante el método `copy(Path, Path, CopyOption...)`.
- Cuando se copian directorios, los archivos que incluyen no se copian.

Parámetros de StandardCopyOption

```
//...
copy(Path, Path, CopyOption...)
//...
```

**REPLACE_EXISTING**
**COPY_ATTRIBUTES**
**NOFOLLOW_LINKS**

- Ejemplo:

```
import static java.nio.file.StandardCopyOption.*;
//...
Files.copy(source, target, REPLACE_EXISTING, NOFOLLOW_LINKS);
```

ORACLE







Puede copiar un archivo o un directorio mediante el método `copy(Path, Path, CopyOption...)`. Si el archivo de destino existe, la operación de copia fallará, a menos que se haya especificado la opción `REPLACE_EXISTING`.

Si bien se pueden copiar directorios, los archivos dentro del directorio no se copian, por lo que el nuevo directorio quedará vacío incluso aunque el directorio original contenga archivos.

Cuando se copia un enlace simbólico, se copia el destino del enlace. Si desea copiar el enlace en sí, y no su contenido, especifique la opción `NOFOLLOW_LINKS` o la opción `REPLACE_EXISTING`.

Se soportan las siguientes enumeraciones `StandardCopyOption` y `LinkOption`:

- **`REPLACE_EXISTING`:** realiza la copia, incluso si ya existe el archivo de destino. Si el destino es un enlace simbólico, se copia el enlace en sí (y no el destino del enlace). Si el destino es un directorio que no está vacío, la operación fallará con la excepción `FileAlreadyExistsException`.
- **`COPY_ATTRIBUTES`:** copia los atributos de archivo asociados con el archivo en el archivo de destino. Los atributos de archivo concretos que se soportan son dependientes del sistema de archivos y de la plataforma, pero el atributo de momento de la última modificación es uno que soportan las distintas plataformas y se copia en el archivo de destino.
- **`NOFOLLOW_LINKS`:** indica que no se deben seguir los enlaces simbólicos. Si el archivo que se va a copiar es un enlace simbólico, se copia el enlace en sí (y no el destino del enlace).

# Copia entre un flujo y una ruta

También puede interesarle poder copiar (o escribir) desde un objeto `Stream` a un archivo o desde un archivo a un objeto `Stream`. La clase `Files` proporciona dos métodos que facilitan esta tarea:

```
copy(InputStream source, Path target, CopyOption... options)
copy(Path source, OutputStream out)
```

- Un uso interesante de este primer método es copiar desde una página web y guardar en un archivo:

```
 Path path = Paths.get("D:/Temp/oracle.html");
URI u = URI.create("http://www.oracle.com/");
try (InputStream in = u.toURL().openStream()) {
     Files.copy(in, path, StandardCopyOption.REPLACE_EXISTING);
} catch (final MalformedURLException | IOException e) {
     System.out.println("Exception: " + e);
}
```

ORACLE



La alternativa a la copia del flujo a la ruta es el método de ruta a flujo. Este método se puede usar para escribir un archivo en un socket o en otro tipo de flujo.

# Desplazamiento de un directorio o un archivo

- Puede desplazar un archivo o un directorio mediante el método `move(Path, Path, CopyOption...)`.
- Al desplazar un directorio, no se desplazará su contenido.

Parámetros de StandardCopyOption

```
//...
move(Path, Path, CopyOption...)
//...
```

**REPLACE_EXISTING**
**ATOMIC_MOVE**

- Ejemplo:

```
import static java.nio.file.StandardCopyOption.*;
//...
Files.move(source, target, REPLACE_EXISTING);
```

ORACLE







Directrices para realizar desplazamientos:

- Si la ruta de destino es un directorio y ese directorio está vacío, el desplazamiento se realizará correctamente si se ha definido `REPLACE_EXISTING`.
- Si el directorio de destino no existe, el desplazamiento se realizará correctamente. Básicamente, esto es cambiar de nombre el directorio.
- Si el directorio de destino existe y no está vacío, se devuelve `DirectoryNotEmptyException`.
- Si el origen es un archivo, el destino es un directorio que existe y se define `REPLACE_EXISTING`, la operación de desplazamiento cambiará el nombre del archivo al nombre de directorio en cuestión.

Para desplazar un directorio con archivos a otro directorio, esencialmente necesita copiar de forma recurrente el contenido del directorio y suprimir el directorio antiguo.

También puede hacer el desplazamiento como una operación de archivo atómica mediante `ATOMIC_MOVE`.

- Si el sistema de archivos no soporta desplazamientos atómicos, se devuelve una excepción. `ATOMIC_MOVE` le permite desplazar un archivo a un directorio con la garantía de que ningún proceso que esté observando el directorio acceda a un archivo entero.

# Listado del contenido de un directorio

La clase `DirectoryStream` proporciona un mecanismo para iterar sobre todas las entradas de un directorio.

```
1 Path dir = Paths.get("D:/Temp");
2 // DirectoryStream is a stream, so use try-with-resources
3 // or explicitly close it when finished
4 try (DirectoryStream<Path> stream =
5                       Files.newDirectoryStream(dir, "*.zip")) {
6     for (Path file : stream) {
7         System.out.println(file.getFileName());
8     }
9 } catch (PatternSyntaxException | DirectoryIteratorException |
10          IOException x) {
11     System.err.println(x);
12 }
```

- `DirectoryStream` escala para soportar directorios de gran tamaño.

ORACLE



La clase `Files` proporciona un método para devolver un objeto `DirectoryStream`, el cual se puede usar para iterar sobre todos los archivos y directorios desde cualquier directorio `Path` (raíz).

Si se produce un error de E/S al iterar sobre todas las entradas en el directorio especificado, se devuelve `DirectoryIteratorException`.

Si el patrón proporcionado (segundo argumento del método) no es válido, se devuelve `PatternSyntaxException`.

# Lectura o escritura de todos los bytes o líneas de un archivo

- Los métodos `readAllBytes` o `readAllLines` leen el contenido completo de un archivo en una sola transferencia.
- Ejemplo:

```
Path source = ...;
List<String> lines;
Charset cs = Charset.defaultCharset();
lines = Files.readAllLines(file, cs);
```

- Utilice métodos `write` para escribir bytes o líneas en un archivo.

```
Path target = ...;
Files.write(target, lines, cs, CREATE, TRUNCATE_EXISTING, WRITE);
```

Enumeraciones `StandardOpenOption`.

ORACLE



Si tiene un archivo pequeño y quiere leer todo su contenido en una sola transferencia, puede usar el método `readAllBytes(Path)` o el método `readAllLines(Path, Charset)`. Estos métodos se ocupan de la mayor parte del trabajo, como abrir y cerrar el flujo, pero al traer el archivo entero a la memoria de una sola vez, no deben usarse para manejar archivos de gran tamaño.

Puede usar uno de los métodos de escritura para escribir bytes o líneas en un archivo.

- `write(Path, byte[], OpenOption...)`
- `write(Path, Iterable<? extends CharSequence>, Charset, OpenOption...)`

# Canales y `ByteBuffers`

- La E/S de flujo lee un carácter cada vez, mientras que la E/S de canal lee un buffer cada vez.
- La interfaz `ByteChannel` proporciona una funcionalidad de lectura y escritura básica.
- Un objeto `SeekableByteChannel` es un objeto `ByteChannel` que tiene la capacidad de mantener una posición en el canal y cambiarla.
- Los dos métodos para escribir y leer E/S de canal son:

```
newByteChannel(Path, OpenOption...)
newByteChannel(Path, Set<? extends OpenOption>, FileAttribute<?>...)
```

- La capacidad de realizar desplazamientos a diferentes puntos en un archivo y después leer o escribir desde esa ubicación hace posible un acceso aleatorio a un archivo.





NIO.2 soporta E/S de canal y de flujo en buffer.

La E/S de flujo lee un carácter cada vez, mientras que la E/S de canal lee un buffer cada vez. La interfaz `ByteChannel` proporciona una funcionalidad de lectura y escritura básica. Un objeto `SeekableByteChannel` es un objeto `ByteChannel` que tiene la capacidad de mantener una posición en el canal y realizar una consulta al archivo según su tamaño.

La capacidad de realizar desplazamientos a diferentes puntos en un archivo y después leer o escribir desde esa ubicación hace posible un acceso aleatorio a un archivo.

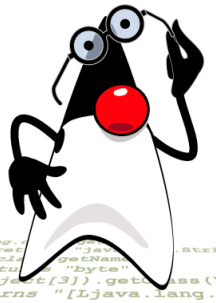
Hay dos métodos para escribir y leer E/S de canal:

- `newByteChannel(Path, OpenOption...)`
- `newByteChannel(Path, Set<? extends OpenOption>, FileAttribute<?>...)`

**Nota:** el método `newByteChannel` devuelve una instancia de `SeekableByteChannel`. Con un sistema de archivos por defecto se puede difundir este canal de bytes susceptible de búsqueda a `FileChannel`, y con ello proporcionar acceso a funciones más avanzadas, como asignar una zona del archivo directamente a la memoria para un acceso más rápido, bloquear una zona del archivo para que no puedan acceder a ella otros procesos, o leer y escribir bytes desde una posición absoluta sin afectar a la posición actual del canal. Consulte la lección “Conceptos fundamentales de E/S” para ver un ejemplo del uso de `FileChannel`.

# Archivos de acceso aleatorio

- Los archivos de acceso aleatorio permiten acceder aleatoria y no secuencialmente al contenido de un archivo.
- Para acceder a un archivo aleatoriamente, abra el archivo, busque una ubicación concreta y lea desde el archivo o escriba en él.
- La funcionalidad de acceso aleatorio se activa mediante la interfaz `SeekableByteChannel`.



**position()**

**write(ByteBuffer)**

**position(long)**

**read(ByteBuffer)**

**truncate(long)**

ORACLE



Los archivos de acceso aleatorio permiten acceder aleatoria y no secuencialmente al contenido de un archivo. Para acceder a un archivo aleatoriamente, abra el archivo, busque una ubicación concreta y lea desde el archivo o escriba en él.

Esta funcionalidad se activa mediante la interfaz `SeekableByteChannel`. La interfaz `SeekableByteChannel` amplía la E/S de canal con la noción de una posición actual. Los métodos le permiten establecer o consultar la posición y después leer datos desde esa ubicación o escribir datos en ella. La API incluye unos pocos métodos fáciles de usar:

- **`position()`:** devuelve la posición actual del canal.
- **`position(long)`:** establece la posición del canal.
- **`read(ByteBuffer)`:** lee bytes en el buffer desde el canal.
- **`write(ByteBuffer)`:** escribe bytes desde el buffer al canal.
- **`truncate(long)`:** trunca el archivo (u otra entidad) conectado al canal.

# Métodos de E/S en buffer para archivos de texto

- El método `newBufferedReader` abre un archivo para su lectura.

```
//...
BufferedReader reader = Files.newBufferedReader(file, charset);
line = reader.readLine();
```

- El método `newBufferedWriter` escribe en un archivo a través de un objeto `BufferedWriter`.

```
//...
BufferedWriter writer = Files.newBufferedWriter(file, charset);
writer.write(s, 0, s.length());
```



**Lectura de un archivo mediante E/S de flujo en buffer**

El método `newBufferedReader(Path, Charset)` abre un archivo para su lectura y devuelve un objeto `BufferedReader` que se puede utilizar para leer texto desde un archivo de manera eficiente.

# Flujos de bytes

- NIO.2 soporta también métodos para abrir flujos de bytes.

```
InputStream in = Files.newInputStream(file);
BufferedReader reader = new BufferedReader(new InputStreamReader(in));
line = reader.readLine();
```

- Para crear, agregar a un archivo o escribir en él, utilice el método `newOutputStream`.

```
import static java.nio.file.StandardOpenOption.*;
//...
Path logfile = ...;
String s = ...;
byte data[] = s.getBytes();
OutputStream out =
       new BufferedOutputStream(file.newOutputStream(CREATE, APPEND));
out.write(data, 0, data.length);
```

# Gestión de metadatos

| Método | Explicación |
|---|---|
| `size` | Devuelve el tamaño del archivo especificado en bytes. |
| `isDirectory` | Devuelve true si el objeto `Path` especificado localiza un archivo que es un directorio. |
| `isRegularFile` | Devuelve true si el objeto `Path` especificado localiza un archivo que es un archivo normal. |
| `isSymbolicLink` | Devuelve true si el objeto `Path` especificado localiza un archivo que es un enlace simbólico. |
| `isHidden` | Devuelve true si el objeto `Path` especificado localiza un archivo considerado oculto por el sistema de archivos. |
| `getLastModifiedTime` | Devuelve o establece la fecha de última modificación del archivo especificado. |
| `setLastModifiedTime` | |
| `getAttribute` | Devuelve o establece el valor de un atributo de archivo. |
| `setAttribute` | |

ORACLE



Si un programa necesita usar varios atributos de archivo al mismo tiempo, puede que usar métodos que recuperen un solo atributo no sea eficiente. Acceder repetidamente al sistema de archivos para recuperar un solo atributo puede afectar negativamente al rendimiento. Por esto motivo, la clase `Files` proporciona dos métodos `readAttributes` para recuperar los atributos de un archivo en una sola operación en bloque.

- `readAttributes(Path, String, LinkOption...)`
- `readAttributes(Path, Class<A>, LinkOption...)`

# Atributos de archivo (DOS)

- Los atributos de archivo se pueden leer desde un archivo o un directorio en una sola llamada:

```
DosFileAttributes attrs =
   Files.readAttributes (path, DosFileAttributes.class);
```

- Los sistemas de archivos DOS pueden modificar atributos después de crearse los archivos:

```
Files.createFile (file);
Files.setAttribute (file, "dos:hidden", true);
```



Los tipos `setAttribute` (en DOS) amplían `BasicFileAttributeView` y ven los cuatro bits estándar en sistemas de archivos que soportan atributos de DOS:

- `dos:hidden`
- `dos:readonly`
- `dos:system`
- `dos:archive`

Otras vistas de atributos soportadas son:

- `BasicFileAttributeView`: proporciona un juego de atributos básicos que soportan todas las implantaciones de sistemas de archivos.
- `PosixFileAttributeView`: amplía `BasicFileAttributeView` con atributos que soportan la familia de estándares POSIX, como UNIX.
- `FileOwnerAttributeView`: la soporta cualquier implantación de sistema de archivos que soporte el concepto de propiedad de archivo.
- `AclFileAttributeView`: soporta la lectura o actualización de la lista de control de acceso (ACL) de un archivo. Se soporta el modelo NFSv4 ACL.
- `UserDefinedFileAttributeView`: permite usar metadatos definidos por el usuario.

# Atributos de archivo DOS: ejemplo

```
DosFileAttributes attrs = null;
Path file = ...;
try { attrs =
          Files.readAttributes(file, DosFileAttributes.class);
} catch (IOException e) { ///... }
FileSystem creation = attrs.creationTime();
FileSystem modified = attrs.lastModifiedTime();
FileSystem lastAccess = attrs.lastAccessTime();
if (!attrs.isDirectory()) {
   long size = attrs.size();
}
// DosFileAttributes adds these to BasicFileAttributes
boolean archive = attrs.isArchive();
boolean hidden = attrs.isHidden();
boolean readOnly = attrs.isReadOnly();
boolean systemFile = attrs.isSystem();
```

ORACLE



El fragmento de código en la diapositiva ilustra el uso de la clase `DosFileAttributes`. En la llamada (única) al método `readAttributes`, se devuelven los atributos del archivo (o del directorio).

# Permisos de POSIX

NIO.2 permite crear archivos y directorios en sistemas de archivos POSIX con sus primeros permisos establecidos.

```
1  Path p = Paths.get(args[0]);
2  Set<PosixFilePermission> perms =
3      PosixFilePermissions.fromString("rwxr-x---");
4  FileAttribute<Set<PosixFilePermission>> attrs =
5      PosixFilePermissions.asFileAttribute(perms);
6  try {
7      Files.createFile(p, attrs);
8  } catch (FileAlreadyExistsException f) {
9      System.out.println("FileAlreadyExists" + f);
10 } catch (IOException i) {
11     System.out.println("IOException:" + i);
12 }
```

Crear un archivo en `Path p` con atributos opcionales.

ORACLE







Los sistemas de archivos que implantan el estándar POSIX (Portable Operating System Interface) pueden crear archivos y directorios con sus primeros permisos establecidos. Esto resuelve un problema común en la creación de archivos a la hora de programar las operaciones de E/S. Los permisos en ese archivo se pueden cambiar antes de la siguiente ejecución para establecer permisos.

Se pueden establecer permisos solo para sistemas de archivos compatibles con POSIX, como MacOS, Linux y Solaris. Windows (basado en DOS) no es compatible con POSIX. Los archivos y directorios basados en DOS no tienen permisos, sino más bien atributos de archivo.

**Nota:** puede determinar si un sistema de archivos soporta o no POSIX en cuanto a programación consultando las vistas de atributos de archivo soportadas. Por ejemplo:

```
boolean unixFS = false;
Set<String> views =
    FileSystems.getDefault().supportedFileAttributeViews();
for (String s : views) {
    if (s.equals("posix")) unixFS = true;
}
```

# Prueba

Dado el siguiente fragmento:

```
Path p1 = Paths.get("/export/home/peter");
Path p2 = Paths.get("/export/home/peter2");
Files.move(p1, p2, StandardCopyOption.REPLACE_EXISTING);
```

Si el directorio `peter2` no existe, y el directorio `peter` tiene dentro subcarpetas y archivos, ¿cuál es el resultado?

a. `DirectoryNotEmptyException`
b. `NotDirectoryException`
c. Se crea el directorio `peter2`.
d. Se copia el directorio `peter` en `peter2`.
e. Se crea el directorio `peter2` y se copian en él los archivos y directorios de `peter`.

ORACLE

# Prueba

Dado el siguiente fragmento:

```
Path source = Paths.get(args[0]);
Path target = Paths.get(args[1]);
File.copy(source, target);
```

Suponiendo que `source` y `target` no son directorios, ¿cómo puede evitar que esta operación de copia genere `FileAlreadyExistsException`?

a. Suprimiendo el archivo `target` antes de copiar.
b. Utilizando en su lugar el método `move`.
c. Utilizando en su lugar el método `copyExisting`.
d. Agregando la opción `REPLACE_EXISTING` al método.

ORACLE

# Prueba

Dado el siguiente fragmento:

```
Path source = Paths.get("/export/home/mcginn/HelloWorld.java");
Path newdir = Paths.get("/export/home/heimer");
Files.copy(source, newdir.resolve(source.getFileName());
```

Suponiendo que no hay excepciones, ¿cuál sería el resultado?

a. El contenido de `mcginn` se copiará en `heimer`.
b. `HelloWorld.java` se copiará en `/export/home`.
c. `HelloWorld.java` se copiará en `/export/home/heimer`.
d. El contenido de `heimer` se copiará en `mcginn`.

# Visión general de la práctica 11-1: Escritura de una aplicación de fusión de archivos

En esta práctica, utilice la interfaz `Path` y la clase `Files` para abrir un formulario de plantilla de carta, y sustituya el nombre en la plantilla por un nombre extraído de un archivo que contiene una lista de nombres.

- Utilice la interfaz `Path` para crear un nuevo nombre de archivo para la carta personalizada.
- Utilice la clase `Files` para leer todas las cadenas de ambos archivos a objetos `List`.
- Utilice las clases `Matcher` y `Pattern` para buscar el token que sustituir en la plantilla.

ORACLE

# Operaciones recursivas

La clase `Files` ofrece un método para recorrer el árbol de archivos en busca de operaciones recursivas, como de copia y de supresión.

- `walkFileTree (Path start, FileVisitor<T>)`
- Ejemplo:

```
public class PrintTree implements FileVisitor<Path> {
    public FileVisitResult preVisitDirectory(Path, BasicFileAttributes){}
    public FileVisitResult postVisitDirectory(Path, BasicFileAttributes){}
    public FileVisitResult visitFile(Path, BasicFileAttributes){}
    public FileVisitResult visitFileFailed(Path, BasicFileAttributes){}
}
```

```
public class WalkFileTreeExample {
    public printFileTree(Path p) {
        Files.walkFileTree(p, new PrintTree());
    }
}
```

El árbol de archivos se explora de forma recurrente. Se llama como directorios a los métodos definidos por `PrintTree` y se llega a los archivos en el árbol. Cada método se transfiere a la ruta actual como el primer argumento del método.

ORACLE



La interfaz `FileVisitor` incluye métodos a los que se va llamando a medida que se visita cada uno de los nodos del árbol:

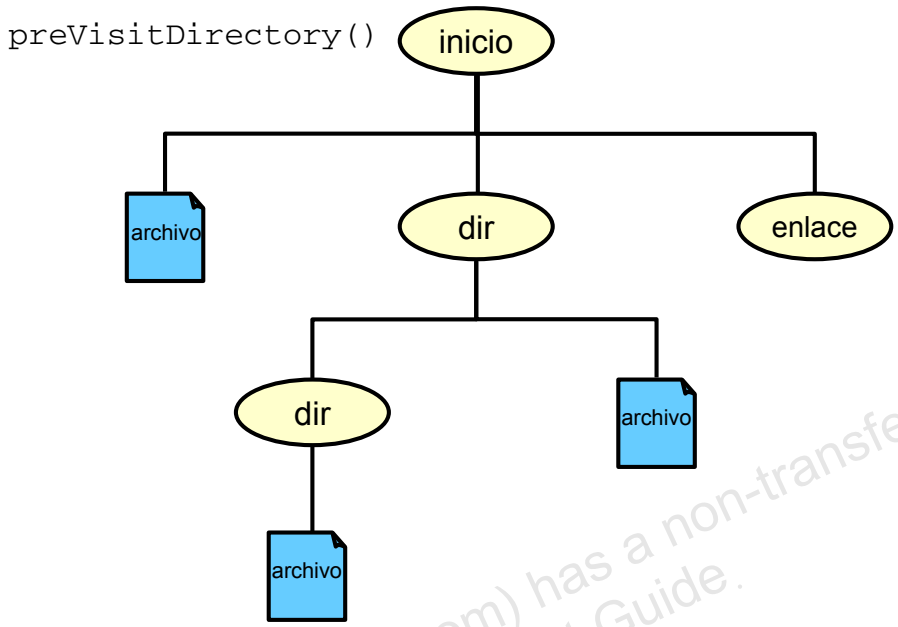
- `preVisitDirectory`: se invoca en un directorio antes de que se visiten las entradas del directorio.
- `visitFile`: se invoca para un archivo en un directorio.
- `postVisitDirectory`: se invoca después de que se hayan visitado todas las entradas en un directorio y sus descendientes.
- `visitFileFailed`: se invoca para un archivo que no se haya podido visitar.

El resultado devuelto de cada uno de los métodos llamados determina las acciones que tomar tras alcanzar un nodo (antes o después). Estas acciones se enumeran en la clase `FileVisitResult`:

- `CONTINUE`: continuar al siguiente nodo.
- `SKIP_SIBLINGS`: continuar sin visitar a los hermanos del archivo o del directorio.
- `SKIP_SUBTREE`: continuar sin visitar las entradas en este directorio.
- `TERMINATE`

**Nota:** existe también una clase, `SimpleFileVisitor`, que implanta cada método en `FileVisitor` con un tipo de retorno `FileVisitResult.CONTINUE` o que devuelve `IOException`. Si planea utilizar algunos de los métodos de la interfaz `FileVisitor`, esta clase es más fácil de ampliar sustituyendo los métodos que necesite.

# Orden del método `FileVisitor`





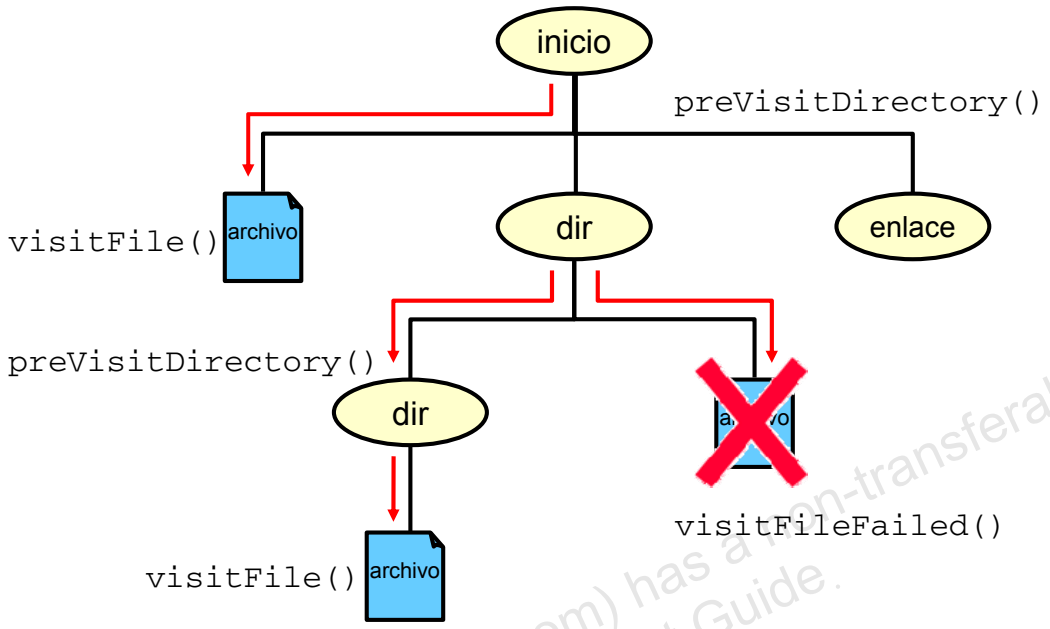

**`FileVisitor`**

Empezando por el primer nodo de directorio y en cada subdirectorio que se encuentre, al método `preVisitDirectory(Path, BasicFileAttributes)` se le llama en la clase transferida al método `walkFileTree`.

Si el tipo de retorno al invocar a `preVisitDirectory()` es `FileVisitResult.CONTINUE`, se explorará el nodo siguiente.

**Nota:** la transversal del árbol de archivos es una transversal de profundidad con `FileVisitor` invocado con cada archivo encontrado. La transversal del árbol de archivos finaliza cuando se han visitado todos los archivos accesibles en el árbol, o cuando un método de visita devuelve un resultado `TERMINATE`. Cuando un método de visita termina porque se produce una excepción `IOException`, un error uncaught o una excepción de tiempo de ejecución, la transversal se termina y se propaga el error o la excepción al emisor del método.

## Orden del método `FileVisitor`

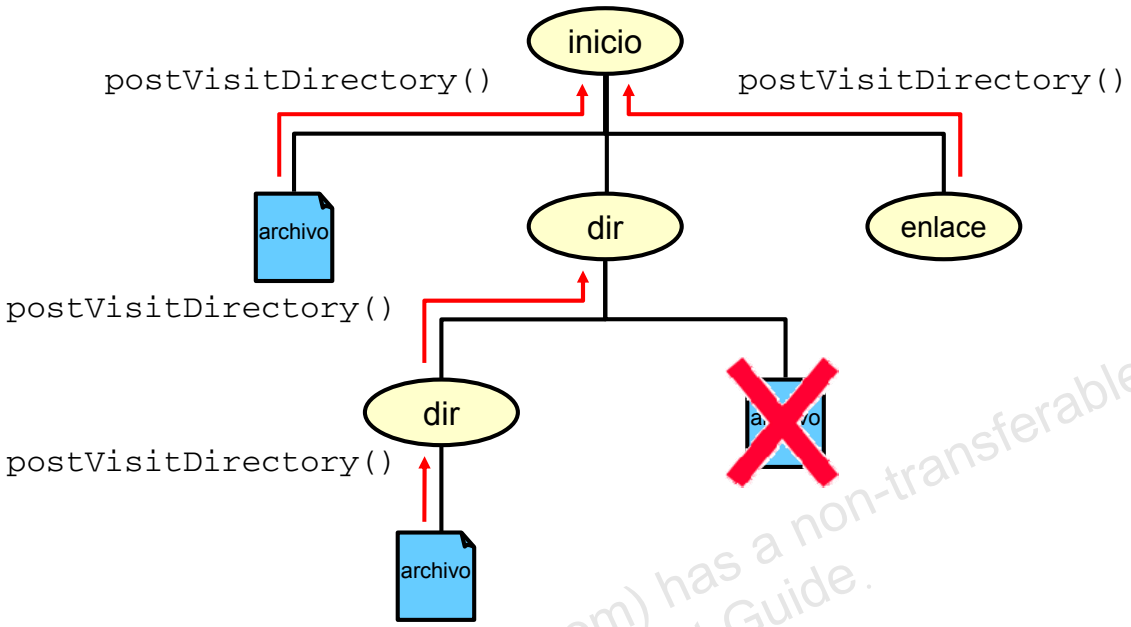





Cuando se encuentra un archivo en el árbol, el método `walkFileTree` intenta leer sus atributos `BasicFileAttributes`. Si el archivo no es un directorio, se llama al método `visitFile` con los atributos de archivo. Si los atributos del archivo no se pudieran leer debido a una excepción de E/S, se llama al método `visitFileFailed` con la excepción de E/S.

.

## Orden del método `FileVisitor`



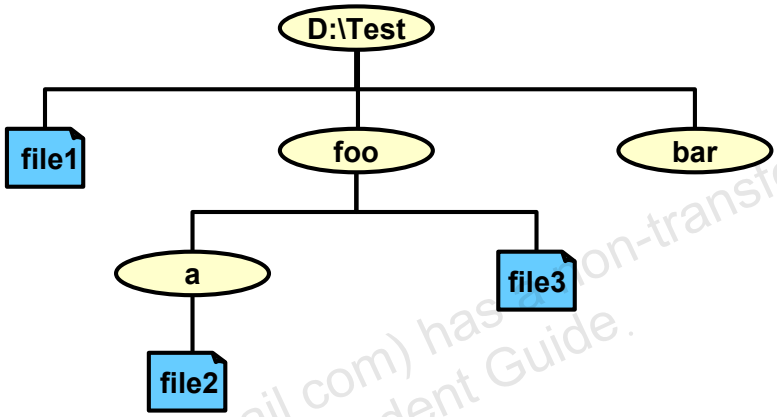



Una vez se han alcanzado todos los objetos secundarios del nodo, se invoca al método `postVisitDirectory` en cada uno de los directorios.

**Nota:** en la progresión que aquí se muestra, se entiende que el tipo de retorno `FileVisitResult` es `CONTINUE` por cada uno de los métodos `FileVisitor`.

# Ejemplo: WalkFileTreeExample

```
Path path = Paths.get("D:/Test");
try {
    Files.walkFileTree(path, new PrintTree());
} catch (IOException e) {
    System.out.println("Exception: " + e);
}
```






En este ejemplo, la clase `PrintTree` implanta cada uno de los métodos en `FileVisitor` e imprime el tipo, el nombre y el tamaño del directorio y el archivo en cada nodo. Si se usara el diagrama que se muestra en la diapositiva, el resultado sería el siguiente (en Windows):

```
preVisitDirectory: Directory: D:\Test (0 bytes)
preVisitDirectory: Directory: D:\Test\bar (0 bytes)
postVisitDirectory: Directory: D:\Test\bar
visitFile: Regular file: D:\Test\file1 (328 bytes)
preVisitDirectory: Directory: D:\Test\foo (0 bytes)
preVisitDirectory: Directory: D:\Test\foo\a (0 bytes)
visitFile: Regular file: D:\Test\foo\a\file2 (22 bytes)
postVisitDirectory: Directory: D:\Test\foo\a
visitFile: Regular file: D:\Test\foo\file3 (12 bytes)
postVisitDirectory: Directory: D:\Test\foo
postVisitDirectory: Directory: D:\Test
```

El código completo para este ejemplo está en el proyecto `examples/WalkFileTreeExample`.

# Búsqueda de archivos

Para localizar un archivo, generalmente se examina un directorio. Se podría usar una herramienta de búsqueda, o un comando como:

```
dir /s *.java
```

- Este comando examinará de forma recurrente el árbol del directorio, empezando por donde se encuentre y mirando todos los archivos que contengan la extensión `.java`.

La interfaz `java.nio.file.PathMatcher` incluye un método de coincidencia para determinar si un objeto `Path` coincide con una cadena de búsqueda especificada.

- Cada implantación de sistema de archivos proporciona un objeto `PathMatcher` recuperable mediante la fábrica `FileSystems`:

```
PathMatcher matcher = FileSystems.getDefault().getPathMatcher
(String syntaxAndPattern);
```

ORACLE

# Patrón y sintaxis de PathMatcher

- La cadena `syntaxAndPattern` presenta la siguiente forma:
  *`sintaxis:patrón`*
  Donde *`sintaxis`* puede ser “glob” y “regex”.
- La sintaxis glob es parecida a las expresiones regulares, pero más simple:

| Ejemplo de patrón | Coincidencias |
|---|---|
| `*.java` | Una ruta que representa el nombre de archivo que termina en `.java`. |
| `*.*` | Encuentra coincidencias de nombres de archivos que tienen un punto. |
| `*.{java,class}` | Encuentra coincidencias de nombres de archivos que terminan en `.java` o en `.class`. |
| `foo.?` | Encuentra coincidencias de nombres de archivos que empiezan por `foo.` y tienen una extensión de un único carácter. |
| `C:\\*` | Encuentra coincidencias de `C:\foo` y `C:\bar` en la plataforma de Windows (observe cómo la barra diagonal inversa tiene carácter de escape. Como un literal de cadena en el lenguaje Java, el patrón sería `C:\\*`). |

ORACLE



Las reglas a continuación se usan para interpretar patrones glob:

- El carácter `*` busca coincidencias de cero o más caracteres de un componente nombre sin cruzar límites de directorios.
- Los caracteres `**` encuentran coincidencias de cero o más caracteres cruzando límites de directorios.
- El carácter `?` busca coincidencias exactas de un carácter de un componente nombre.
- El carácter de barra diagonal inversa (`\`) se utiliza para incluir un escape en caracteres que si no se interpretarían como caracteres especiales. La expresión `\\` busca coincidencias de una barra diagonal inversa, mientras que `\{` busca coincidencias de un corchete de apertura, por ejemplo.
- Los caracteres `[ ]` son una expresión de corchetes cuadrados que buscan coincidencias de un solo carácter en el componente nombre de un juego de caracteres. Por ejemplo, `[abc]` busca coincidencias de `a`, `b` o `c`. El guion (-) especifica un rango. Así, `[a-z]` especifica un rango que busca coincidencias desde `a` hasta `z` (incluidas). Estas formas se pueden también mezclar, de modo que `[abce-g]` buscará coincidencias de `a`, `b`, `c`, `e`, `f` o `g`. Si el carácter después de `[` es `!`, entonces se usa para indicar una negación. Así, `[!a-c]` buscará coincidencias de cualquier carácter salvo `a`, `b` o `c`.

- Dentro de una expresión entre corchetes cuadrados, los caracteres *, ? y \ coinciden consigo mismos. El carácter (-) coincide consigo mismo si el primer carácter dentro de los corchetes, o el primer carácter después del ! si se trata de una negación.
- Los caracteres { } son un grupo de subpatrones, donde el grupo coincide si coincide alguno de los subpatrones del grupo. El carácter "," se utiliza para separar los subpatrones. Los grupos no se pueden anidar.
- Los caracteres iniciales de punto en un nombre de archivo se tratan como caracteres normales en operaciones de coincidencia. Por ejemplo, el patrón "*" glob coincide con el nombre de archivo `.login`. El método `Files.isHidden(java.nio.file.Path)` se puede usar para probar si un archivo se considera oculto.
- El resto de los caracteres coinciden con ellos mismos de forma dependiente con la implantación. Esto incluye caracteres que representen cualquier separador de nombres.
- La coincidencia de los componentes raíz depende en gran manera de la implantación y no se especifica.

Cuando la sintaxis es "regex", el componente patrón es una expresión regular definida por la clase Pattern.

En ambas sintaxis, la glob y la regex, los detalles coincidentes, como si la coincidencia es por el uso de mayúsculas o minúsculas, dependen de la implantación y, por lo tanto, no se especifican.

# PathMatcher: ejemplo

```
1 public static void main(String [] args) {
2     // ... check for two arguments
3     Path root = Paths.get(args[0]);
4     // ... check that the first argument is a directory
5     PathMatcher matcher =
6         FileSystems.getDefault().getPathMatcher("glob:" + args[1]);
7     // Finder is class that implements FileVisitor
8     Finder finder = new Finder(root, matcher);
9     try {
10         Files.walkFileTree(root, finder);
11     } catch (IOException e) {
12         System.out.println("Exception: " + e);
13     }
14     finder.done();
15 }
```





En este fragmento de código en la diapositiva (el ejemplo completo está en el directorio de ejemplos), se transfieren dos argumentos al principal.

El primer argumento se prueba para ver si es un directorio. El segundo argumento se usa para crear una instancia de `PathMatcher` con una expresión regular mediante la fábrica `FileSystems`.

`Finder` es una clase que implanta la interfaz `FileVisitor`, de modo que se puede transferir a un método `walkFileTree`. Esta clase se usa para llamar al método de coincidencia en todos los archivos visitados en el árbol.

# Clase Finder

```
1 public class Finder extends SimpleFileVisitor<Path> {
2     private Path file;
3     private PathMatcher matcher;
4     private int numMatches;
5     // ... constructor stores Path and PathMatcher objects
6     private void find(Path file) {
7         Path name = file.getFileName();
8         if (name != null && matcher.matches(name)) {
9             numMatches++;
10             System.out.println(file);
11         }
12     }
13     @Override
14     public FileVisitResult visitFile(Path file,
15                                      BasicFileAttributes attrs) {
16         find(file);
17         return CONTINUE;
18     }
19     //...
20 }
```

ORACLE



La diapositiva muestra una porción de la clase `Finder`. Esta clase se utiliza para recorrer el árbol en busca de coincidencias entre el archivo y el archivo alcanzado por el método `visitFile`.

# Otras clases útiles de NIO.2

- La clase `FileStore` es útil para proporcionar información de uso sobre el sistema de archivos, como el total de espacio en disco utilizable y asignado.

```
Filesystem                  kbytes          used         avail
System (C:)              209748988      72247420     137501568
Data (D:)                 81847292        429488      81417804
```

- Se puede usar una instancia de la interfaz `WatchService` para informar sobre cambios en los objetos `Path` registrados. `WatchService` se puede usar para identificar el momento de adición de los archivos a un directorio, así como su supresión o modificación.

```
ENTRY_CREATE: D:\test\New Text Document.txt
ENTRY_CREATE: D:\test\Foo.txt
ENTRY_MODIFY: D:\test\Foo.txt
ENTRY_MODIFY: D:\test\Foo.txt
ENTRY_DELETE: D:\test\Foo.txt
```

ORACLE



La diapositiva muestra resultados de ejemplos del proyecto `DiskUsageExample` y el proyecto `WatchDirExample`.

# Cambio a NIO.2

Se ha agregado un método a la clase `java.io.File` para que JDK 7 pueda proporcionar compatibilidad con versiones posteriores de NIO.2.

```
Path path = file.toPath();
```

- Esto le permite aprovechar NIO.2 sin tener que reescribir una gran cantidad de código.
- Además, también puede sustituir el código existente para mejorar el mantenimiento futuro. Por ejemplo, puede sustituir `file.delete();` por:

```
Path path = file.toPath();
Files.delete(path);
```

- Por el contrario, la interfaz `Path` proporciona un método para construir un objeto `java.io.File`:

```
File file = path.toFile();
```

ORACLE







### Código java.io.File de legado

Una de las ventajas del paquete NIO.2 es que puede activar código de legado para aprovechar la nueva API.

# Resumen

En esta lección, debe haber aprendido a hacer lo siguiente:

- Utilizar la interfaz `Path` para realizar operaciones en archivos y en rutas de acceso a directorios
- Utilizar la clase `Files` para comprobar, suprimir, copiar o mover un archivo o un directorio
- Utilizar métodos de la clase `Files` para leer y escribir archivos mediante E/S de canales o E/S de flujos
- Leer y cambiar atributos de archivos y de directorios
- Acceder de forma recurrente al árbol de un directorio
- Localizar un archivo a través de la clase `PathMatcher`

ORACLE

# Prueba

Para copiar, desplazar o abrir un archivo o un directorio con NIO.2, primero debe crearse una instancia de:

a. `Path`
b. `Files`
c. `FileSystem`
d. `Channel`

# Prueba

Para una ruta de directorio de inicio dada, ¿qué métodos `FileVisitor` utilizaría para suprimir un árbol de archivos?

a. `preVisitDirectory()`
b. `postVisitDirectory()`
c. `visitFile()`
d. `visitDirectory()`

# Prueba

En el caso de una aplicación en la que se desea contar la profundidad de un árbol de archivo (el número de niveles de directorios), ¿qué método FileVisitor deberá usar?

a. `preVisitDirectory()`
b. `postVisitDirectory()`
c. `visitFile()`
d. `visitDirectory()`

# Visión general de la práctica 11-2: Copia recursiva

Esta práctica aborda la creación de una clase mediante la implantación de `FileVisitor` para copiar de forma recurrente un árbol de directorios a otra ubicación.

- Permita al usuario de su aplicación decidir si sustituir o no un directorio existente.

ORACLE

# (Opcional) Visión general de la práctica 11-3: Uso de `PathMatcher` para realizar una supresión recursiva

En esta práctica, se abordan los siguientes temas:

- Creación de una clase mediante la implantación de `FileVisitor` para suprimir un archivo mediante un comodín (es decir, suprimir todos los archivos de texto con `*.txt`).
- (Opcional) Ejecución de `WatchDirExample` en el directorio de ejemplos mientras se suprimen archivos de un directorio (o se usa la aplicación de copia recursiva) para supervisar posibles cambios.

ORACLE

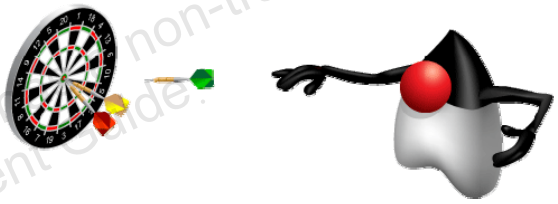
# 12

# Threads

# Objetivos

Al finalizar esta lección, debería estar capacitado para:

- Describir la programación de tareas del sistema operativo
- Definir un thread
- Crear threads
- Gestionar threads
- Sincronizar threads que acceden a datos compartidos
- Identificar posibles problemas de threads



ORACLE

# Programación de tareas

Los sistemas operativos modernos utilizan una multitarea preferente para asignar el tiempo de CPU a las aplicaciones. Existen dos tipos de tareas que se pueden programar para la ejecución:

- Procesos: un proceso es un área de la memoria que contiene tanto código como datos. Un proceso tiene un thread de ejecución que se programa para recibir porciones de tiempo de CPU.
- Thread: un thread es la ejecución programada de un proceso. Es posible que existan threads simultáneos. Todos los threads de un proceso comparten la misma memoria de datos, pero es posible que sigan diferentes rutas de acceso a través de una sección de código.

ORACLE







**Multitarea preferente**

Las computadoras modernas a menudo tienen más tareas que ejecutar que las CPU. A cada tarea se le proporciona una cantidad de tiempo (denominada porción de tiempo) en la que se puede ejecutar en una CPU. Una porción de tiempo normalmente se mide en milisegundos. Cuando transcurre la porción de tiempo, la tarea se elimina de forma forzosa de la CPU y se le da la oportunidad a otra tarea para que se ejecute.

# Importancia de los threads

Para ejecutar un programa tan rápido como sea posible, debe evitar cuellos de botella de rendimiento. Algunos de estos cuellos de botella son:

- Contención de recursos: dos o más tareas esperan uso exclusivo de un recurso
- Operaciones de E/S de bloqueo: no realizar ninguna acción a la espera de las transferencias de datos de disco o de red
- Infrautilización de CPU: una aplicación de thread único utiliza solo una CPU única

ORACLE







**Servidores multithread**

Aunque no escriba el código para crear nuevos threads de ejecución, el código se podría ejecutar en un entorno multithread. Debe tener en cuenta cómo funcionan los threads y cómo escribir el código con protección de thread. Si crea el código para su ejecución en otra parte del software (como un servidor de aplicaciones o Middleware), debe consultar la documentación de los productos para detectar si se creará los threads de forma automática. Por ejemplo, en un servidor de aplicaciones Java EE, existe un componente denominado servlet que se utiliza para manejar solicitudes HTTP. Los servlets siempre deben tener protección de thread porque el servidor inicia un nuevo thread para cada solicitud HTTP.

# Clase `Thread`

La clase `Thread` se utiliza para crear e iniciar threads. El código que va a ejecutar un thread se debe colocar en una clase, lo que:

- Amplía la clase `Thread`
  - Código más simple
- Implanta la interfaz `Runnable`
  - Más flexible
  - `extends` es aún libre

# Ampliación de `Thread`

Ampliar `java.lang.Thread` y sustituir el método `run`:

```
public class ExampleThread extends Thread {
    @Override
    public void run() {
        for(int i = 0; i < 100; i++) {
            System.out.println("i:" + i);
        }
    }
}
```

ORACLE







### Método `run`

El código que se va a ejecutar en un nuevo thread de ejecución se debe colocar en un método `run`. Debe evitar llamar al método `run` de forma directa. Al llamar al método `run` no se inicia un nuevo thread y el efecto no sería diferente de llamar a cualquier otro método.

# Inicio de `Thread`

Tras crear un nuevo `Thread`, se debe iniciar llamando al método `Thread` `start`:

```
public static void main(String[] args) {
    ExampleThread t1 = new ExampleThread();
    t1.start();
}
```

Programa el método run que se va a llamar.

ORACLE







### Método `start`

El método `start` se utiliza para comenzar a ejecuta un thread. Java Virtual Machine llamará al método `run` de `Thread`. Exactamente cuando el método `run` comienza la ejecución, la acción está fuera de su control. Un `Thread` se puede iniciar solo una vez.

# Implantación de `Runnable`

Implantar `java.lang.Runnable` y el método `run`:

```
public class ExampleRunnable implements Runnable {
    @Override
    public void run() {
        for(int i = 0; i < 100; i++) {
            System.out.println("i:" + i);
        }
    }
}
```

ORACLE







### Método `run`

Al igual que al ampliar `Thread`, al llamar al método `run` no se inicia un nuevo thread. La ventaja de implantar `Runnable` es que puede seguir ampliando una clase de su elección.

# Ejecución de instancias `Runnable`

Tras crear un nuevo `Runnable`, se debe transferir a un constructor de `Thread`. El método `Thread` `start` comienza la ejecución:

```
public static void main(String[] args) {
    ExampleRunnable r1 = new ExampleRunnable();
    Thread t1 = new Thread(r1);
    t1.start();
}
```



**Método `start`**

El método `start` de `Thread` se utiliza para comenzar a ejecutar un thread. Una vez iniciado el thread, Java Virtual Machine llamará al método `run` en el `Runnable` asociado al `Thread`.

# `Runnable` con datos compartidos

Los threads comparten potencialmente campos estáticos y de instancia.

```
public class ExampleRunnable implements Runnable {
    private int i;

    @Override
    public void run() {
        for(i = 0; i < 100; i++) {
            System.out.println("i:" + i);
        }
    }
}
```

Variable compartida potencialmente

ORACLE

# Un ejecutable: varios threads

Varios threads que hacen referencia a un objeto pueden producir que se acceda de forma simultánea a los campos de instancia.

```
public static void main(String[] args) {
    ExampleRunnable r1 = new ExampleRunnable();
    Thread t1 = new Thread(r1);
    t1.start();
    Thread t2 = new Thread(r1);
    t2.start();
}
```

Una única instancia `Runnable`

ORACLE







### Varios threads con un `Runnable`

Es posible transferir una única instancia `Runnable` a varias instancias `Thread`. Solo hay tantas instancias `Runnable` como haya creado. Varias instancias de `Thread` comparten los campos de la instancia `Runnable`.

Varios threads también puede acceder de forma simultánea a campos estáticos.

# Prueba

La creación de un nuevo thread requiere el uso de:

a. `java.lang.Runnable`
b. `java.lang.Thread`
c. `java.util.concurrent.Callable`

# Problemas con datos compartidos

A los datos compartidos se debe acceder con cuidado.
Campos de instancia y estáticos:

- Se crean en un área de memoria conocida como espacio de pila.
- Cualquier thread los puede compartir de forma potencial.
- Varios threads los pueden cambiar de forma simultanea.
  - No hay ningún compilador o advertencias de IDE.
  - El acceso “de forma segura” a campos compartidos es su responsabilidad.

Las diapositivas anteriores pueden producir lo siguiente:

```
i:0,i:0,i:1,i:2,i:3,i:4,i:5,i:6,i:7,i:8,i:9,i:10,i:12,i:11 ...
```

El cero se ha producido dos veces

Fuera de la secuencia

ORACLE



### Depuración de threads

La depuración de threads puede ser difícil porque la frecuencia y la duración de tiempo de cada thread asignada puede variar por diferente motivos, entre los que se incluyen:

- Un sistema operativo maneja la programación de threads y los sistemas operativos pueden usar diferentes algoritmos de programación.
- Las máquinas tienen diferentes recuentos y velocidades de CPU.
- Otras aplicaciones pueden estar poniendo carga en el sistema.

Este es uno de esos casos en los que una aplicación puede parecer que funciona a la perfección mientras se está desarrollando, pero los problemas extraños se pueden manifestar después de su producción debido a las variaciones de programación. Es su responsabilidad proteger el acceso a las variables compartidas.

# Datos no compartidos

Algunos tipos de variables nunca se comparten. Los siguientes tipos siempre tienen protección de thread:

- Variables locales
- Parámetros del método
- Parámetros de manejador de excepciones





**Datos con protección de thread compartidos**

Cualquier dato compartido que sea inmutable, como los objetos `String` o campos finales, tiene protección de thread porque solo se pueden leer y no escribir.

# Prueba

Las variables tiene protección de thread si son:

a. `local`
b. `static`
c. `final`
d. `private`

# Operaciones atómicas

Las operaciones atómicas funcionan como una operación única. Una única sentencia en el lenguaje Java no siempre es atómica.

- `i++;`
  - Crea una copia temporal del valor en `i`.
  - Incrementa la copia temporal.
  - Anota el nuevo valor en `i`.
- `l = 0xffff_ffff_ffff_ffff;`
  - Es posible acceder a variables de 64 bits mediante dos operaciones de 32 bits independientes.

¿Qué inconsistencias podrían encontrar dos threads al incrementar el mismo campo?

¿Qué ocurre si dicho campo es largo?

ORACLE







### Comportamiento inconsistente

Un posible problema con dos threads que incrementan el mismo campo es que se puede producir una actualización perdida. Imagine si ambos threads leen un valor de 41 de un campo, incrementan el valor en uno y, a continuación, anotan los resultados en el campo. Ambos threads realizarán un incremento pero el valor resultante es solo 42. Según cómo Java Virtual Machine se implante y el tipo de CPU física que se esté usando, es posible que nunca o en raras ocasiones vea este comportamiento. Sin embargo, siempre debe asumir que esto puede ocurrir.

Si tiene un valor largo de `0x0000_0000_ffff_ffff` y lo incrementa en `1`, el resultado debe ser `0x0000_0001_0000_0000`. Sin embargo, debido a que es válido acceder a un campo de 64 bits mediante dos escrituras de 32 bits independientes, podría haber temporalmente un valor de `0x0000_0001_ffff_ffff` o incluso `0x0000_0000_0000_0000` según los bits que se modifiquen en primer lugar. Si se permite que un segundo thread lea un campo de 64 bits mientras otro thread lo está modificando, se podría recuperar un valor incorrecto.

# Ejecución desordenada

- Las operaciones realizadas en un thread puede parecer que no se ejecutan en orden al observar los resultados desde otro thread.
  - La optimización de código puede dar como resultado una operación desordenada.
  - Los threads funcionan en copias almacenadas en caché de variables compartidas.
- Para garantizar el comportamiento consistente en los threads, debe sincronizar sus acciones.
  - Necesita una forma de establecer que una acción ocurra antes que otra.
  - Necesita una forma de vaciar de nuevo los cambios en variables compartidas en la memoria principal.

ORACLE







### Sincronización de acciones

Cada thread tiene una *memoria de trabajo* en la que mantiene su propia *copia de trabajo* de las variables que se deben usar o asignar. A medida que el thread ejecuta un programa, funciona en estas copias de trabajo. Existen varias acciones que sincronizarán una *memoria de trabajo* del thread con la memoria principal:

- Una lectura o escritura volátil de una variable (palabra clave `volatile`)
- Bloqueo o desbloqueo de una supervisión (palabra clave `synchronized`)
- La primera y última acción de un thread
- Acciones que inician un thread o detectan que un thread ha terminado

# Prueba

¿Cuál de las siguientes opciones hace que un thread sincronice variables?

a. Lectura de un campo volátil
b. Llamada a `isAlive()` en un thread
c. Inicio de un nuevo thread
d. Terminación de un bloque de código sincronizado

# Palabra clave `volatile`

Un campo puede tener el modificador `volatile` aplicado:

```
public volatile int i;
```

- La lectura o escritura de un campo `volatile` hará que un thread sincronice su memoria de trabajo con la memoria principal.
- `volatile` no significa atómico.
  - Si `i` es `volatile`, `i++` todavía no es una operación con protección de thread.

ORACLE



Debido a que la manipulación de los campos `volatile` puede no ser atómica, no es suficiente con realizar lecturas y escrituras de variables únicas con protección de thread. Un buen ejemplo del uso de `volatile` se muestra en los ejemplos de las siguientes diapositivas sobre la parada de un thread.

# Parada de un thread

Un thread se para al terminar su método `run`.

```
public class ExampleRunnable implements Runnable {
    public volatile boolean timeToQuit = false;

    @Override
    public void run() {
        System.out.println("Thread started");
        while(!timeToQuit) {
            // ...
        }
        System.out.println("Thread finishing");
    }
}
```

Variable volátil compartida

# Parada de un thread

```
public static void main(String[] args) {
    ExampleRunnable r1 = new ExampleRunnable();
    Thread t1 = new Thread(r1);
    t1.start();
    // ...
    r1.timeToQuit = true;
}
```







### Threadprincipal

El método `main` en una aplicación Java SE que se ejecuta en un thread, a veces denominado thread principal, que JVM crea automáticamente. Solo con cualquier thread, cuando el thread principal escribe en el campo `timeToQuit`, es importante que el thread t1 vea la escritura. Si el campo `timeToQuit` no es `volatile`, no existe ninguna garantía de que la escritura se vea inmediatamente. Tenga en cuenta que si olvida declarar un campo similar como `volatile`, una aplicación puede funcionar perfectamente para usted pero, de forma ocasional, puede fallar para otros usuarios.

# Palabra clave `volatile`

La palabra clave `synchronized` se utiliza para crear bloques de código con protección de thread. Un bloque de código `synchronized`:

- Hace que un thread escriba todos sus cambios en la memoria principal cuando se alcanza el final del bloque.
  - Similar a `volatile`
- Se usa para agrupar bloques de código para una ejecución exclusiva.
  - Bloque de threads hasta que obtienen acceso exclusivo
  - Resuelve el problema atómico

# Métodos synchronized

```
public class ShoppingCart {
    private List<Item> cart = new ArrayList<>();
    public synchronized void addItem(Item item) {
        cart.add(item);
    }
    public synchronized void removeItem(int index) {
        cart.remove(index);
    }
    public synchronized void printCart() {
        Iterator<Item> ii = cart.iterator();
        while(ii.hasNext()) {
            Item i = ii.next();
            System.out.println("Item:" + i.getDescription());
        }
    }
}
```

ORACLE







### Comportamiento de método sincronizado

En el ejemplo de la diapositiva, puede llamar solo a un método al mismo tiempo en un objeto `ShoppingCart` porque todos sus métodos están `synchronized`. En este ejemplo, la sincronización es por `ShoppingCart`. Dos instancias `ShoppingCart` se podrían usar de forma simultánea.

Si los métodos no estuvieran `synchronized`, al llamar a `removeItem` mientras `printCart` itera con la recopilación de `Item` podría darse un comportamiento inesperado. Un iterador puede soportar un comportamiento de fallo rápido. Un iterador de fallo rápido devolverá `java.util.ConcurrentModificationException`, una subclase de `RuntimeException`, si la recopilación del iterador se modifica mientras se está utilizando.

# Bloques `synchronized`

```
public void printCart() {
    StringBuilder sb = new StringBuilder();
    synchronized (this) {
        Iterator<Item> ii = cart.iterator();
        while (ii.hasNext()) {
            Item i = ii.next();
            sb.append("Item:");
            sb.append(i.getDescription());
            sb.append("\n");
        }
    }
    System.out.println(sb.toString());
}
```

ORACLE







### Cuellos de botella de sincronización

La sincronización en aplicaciones multithread garantiza un comportamiento fiable. Debido a que los bloques y métodos `synchronized` se utilizan para restringir una sección de código a un único thread, está creando de forma potencial cuellos de botella de rendimiento. Se pueden usar los bloques `synchronized` en lugar de métodos `synchronized` para reducir el número de líneas que son exclusivas de un único thread.

Utilice la sincronización lo menos posible para el rendimiento, pero tanto como sea necesario para garantizar la fiabilidad.

# Bloqueo de supervisión de objeto

Cada objeto en Java se asocia a una supervisión, que un thread puede bloquear o desbloquear.

- Los métodos `synchronized` utilizan la supervisión para el objeto `this`.
- Los métodos `static synchronized` utilizan la supervisión de clases.
- Los bloques `synchronized` deben especificar qué supervisión del objeto bloquear o desbloquear.

```
synchronized ( this ) { }
```

- Los bloques `synchronized` pueden ser anidados.

ORACLE



### Bloques `synchronized` anidados

Un thread puede bloquear varias supervisiones de forma simultanea mediante bloques `synchronized` anidados.

# Detección de interrupción

La interrupción de un thread es otra posible forma de solicitar que un thread pare la ejecución.

```
public class ExampleRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Thread started");
        while(!Thread.interrupted()) {
            // ...
        }
        System.out.println("Thread finishing");
    }
}
```

Método `Thread` estático



### Interrupción no significa parada

Cuando se interrumpe un thread, depende del usuario decidir qué acción realizar. Dicha acción podría ser volver del método `run` o continuar con la ejecución del código.

# Interrupción de un thread

Cada thread tiene un método `interrupt()` e `isInterrupted()`.

```
public static void main(String[] args) {
    ExampleRunnable r1 = new ExampleRunnable();
    Thread t1 = new Thread(r1);
    t1.start();
    // ...
    t1.interrupt();
}
```

Interrupción de un thread

ORACLE







### Ventajas de la interrupción

El uso de funciones de interrupción de `Thread` es un modo práctico de parar un thread. Además de eliminar la necesidad de escribir su propia lógica de parada de thread, también puede interrumpir un thread bloqueado. Para obtener más información, consulte http://download.java.net/jdk7/docs/api/java/lang/Thread.html#interrupt().

# Thread.sleep()

Un `Thread` puede realizar una ejecución durante un tiempo.

```
long start = System.currentTimeMillis();
try {
    Thread.sleep(4000);
} catch (InterruptedException ex) {
    // What to do?
}
long time = System.currentTimeMillis() - start;
System.out.println("Slept for " + time + " ms");
```

`interrupt()` se llama durante la suspensión

ORACLE







**¿Cuánto tiempo se suspenderá un thread?**

Una solicitud de `Thread.sleep(4000)` significa que un thread desea parar la ejecución durante 4 segundos. Tras 4 segundos transcurridos, el thread se vuelve a programar para la ejecución. Esto no significa que el thread se inicie exactamente 4 segundos después de la llamada a `sleep()`, si no que comenzará la ejecución 4 segundos o más después de iniciar la suspensión. La duración exacta de la suspensión se ve afectada por el hardware de la máquina, el sistema operativo y la carga del sistema.

**Suspensión interrumpida**

Si llama a `interrupt()` en un thread que está suspendido, la llamada a `sleep()` devolverá una excepción `InterruptedException` que se debe manejar. La forma de manejar la excepción depende de cómo esté diseñada la aplicación. Si la llamada a `interrupt()` simplemente intenta interrumpir la llamada de `sleep()` y no la ejecución de un thread, debe aceptar la excepción. Otros casos pueden requerir que devuelva la excepción o regrese desde un método `run()`.

# Prueba

Una llamada a `Thread.sleep(4000)` causará la ejecución del thread para que siempre se suspenda durante 4 segundos

a. Verdadero
b. Falso

# Métodos `Thread` adicionales

- Existen muchos más `Thread` y métodos relacionados con thread:
  - `setName(String)`, `getName()` y `getId()`
  - `isAlive()`: ¿Ha finalizado un thread?
  - `isDaemon()` y `setDaemon(boolean)`: JVM puede cerrarse mientras los threads del daemon están en ejecución.
  - `join()`: un thread actual espera que otro thread finalice.
  - `Thread.currentThread()`: las instancias `Runnable` pueden recuperar la instancia `Thread` que actualmente esté en ejecución.
- La clase `Object` también tiene métodos relacionados con thread:
  - `wait()`, `notify()` y `notifyAll()`: los threads se pueden suspender durante un tiempo indeterminado, reactivándose solo cuando el `Object` esperado recibe una notificación de reactivación.

ORACLE







**Más información**

Los threads de daemon son threads en segundo plano que son menos importantes que los threads normales. Debido a que el thread principal no es un thread de daemon, todos los threads creados también serán threads no de daemon. Cualquier thread no de daemon que esté en ejecución (activo) evitará que se cierre JVM incluso si se ha devuelto el método main. Si un thread no debe evitar que JVM se cierre, se debe establecer como un thread de daemon. Hay más conceptos y métodos de multithread sobre los que obtener información. Para obtener material de lectura adicional, consulte http://download.oracle.com/javase/tutorial/essential/concurrency/further.html.

# Métodos a evitar

Se deben evitar algunos métodos `Thread`:

- `setPriority(int)` y `getPriority()`
  - Es posible que no causen ningún impacto o que provoquen problemas
- Los siguientes métodos están anticuados y nunca se deben usar:
  - `destroy()`
  - `resume()`
  - `suspend()`
  - `stop()`

ORACLE







**Métodos anticuados**

Clases, interfaces, métodos, variables y otros componentes de cualquier biblioteca de Java puede aparecer marcados como anticuados. Los componentes anticuados puede causar un comportamiento no predecible o simplemente puede que no hayan seguido las convenciones de nomenclatura adecuadas. Debe evitar usar cualquier API anticuada en las aplicaciones. Las API anticuadas aún se incluyen en las bibliotecas para asegurar la compatibilidad con versiones anteriores, pero se podrían eliminar de forma potencial en futuras versiones de Java.

Para obtener más información sobre los métodos anticuados mencionados anteriormente, consulte docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html, que está disponible en línea en http://download.oracle.com/javase/7/ o como parte de la documentación JDK descargable.

# Interbloqueo

El interbloqueo se produce cuando dos o más threads se bloquean para siempre, en espera el uno del otro.

```
synchronized(obj1) {
  synchronized(obj2) {
  }
}
```

El thread 1 realiza una pausa después de bloquear la supervisión de obj1.

```
synchronized(obj2) {
  synchronized(obj1) {
  }
}
```

El thread 2 realiza una pausa después de bloquear la supervisión de obj2.

ORACLE







La inanición e interbloqueo activo son problemas mucho menos comunes que el interbloqueo, pero aún son problemas que es probable que los diseñadores de un software actual encuentren.

**Inanición**

La inanición describe una situación en la que un thread no ha podido obtener acceso normal a recursos compartidos y no podrá realizar progresos. Esto ocurre cuando los recursos compartidos dejan de estar disponibles durante largos períodos por threads "abusivos". Por ejemplo, suponga que un objeto proporciona un método sincronizado que a menudo tarda mucho tiempo en volver. Si un thread llama a este método de forma frecuente, otros threads que también necesiten acceso sincronizado de forma frecuente al mismo objeto se bloquearán con frecuencia.

**Interbloqueo activo**

Un thread a menudo actúa en respuesta a la acción de otro thread. Si la acción del otro thread es también respuesta a la acción de otro thread, puede da lugar a un *interbloqueo activo*. Al igual que ocurre con un interbloqueo, los threads con un interbloqueo activo no pueden avanzar. Sin embargo, los threads que no están bloqueados, simplemente están demasiado ocupados respondiéndose como para reanudar el trabajo.

# Resumen

En esta lección, debe haber aprendido a hacer lo siguiente:

- Describir la programación de tareas del sistema operativo
- Definir un thread
- Crear threads
- Gestionar threads
- Sincronizar threads que acceden a datos compartidos
- Identificar posibles problemas de threads

ORACLE

# Visión general de la práctica 12-1: Sincronización de acceso a datos compartidos

En esta práctica, se abordan los siguientes temas:

- Impresión de ID de thread
- Uso de `Thread.sleep()`
- Sincronización de un bloque de código

ORACLE



En esta práctica, escribirá una clase agregada a una aplicación multithread existente. Debe crear una clase con protección de thread.

# Visión general de la práctica 12-2: Implantación de un programa multithread

En esta práctica, se abordan los siguientes temas:

- Implantación de `Runnable`
- Inicio de `Thread`
- Comprobación del estado de un `Thread`
- Interrupción de un `Thread`



En esta práctica, creará, iniciará e interrumpirá threads básicos mediante la interfaz `Runnable`.

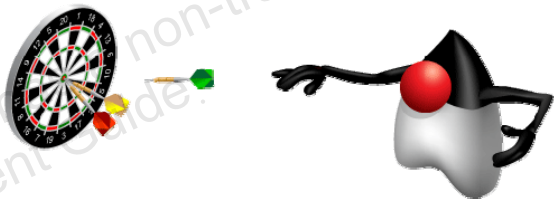
# 13

# Simultaneidad

# Objetivos

Al finalizar esta lección, debería estar capacitado para:

- Usar variables atómicas
- Usar `ReentrantReadWriteLock`
- Usar recopilaciones `java.util.concurrent`
- Describir las clases de sincronizador
- Usar `ExecutorService` para ejecutar tareas de forma simultánea
- Aplicar el marco Fork-Join



ORACLE

# Paquete `java.util.concurrent`

Java 5 introdujo el paquete `java.util.concurrent`, que contiene clases que son útiles en la programación simultánea. Sus funciones incluyen:

- Recopilaciones simultáneas
- Alternativas de sincronización y bloqueo
- Pools de threads
  - Pools de recuento de threads dinámicos y fijos disponibles
  - Metodología "divide y vencerás" paralela (Fork-Join), nueva en Java 7

## Paquete `java.util.concurrent.atomic`

El paquete `java.util.concurrent.atomic` contiene clases que soportan la programación con protección de thread y bloqueo libre en variables únicas.

```
AtomicInteger ai = new AtomicInteger(5);
if(ai.compareAndSet(5, 42)) {
    System.out.println("Replaced 5 with 42");
}
```

Una operación atómica garantiza que el valor actual sea 5 y, a continuación, se defina en 42.



**Operación no bloqueante**

En arquitecturas de CPU que soportan una operación de definición y comparación nativa, no será necesario bloquear al ejecutar el ejemplo que se muestra. Otras arquitecturas pueden requerir alguna forma de bloqueo interno.

## Paquete `java.util.concurrent.locks`

El paquete `java.util.concurrent.locks` es un marco para bloquear y esperar condiciones que es distinto de las supervisiones y sincronización incorporadas.

```
public class ShoppingCart {
    private final ReentrantReadWriteLock rwl =
        new ReentrantReadWriteLock();

    public void addItem(Object o)
        rwl.writeLock().lock();
        // modify shopping cart
        rwl.writeLock().unlock();
    }
```

Bloqueo de único escritor, varios lectores

Bloqueo de escritura

ORACLE



### Bloqueo de varios lectores, único escritor

Una de las funciones del paquete `java.util.concurrent.locks` es la implantación de un bloqueo de varios lectores, único escritor. Es posible que un thread no tenga ni obtenga un bloqueo de lectura mientras está en uso un bloqueo de escritura. Varios threads pueden adquirir simultáneamente el bloqueo de lectura pero solo uno puede adquirir el bloqueo de escritura. El bloqueo es reentrante; un thread que ya ha adquirido el bloqueo de escritura puede llamar a métodos adicionales que también obtengan el bloqueo de escritura sin miedo al bloqueo.

## java.util.concurrent.locks

```
public String getSummary() {
    String s = "";
    rwl.readLock().lock();
    // read cart, modify s
    rwl.readLock().unlock();
    return s;
}
public double getTotal() {
    // another read-only method
}
}
```

**Bloqueo de lectura**

Todos los métodos de solo lectura se pueden ejecutar de forma simultánea.





### Varias lecturas simultáneas

En el ejemplo, todos los métodos determinados como de solo lectura pueden agregar el código necesario para bloquear y desbloquear un bloqueo de lectura. `ReentrantReadWriteLock` permite la ejecución simultánea de ambos, un único método de solo lectura y varios métodos de solo lectura.

## Recopilaciones con protección de thread

Las recopilaciones de `java.util` no tienen protección de thread. Para usar recopilaciones en modo de protección de thread:

- Usar bloques de código sincronizados para todos los accesos a una recopilación si se realizan escrituras
- Crear un envoltorio sincronizado mediante métodos de biblioteca, como `java.util.Collections.synchronizedList(List<T>)`
- Usar recopilaciones `java.util.concurrent`

**Nota:** el hecho de que una `Collection` se cree con protección de thread, no hace que sus elementos tengan protección de thread.

ORACLE







### Recopilaciones simultáneas

La clase `ConcurrentLinkedQueue` proporciona una cola FIFO no bloqueante con protección de thread escalable eficaz. Cinco implantaciones en `java.util.concurrent` soportan la interfaz ampliada `BlockingQueue`, que define las versiones de bloqueo de colocación y captura: `LinkedBlockingQueue`, `ArrayBlockingQueue`, `SynchronousQueue`, `PriorityBlockingQueue` y `DelayQueue`.

Además de las colas, este paquete proporciona implantaciones de `Collection` diseñadas para su uso en contextos multithread: `ConcurrentHashMap`, `ConcurrentSkipListMap`, `ConcurrentSkipListSet`, `CopyOnWriteArrayList` y `CopyOnWriteArraySet`. Cuando se espera que muchos threads accedan a la recopilación proporcionada, normalmente se prefiere `ConcurrentHashMap` a `HashMap` sincronizado, y normalmente se prefiere `ConcurrentSkipListMap` a `TreeMap` sincronizado. Se prefiere `CopyOnWriteArrayList` a `ArrayList` sincronizado cuando el número esperado de lecturas y transversales supera en gran medida el número de actualizaciones en una lista.

# Prueba

`CopyOnWriteArrayList` garantiza la protección de thread de los objetos agregados a `List`.

a. Verdadero
b. Falso

# Sincronizadores

El paquete `java.util.concurrent` proporciona cinco clases que ayudan a las expresiones de sincronización con un objetivo especial común.

| Clase | Descripción |
|---|---|
| Semaphore | Semaphore es una herramienta de simultaneidad clásica. |
| CountDownLatch | Utilidad todavía muy simple y muy común para bloquear hasta que se contenga un número determinado de señales, eventos o condiciones. |
| CyclicBarrier | Punto de sincronización multidireccional reajustable útil en algunos estilos de programación paralela. |
| Phaser | Proporciona una forma más flexible de barrera que se puede usar para controlar el cálculo en fases entre varios threads. |
| Exchanger | Permite a dos threads intercambiar objetos en un punto de encuentro y es útil en distintos diseños de pipeline. |

ORACLE







Las clases de sincronizador permiten a los threads bloquear hasta que se alcanza determinado estado o acción.

`Semaphore`: `Semaphore` mantiene un juego de permisos. Los threads tratan de adquirir permisos y se pueden bloquear hasta que otros threads liberen permisos.

`CountDownLatch`: `CountDownLatch` permite a uno o más threads esperar (bloquear) hasta la finalización de una cuenta atrás. Una vez finalizada la cuenta atrás, todos los threads en espera continúan. `CountDownLatch` no se puede volver a usar.

`CyclicBarrier`: se crea con un recuento de terceros. Después de llamar a un número de partes (threads) en espera de `CyclicBarrier`, se liberarán (desbloquearán). `CyclicBarrier` se puede volver a usar.

`Phaser`: una versión más versátil de `CyclicBarrier`, nuevo en Java 7. Las partes pueden registrar y anular el registro durante un período de tiempo, lo que provoca que el número de threads necesarios antes del avance cambie.

`Exchanger`: permite a dos threads intercambiar un par de objetos, bloqueando hasta que se realice el intercambio. Es una alternativa bidireccional de memoria eficaz a `SynchronousQueue`.

## java.util.concurrent.CyclicBarrier

`CyclicBarrier` es un ejemplo de categoría de sincronizador de clases proporcionada por `java.util.concurrent`.

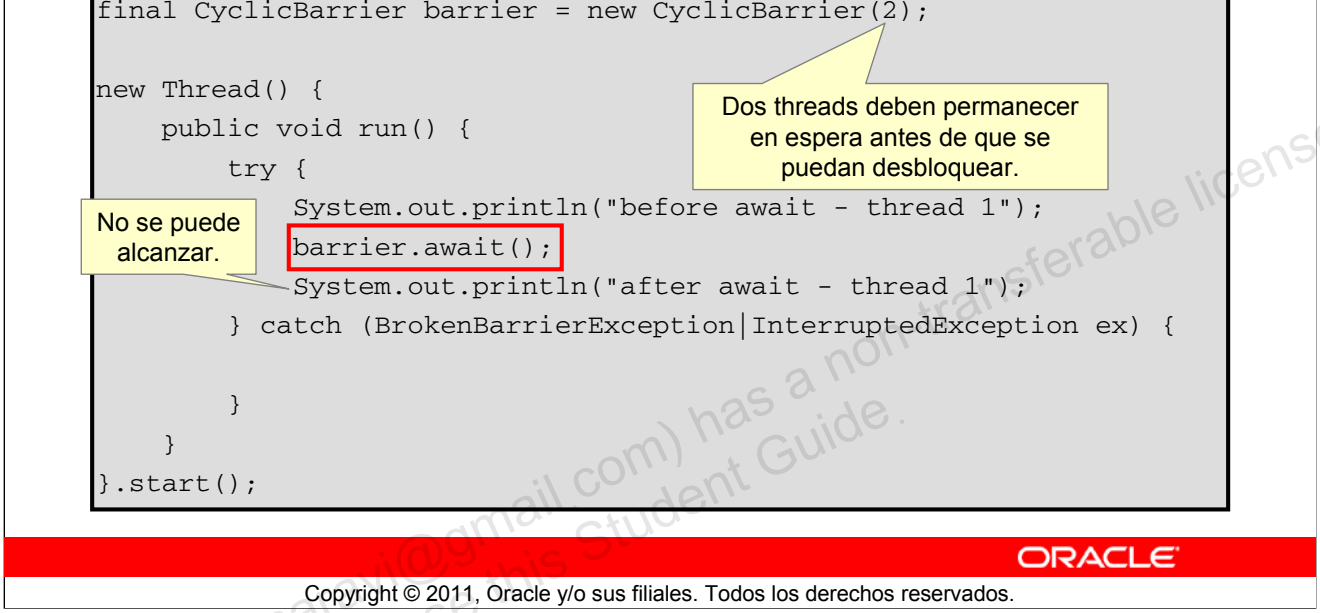
```
final CyclicBarrier barrier = new CyclicBarrier(2);

new Thread() {
    public void run() {
        try {
            System.out.println("before await - thread 1");
            barrier.await();
            System.out.println("after await - thread 1");
        } catch (BrokenBarrierException|InterruptedException ex) {

        }
    }
}.start();
```

Dos threads deben permanecer en espera antes de que se puedan desbloquear.

No se puede alcanzar.





### `CyclicBarrier`: comportamiento

En este ejemplo, si solo un thread llama a `await()` en la barrera, dicho thread se puede bloquear para siempre. Después de que un segundo thread llame a `await()`, cualquier llamada adicional a `await()` se volverá a bloquear hasta que se alcance el número de threads necesario. `CyclicBarrier` contiene un método `await(long timeout, TimeUnit unit)`, que se bloqueará durante una duración especificada y devolverá una excepción `TimeoutException` si se alcanza dicha duración.

## Alternativas de threads de alto nivel

Puede resultar difícil usar las API relacionadas con el `Thread` tradicional de forma correcta. Las alternativas incluyen:

- `java.util.concurrent.ExecutorService`, mecanismo de mayor nivel usado para ejecutar tareas
  - Puede crear y volver a usar objetos de `Thread` para el usuario.
  - Permite ejecutar el trabajo y comprobar los resultados en el futuro.
- Marco Fork-Join, servicio `ExecutorService` de extracción de trabajo especializado, nuevo en Java 7

ORACLE







### Alternativas de sincronización

Los bloques de código sincronizados se utilizan para garantizar que a los datos que no tienen protección de thread no podrán acceder de forma simultánea varios threads. Sin embargo, el uso de bloques de código sincronizados puede generar cuellos de botella de rendimiento. Varios componentes del paquete `java.util.concurrent` proporcionan alternativas para usar bloques de código sincronizados. Además de aprovechar recopilaciones simultáneas, colas y sincronizadores, existe otra forma de garantizar que a los datos no accederán de manera incorrecta varios threads: simplemente no permitir que varios threads procesen los mismos datos. En algunos casos, puede ser posible crear varias copias de los datos en RAM y permitir que cualquier thread procese una única copia.

## java.util.concurrent.ExecutorService

`ExecutorService` se utiliza para ejecutar tareas.

- Elimina la necesidad de crear y gestionar threads de forma manual.
- Las tareas **se pueden** ejecutar en paralelo según la implantación de `ExecutorService`.
- Las tareas pueden ser:
  - `java.lang.Runnable`
  - `java.util.concurrent.Callable`
- La implantación de instancias se puede obtener con `Executors`.

```
ExecutorService es = Executors.newCachedThreadPool();
```



### Comportamiento de `ExecutorService`

Un pool de threads almacenado en caché `ExecutorService`:

- Crea nuevos threads según sea necesario.
- Vuelve a usar sus threads (dichos threads no mueren tras la finalización de la tarea).
- Termina los threads que han estado inactivos durante 60 segundos.

Otros tipos de implantaciones de `ExecutorService` disponibles:

```
int cpuCount = Runtime.getRuntime().availableProcessors();
ExecutorService es = Executors.newFixedThreadPool(cpuCount);
```

Un pool de threads fijo `ExecutorService`:

- Contiene un número fijo de threads.
- Vuelve a usar sus threads (dichos threads no mueren tras la finalización de la tarea).
- Se pone en cola hasta que un thread está disponible.
- Se podría usar para evitar el exceso de trabajo en un sistema con tareas con más uso de CPU.

# java.util.concurrent.Callable

La interfaz `Callable`:

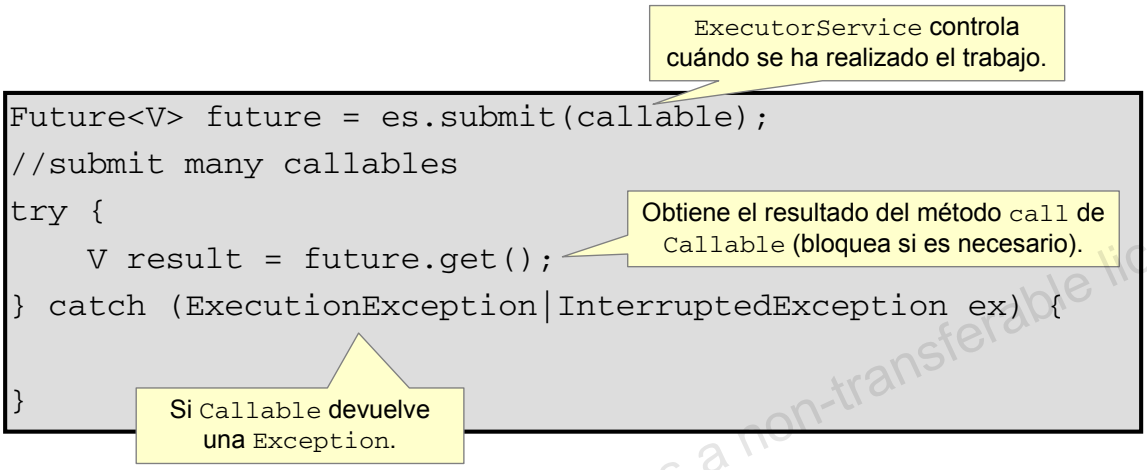
- Define una tarea ejecutada en `ExecutorService`.
- Es similar en naturaleza a `Runnable`, pero puede:
  - Devolver un resultado mediante genéricos.
  - Devolver una excepción comprobada.

```
package java.util.concurrent;
public interface Callable<V> {
    V call() throws Exception;
}
```

ORACLE

## java.util.concurrent.Future

La interfaz `Future` se utiliza para obtener resultados de un método `V call()` de `Callable`.






### Espera de un futuro

Debido a que la llamada a `Future.get()` se bloqueará, debe realizar una de las siguientes acciones:

- Envíe todo el trabajo a `ExecutorService` antes de llamar a ningún método `Future.get()`.
- Esté preparado para esperar que `Future` obtenga el resultado.
- Utilice un método no bloqueante como `Future.isDone()` antes de llamar a `Future.get()` o utilice `Future.get(long timeout, TimeUnit unit)`, que devolverá una excepción `TimeoutException` si el resultado no está disponible en una duración determinada.

# Cierre de `ExecutorService`

El cierre de `ExecutorService` es importante porque sus threads son threads de no daemons y evitarán que JVM se cierre.

```
es.shutdown();

try {
    es.awaitTermination(5, TimeUnit.SECONDS);
} catch (InterruptedException ex) {
    System.out.println("Stopped waiting early");
}
```

Pare la aceptación de nuevos `Callable`.

Si desea esperar que las acciones `Callable` finalicen.

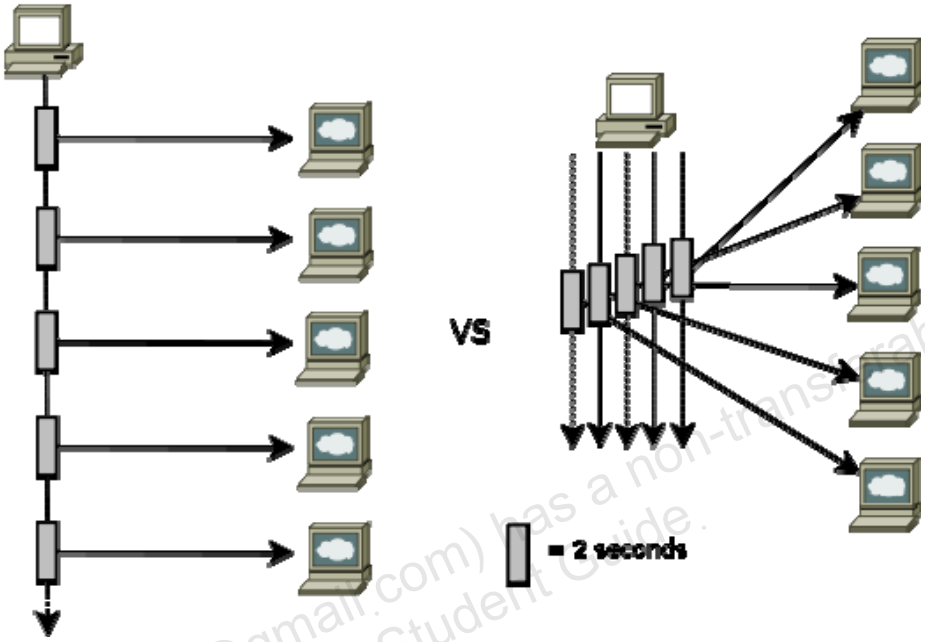
# Prueba

`ExecutorService` siempre intentará usar todas las CPU disponibles en un sistema.

a. Verdadero
b. Falso

# E/S simultánea

Las llamadas de bloqueo secuencial se ejecutan en una duración de tiempo más larga que las llamadas de bloqueo simultáneo.




ORACLE



**Reloj**

Existen diferentes formas de medir el tiempo. En el gráfico se muestran cinco llamadas secuenciales a servidores de red que tardarán aproximadamente 10 segundos si cada llamada dura 2 segundos. En la parte derecha del gráfico, cinco llamadas simultáneas a los servidores de red solo tardan un poco más de 2 segundos si cada llamada dura 2 segundos. Ambos ejemplos usan aproximadamente la misma cantidad de tiempo de CPU, la cantidad de ciclos de CPU consumidos, pero tienen diferentes duraciones en conjunto o tiempo real.

# Cliente de red de thread único

```
public class SingleThreadClientMain {
    public static void main(String[] args) {
        String host = "localhost";
        for (int port = 10000; port < 10010; port++) {
            RequestResponse lookup =
             new RequestResponse(host, port);
            try (Socket sock = new Socket(lookup.host, lookup.port);
                 Scanner scanner = new Scanner(sock.getInputStream());){
                lookup.response = scanner.next();
                System.out.println(lookup.host + ":" + lookup.port + " " +
                    lookup.response);
            } catch (NoSuchElementException|IOException ex) {
                System.out.println("Error talking to " + host + ":" +
                    port);
            }
        }
    }
}
```

ORACLE



### Llamada síncrona

En el ejemplo de esta diapositiva, estamos intentado detectar el proveedor que ofrece el precio más bajo para un artículo. El cliente comunicará con los 10 servidores de red distintos, cada servidor tardará aproximadamente dos segundos en buscar los datos solicitados y devolverlos. Es posible que haya retrasos adicionales introducidos por la latencia de red.

Este cliente de thread único debe esperar que cada servidor responda antes de moverse a otro servidor. Son necesarios cerca de 20 segundos para recuperar todos los datos.

# Cliente de red multithread (parte 1)

```
public class MultiThreadedClientMain {
    public static void main(String[] args) {
        //ThreadPool used to execute Callables
        ExecutorService es = Executors.newCachedThreadPool();
        //A Map used to connect the request data with the result
        Map<RequestResponse,Future<RequestResponse>> callables =
            new HashMap<>();

        String host = "localhost";
        //loop to create and submit a bunch of Callable instances
        for (int port = 10000; port < 10010; port++) {
            RequestResponse lookup = new RequestResponse(host, port);
            NetworkClientCallable callable =
                new NetworkClientCallable(lookup);
            Future<RequestResponse> future = es.submit(callable);
            callables.put(lookup, future);
        }
```

ORACLE







### Llamada asíncrona

En el ejemplo de esta diapositiva, estamos intentado detectar el proveedor que ofrece el precio más bajo para un artículo. El cliente comunicará con los 10 servidores de red distintos, cada servidor tardará aproximadamente dos segundos en buscar los datos solicitados y devolverlos. Es posible que haya retrasos adicionales introducidos por la latencia de red.

Este cliente multithread *no* espera que cada servidor responda antes de intentar comunicarse con otro servidor. Son necesarios cerca de 2 segundos en lugar de 20 para recuperar todos los datos.

# Cliente de red multithread (parte 2)

```
//Stop accepting new Callables
es.shutdown();

try {
    //Block until all Callables have a chance to finish
    es.awaitTermination(5, TimeUnit.SECONDS);
} catch (InterruptedException ex) {
    System.out.println("Stopped waiting early");
}
```

# Cliente de red multithread (parte 3)

```
        for(RequestResponse lookup : callables.keySet()) {
            Future<RequestResponse> future = callables.get(lookup);
            try {
                lookup = future.get();
                System.out.println(lookup.host + ":" + lookup.port + " " +
                    lookup.response);
            } catch (ExecutionException|InterruptedException ex) {
                //This is why the callables Map exists
                //future.get() fails if the task failed
                System.out.println("Error talking to " + lookup.host +
                    ":" + lookup.port);
            }
        }
    }
}
```

ORACLE

# Cliente de red multithread (parte 4)

```
public class RequestResponse {
    public String host; //request
    public int port; //request
    public String response; //response

    public RequestResponse(String host, int port) {
        this.host = host;
        this.port = port;
    }

    // equals and hashCode

}
```

# Cliente de red multithread (parte 5)

```
public class NetworkClientCallable implements Callable<RequestResponse> {
    private RequestResponse lookup;

    public NetworkClientCallable(RequestResponse lookup) {
        this.lookup = lookup;
    }

    @Override
    public RequestResponse call() throws IOException {
        try (Socket sock = new Socket(lookup.host, lookup.port);
             Scanner scanner = new Scanner(sock.getInputStream());) {
            lookup.response = scanner.next();
            return lookup;
        }
    }
}
```

ORACLE

# Paralelismo

Los sistemas modernos contienen varias CPU. Para sacar partido de la potencia de procesamiento en un sistema es preciso ejecutar tareas en paralelo en varias CPU.

- Divide y vencerás: una tarea se debe dividir en subtareas. Debe intentar identificar aquellas subtareas que se puedan ejecutar en paralelo.
- Puede ser difícil ejecutar algunos problemas como tareas paralelas.
- Algunos problemas son más sencillos. Los servidores que soportan varios clientes pueden usar una tarea independiente para manejar cada cliente.
- Tenga cuidado con el hardware. La programación de demasiadas tareas paralelas puede afectar de forma negativa al rendimiento.

ORACLE







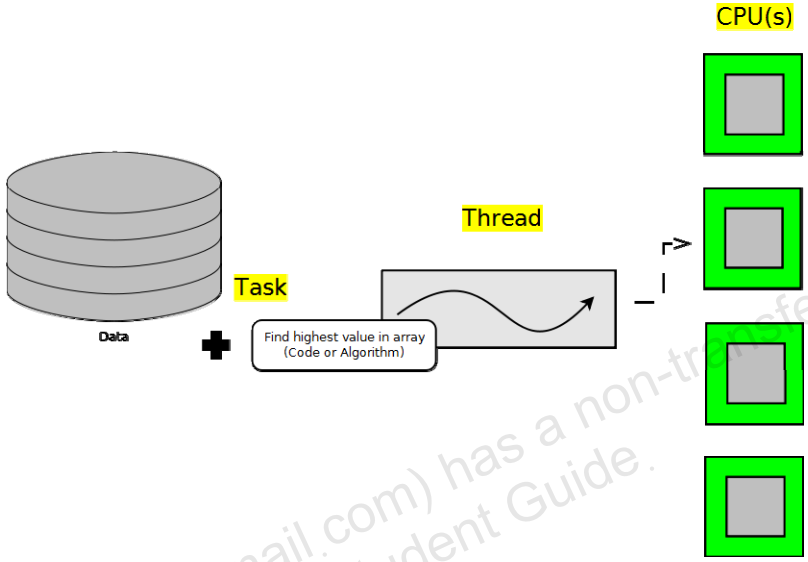
### Recuento de CPU

Si las tareas requieren muchos cálculos, al contrario de operaciones que generan muchas E/S, el número de tareas paralelas no debe superar en gran cantidad el número de procesadores del sistema. Puede detectar el número de procesadores de forma sencilla en Java:

```
int count = Runtime.getRuntime().availableProcessors();
```

# Sin paralelismo

Los sistemas modernos contienen varias CPU. Si no aprovecha los threads de alguna forma, solo se utilizará una parte de la potencia de procesamiento del sistema.




ORACLE





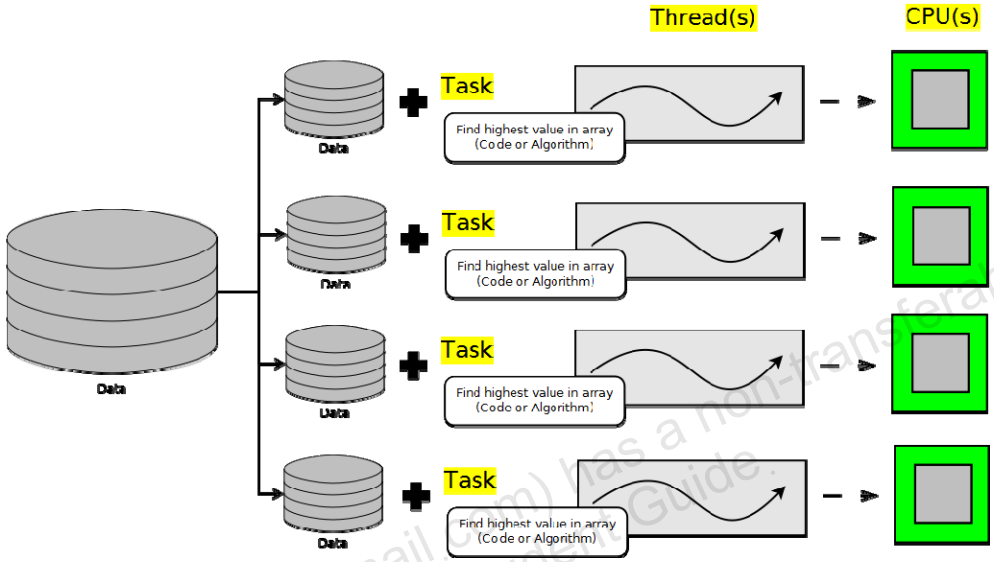


**Definición de etapa**

Si tiene una gran cantidad de datos que procesar pero solo un thread para procesar dichos datos, se utilizará una única CPU. En el gráfico de la diapositiva, se va a procesar un gran número de datos (una matriz, posiblemente). El procesamiento de matriz podría ser una tarea simple, como buscar el valor más alto en la matriz. En un sistema de cuatro CPU, debe haber tres CPU inactivas mientras se procesa la matriz.

# Paralelismo Naive

Una solución paralela simple divide los datos que se van a procesar en varios juegos. Un juego de datos para cada CPU y un thread para procesar cada juego de datos.




ORACLE







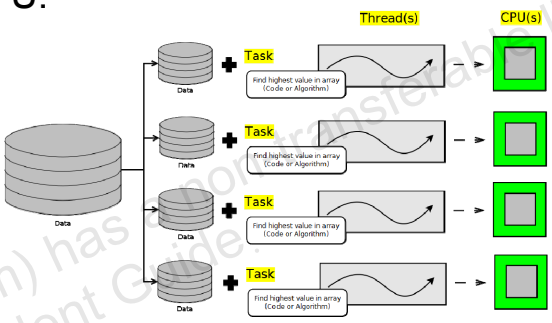
### División de datos

En el gráfico de la diapositiva, un gran juego de datos (una matriz, posiblemente) se divide en cuatro subjuegos de datos, uno para cada CPU. Se crea un thread por CPU para procesar los datos. Tras el procesamiento de los subjuegos de datos, los resultados se tendrán que combinar de una forma significativa. Hay distintas formas de subdividir el juego de datos grande que se va a procesar. Se usaría demasiada memoria para crear una nueva matriz por thread que contenga una copia de una parte de la matriz original. Cada matriz puede compartir una referencia a una única matriz grande pero solo acceder a un subjuego de una forma con protección de thread no bloqueante.

# La necesidad de un marco Fork-Join

La división de juegos de datos en subjuegos con el mismo tamaño para cada thread que se va a procesar tiene un par de problemas. Lo ideal es que todas las CPU se utilicen completamente hasta que la tarea finalice pero:

- Las CPU se pueden ejecutar a diferentes velocidades.
- Las tareas que no son de Java requieren tiempo de CPU y pueden reducir el tiempo del que dispone un thread de Java para la ejecución en una CPU.
- Los datos que se analizan pueden requerir diferentes cantidades de tiempo para el proceso.



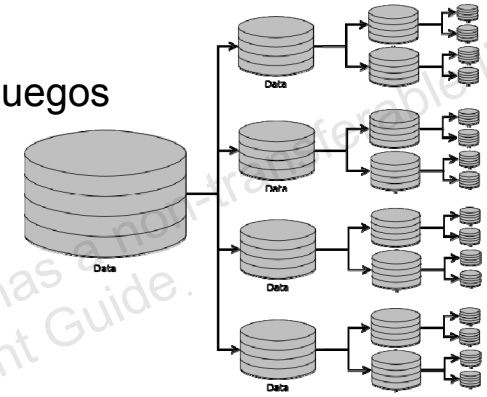

ORACLE

# Extracción de trabajo

Para mantener varios threads ocupados:

- Divida los datos que se van a procesar en un gran número de subjuegos.
- Asigne los subjuegos de datos a una cola de procesamiento de threads.
- Cada thread tendrá muchos subjuegos en cola.

Si un thread finaliza todos sus subjuegos pronto, puede “extraer” subjuegos de otro thread.




ORACLE



### Granularidad de trabajo

Al subdividir los datos que se van a procesar hasta que haya más subjuegos que threads, se está facilitando la "extracción de trabajo". En la extracción de trabajo, un thread que se queda sin trabajo puede extraer trabajo (un subjuego de datos) de la cola de procesamientos de otro thread. Debe determinar el tamaño óptimo del trabajo que desee agregar a cada cola de procesamiento de thread. La subdivisión excesiva de datos que se van a procesar pueden causar una sobrecarga innecesaria, mientras que una división insuficiente de datos puede dar como resultado una infrautilización de la CPU.

# Ejemplo de thread único

```
int[] data = new int[1024 * 1024 * 256]; //1G

for (int i = 0; i < data.length; i++) {
    data[i] = ThreadLocalRandom.current().nextInt();
}

int max = Integer.MIN_VALUE;
for (int value : data) {
    if (value > max) {
        max = value;
    }
}
System.out.println("Max value found:" + max);
```

Juego de datos muy grande.

Llenar la matriz con valores.

Buscar de forma secuencial la matriz para el valor mayor.



**Paralelo potencial**

En este ejemplo hay dos tareas independientes que se podrían ejecutar en paralelo. La inicialización de la matriz con valores aleatorios y la búsqueda de la matriz del mayor valor posible podrían hacerse en paralelo.

## java.util.concurrent.ForkJoinTask<V>

Un objeto `ForkJoinTask` representa una tarea que se va a ejecutar.

- Una tarea contiene el código y los datos que se van a procesar. Similar a `Runnable` o `Callable`.
- Un número pequeño de threads en un pool Fork-Join crea y procesa un gran número de tareas.
  - `ForkJoinTask` normalmente crea más instancias `ForkJoinTask` hasta que los datos que se van procesar se subdividen de forma adecuada.
- Los desarrolladores normalmente utilizan las siguientes subclases:
  - `RecursiveAction`: si una tarea no tiene que devolver un resultado.
  - `RecursiveTask`: si una tarea tiene que devolver un resultado.

ORACLE

# Ejemplo de `RecursiveTask`

```
public class FindMaxTask extends RecursiveTask<Integer> {
    private final int threshold;
    private final int[] myArray;
    private int start;
    private int end;

    public FindMaxTask(int[] myArray, int start, int end,
int threshold) {
        // copy parameters to fields
    }
    protected Integer compute() {
        // shown later
    }
}
```

Tipo de resultado de la tarea.

Datos a procesar.

Dónde se realiza el trabajo. Observe el tipo de devolución genérica.

# Estructura de `compute`

```
protected Integer compute() {
    if DATA_SMALL_ENOUGH {
        PROCESS_DATA
        return RESULT;
    } else {
        SPLIT_DATA_INTO_LEFT_AND_RIGHT_PARTS
        TASK t1 = new TASK(LEFT_DATA);
        t1.fork();
        TASK t2 = new TASK(RIGHT_DATA);
        return COMBINE(t2.compute(), t1.join());
    }
}
```

Ejecución asíncrona

Proceso en el thread actual

Bloquear hasta que se termine

ORACLE

# Ejemplo de `compute` (por debajo del umbral)

```
protected Integer compute() {
    if (end - start < threshold) {
        int max = Integer.MIN_VALUE;
        for (int i = start; i <= end; i++) {
            int n = myArray[i];
            if (n > max) {
                max = n;
            }
        }
        return max;
    } else {
        // split data and create tasks
    }
}
```

Umbral decidido por el usuario

Rango en la matriz

ORACLE

# Ejemplo de `compute` (por encima del umbral)

```
protected Integer compute() {
    if (end - start < threshold) {
        // find max
    } else {
        int midway = (end - start) / 2 + start;
        FindMaxTask a1 =
    new FindMaxTask(myArray, start, midway, threshold);
        a1.fork();
        FindMaxTask a2 =
    new FindMaxTask(myArray, midway + 1, end, threshold);
        return Math.max(a2.compute(), a1.join());
    }
}
```

Tarea para la mitad izquierda de los datos

Tarea para la mitad derecha de los datos

ORACLE



### Gestión de memoria

Observe que la misma matriz se transfiere a cada tarea pero con diferentes valores de inicio y finalización. Si el subjuego de valores que se va a procesar se copiara en una matriz cada vez que se crea una tarea, el uso de la memoria aumentaría rápidamente.

# Ejemplo de `ForkJoinPool`

`ForkJoinPool` se utiliza para ejecutar `ForkJoinTask`.
Crea un thread para cada CPU en el sistema por defecto.

```
ForkJoinPool pool = new ForkJoinPool();
FindMaxTask task =
 new FindMaxTask(data, 0, data.length-1, data.length/16);
Integer result = pool.invoke(task);
```

El método `compute` de la tarea se llama automáticamente.

# Recomendaciones del marco Fork-Join

- Evite operaciones de bloqueo o E/S.
  - Solo se crea un thread por CPU por defecto. Las operaciones de bloqueo evitarán el uso de todos los recursos de CPU.
- Conozca el hardware.
  - Una solución Fork-Join se ejecutará de forma más lenta en un sistema de una CPU que en una solución secuencial estándar.
  - Algunas CPU aumentan la velocidad solo cuando usan un único núcleo, lo que podría compensar de forma potencial cualquier aumento de rendimiento proporcionado por Fork-Join.
- Conozca el problema.
  - Muchos de los problemas tienen una sobrecarga adicional si se ejecutan en paralelo (ordenación paralela, por ejemplo).



**Ordenación paralela**

Al utilizar Fork-Join para ordenar una matriz en paralelo, se termina ordenando muchas pequeñas matrices que, a continuación, se combinan en matrices ordenadas más grandes. Para ver un ejemplo, consulte la aplicación de ejemplo proporcionada con JDK en `C:\Program Files\Java\jdk1.7.0\sample\forkjoin\mergesort`.

# Prueba

La aplicación del marco Fork-Join siempre dará como resultado un aumento del rendimiento.

a. Verdadero
b. Falso

# Resumen

En esta lección, debe haber aprendido a hacer lo siguiente:

- Usar variables atómicas
- Usar `ReentrantReadWriteLock`
- Usar recopilaciones `java.util.concurrent`
- Describir las clases de sincronizador
- Usar `ExecutorService` para ejecutar tareas de forma simultánea
- Aplicar el marco Fork-Join

ORACLE

# (Opcional) Visión general de la práctica 13-1: Uso del paquete `java.util.concurrent`

En esta práctica, se abordan los siguientes temas:

- Uso de un pool de threads almacenado en caché (`ExecutorService`)
- Implantación de `Callable`
- Recepción de resultados de `Callable` con `Future`



En esta práctica, va a crear un cliente de red multithread.

# (Opcional) Visión general de la práctica 13-2: Uso del marco Fork-Join

En esta práctica, se abordan los siguientes temas:

- Ampliación de `RecursiveAction`
- Creación y uso de `ForkJoinPool`



En esta práctica, va a crear un cliente de red multithread.

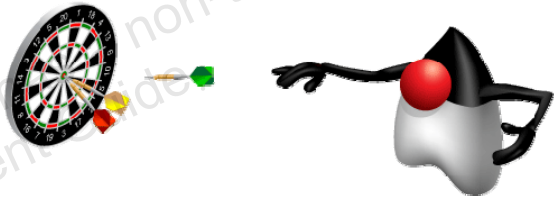
# 14

# Creación de aplicaciones de base de datos con JDBC

# Objetivos

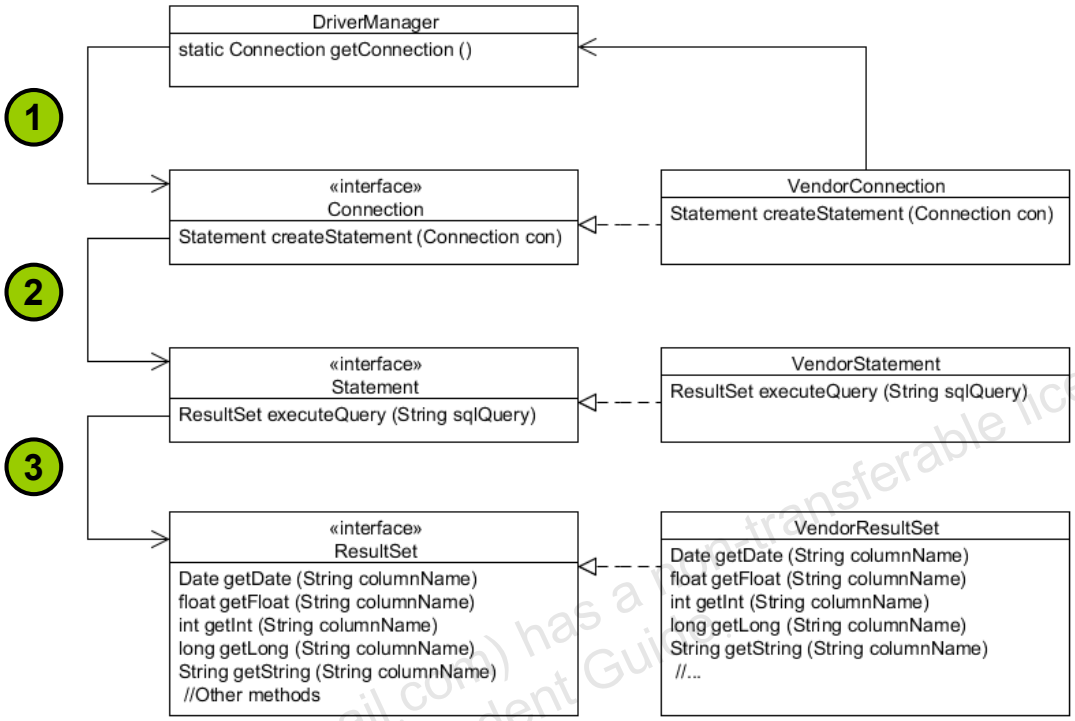
Al finalizar esta lección, debería estar capacitado para:

- Definir el diseño de la API de JDBC
- Conectarse a una base de datos mediante un controlador JDBC
- Enviar consultas y obtener resultados de la base de datos
- Especificar información sobre el controlador JDBC de forma externa
- Usar transacciones con JDBC
- Usar `RowSetProvider` y `RowSetFactory` de JDBC 4.1
- Usar un patrón de objeto de acceso a datos para separar datos y métodos de negocio



ORACLE

## Uso de la API de JDBC








La API de JDBC se compone de algunas clases determinadas, como `Date`, `Time` y `SQLException`, así como de un juego de interfaces que se implantan en una clase de controlador proporcionada por el proveedor de base de datos.

Debido a que la implantación es una instancia válida de la firma de método de interfaz, una vez que se cargan las clases de controlador del proveedor de base de datos, puede acceder a ellas siguiendo la secuencia que se muestra en la diapositiva:

1. Utilice la clase `DriverManager` para obtener una referencia a un objeto `Connection` mediante el método `getConnection`. La firma típica de este método es `getConnection (url, name, password)`, donde `url` es la URL de JDBC y `name` y `password` son cadenas que la base de datos aceptará para una conexión.
2. Use el objeto `Connection` (implantado por alguna clase proporcionada por el proveedor) para obtener una referencia a un objeto `Statement` mediante el método `createStatement`. La firma típica para este método es `createStatement ()` sin argumentos.
3. Utilice el objeto `Statement` para obtener una instancia de `ResultSet` a través de un método `executeQuery (query)`. Este método normalmente acepta una cadena (`query`) donde `query` es una SQL de cadena estática.

# Uso de clases de controlador de proveedor

La clase `DriverManager` se utiliza para obtener una instancia de un objeto de conexión, mediante el controlador JDBC nombrado en la URL de JDBC:

```
String url = "jdbc:derby://localhost:1527/EmployeeDB";
Connection con = DriverManager.getConnection (url);
```

- La sintaxis de URL para un controlador JDBC es:

```
jdbc:<driver>:[subsubprotocol:][databaseName][;attribute=value]
```

- Cada proveedor puede implantar su propio subprotocolo.
- La sintaxis de URL para un controlador Thin de Oracle es:

```
jdbc:oracle:thin:@//[HOST][:PORT]/SERVICE
```

Ejemplo:

```
jdbc:oracle:thin:@//myhost:1521/orcl
```

ORACLE







### DriverManager

Los controladores JDBC 4.0 que se encuentran en la ruta de acceso de clase se cargan automáticamente. El método `DriverManager.getConnection` intentará cargar la clase de controlador consultando el archivo `META_INF/services/java.sql.Driver`. Este archivo contiene el nombre de la implantación del controlador JDBC de `java.sql.Driver`. Por ejemplo, el contenido del archivo `META-INF/services/java.sql.driver` en `derbyclient.jar` contiene `org.apache.derby.jdbc.ClientDriver`.

Los controladores anteriores a JDBC 4.0 se deben cargar manualmente mediante:

```
try {
    java.lang.Class.forName("<fully qualified path of the driver>");
} catch (ClassNotfoundException c) {
}
```

Las clases de controlador también se pueden transferir al intérprete en la línea de comandos:

```
java -djdbc.drivers=<fully qualified path to the driver> <class to run>
```

## Componentes de la API de JDBC clave

Cada clase de controlador JDBC del proveedor también implanta las clases de API clave que usará para conectarse a la base de datos, ejecutar consultas y manipular datos:

- `java.sql.Connection`: conexión que representa la sesión entre la aplicación Java y la base de datos.

```
Connection con = DriverManager.getConnection(url,
   username, password);
```

- `java.sql.Statement`: objeto usado para ejecutar una sentencia SQL estática y devolver el resultado.

```
Statement stmt = con.createStatement();
```

- `java.sql.ResultSet`: objeto que representa un juego de resultados de la base de datos.

```
String query = "SELECT * FROM Employee";
ResultSet rs = stmt.executeQuery(query);
```

ORACLE







### Objetos Connection, Statement y ResultSets

La verdadera virtud de la API de JDBC es que proporciona una manera flexible y portátil de comunicarse con la base de datos.

El controlador JDBC proporcionado por un proveedor de base de datos implanta cada una de estas interfaces Java. El código Java puede usar la interfaz sabiendo que el proveedor de base de datos ha proporcionado la implantación de cada uno de los método de la interfaz.

`Connection` es una interfaz que proporciona una sesión con la base de datos. Mientras el objeto de conexión está abierto, puede acceder a la base de datos, crear sentencias, obtener resultados y manipular la base de datos. Al cerrar una conexión, el acceso a la base de datos termina y la conexión abierta se cierra.

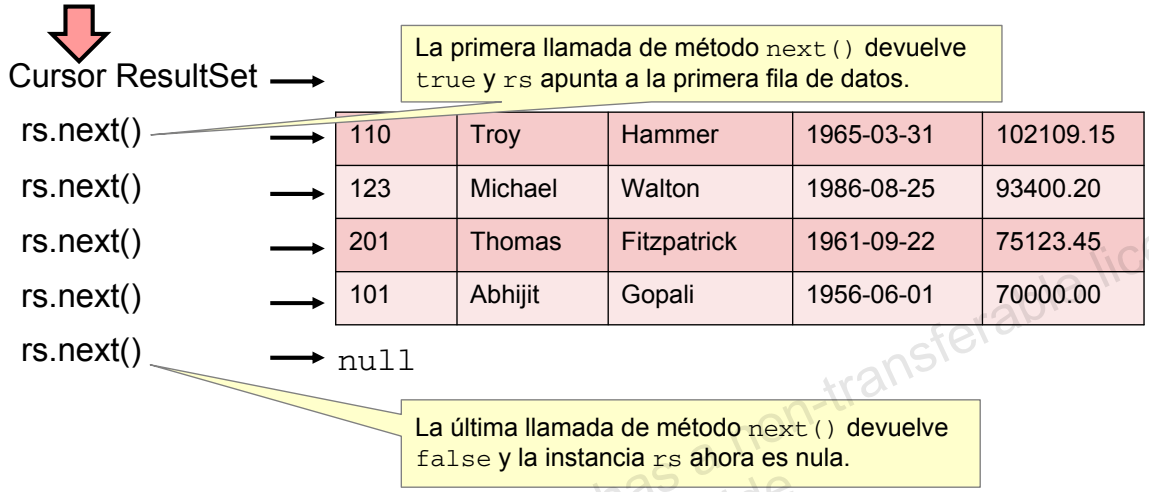
`Statement` es una interfaz que proporciona una clase para ejecutar sentencias SQL y devolver resultados. La interfaz `Statement` se utiliza para consultas SQL estáticas. Hay dos subinterfaces más: `PreparedStatement`, que amplía `Statement`, y `CallableStatement`, que amplía `PreparedStatement`.

`ResultSet` es una interfaz que gestiona los datos resultantes devueltos de `Statement`.

**Nota:** las palabras clave y comandos SQL no son sensibles a mayúsculas/minúsculas, es decir puede usar `SELECT` o `Select`. Los nombres de columna y tabla SQL (identificadores) pueden ser sensibles a mayúsculas/minúsculas o no según la base de datos. Los identificadores SQL no son sensibles a mayúsculas/minúsculas en la base de datos Derby (a menos que se delimite).

# Uso de un objeto `ResultSet`

```
String query = "SELECT * FROM Employee";
ResultSet rs = stmt.executeQuery(query);
```








## Objetos `ResultSet`

- `ResultSet` mantiene un cursor en las filas devueltas. El cursor está apuntando inicialmente antes de la primera fila.
- Se llama al método `ResultSet.next()` para colocar el curso en la siguiente fila.
- El objeto por defecto `ResultSet` no se puede actualizar y tiene un cursor que solo apunta hacia adelante.
- Es posible producir objetos `ResultSet` que se pueden desplazar o actualizar. El siguiente fragmento de código, en el que `con` es un objeto `Connection` válido, muestra cómo crear un juego de resultados desplazable, en el que otros usuarios no puedan realizar actualizaciones y que se puede actualizar:

```
Statement stmt
       = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                             ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT a, b FROM TABLE2");
```

**Nota:** no todas las bases de datos soportan juegos de resultados desplazables.

`ResultSet` tiene métodos de acceso para leer el contenido de cada columna devuelta en una fila. `ResultSet` tiene un método `getter` para cada tipo.

# Unión de todo

```
1 package com.example.text;
2
3 import java.sql.DriverManager;
4 import java.sql.ResultSet;
5 import java.sql.SQLException;
6 import java.util.Date;
7
8 public class SimpleJDBCTest {
9
10    public static void main(String[] args) {
11        String url = "jdbc:derby://localhost:1527/EmployeeDB";
12        String username = "public";
13        String password = "tiger";
14        String query = "SELECT * FROM Employee";
15        try (Connection con =
16             DriverManager.getConnection (url, username, password);
17             Statement stmt = con.createStatement ();
18             ResultSet rs = stmt.executeQuery (query)) {
```

La contraseña, nombre de usuario y URL de JDBC codificada sirven solo de ejemplo.

ORACLE

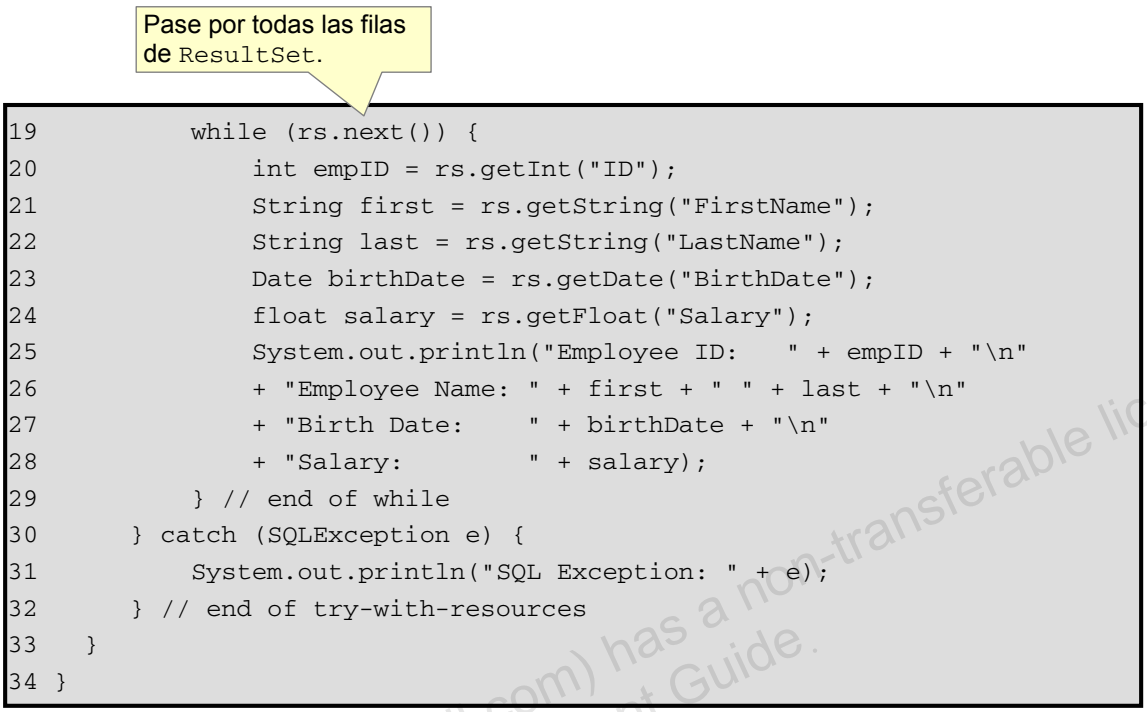


En esta diapositiva y la siguiente, verá un ejemplo completo de una aplicación JDBC, una simple que lee todas las filas de una base de datos de empleado y devuelve los resultados como cadenas a la consola.

- **Línea 15–16:** utilice una sentencia `try`-with-resources para obtener una instancia de un objeto que implanta la interfaz `Connection`.
- **Línea 17:** utilice dicho objeto para obtener una instancia de un objeto que implanta la interfaz `Statement` del objeto `Connection`.
- **Línea 18:** cree un objeto `ResultSet` ejecutando la consulta de cadena mediante el objeto `Statement`.

**Nota:** la codificación de la URL de JDBC, nombre de usuario y contraseña hace que una aplicación sea menos portátil. En su lugar, considere usar `java.io.Console` para leer el nombre de usuario y la contraseña, o algún tipo de servicio de autenticación.

# Unión de todo




- **Líneas 20–24:** obtiene los resultados de cada uno de los campos de datos en cada fila leída de la tabla `Employee`.
- **Líneas 25–28:** imprima los campos de datos resultantes en la consola del sistema.
- **Línea 30:** `SQLException`: esta clase amplía la `Exception` devuelta por los métodos `DriverManager`, `Statement` y `ResultSet` (puede obtener más información sobre esta clase de excepción en la siguiente diapositiva).
- **Línea 32:** se trata del corchete de cierre para la sentencia `try`-with-resources de la línea 15.

Este ejemplo se extrae del proyecto `SimpleJDBCExample`.

Resultado:

```
run:
Employee ID:   110
Employee Name: Troy Hammer
Birth Date:    1965-03-31
Salary:        102109.15
```

etc.

# Escritura de código JDBC portátil

El controlador JDBC proporciona una capa "aislante" mediante programación entre la aplicación Java y la base de datos. Sin embargo, también debe considerar la semántica y la sintaxis SQL al escribir aplicaciones de base de datos.

- La mayoría de las bases de datos soportan un juego estándar de semántica y sintaxis SQL descrita por la especificación de nivel de entrada SQL-92 de ANSI (American National Standards Institute).
- Puede comprobar mediante programación el soporte para esta especificación desde su controlador:

```
Connection con = DriverManager.getConnection(url, username,
   password);
DatabaseMetaData dbm = con.getMetaData();
if (dbm.supportsANSI92EntrySQL()) {
   // Support for Entry-level SQL-92 standard
}
```

ORACLE



En general, probablemente escribirá una aplicación que aprovecha las capacidades y funciones de la base de datos con la que está trabajando. Sin embargo, si desea escribir una aplicación portátil, debe considerar el soporte que cada base de datos le proporcionará para la funcionalidad y tipos de SQL. Afortunadamente, puede consultar el controlador de base de datos mediante programación para determinar el nivel de soporte que proporciona el controlador. La interfaz `DatabaseMetaData` tiene un juego de métodos que el desarrollador del controlador usa para indicar lo que soporta el controlador, incluido el soporte para la entrada, soporte intermedio o completo para SQL-92.

La interfaz `DatabaseMetaData` también incluye otros métodos que determinan el tipo de soporte que la base de datos proporciona para consultas, tipos, transacciones, etc.

# Clase SQLException

`SQLException` se puede usar para notificar detalles sobre los errores de la base de datos resultantes. Para informar de todas las excepciones devueltas, puede iterar con la `SQLException` devuelta:

```
1 catch(SQLException ex) {
2     while(ex != null) {
3         System.out.println("SQLState:   " + ex.getSQLState());
4         System.out.println("Error Code:" + ex.getErrorCode());
5         System.out.println("Message:    " + ex.getMessage());
6         Throwable t = ex.getCause();
7         while(t != null) {
8             System.out.println("Cause:" + t);
9             t = t.getCause();
10         }
11         ex = ex.getNextException();
12     }
13 }
```

Códigos de estado, códigos de error y mensajes dependientes del proveedor

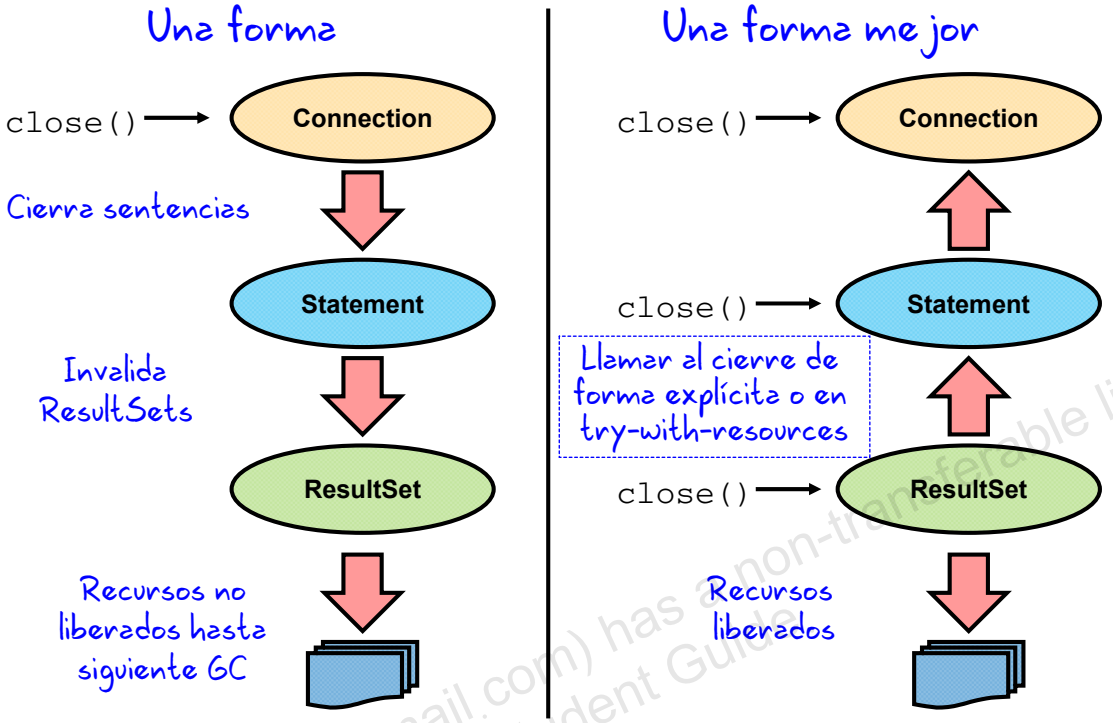
ORACLE



- Se devuelve una `SQLException` de los errores producidos en uno de los siguientes tipos de acciones: métodos de controlador, métodos que acceden a la base de datos o intentos de obtener una conexión a la base de datos.
- La clase `SQLException` también implanta `Iterable`. Las excepciones se pueden encadenar juntas y devolver como un único objeto.
- `SQLException` se devuelve si la conexión a la base de datos no se puede realizar debido a información de contraseña o nombre de usuario incorrecta o simplemente a que la base de datos esté fuera de línea.
- `SQLException` también puede resultar al intentar acceder a un nombre de columna que no forma parte de la consulta SQL.
- `SQLException` es también una subclase, que proporciona granularidad de la excepción actual devuelta.

**Nota:** los valores `SQLState` y `SQLErrorCode` dependen de la base de datos. Para Derby, los valores de `SQLState` se definen aquí: http://download.oracle.com/javadb/10.8.1.2/ref/rrefexcept71493.html.

## Cierre de objetos de JDBC








- Al cerrar un objeto `Connection` se cerrará automáticamente cualquier objeto `Statement` creado con este objeto `Connection`.
- Al cerrar un objeto `Statement` se cerrarán e invalidarán las instancias de `ResultSet` creadas por el objeto `Statement`.
- Los recursos contenidos por `ResultSet` no se liberarán hasta que no se realice la recolección de basura, por lo que se trata de un método recomendable para cerrar de forma explícita los objetos `ResultSet` que ya no sean necesarios.
- Al ejecutar el método `close()` en `ResultSet`, se liberan los recursos externos.
- Los objetos `ResultSet` también se cierran de forma implícita al volver a ejecutar un objeto Statement `asociado`.

En resumen, se recomienda cerrar de forma explícita los objetos `Connection`, `Statement` y `ResultSet` de JDBC cuando ya no son necesarios.

**Nota:** una conexión a la base de datos puede ser una operación costosa. Se recomienda mantener los objetos `Connection` lo máximo posible o usar un pool de conexiones.

# Construcción `try`-with-resources

Con la siguiente sentencia `try`-with-resources:

```
try (Connection con =
     DriverManager.getConnection(url, username, password);
     Statement stmt = con.createStatement();
     ResultSet rs = stmt.executeQuery (query)){
```

- El compilador comprueba para ver que el objeto entre paréntesis implanta `java.lang.AutoCloseable`.
  - Esta interfaz incluye un método: `void close()`.
- El método de cierre se llama automáticamente al final de bloque try en el orden adecuado (de la última declaración a la primera).
- Es posible incluir varios recursos que se puedan cerrar en el bloque try, separados por punto y coma.

ORACLE



Una de las funciones de JDK 7 es la sentencia `try`-with-resources. Se trata de una mejora que cierra automáticamente los recursos abiertos.

Con JDBC 4.1, las clases de API de JDBC que incluyen `ResultSet`, `Connection` y `Statement` implantan `java.lang.AutoCloseable`. El método `close()` de los objetos `ResultSet`, `Statement` y `Connection` se llamará en este ejemplo.

## `try`-with-resources: práctica incorrecta

Puede ser tentador escribir `try`-with-resources de forma más compacta:

```
try (ResultSet rs = DriverManager.getConnection(url, username,
password).createStatement().executeQuery(query)) {
```

- Sin embargo, solo se llama el método de cierre de `ResultSet`, lo cual no es una buena práctica.
- Recuerde siempre qué recursos debe cerrar al usar `try`-with-resources.



### Evitar este problema de `try`-with-resources

Puede parecer que es una forma de ahorrar tiempo escribir estas tres sentencias, pero el efecto final es que `Connection` devuelto por `DriverManager` nunca se cierra de forma explícita al final del bloque try por lo que no es un método recomendable.

# Escritura de consultas y obtención de resultados

Para ejecutar consultas SQL con JDBC, debe crear un objeto de envoltorio de consulta SQL, una instancia del objeto Statement.

```
Statement stmt = con.createStatement();
```

- Utilice la instancia Statement para ejecutar una consulta SQL:

```
ResultSet rs = stmt.executeQuery (query);
```

- Tenga en cuenta que hay tres métodos de ejecución de Statement:

| Método | Devuelve | Se utiliza para |
|---|---|---|
| `executeQuery(sqlString)` | `ResultSet` | Sentencia `SELECT` |
| `executeUpdate(sqlString)` | `int` (filas afectadas) | `INSERT`, `UPDATE`, `DELETE` o DDL |
| `execute(sqlString)` | `boolean` (true si hay `ResultSet`) | Comandos o comando SQL |

ORACLE



Una sentencia SQL se ejecuta en una base de datos mediante una instancia de un objeto Statement. El objeto Statement es un objeto de envoltorio para una consulta. Un objeto Statement se obtiene a través de un objeto Connection, la conexión a la base de datos. Así que tiene sentido que, a partir de Connection, obtenga un objeto que se puede utilizar para escribir sentencias en la base de datos.

La interfaz Statement proporciona tres métodos para crear consultas SQL y devolver un resultado. El método que use depende del tipo de sentencia SQL que desee utilizar:

- `executeQuery(sqlString)`: para una sentencia `SELECT`, devuelve un objeto `ResultSet`.
- `executeUpdate(sqlString)`: para sentencias `INSERT`, `UPDATE` y `DELETE`, devuelve `int` (número de filas afectadas) o 0 cuando la sentencia es una sentencia de Lenguaje de definición de datos (DDL), como `CREATE TABLE`.
- `execute(sqlString)`: para sentencias SQL, devuelve un `boolean` que indica si se ha devuelto `ResultSet`. Se pueden ejecutar varias sentencias SQL con `execute`.

# Visión general de la práctica 14-1: Trabajo con la base de datos Derby y JDBC

En esta práctica, se abordan los siguientes temas:

- Inicio de la base de datos JavaDB (Derby) desde NetBeans IDE
- Relleno de la base de datos con datos (tabla Employee)
- Ejecución de consultas SQL para examinar los datos
- Compilación y ejecución de la aplicación JDBC de ejemplo

ORACLE



En esta práctica, iniciará la base de datos desde NetBeans, rellenará la base de datos con datos, ejecutará algunas consultas SQL, y compilará y ejecutará una aplicación simple que devuelva filas de la tabla de base de datos Employee.

# ResultSetMetaData

En un momento dado puede necesitar detectar dinámicamente el número de columnas y su tipo.

```
1 int numCols = rs.getMetaData().getColumnCount();
2 String [] colNames = new String[numCols];
3 String [] colTypes = new String[numCols];
4 for (int i= 0; i < numCols; i++) {
5     colNames[i] = rs.getMetaData().getColumnName(i+1);
6     colTypes[i] = rs.getMetaData().getColumnTypeName(i+1);
7 }
8 System.out.println ("Number of columns returned: " + numCols);
9 System.out.println ("Column names/types returned: ");
10 for (int i = 0; i < numCols; i++) {
11     System.out.println (colNames[i] + " : " + colTypes[i]);
12 }
```

Observe que estos métodos se indexan desde 1, no 0.



La clase `ResultSetMetaData` se obtiene a partir de un objeto `ResultSet`.

`getColumnCount` devuelve el número de columnas devueltas en la consulta que ha producido `ResultSet`.

Los métodos `getColumnName` y `getColumnTypeName` devuelven cadenas. Se puede usar para realizar una recuperación dinámica de los datos de la columna.

**Nota:** estos métodos utilizan 1 para indicar la primera columna, no 0.

Con una consulta de "SELECT * FROM Employee" y la tabla de datos Employee de las prácticas, este fragmento producirá este resultado:

```
Number of columns returned: 5
Column names/types returned:
ID : INTEGER
FIRSTNAME : VARCHAR
LASTNAME : VARCHAR
BIRTHDATE : DATE
SALARY : REAL
```

# Obtención de recuento de filas

Una pregunta común al ejecutar una consulta es cuántas filas se han devuelto.

```
1 public int rowCount(ResultSet rs) throws SQLException{
2     int rowCount = 0;
3     int currRow = rs.getRow();
4     // Valid ResultSet?
5     if (!rs.last()) return -1;
6     rowCount = rs.getRow();
7     // Return the cursor to the current position
8     if (currRow == 0) rs.beforeFirst();
9     else rs.absolute(currRow);
10    return rowCount;
11 }
```

Mueve el cursor hasta la última fila; este método devuelve `false` si el objeto `ResultSet` está vacío.

La devolución del cursor de fila a su posición original antes de la llamada es un método recomendable.

- Para usar esta técnica, `ResultSet` debe ser desplazable.

ORACLE



**Nota:** recuerde que para crear un objeto `ResultSet` desplazable debe definir el tipo `ResultSet` en el método `createStatement`:

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
```

Hay otra técnica para el objeto ResultSets no desplazable. Al usar la función SQL COUNT, una consulta determina el número de filas y una segunda lee los resultados. Tenga en cuenta que esta técnica requiere el bloqueo del control en las tablas para garantizar que el recuento no cambia durante la operación:

```
ResultSet rs = stmt.executeQuery("SELECT COUNT(*) FROM EMPLOYEE");
rs.next();
int count = rs.getInt(1);
rs.stmt.excuteQuery ("SELECT * FROM EMPLOYEE");
// process results
```

# Control del tamaño de recuperación de `ResultSet`

Por defecto, el número de filas recuperadas a la vez por una consulta se determina mediante el controlador JDBC. Es posible que desee controlar este comportamiento en el caso de juegos de datos grandes.

- Por ejemplo, si desea limitar el número de filas recuperadas en la caché a 25, puede definir el tamaño de recuperación:

```
rs.setFetchSize(25);
```

- Las llamadas a `rs.next()` devuelven los datos en la caché hasta la fila 26ª, momento en el que el controlador recuperará otras 25 filas.

ORACLE



**Nota:** normalmente el tamaño de recuperación más eficaz es el valor por defecto para el controlador.

# Uso de `PreparedStatement`

`PreparedStatement` es una subclase de Statement que permite transferir argumentos a una sentencia SQL precompilada.

```
double value = 100_000.00;
String query = "SELECT * FROM Employee WHERE Salary > ?";
PreparedStatement pStmt = con.prepareStatement(query);
pStmt.setDouble(1, value);
ResultSet rs = pStmt.executeQuery();
```

Parámetro para sustitución.

Sustituye `value` para el primer parámetro en la sentencia preparada.

- En este fragmento de código, una sentencia preparada devuelve todas las columnas de todas las filas cuyo salario es mayor de 100 000 dólares.
- `PreparedStatement` es útil cuando tiene sentencias SQL que va a ejecutar varias veces.

ORACLE







**`PreparedStatement`**

La sentencia SQL en el ejemplo de la diapositiva se precompila y almacena en el objeto `PreparedStatement`. Esta sentencia se puede usar de forma eficaz para ejecutar esta sentencia varias veces. Este ejemplo podría estar en un bucle, consultando valores diferentes.

Las sentencias preparadas también se pueden usar para evitar ataques de inyección de SQL. Por ejemplo, donde un usuario puede introducir una cadena que, cuando se ejecuta como parte de una sentencia SQL, permite al usuario modificar la base de datos de formas no intencionadas (como otorgar permisos en sí).

**Nota:** los métodos `PreparedStatement` setXXXX indexan parámetros a partir de 1, no 0.

# Uso de `CallableStatement`

`CallableStatement` permite que sentencias no SQL (como procedimientos almacenados) se ejecuten en la base de datos.

```
CallableStatement cStmt
       = con.prepareCall("{CALL EmplAgeCount (?, ?)}");
int age = 50;
cStmt.setInt (1, age);
ResultSet rs = cStmt.executeQuery();
cStmt.registerOutParameter(2, Types.INTEGER);
boolean result = cStmt.execute();
int count = cStmt.getInt(2);
System.out.println("There are " + count +
          " Employees over the age of " + age);
```

El parámetro `IN` se transfiere al procedimiento almacenado.

El parámetro `OUT` se devuelve desde el procedimiento almacenado.

- Los procedimientos almacenados se ejecutan en la base de datos.

ORACLE







### Procedimientos almacenados Derby

La base de datos Derby utiliza un lenguaje de programación Java para sus procedimientos almacenados.

En el ejemplo que muestra la diapositiva, el procedimiento almacenado se declara mediante la siguiente sintaxis:

```
CREATE PROCEDURE EmplAgeCount (IN age INTEGER, OUT num INTEGER) DYNAMIC
RESULT SETS 0
LANGUAGE JAVA
EXTERNAL NAME 'DerbyStoredProcedure.countAge'
PARAMETER STYLE JAVA
READS SQL DATA;
```

Una clase Java se carga en la base de datos Derby mediante la siguiente sintaxis:

```
CALL SQLJ.install_jar ('D:\temp\DerbyStoredProcedure.jar',
'PUBLIC.DerbyStoredProcedure', 0);
```

```
CALL syscs_util.syscs_set_database_property('derby.database.classpath',
'PUBLIC.DerbyStoredProcedure');
```

La clase Java almacenada en la base de datos Derby que realiza el procedimiento almacenado calcula una fecha en años en el pasado según la fecha actual. La consulta SQL cuenta el número de empleados únicos que son mayores (o igual que) el número de años pasados y devuelve dicho recuento como el segundo parámetro del procedimiento almacenado. El código de este ejemplo debe tener el siguiente aspecto:

```
import java.sql.Connection;
import java.sql.Date;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Calendar;

public class DerbyStoredProcedure {
    public static void countAge (int age, int[] count) throws SQLException {
        String url = "jdbc:default:connection";
        Connection con = DriverManager.getConnection(url);
        String query = "SELECT COUNT(DISTINCT ID) " +
                       "AS count FROM Employee " +
                       "WHERE Birthdate <= ?";
        PreparedStatement ps = con.prepareStatement(query);
        Calendar now = Calendar.getInstance();
        now.add(Calendar.YEAR, (age*-1));
        Date past = new Date (now.getTimeInMillis());
        ps.setDate(1, past);
        ResultSet rs = ps.executeQuery();
        if (rs.next()) {
            count[0] = rs.getInt(1);
        } else {
            count[0] = 0;
        }
        con.close();
    }
}
```

Consulte el manual de referencia, y la guía de utilidades y herramientas de Derby para obtener más información sobre la creación de procedimientos almacenados.

# ¿Qué es una transacción?

- Una transacción es un mecanismo para manejar grupos de operaciones como si fueran solo una.
- Puede darse el caso de que ocurran todas las operaciones de una transacción o ninguna en absoluto.
- Las operaciones implicadas en una transacción pueden depender de una o más bases de datos.





Un ejemplo clásico de cuándo usar una transacción sería: suponga que una aplicación cliente necesita realizar una solicitud de servicio que puede implicar varias operaciones de lectura y escritura en una base de datos. Si alguna de las llamadas no se realiza correctamente, se debe realizar un rollback de los estados escritos (en memoria o, con más frecuencia, en una base de datos).

Piense en una solicitud de transferencia de fondos interbancarios en la que el dinero se transfiere de un banco a otro.

La operación de transferencia requiere que el servidor realice las siguientes llamadas:

1. Llamada al método de débito en una cuenta en el primer banco
2. Llamada al método de crédito en otra cuenta en el segundo banco

Si la llamada de crédito en el segundo banco falla, la aplicación bancaria debe realizar un rollback de la llamada de débito anterior en el primer banco.

**Nota:** cuando una transacción abarca varias bases de datos, es posible que se requieran servicios de transacción más complicados.

# Propiedades ACID de una transacción

Una transacción formalmente la define el juego de propiedades que se conoce por el acrónimo ACID.

- **Atomicidad:** una transacción se hace o se deshace completamente. En caso de fallo, todas las operaciones y procedimientos se deshacen y se realiza un rollback de todos los datos a su estado anterior.
- **Consistencia:** una transacción transforma un sistema desde un estado consistente a otro estado consistente.
- **Aislamiento:** cada transacción ocurre de forma independiente de otras transacciones que ocurren al mismo tiempo.
- **Durabilidad:** las transacciones completadas permanecen como permanentes, incluso durante un fallo del sistema.

ORACLE





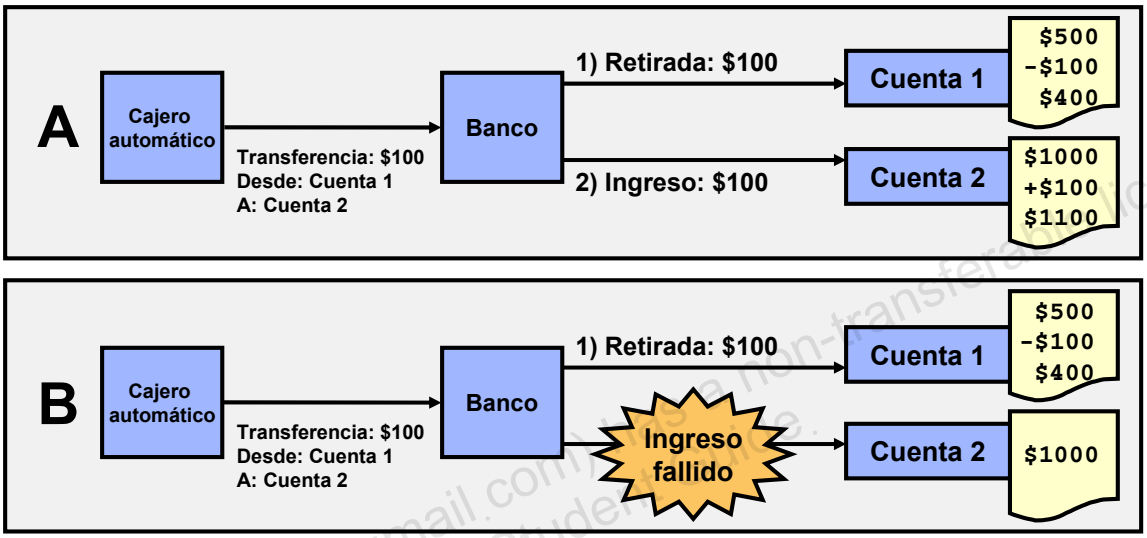


Las transacciones deben tener las siguientes propiedades ACID:

- **Atomicidad:** todo o nada, se implantan todas las operaciones implicadas en la transacción o no lo hace ninguna.
- **Consistencia:** la base de datos se debe modificar de un estado consistente a otro. En el caso de que el sistema o la base de datos falle durante la transacción, el estado original se restaura (rollback).
- **Aislamiento:** una transacción en ejecución se aísla de otras transacciones en ejecución en términos de los registros de base de datos a los que está accediendo.
- **Durabilidad:** después de que se confirme una transacción, se puede restaurar a su estado en el caso de un fallo del sistema o la base de datos.

# Transferencia sin transacciones

- Transferencia correcta (A)
- Transferencia incorrecta (las cuentas se dejan en un estado inconsistente) (B)




ORACLE



Las transacciones son adecuadas en los siguientes casos. Cada situación describe un modelo de transacción soportado por la implantación del modelo de transacción local del recurso en la instancia EntityManager.

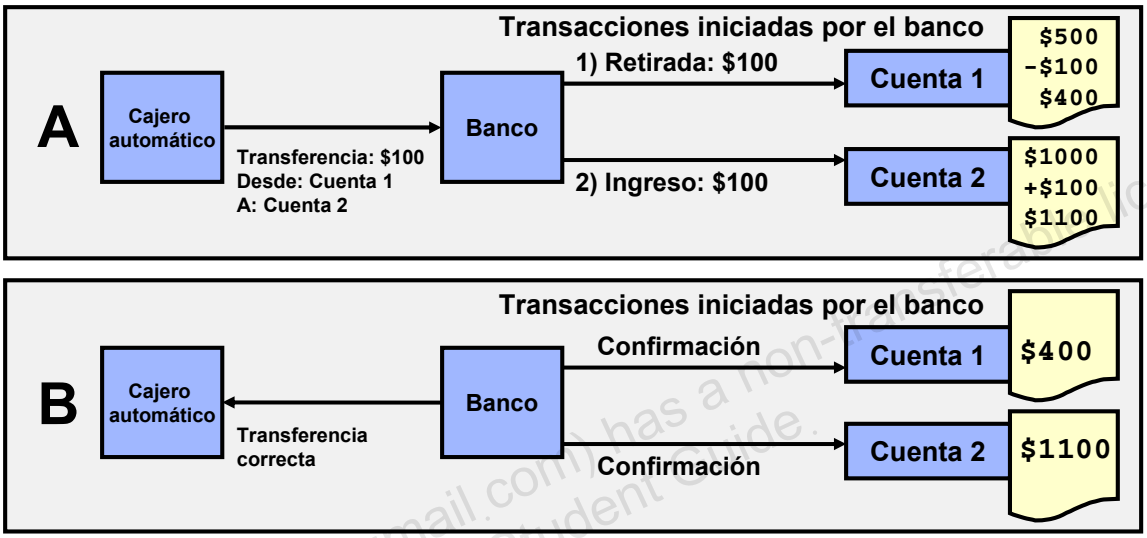
Una aplicación cliente debe conversar con un objeto gestionado y debe realizar varias llamadas en una instancia de objeto específica. La conversación se puede caracterizar por uno o más de los siguientes tipos:

A. Los datos se almacenan en caché en la memoria o se escriben en una base de datos durante o después de cada llamada sucesiva.
B. Los datos se escriben en una base de datos al final de la conversación.
C. La aplicación cliente requiere que el objeto mantenga un contexto en memoria entre cada llamada; cada llamada sucesiva utiliza los datos que se mantienen en la memoria.
D. Al final de la conversación, la aplicación cliente requiere la capacidad de cancelar todas las operaciones de escritura de la base de datos que se han producido durante o al final de la conversación.

Considere una aplicación de carro de compra. Los usuarios de la aplicación cliente examinan un catálogo en línea y crean varias selecciones de compra. Continúan a la fase de pago e introducen la información de tarjeta de crédito para realizar la compra. Si falla la verificación de la tarjeta de crédito, la aplicación de compra debe cancelar todas las selecciones de compra pendientes en el carro de la compra o realizar un rollback de las transacciones de compra realizadas durante la conversación.

# Transferencia correcta con transacciones

- Los cambios en una transacción se almacenan en buffer. (A)
- Si una transferencia es correcta, los cambios se confirman (se hacen permanentes). (B)






Si la transacción se ha realizado de forma correcta, los cambios almacenados en buffer se confirman, es decir, se hacen permanentes.

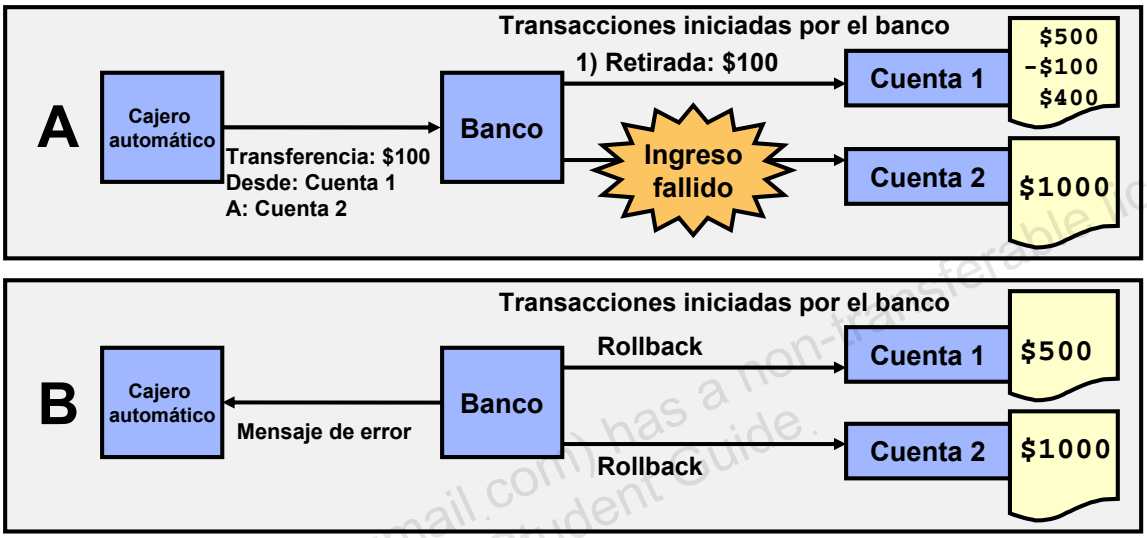
En el ámbito de una llamada de cliente en un objeto, el objeto realiza varios cambios en los datos en una base de datos. Si un cambio falla, el objeto debe realizar un rollback de todos los cambios. Piense en una aplicación bancaria. El cliente llama a la operación de transferencia en un objeto de cajero automático. La operación requiere que el objeto de cajero automático realice las siguientes llamadas en la base de datos del banco:

1. Llamada al método de débito en una cuenta
2. Llamada al método de crédito en otra cuenta

Si la llamada de crédito en la base de datos del banco falla, la aplicación bancaria debe realizar un rollback de la llamada de débito anterior.

# Transferencia incorrecta con transacciones

- Los cambios en una transacción se almacenan en buffer. (A)
- Si se produce un problema, la transacción realiza un rollback al último estado consistente. (B)






Si la transacción no se ha realizado de forma correcta, los cambios almacenados en buffer se devuelven y la base de datos realiza un rollback al último estado consistente.

# Transacciones JDBC

Por defecto, cuando se crea un objeto `Connection`, se hace en modo de confirmación automática.

- Cada sentencia SQL individual se trata como una transacción y se confirma automáticamente después de la ejecución.
- Para agrupar dos o más sentencias, debe desactivar el modo de confirmación automática.

```
con.setAutoCommit (false);
```

- Debe llamar de forma explícita el método de confirmación para completar la transacción con la base de datos.

```
con.commit();
```

- También puede realizar un rollback mediante programación de las transacciones en caso de fallo.

```
con.rollback();
```







Por defecto, JDBC confirma de forma automática todas las sentencias SQL. Sin embargo, si desea crear una operación atómica que implique varias sentencias SQL, debe desactivar la confirmación automática.

Después de desactivar la confirmación automática, no se confirma ninguna sentencia SQL en la base de datos hasta que de forma explícita llame al método de confirmación.

La otra ventaja de gestionar sus propias transacciones es la capacidad de realizar un rollback a un juego de sentencias SQL en caso de fallo mediante el método de rollback.

**Nota:** JDBC no tiene una API para comenzar de forma explícita una transacción. JSR (221) de JDBC proporciona las siguientes instrucciones:

- Cuando se desactiva una confirmación automática para un objeto `Connection`, todos los objetos `Statements` posteriores están en un contexto de transacción hasta que se ejecuta el método de rollback o de confirmación de `Connection`.
- Si se cambia el valor de confirmación automática en mitad de una transacción, la transacción actual se confirma.

Además, la documentación del controlador Derby agrega lo siguiente:

- Un contexto de transacción se asocia a un único objeto `Connection` (y base de datos). Una transacción no puede abarcar varios objetos `Connection` (o bases de datos).

**Nota:** una aplicación de ejemplo que usa transacciones es el archivo de proyecto `JDBCTransactionsExample` en el directorio de ejemplos.

## RowSet 1.1: `RowSetProvider` y `RowSetFactory`

La especificación de la API de JDK 7 presenta la nueva API de RowSet 1.1. Una de las nuevas funciones de esta API es `RowSetProvider`.

- `javax.sql.rowset.RowSetProvider` se usa para crear un objeto `RowSetFactory`:

```
myRowSetFactory = RowSetProvider.newFactory();
```

- La implantación por defecto `RowSetFactory` es:

```
com.sun.rowset.RowSetFactoryImpl
```

- `RowSetFactory` se usa para crear uno de los tipos de objeto `RowSet` de RowSet 1.1.

ORACLE







**RowSet 1.1**

Nuevas clases para JDK7: `javax.sql.rowset.RowSetProvider` y `javax.sql.rowset.RowSetFactory`. Estas dos clases se usan para crear instancias de RowSets, como se trata en la siguiente diapositiva.

# Uso de `RowSetFactory` de `RowSet 1.1`

`RowSetFactory` se usa para crear instancias de implantaciones de `RowSet`:

| Tipo de RowSet | Proporciona |
|---|---|
| `CachedRowSet` | Un contenedor para filas de datos que almacenan en caché sus filas en memoria |
| `FilteredRowSet` | Un objeto RowSet que proporciona métodos para el soporte de filtrado |
| `JdbcRowSet` | Un envoltorio alrededor de ResultSet para tratar un juego de resultados como un componente de JavaBeans |
| `JoinRowSet` | Un objeto RowSet que proporciona mecanismos para combinar datos relacionados de diferentes objetos RowSet |
| `WebRowSet` | Un objeto RowSet que soporta el formato de documento XML estándar requerido al describir un objeto RowSet en XML |

ORACLE







- **`CachedRowSet`**: el objeto `CachedRowSet` es un contenedor para filas de datos que almacena en caché sus filas en memoria. Esto hace que sea posible operar sin estar siempre conectado a su origen de datos. Además, es un componente de JavaBeans que se puede desplazar, actualizar y serializar. Un objeto `CachedRowSet` normalmente contiene filas de un juego de resultados, pero también puede contener filas de archivos con un formato tabular, como una hoja de cálculo. La implantación de referencia soporta la obtención de datos solo desde un objeto `ResultSet`, pero los desarrolladores pueden ampliar las implantaciones de `SyncProvider` para proporcionar acceso a otras fuentes de datos tabulares.
- **`FilteredRowSet`**: implantación estándar de `FilteredRowSet` de JDBC que implanta las interfaces `RowSet` y amplía la clase `CachedRowSet`. La clase `CachedRowSet` proporciona un juego de métodos de manipulación de cursor protegidos, que una implantación de `FilteredRowSet` puede sustituir para proporcionar soporte de filtrado.

- **`JdbcRowSet`**: **`JdbcRowSet`** es un envoltorio alrededor del objeto `ResultSet` que hace posible usar el juego de resultados como un componente de JavaBeans. Por tanto, un objeto `JdbcRowSet` puede ser uno de los beans que una herramienta pone a disposición para componer una aplicación. Debido a que `JdbcRowSet` es un juego de filas conectado, es decir, que mantiene continuamente su conexión a la base de datos mediante un controlador con tecnología JDBC, también hace que el controlador sea un componente de JavaBeans.
- **`JoinRowSet`**: la interfaz `JoinRowSet` proporciona un mecanismo para combinar datos relacionados de diferentes objetos `RowSet` en un objeto `JoinRowSet`, lo que representa una JOIN SQL. Es decir, un objeto `JoinRowSet` actúa como contenedor para los datos a partir de los objetos de `RowSet` que forman una relación JOIN SQL.
- **`WebRowSet`**: la interfaz `WebRowSet` describe el formato de documento XML estándar necesario al describir un objeto `RowSet` en XML y lo deben usar todas las implantaciones estándar de la interfaz `WebRowSet` para garantizar la interoperabilidad. Además, el esquema `WebRowSet` usa anotaciones de esquema SQL/XML específicas, lo que asegura una interoperabilidad entre plataformas mayor. Se trata de un esfuerzo en curso bajo los estándares de la organización ISO.

# Ejemplo: Uso de `JdbcRowSet`

```
1 try (JdbcRowSet jdbcRs =
2       RowSetProvider.newFactory().createJdbcRowSet()) {
3      jdbcRs.setUrl(url);
4      jdbcRs.setUsername(username);
5      jdbcRs.setPassword(password);
6      jdbcRs.setCommand("SELECT * FROM Employee");
7      jdbcRs.execute();
8      // Now just treat JDBC Row Set like a ResultSet object
9      while (jdbcRs.next()) {
10          int empID = jdbcRs.getInt("ID");
11          String first = jdbcRs.getString("FirstName");
12          String last = jdbcRs.getString("LastName");
13          Date birthDate = jdbcRs.getDate("BirthDate");
14           float salary = jdbcRs.getFloat("Salary");
15      }
16    //... other methods
17 }
```

ORACLE



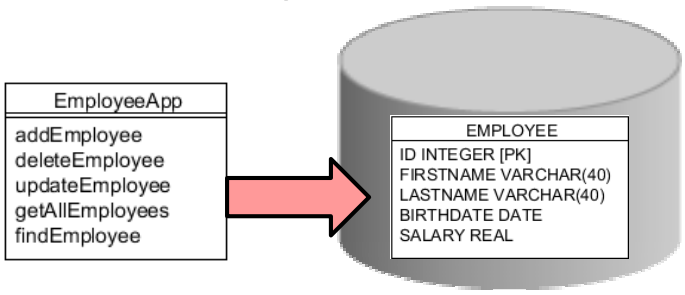
En el fragmento de código de la diapositiva, se crea una instancia `JdbcRowSet` a partir de `RowSetProviderFactory`.

A continuación, se trata el objeto como un JavaBean `RowSet`. Puede usar métodos `setter` para definir `url`, `username` y `password` y, a continuación, ejecutar un comando SQL y obtener `ResultSet` para recuperar los valores de columna.

Este ejemplo se extrae del proyecto `SimpleJDBCRowSetExample`.

# Objetos de acceso a datos

Piense en una tabla de empleado como la del código JDBC del ejemplo.




- Al combinar el código que accede a la base de datos con la lógica "negocio", los métodos de acceso a datos y la tabla Employee se acoplan.
- Los cambios en la tabla (como la adición de un campo) requerirán un cambio completo en la aplicación.
- Los datos del empleado no se encapsulan en la aplicación del ejemplo.

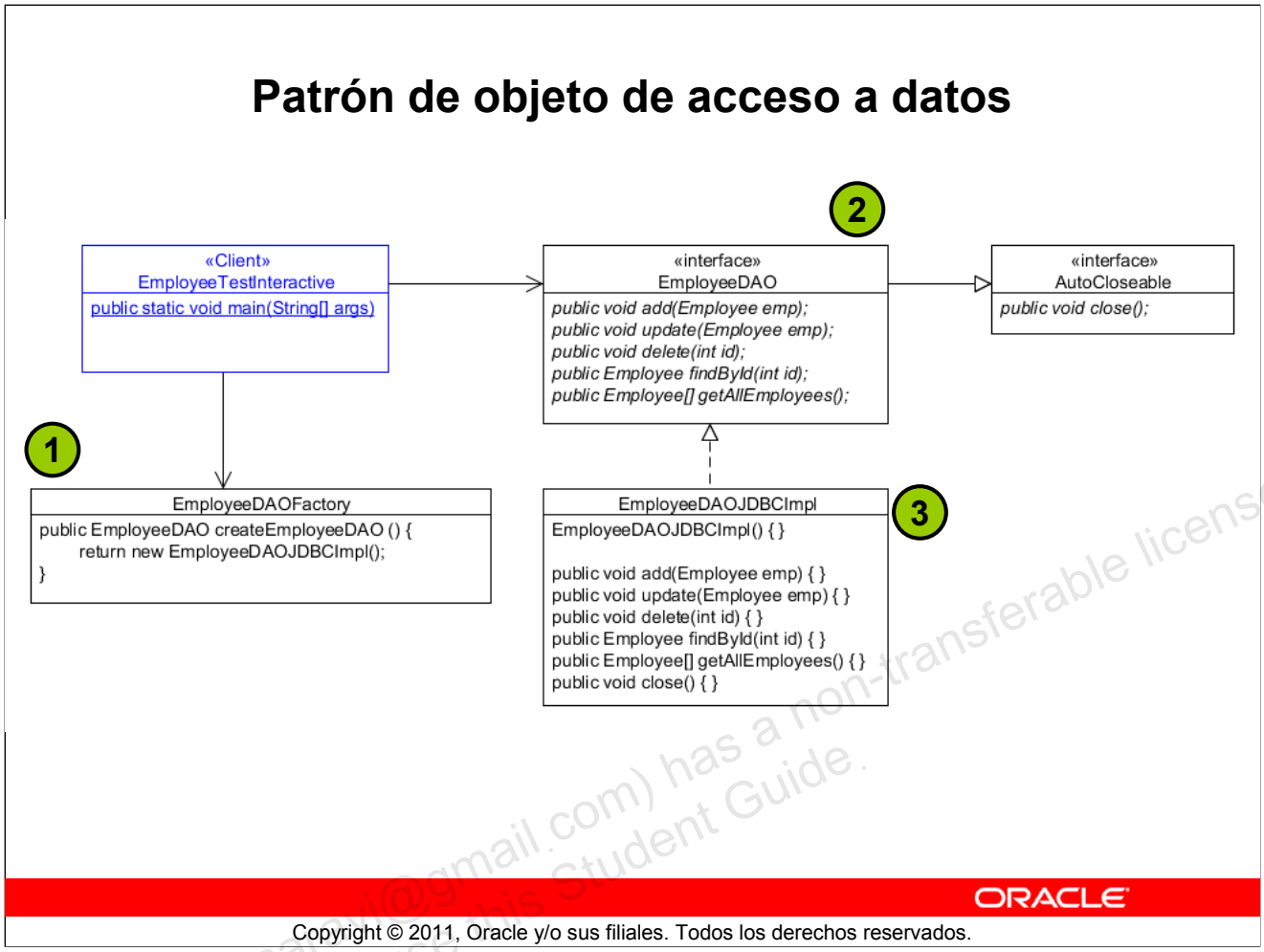
ORACLE



**Tabla Employee**

En la aplicación `SimpleJDBCExample` que se muestra en la diapositiva anterior, hay un acoplamiento estrecho entre las operaciones utilizadas para acceder a los datos y la tabla Employee en sí. El ejemplo es simple, pero si se imagina este tipo de acceso en una aplicación mayor, tal vez con varias tablas con relaciones entre ellas, comprobará que, si se accede directamente a la base de datos en la misma clase que los métodos de negocio, podría haber problemas en el futuro si cambia la tabla Employee.

Además, puesto que se accede a los datos directamente, no tiene forma de transferir la noción de un empleado. Debe tratar un empleado como un objeto.




### Patrón de fábrica y objeto de acceso a datos

El propósito de un objeto de acceso a datos (DAO) es separar actividades relacionadas con la base de datos del modelo de negocio. En este patrón de diseño, hay dos técnicas para garantizar la flexibilidad de diseño futuro.

1. Una fábrica se utiliza para generar instancias (referencias) a una implantación de la interfaz `EmployeeDAO`. Una fábrica hace que sea posible aislar el desarrollador mediante DAO a partir de los detalles sobre cómo se instancia una implantación de DAO. Como ha visto, se utiliza el mismo patrón para crear una implantación en la que se almacenan los datos en memoria.
2. Una interfaz `EmployeeDAO` está diseñada para modelar el comportamiento que desee permitir en los datos de empleado. Tenga en cuenta que esta técnica de separación de los datos demuestra una *separación de problemas*. La interfaz `EmployeeDAO` fomenta la separación adicional entre la implantación de los métodos necesarios para soportar DAO y las referencias a los objetos `EmployeeDAO`.
3. `EmployeeDAOJDBCImpl` implanta la interfaz `EmployeeDAO`. La clase de implantación se puede sustituir por una implantación diferente sin que afecte a la aplicación cliente.

# Resumen

En esta lección, debe haber aprendido a hacer lo siguiente:

- Definir el diseño de la API de JDBC
- Conectarse a una base de datos mediante un controlador JDBC
- Enviar consultas y obtener resultados de la base de datos
- Especificar información sobre el controlador JDBC de forma externa
- Usar transacciones con JDBC
- Usar `RowSetProvider` y `RowSetFactory` de JDBC 4.1
- Usar un patrón de objeto de acceso a datos para separar datos y métodos de negocio

# Prueba

¿Qué método `Statement` ejecuta una sentencia SQL y devuelve el número de filas afectadas?

a. `stmt.execute(query);`
b. `stmt.executeUpdate(query);`
c. `stmt.executeQuery(query);`
d. `stmt.query(query);`

ORACLE

# Prueba

Al usar un objeto `Statement` para ejecutar una consulta que devuelve solo un registro, no es necesario usar el método `next()` de `ResultSet`.

a. Verdadero
b. Falso

# Prueba

La siguiente sentencia `try`-with-resources cerrará adecuadamente los recursos JDBC:

```
try (Statement stmt = con.createStatement();
     ResultSet rs = stmt.executeQuery(query)){
    //...
} catch (SQLException s) {
}
```

a. Verdadero
b. Falso

# Prueba

Teniendo en cuenta:

```
10 String[] params = {"Bob", "Smith"};
11 String query = "SELECT itemCount FROM Customer " +
12                "WHERE lastName='?' AND firstName='?'";
13 try (PreparedStatement pStmt = con.prepareStatement(query)) {
14     for (int i = 0; i < params.length; i++)
15         pStmt.setObject(i, params[i]);
16     ResultSet rs = pStmt.executeQuery();
17     while (rs.next()) System.out.println (rs.getInt("itemCount"));
18 } catch (SQLException e){ }
```

Suponiendo que hay un objeto Connection válido y que la consulta SQL producirá al menos una fila, ¿cuál es el resultado?

a. Cada valor `itemCount` para el cliente Bob Smith
b. Error del compilador
c. Error de tiempo de ejecución
d. `SQLException`

ORACLE

# Visión general de la práctica 14-2: Uso del patrón de objeto de acceso a datos

En esta práctica, se abordan los siguientes temas:

- Refactorización de la aplicación DAO basado en memoria para usar JDBC.
- Uso de la aplicación cliente Employee interactiva, prueba del código.

ORACLE



En esta práctica, refactorizará el DAO basado en memoria desde excepciones y afirmaciones para usar JDBC en su lugar. Se proporciona un cliente interactivo para que pueda experimentar con la creación, lectura, actualización y supresión de registros mediante JDBC.

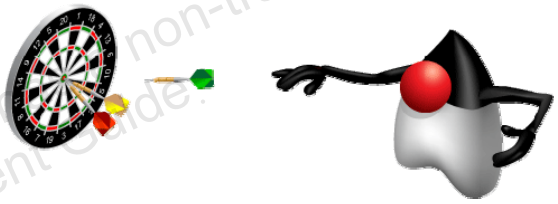
# 15

# Localización

# Objetivos

Al finalizar esta lección, debería estar capacitado para:

- Describir las ventajas de localizar una aplicación
- Definir lo que representa una configuración regional
- Leer y definir la configuración regional mediante el objeto `Locale`
- Crear un grupo de recursos para cada configuración regional
- Llamar a un grupo de recursos desde una aplicación
- Cambiar la configuración regional para un grupo de recursos
- Aplicar formato a texto para la localización mediante `NumberFormat` y `DateFormat`



ORACLE

# ¿Por qué localizar?

La decisión de crear una versión de una aplicación para uso internacional se suele presentar al inicio de un proyecto de desarrollo.

- Software que tiene en cuenta el idioma y la región
- Fechas, números y monedas con formato para países específicos
- Capacidad para conectarse a datos específicos del país sin cambiar el código

ORACLE



La localización es el proceso de adaptación de software para una región o idioma específico mediante la adición de componentes específicos de la configuración regional y la traducción del texto.

Además de los cambios de idioma, los componentes culturales, como fechas, números, monedas, etc. se deben traducir.

El objetivo es diseñar la localización de modo que no se requieren cambios de codificación.

# Aplicación de ejemplo

Localizar una aplicación de ejemplo:

- Interfaz de usuario basada en texto
- Localización de menús
- Muestra de localizaciones de moneda y fecha

```
=== Localization App ===
1. Set to English
2. Set to French
3. Set to Chinese
4. Set to Russian
5. Show me the date
6. Show me the money!
q. Enter q to quit
Enter a command:
```

ORACLE



En el resto de esta lección, esta sencilla interfaz de usuario basada en texto se localizará en francés, chino simplificado y ruso. Introduzca el número que indica el menú y dicha opción de menú se aplicará a la aplicación. Introduzca `q` para salir de la aplicación.

# Locale

`Locale` especifica un idioma y país determinado:

- Idioma
  - Código ISO 639: Alfa-2 o Alfa-3
  - “en” para inglés, “es” para español
  - Siempre utiliza minúscula
- País
  - Utiliza el código de país ISO 3166: Alfa-2 o el código de área numérico UN M.49
  - "US" para Estados Unidos, "ES" para España
  - Siempre utiliza mayúscula
- Consulte los tutoriales de Java para obtener más información sobre todos los estándares utilizados







En Java, se especifica una configuración regional mediante dos valores: idioma y país. Consulte el tutorial de Java para ver los estándares utilizados:
`http://download.oracle.com/javase/tutorial/i18n/locale/create.html`

Ejemplos de idiomas

- de: alemán
- en: inglés
- fr: francés
- zh: chino

Ejemplos de país

- DE: Alemania
- US: Estados Unidos
- FR: Francia
- CN: China

# Grupo de recursos

- La clase `ResourceBundle` aísla los datos específicos de la configuración regional:
  - Devuelve pares clave/valor almacenados de forma independiente.
  - Puede ser una clase o un archivo `.properties`.
- Pasos que usar:
  - Crear archivos de grupo para cada configuración regional.
  - Llamar a una configuración regional específica desde la aplicación.



El diseño para la localización comienza con el diseño de la aplicación, de modo que todo el texto, los sonidos y las imágenes se pueden sustituir en tiempo de ejecución por los elementos adecuados para la región y la cultura deseada. Los grupos de recursos contienen pares clave/valor que se pueden codificar en una clase o localizar en un archivo `.properties`.

# Archivo de grupo de recursos

- El archivo de propiedades contiene un juego de pares clave/valor.
  - Cada clave identifica un componente de aplicación específico.
  - Los nombres de archivo especiales utilizan códigos de idioma y país.
- Valor por defecto para la aplicación de ejemplo:
  - Menú convertido en grupo de recursos

**MessageBundle.properties**

```
menu1 = Set to English
menu2 = Set to French
menu3 = Set to Chinese
menu4 = Set to Russian
menu5 = Show the Date
menu6 = Show me the money!
menuq = Enter q to quit
```

ORACLE



La diapositiva muestra un archivo del grupo de recursos de ejemplo para esta aplicación. Cada opción de menú se ha convertido en un par nombre/valor. Se trata del archivo por defecto para la aplicación. Para idiomas alternativos, se utiliza una convención de nomenclatura especial:

```
MessageBundle_xx_YY.properties
```

donde `xx` es el código de idioma e `YY` es el código de país.

# Archivos del grupo de recursos de ejemplo

## Ejemplos para francés y chino

**MessagesBundle_fr_FR.properties**

```
menu1 = Régler à l'anglais
menu2 = Régler au français
menu3 = Réglez chinoise
menu4 = Définir pour la Russie
menu5 = Afficher la date
menu6 = Montrez-moi l'argent!
menuq = Saisissez q pour quitter
```

**MessagesBundle_zh_CN.properties**

```
menu1 = 设置为英语
menu2 = 设置为法语
menu3 = 设置为中文
menu4 = 设置到俄罗斯
menu5 = 显示日期
menu6 = 显示我的钱!
menuq = 输入q退出
```

ORACLE



La diapositiva muestra los archivos del grupo de recursos para francés y chino. Tenga en cuenta que los nombres de archivo incluyen idioma y país. La opción de menú inglés se ha sustituido por las alternativas francés y chino.

# Prueba

¿Qué archivo del grupo representa el idioma español y el código de país de Estados Unidos?

a. `MessagesBundle_ES_US.properties`
b. `MessagesBundle_es_es.properties`
c. `MessagesBundle_es_US.properties`
d. `MessagesBundle_ES_us.properties`

# Inicialización de la aplicación de ejemplo

```
PrintWriter pw = new PrintWriter(System.out, true);
    // More init code here

    Locale usLocale = Locale.US;
    Locale frLocale = Locale.FRANCE;
    Locale zhLocale = new Locale("zh", "CN");
    Locale ruLocale = new Locale("ru", "RU");
    Locale currentLocale = Locale.getDefault();

    ResourceBundle messages = ResourceBundle.getBundle("MessagesBundle",
    currentLocale);

    // more init code here

    public static void main(String[] args){
        SampleApp ui = new SampleApp();
        ui.run();
    }
```



Con los grupos de recursos creados, simplemente necesita cargar los grupos en la aplicación. El código de origen en la diapositiva muestra los pasos. En primer lugar, cree un objeto `Locale` que especifique el idioma y el país. A continuación, cargue el grupo de recursos especificando el nombre del archivo base para el grupo y el `Locale` actual.

Tenga en cuenta que hay dos formas de definir un objeto `Locale`. La clase `Locale` incluye constantes por defecto para algunos países. Si no está disponible una constante, puede usar el código de idioma con el código de país para definir la ubicación. Finalmente, puede usar el método `getDefault()` para obtener la ubicación por defecto.

# Aplicación de ejemplo: bucle principal

```
public void run(){
    String line = "";
    while (!(line.equals("q"))){
        this.printMenu();
        try { line = this.br.readLine(); }
        catch (Exception e){ e.printStackTrace(); }

        switch (line){
            case "1": setEnglish(); break;
            case "2": setFrench(); break;
            case "3": setChinese(); break;
            case "4": setRussian(); break;
            case "5": showDate(); break;
            case "6": showMoney(); break;
        }
    }
}
```

ORACLE



Para esta aplicación, un método de ejecución contiene el bucle principal. El bucle se ejecuta hasta que se introduce la letra "q" como entrada. Una conmutación de cadena se utiliza para realizar una operación según el número introducido. Se realiza una simple llamada a cada método para realizar los cambios de configuración regional y mostrar la salida con formato.

# Método `printMenu`

En lugar de texto, se utiliza el grupo de recursos.

- `messages` es un grupo de recursos.
- Se utiliza una clave para recuperar cada opción de menú.
- El idioma se selecciona según la definición de `Locale`.

```
public void printMenu(){
    pw.println("=== Localization App ===");
    pw.println("1. " + messages.getString("menu1"));
    pw.println("2. " + messages.getString("menu2"));
    pw.println("3. " + messages.getString("menu3"));
    pw.println("4. " + messages.getString("menu4"));
    pw.println("5. " + messages.getString("menu5"));
    pw.println("6. " + messages.getString("menu6"));
    pw.println("q. " + messages.getString("menuq"));
    System.out.print(messages.getString("menucommand")+" ");
 }
```

ORACLE



En lugar de imprimir el texto, se llama al grupo de recursos (`messages`) y el objeto `Locale` actual determina el idioma presente para el usuario.

# Cambio de `Locale`

Para cambiar `Locale`:

- Defina `currentLocale` en el idioma deseado.
- Vuelva a cargar el grupo mediante la configuración regional actual.

```
public void setFrench(){
    currentLocale = frLocale;
    messages = ResourceBundle.getBundle("MessagesBundle",
    currentLocale);
}
```

ORACLE



Una vez que se actualiza el grupo del menú con la configuración regional correcta, el texto de la interfaz se agrega con el idioma seleccionado.

# Interfaz de ejemplo con francés

Después de seleccionar la opción de francés, la interfaz de usuario actualizada es parecida a la siguiente:

```
=== Localization App ===
1. Régler à l'anglais
2. Régler au français
3. Réglez chinoise
4. Définir pour la Russie
5. Afficher la date
6. Montrez-moi l'argent!
q. Saisissez q pour quitter
Entrez une commande:
```



La interfaz de usuario actualizada se muestra en la diapositiva. La primera y la última línea de la aplicación también se podrían localizar.

# Formato de fecha y moneda

- Los números se pueden localizar y mostrar en su formato local.
- Las clases de formato especial incluyen:
  - `DateFormat`
  - `NumberFormat`
- Cree objetos mediante `Locale`.



El cambio de texto no es la única herramienta de localización disponible. Las fechas y números también se pueden formatear según los estándares locales.

# Inicialización de fecha y moneda

La aplicación puede mostrar la moneda y la fecha con formato local. Las variables se inicializan de la siguiente forma:

```
// More init code precedes
NumberFormat currency;
Double money = new Double(1000000.00);

Date today = new Date();
DateFormat df;
```



Antes de realizar ninguna acción de formato, se deben configurar los objetos de fecha y número. Tanto la fecha actual como un objeto Double se utilizan en esta aplicación.

# Visualización de fecha

- Aplicación de formato a una fecha:
  - Obtenga un objeto `DateFormat` basado en el objeto `Locale`.
  - Llame al método `format` transfiriendo la fecha al formato.

```
public void showDate(){

    df = DateFormat.getDateInstance(DateFormat.DEFAULT, currentLocale);
    pw.println(df.format(today) + " " + currentLocale.toString());
}
```

- Fechas de ejemplo:

```
20 juil. 2011 fr_FR
20.07.2011 ru_RU
```



Cree un objeto con formato de fecha mediante la configuración regional y la fecha se formateará para la configuración regional.

# Personalización de fechas

- Las constantes `DateFormat` incluyen:
  - SHORT: es completamente numérica, como 12.13.52 o 3:30 p.m.
  - MEDIUM: es más larga, como 12 ene, 1952
  - LONG: es más larga, como 12 de enero,1952 o 3:30:32 p.m.
  - FULL: se especifica completamente, como martes, 12 de abril, 1952 DC o 3:30:42 p.m. PST
- `SimpleDateFormat`:
  - Una subclase de una clase `DateFormat`

| Letra | Fecha u hora | Presentación | Ejemplos |
|---|---|---|---|
| G | Era | Texto | AD |
| y | Año | Año | 1996; 96 |
| M | Mes en año | Mes | Julio; Jul; 07 |

ORACLE



El objeto `DateFormat` incluye un número de constantes que puede usar para aplicar formato a la salida de fecha.

La clase `SimpleDateFormat` es una subclase de `DateFormat` y permite un mayor control de la salida de fecha. Consulte la documentación para obtener las opciones disponibles.

En algunos casos, el número de letras pueden determinar la salida. Por ejemplo, con el mes:

```
MM 07
MMM Jul
MMMM July
```

# Visualización de moneda

- Aplicación de formato a la moneda:
  - Obtenga una instancia de moneda de `NumberFormat`.
  - Transfiera `Double` al método `format`.

```
public void showMoney(){
   currency = NumberFormat.getCurrencyInstance(currentLocale);
   pw.println(currency.format(money) + " " + currentLocale.toString());
}
```

- Salida de moneda de ejemplo:

```
1 000 000 руб. ru_RU
1 000 000,00   fr_FR
￥1,000,000.00 zh_CN
```



Cree un objeto `NumberFormat` mediante la configuración regional seleccionada y obtenga la salida con formato.

# Prueba

¿Qué constante de formato de fecha proporciona la información más detallada?

a. `LONG`
b. `FULL`
c. `MAX`
d. `COMPLETE`

# Resumen

En esta lección, debe haber aprendido a hacer lo siguiente:

- Describir las ventajas de localizar una aplicación
- Definir lo que representa una configuración regional
- Leer y definir la configuración regional mediante el objeto `Locale`
- Crear un grupo de recursos para cada configuración regional
- Llamar a un grupo de recursos desde una aplicación
- Cambiar la configuración regional para un grupo de recursos
- Aplicar formato a texto para la localización mediante `NumberFormat` y `DateFormat`

ORACLE

# Visión general de la práctica 15-1: Creación de una aplicación de fecha localizada

Esta práctica trata la creación de una aplicación localizada que muestra las fecha en diferentes formatos.

ORACLE

# (Opcional) Visión general de la práctica 15-2: Localización de una aplicación JDBC

Esta práctica trata la creación de una versión localizada de una aplicación JDBC de la lección anterior.

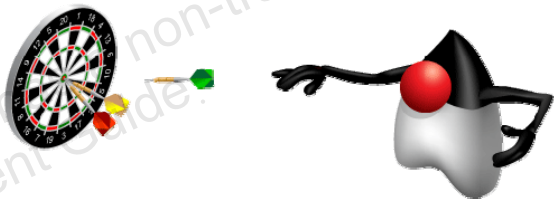
ORACLE

# Descripción general de SQL

# Objetivos

Al finalizar esta lección, debería estar capacitado para:

- Describir la sintaxis de comandos SQL-92/1999 básicos, incluidos:
  - SELECT
  - INSERT
  - UPDATE
  - DELETE
  - CREATE TABLE
  - DROP TABLE
- Definir tipos de datos SQL-92/1999

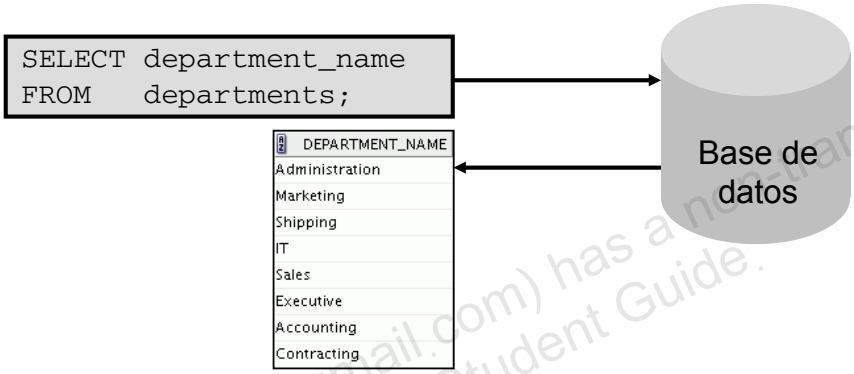


ORACLE

# Uso de SQL para consultar la base de datos

El lenguaje de consulta estructurado (SQL) es:

- Lenguaje estándar de ANSI para el funcionamiento de bases de datos relacionales
- Uso y aprendizaje sencillos y eficaces
- Funcionalidad completa (con SQL, puede definir, recuperar y manipular datos en las tablas)




ORACLE



En una base de datos relacional, no es necesario especificar la ruta de acceso a las tablas y tampoco es necesario saber cómo se organizan los datos de forma física.

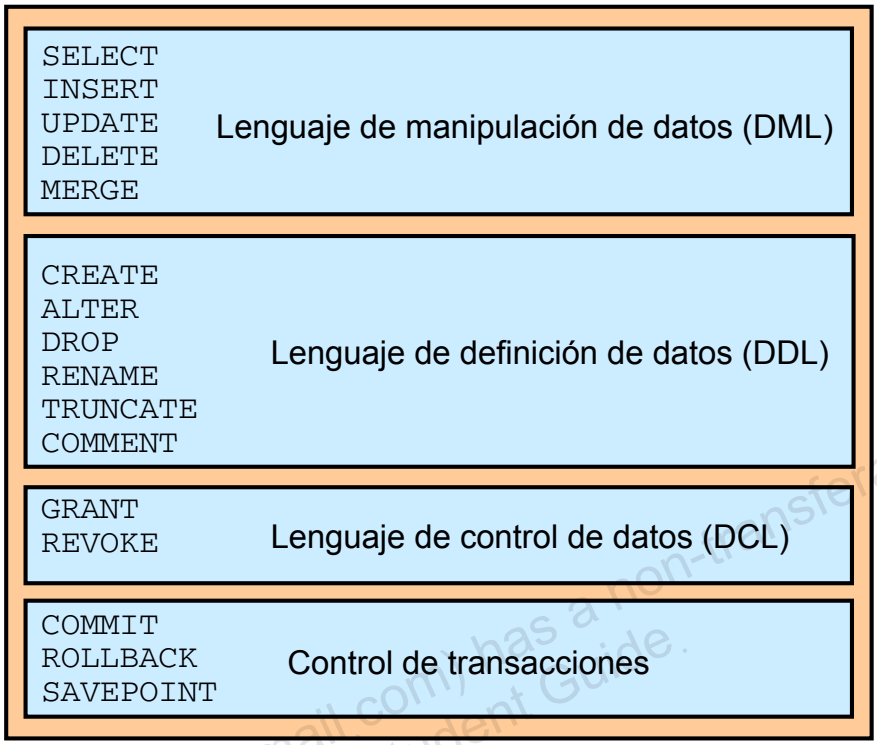
Para acceder a la base de datos, ejecute una sentencia de lenguaje de consulta estructurado (SQL), que es el lenguaje estándar de ANSI (American National Standards Institute) para el funcionamiento de bases de datos relacionales. SQL es un juego de sentencias con el que todos los programas y usuarios acceden a los datos de Oracle Database. Los programas y las herramientas de Oracle a menudo permiten el acceso de usuarios a la base de datos sin utilizar directamente SQL, pero estas aplicaciones a su vez deben utilizar SQL al ejecutar la solicitud del usuario.

SQL proporciona sentencias para distintas tareas, que incluyen las siguientes:

- Consulta de datos
- Inserción, actualización y supresión de filas en una tabla
- Creación, sustitución, modificación y borrado de objetos
- Control de acceso a la base de datos y sus objetos
- Garantía de integridad y consistencia de la base de datos

SQL unifica todas las tareas anteriores en un lenguaje consistente y permite trabajar con datos en el nivel lógico.

## Sentencias SQL




ORACLE





Sentencias SQL soportadas por el cumplimento de Oracle de los estándares de la industria Oracle Corporation asegura el cumplimiento futuro con la evolución de los estándares mediante la implicación de forma activa del personal clave en los comités de estándares SQL. Los comités aceptados por la industria son ANSI e ISO (International Standards Organization). Tanto ANSI como ISO han aceptado SQL como el lenguaje estándar de las bases de datos relacionales.

| Statement | Description |
|---|---|
| SELECT<br>INSERT<br>UPDATE<br>DELETE<br>MERGE | Retrieves data from the database, enters new rows, changes existing rows, and removes unwanted rows from tables in the database, respectively. Collectively known as *data manipulation language* (DML) |
| CREATE<br>ALTER<br>DROP<br>RENAME<br>TRUNCATE<br>COMMENT | Sets up, changes, and removes data structures from tables. Collectively known as *data definition language* (DDL) |
| GRANT<br>REVOKE | Provides or removes access rights to both the Oracle Database and the structures within it |
| COMMIT<br>ROLLBACK<br>SAVEPOINT | Manages the changes made by DML statements. Changes to the data can be grouped together into logical transactions |

# Sentencia SELECT básica

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM    table;
```

- SELECT identifica las columnas que se van a mostrar.
- FROM identifica la tabla que contiene estas columnas.





En su formato más simple, una sentencia SELECT debe incluir lo siguiente:

- Una cláusula SELECT, que especifica las columnas que se van a mostrar.
- Una cláusula FROM, que identifica la tabla que contiene las columnas que se muestran en la cláusula SELECT.

En la sintaxis:

| | |
|---|---|
| SELECT | es una lista de una o más columnas. |
| * | selecciona todas las columnas. |
| DISTINCT | suprime los duplicados. |
| *column\|expression* | selecciona la columna o expresión especificada. |
| *alias* | proporciona diferentes cabeceras de las columnas seleccionadas. |
| FROM *table* | especifica la tabla que contiene las columnas. |

**Nota:** a lo largo de este curso, las palabras *palabra clave*, *cláusula* y *sentencia* se utilizan como se describe a continuación:

- Una *palabra clave* hace referencia a un elemento SQL individual (por ejemplo, `SELECT` y `FROM` son palabras clave).
- Una *cláusula* es parte de una sentencia SQL (por ejemplo, `SELECT employee_id`, `last_name`, etc.).
- Una *sentencia* es una combinación de dos o más cláusulas, por ejemplo, `SELECT * FROM employees`.

# Limitación de las filas seleccionadas

- Restringir las filas devueltas al utilizar la cláusula `WHERE`:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM   table
[WHERE condition(s)];
```

- La cláusula `WHERE` sigue a la cláusula `FROM`.





Puede restringir las filas que devuelve la consulta al utilizar la cláusula `WHERE`. Una cláusula `WHERE` contiene una condición que se debe cumplir e, inmediatamente después, le sigue la cláusula `FROM`. Si la condición es verdadera, se devolverá la fila que cumpla con la condición.

En la sintaxis:

| | |
|---|---|
| `WHERE` | restringe la consulta a filas que cumplan con una condición. |
| *`condition`* | está compuesto por nombres de columna, expresiones, constantes y un operador de comparación. Una condición especifica una combinación de una o más expresiones y operadores lógicos (booleanos) y devuelve un valor de `TRUE`, `FALSE` o `UNKNOWN`. |

La cláusula `WHERE` puede comparar valores en columnas, literales, expresiones aritméticas o funciones. Consta de tres elementos:

- Nombre de la columna
- Condición de comparación
- Nombre de la columna, constante o lista de valores

# Uso de la cláusula ORDER BY

- Ordenar las filas recuperadas con la cláusula ORDER BY:
  - ASC: orden ascendente, valor por defecto
  - DESC: orden descendente
- La cláusula ORDER BY es la última en una sentencia SELECT:

```
SELECT   last_name, job_id, department_id, hire_date
FROM employees
ORDER BY hire_date ;
```

| | LAST_NAME | JOB_ID | DEPARTMENT_ID | HIRE_DATE |
|---|---|---|---|---|
| 1 | King | AD_PRES | 90 | 17-JUN-87 |
| 2 | Whalen | AD_ASST | 10 | 17-SEP-87 |
| 3 | Kochhar | AD_VP | 90 | 21-SEP-89 |
| 4 | Hunold | IT_PROG | 60 | 03-JAN-90 |
| 5 | Ernst | IT_PROG | 60 | 21-MAY-91 |
| 6 | De Haan | AD_VP | 90 | 13-JAN-93 |

...

ORACLE



El orden de las filas devueltas en un resultado de consulta no está definido. La cláusula ORDER BY se puede utilizar para ordenar las filas. Sin embargo, si utiliza la cláusula ORDER BY, debe ser la última cláusula de la sentencia SQL. Además, puede especificar una expresión, un alias o una posición de columna como la condición de ordenación.

**Sintaxis**

```
SELECT          expr
FROM            table
[WHERE          condition(s)]
[ORDER BY  {column, expr, numeric_position} [ASC|DESC]];
```

En la sintaxis:

| | |
|---|---|
| ORDER BY | especifica el orden en el que aparecen las filas recuperadas. |
| ASC | ordena las filas en orden ascendente (orden por defecto). |
| DESC | ordena las filas en orden descendente. |

Si la cláusula ORDER BY no se utiliza, el orden no está definido y puede que el servidor de Oracle no recupere dos veces las filas en el mismo orden para la misma consulta. Utilice la cláusula ORDER BY para mostrar las filas en un orden específico.

**Nota:** utilice las palabras clave NULLS FIRST o NULLS LAST para especificar si las filas devueltas que contengan valores nulos deben aparecer en primer o en último lugar en la secuencia de ordenación.

# Sintaxis de las sentencias `INSERT`

- Agregar nuevas filas a una tabla mediante la sentencia `INSERT`:

```
INSERT INTO   table [(column [, column...])]
VALUES        (value [, value...]);
```

- Con esta sintaxis, solo se inserta una fila cada vez.

ORACLE







Puede agregar nuevas filas a una tabla emitiendo la sentencia `INSERT`.

En la sintaxis:

| | |
|---|---|
| *table* | es el nombre de la tabla. |
| *column* | es el nombre de la columna de la tabla que se debe rellenar. |
| *value* | es el valor correspondiente para la columna. |

**Nota:** esta sentencia con la cláusula `VALUES` agrega solo una fila cada vez a la tabla.

# Sintaxis de sentencias UPDATE

- Modificar los valores existentes en una tabla con la sentencia UPDATE:

```
UPDATE      table
SET         column = value [, column = value, ...]
[WHERE      condition];
```

- Actualizar más de una fila cada vez (si es necesario).



Puede modificar los valores existentes en una tabla mediante la sentencia UPDATE.

En la sintaxis:

| | |
|---|---|
| *table* | es el nombre de la tabla. |
| *column* | es el nombre de la columna de la tabla que se debe rellenar. |
| *value* | es el valor o subconsulta correspondiente para la columna. |
| *condition* | identifica las filas que se deben actualizar y se compone de nombres de columna, expresiones, constantes, subconsultas y operadores de comparación. |

Para confirmar la operación de actualización, consulte la tabla para visualizar las filas actualizadas.

**Nota:** en general, la primera columna de clave primaria se utiliza en la cláusula WHERE para identificar una única fila para la actualización. El uso de otras columnas puede provocar una actualización inesperada de varias filas. Por ejemplo, identificar una fila de la tabla EMPLOYEES por nombre es peligroso, ya que puede que más de un empleado tengan el mismo nombre.

## Sentencia DELETE

Puede eliminar filas existentes de una tabla mediante la sentencia DELETE:

```
DELETE [FROM]   table
[WHERE          condition];
```



### Sintaxis de sentencias DELETE

Puede eliminar filas existentes de una tabla mediante la sentencia DELETE.

En la sintaxis:

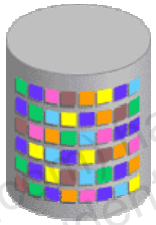
| | |
|---|---|
| *table* | es el nombre de la tabla. |
| *condition* | identifica las filas que se deben suprimir y se compone de nombres de columna, expresiones, constantes, subconsultas y operadores de comparación. |

# Sentencia CREATE TABLE

- Debe tener:
  - El privilegio `CREATE TABLE`
  - Un área de almacenamiento

```
CREATE TABLE [schema.]table
        (column datatype [DEFAULT expr][, ...]);
```

- Debe especificar:
  - El nombre de tabla
  - El nombre de columna, tipo de dato de columna y tamaño de columna



ORACLE







Puede crear tablas para almacenar datos ejecutando la sentencia SQL `CREATE TABLE`. Esta sentencia es una de las sentencias DDL, que son un subjuego de sentencias SQL que se utilizan para crear, modificar o eliminar estructuras de Oracle Database. Estas sentencias tienen un efecto inmediato en la base de datos y registran información en el diccionario de datos.

Para crear una tabla, un usuario debe tener el privilegio `CREATE TABLE` y un área de almacenamiento en la que crear los objetos. El administrador de la base de datos (DBA) utiliza sentencias de lenguaje de control de datos (DCL) para otorgar privilegios a los usuarios.

En la sintaxis:

| | |
|---|---|
| *schema* | es el mismo nombre que el del propietario. |
| *table* | es el nombre de la tabla. |
| DEFAULT *expr* | especifica un valor por defecto si se omite un valor en la sentencia INSERT. |
| *column* | es el nombre de la columna. |
| *datatype* | es el tipo de dato y la longitud de la columna. |

# Definición de restricciones

- Sintaxis:

```
CREATE TABLE [schema.]table
      (column datatype [DEFAULT expr]
      [column_constraint],
      ...
      [table_constraint][,...]);
```

- Sintaxis de restricción a nivel de columna:

```
column [CONSTRAINT constraint_name] constraint_type,
```

- Sintaxis de restricción a nivel de tabla:

```
column,...
  [CONSTRAINT constraint_name] constraint_type
  (column, ...),
```





En la diapositiva se proporciona la sintaxis para la definición de restricciones al crear una tabla. Puede crear las restricciones a nivel de columna o de tabla. Las restricciones definidas a nivel de columna se incluyen al definir la columna. Las restricciones a nivel de tabla se definen al final de la definición de tabla y deben hacer referencia a la columna o las columnas a las que pertenece la restricción en un juego de paréntesis. Se trata principalmente de la sintaxis que diferencia a las dos; de lo contrario, funcionalmente, una restricción a nivel de columna es lo mismo que una restricción a nivel de tabla.

Las restricciones `NOT NULL` se deben definir a nivel de columna.

Las restricciones que se aplican a más de una columna se deben definir a nivel de tabla.

En la sintaxis:

| | |
|---|---|
| `schema` | es el mismo nombre que el del propietario. |
| `table` | es el nombre de la tabla. |
| `DEFAULT expr` | especifica un valor por defecto para utilizarlo si se omite un valor en la sentencia `INSERT`. |
| `column` | es el nombre de la columna. |
| `datatype` | es el tipo de dato y la longitud de la columna. |
| `column_constraint` | es una restricción de integridad como parte de la definición de columna. |
| `table_constraint` | es una restricción de integridad como parte de la definición de tabla. |

# Definición de restricciones

- Ejemplo de una restricción a nivel de columna:

```
CREATE TABLE employees(
  employee_id  NUMBER(6)
    CONSTRAINT emp_emp_id_pk PRIMARY KEY,
  first_name   VARCHAR2(20),
  ...);
```

1

- Ejemplo de una restricción a nivel de tabla:

```
CREATE TABLE employees(
  employee_id  NUMBER(6),
  first_name   VARCHAR2(20),
  ...
  job_id       VARCHAR2(10) NOT NULL,
  CONSTRAINT emp_emp_id_pk
    PRIMARY KEY (EMPLOYEE_ID));
```

2

ORACLE







Las restricciones se crean normalmente al mismo tiempo que la tabla. Las restricciones se pueden agregar a una tabla después de su creación y se pueden desactivar temporalmente.

Ambos ejemplos de la diapositiva crean una restricción de clave primaria en la columna `EMPLOYEE_ID` de la tabla `EMPLOYEES`.

1. En el primer ejemplo se utiliza la sintaxis de nivel de columna para definir la restricción.
2. En el segundo ejemplo se utiliza la sintaxis de nivel de tabla para definir la restricción.

Encontrará más información sobre la restricción de clave primaria más adelante en esta lección.

# Inclusión de restricciones

- Las restricciones aplican reglas a nivel de tabla.
- Las restricciones impiden la supresión de una tabla si hay dependencias.
- Los siguientes tipos de restricciones son válidos:
  - `NOT NULL`
  - `UNIQUE`
  - `PRIMARY KEY`
  - `FOREIGN KEY`
  - `CHECK`





### Restricciones

El servidor de Oracle utiliza restricciones para evitar la introducción de datos no válidos en las tablas.

Puede utilizar restricciones para realizar lo siguiente:

- Aplicar reglas a los datos de la tabla cuando se inserta, actualiza o suprime una fila de la misma. La restricción se debe cumplir para que la operación sea correcta.
- Evitar la supresión de una tabla si hay dependencias de otras tablas.
- Proporcionar reglas para las herramientas de Oracle, como Oracle Developer.

**Restricciones de integridad de datos**

| Constraint | Description |
|---|---|
| NOT NULL | Specifies that the column cannot contain a null value |
| UNIQUE | Specifies a column or combination of columns whose values must be unique for all rows in the table |
| PRIMARY KEY | Uniquely identifies each row of the table |
| FOREIGN KEY | Establishes and enforces a referential integrity between the column and a column of the referenced table such that values in one table match values in another table. |
| CHECK | Specifies a condition that must be true |

# Tipos de datos

| Tipo de dato | Descripción |
|---|---|
| VARCHAR2(*size*) | Datos de caracteres de longitud variable |
| CHAR(*size*) | Datos de caracteres de longitud fija |
| NUMBER(*p*,*s*) | Datos numéricos de longitud variable |
| DATE | Valores de fecha y hora |
| LONG | Datos de caracteres de longitud variable (hasta 2 GB) |
| CLOB | Datos binarios (hasta 4 GB) |
| RAW and LONG RAW | Datos binarios no procesados |
| BLOB | Datos binarios (hasta 4 GB) |
| BFILE | Datos binarios almacenados en un archivo externo (hasta 4 GB) |
| ROWID | Sistema numérico de base -64 que representa la dirección única de una fila en su tabla correspondiente |

ORACLE







## Tipos de datos

Al identificar una columna para una tabla, debe proporcionar un tipo de dato para la columna. Hay varios tipos de datos disponibles:

| Data Type | Description |
|---|---|
| VARCHAR2(*size*) | Variable-length character data (A maximum *size* must be specified: minimum *size* is 1; maximum *size* is 4,000.) |
| CHAR [(*size*)] | Fixed-length character data of length *size* bytes (Default and minimum *size* is 1; maximum *size* is 2,000.) |
| NUMBER [(*p*,*s*)] | Number having precision *p* and scale *s* (Precision is the total number of decimal digits and scale is the number of digits to the right of the decimal point; precision can range from 1 to 38, and scale can range from –84 to 127.) |
| DATE | Date and time values to the nearest second between January 1, 4712 B.C., and December 31, 9999 A.D. |
| LONG | Variable-length character data (up to 2 GB) |
| CLOB | Character data (up to 4 GB) |

| Data Type | Description |
|---|---|
| RAW(*size*) | Raw binary data of length *size* (A maximum *size* must be specified: maximum *size* is 2,000.) |
| LONG RAW | Raw binary data of variable length (up to 2 GB) |
| BLOB | Binary data (up to 4 GB) |
| BFILE | Binary data stored in an external file (up to 4 GB) |
| ROWID | A base-64 number system representing the unique address of a row in its table |

**Directrices**

- Las columnas LONG no se copian al crear una tabla mediante una subconsulta.
- Las columnas LONG no se pueden incluir en una cláusula GROUP BYu ORDER BY.
- Solo se puede utilizar una columna LONG por tabla.
- No se pueden definir restricciones en las columnas LONG.
- Puede que desee utilizar una columna CLOB en lugar de una columna LONG.

# Borrado de una tabla

- Mueve una tabla a la papelera de reciclaje.
- Elimina la tabla y todos sus datos completamente si se especifica la cláusula `PURGE`.
- Invalida objetos dependientes y elimina privilegios de objeto en la tabla.

```
DROP TABLE dept80;
DROP TABLE dept80 succeeded.
```







La sentencia `DROP TABLE` mueve una tabla a la papelera de reciclaje o elimina la tabla y todos sus datos de la base de datos completamente. A menos que especifique la cláusula `PURGE`, la sentencia `DROP TABLE` no vuelve a liberar espacio en los tablespaces para que lo utilicen otros objetos y el espacio sigue contando en la cuota de espacio del usuario. El borrado de una tabla invalida objetos dependientes y elimina privilegios de objeto en la tabla.

Al borrar una tabla, la base de datos pierde todos los datos de la tabla y los índices asociados a los mismos.

**Sintaxis**

`DROP TABLE` *`table`* `[PURGE]`

En la sintaxis, *`table`* es el nombre de la tabla.

**Directrices**

- Se suprimen todos los datos de la tabla.
- Se mantienen las vistas y los sinónimos, pero no son válidos.
- Se confirman las transacciones pendientes.
- Solo el creador de la tabla o un usuario con el privilegio `DROP ANY TABLE` puede eliminar una tabla.

# Resumen

En esta lección, debe haber aprendido a hacer lo siguiente:

- Describir la sintaxis de comandos SQL-92/1999 básicos, incluidos:
  - SELECT
  - INSERT
  - UPDATE
  - DELETE
  - CREATE TABLE
  - DROP TABLE
- Definir tipos de datos SQL-92/1999

ORACLE