Programación Java SE 7

Volumen I • Guía del alumno

D67238CS20
Edición 2.0
Noviembre de 2011
D81763

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

1

Introducción

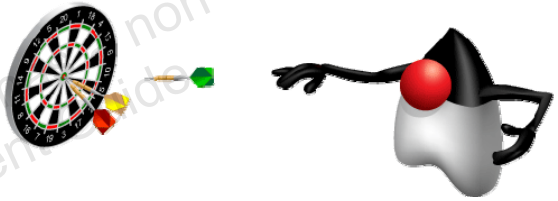
Metas del curso

- Este curso aborda las principales API que se usan para diseñar aplicaciones orientadas a objetos con Java. También aborda la escritura de programas de base de datos con JDBC.
- Utilice este curso para ampliar sus conocimientos del lenguaje Java y prepararse para el examen de programador Oracle Certified Professional, Java SE 7.

Objetivos del curso

Al finalizar este curso, debería estar capacitado para lo siguiente:

- Crear aplicaciones con tecnología Java en las que se usen las funciones orientadas a objetos del lenguaje Java, como la encapsulación, la herencia y el polimorfismo
- Ejecutar una aplicación Java desde la línea de comandos
- Crear aplicaciones que usen el marco Collections
- Implantar técnicas de manejo de errores mediante el manejo de excepciones
- Implantar la funcionalidad de entrada/salida (E/S) de lectura y escritura de datos y archivos de texto y comprender los flujos de E/S avanzados



ORACLE

Objetivos del curso

(continuación)

- Manipular archivos, directorios y sistemas de archivos mediante la especificación JDK7 NIO.2
- Realizar varias operaciones en tablas de bases de datos, incluida la creación, la lectura, la actualización y la supresión mediante la API JDBC
- Procesar cadenas mediante una serie de expresiones regulares
- Crear aplicaciones multithread de alto rendimiento que eviten los interbloqueos
- Localizar aplicaciones Java

ORACLE

Asistentes

Entre el público al que va dirigido se incluyen aquellos que:

- Hayan terminado el curso *Conceptos fundamentales de Java SE 7*, o bien que tengan experiencia con el lenguaje Java y que sean capaces de crear, compilar y ejecutar programas
- Tengan experiencia con al menos un lenguaje de programación
- Comprendan los principios orientados a objetos
- Tengan experiencia con los conceptos básicos y conocimientos básicos de SQL

Requisitos

Para completar este curso satisfactoriamente, debe saber cómo:

- Compilar y ejecutar aplicaciones Java
- Crear clases Java
- Crear instancias de objetos con la palabra clave `new`
- Declarar variables de referencia y primitivas de Java
- Declarar métodos Java con valores de retorno y parámetros
- Usar construcciones condicionales como sentencias `if` y `switch`
- Usar construcciones en bucle, como bucles `for`, `while` y `do`
- Declarar e instanciar matrices Java
- Usar la especificación de la API Java Platform, Standard Edition (Javadocs)

Presentaciones a la clase

Preséntese brevemente:

- Nombre
- Cargo o puesto
- Compañía
- Experiencia con programación Java y aplicaciones Java
- Motivos para asistir

Entorno del curso

PC del aula

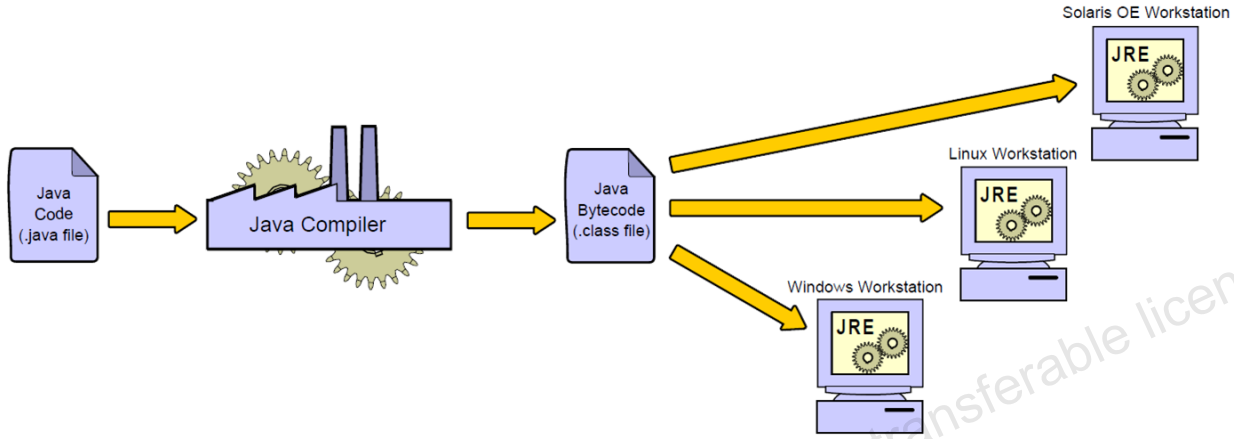
Aplicaciones principales
- JDK 7
- NetBeans 7.0.1

Herramientas adicionales
- Firefox
- Java DB

En este curso están preinstalados los siguientes productos para las prácticas de las lecciones:

- **JDK 7:** Java SE Development Kit incluye el compilador Java de línea de comandos (`javac`) y Java Runtime Environment (JRE), que proporciona el comando `java` necesario para ejecutar aplicaciones Java.
- **Firefox:** se utiliza un explorador web para ver la documentación HTML (documentación Java) para la bibliotecas de la plataforma Java SE.
- **NetBeans 7.0.1:** NetBeans IDE es una herramienta de desarrollo de software gratuita y de código abierto para los profesionales que crean aplicaciones de empresa, web, de escritorio y móviles. NetBeans 7.0.1 cuenta con soporte total de la plataforma Java SE 7. El soporte se proporciona mediante la oferta Oracle Development Tools Support.
- **Java DB:** Java DB es una distribución soportada por Oracle de la base de datos de tecnología Java 100% Apache Derby de código abierto. Se trata de SQL basado en estándares, fácil de usar, seguro y totalmente transaccional, con API JDBC y con Java EE, pero con un tamaño reducido, de tan solo 2,5 MB.

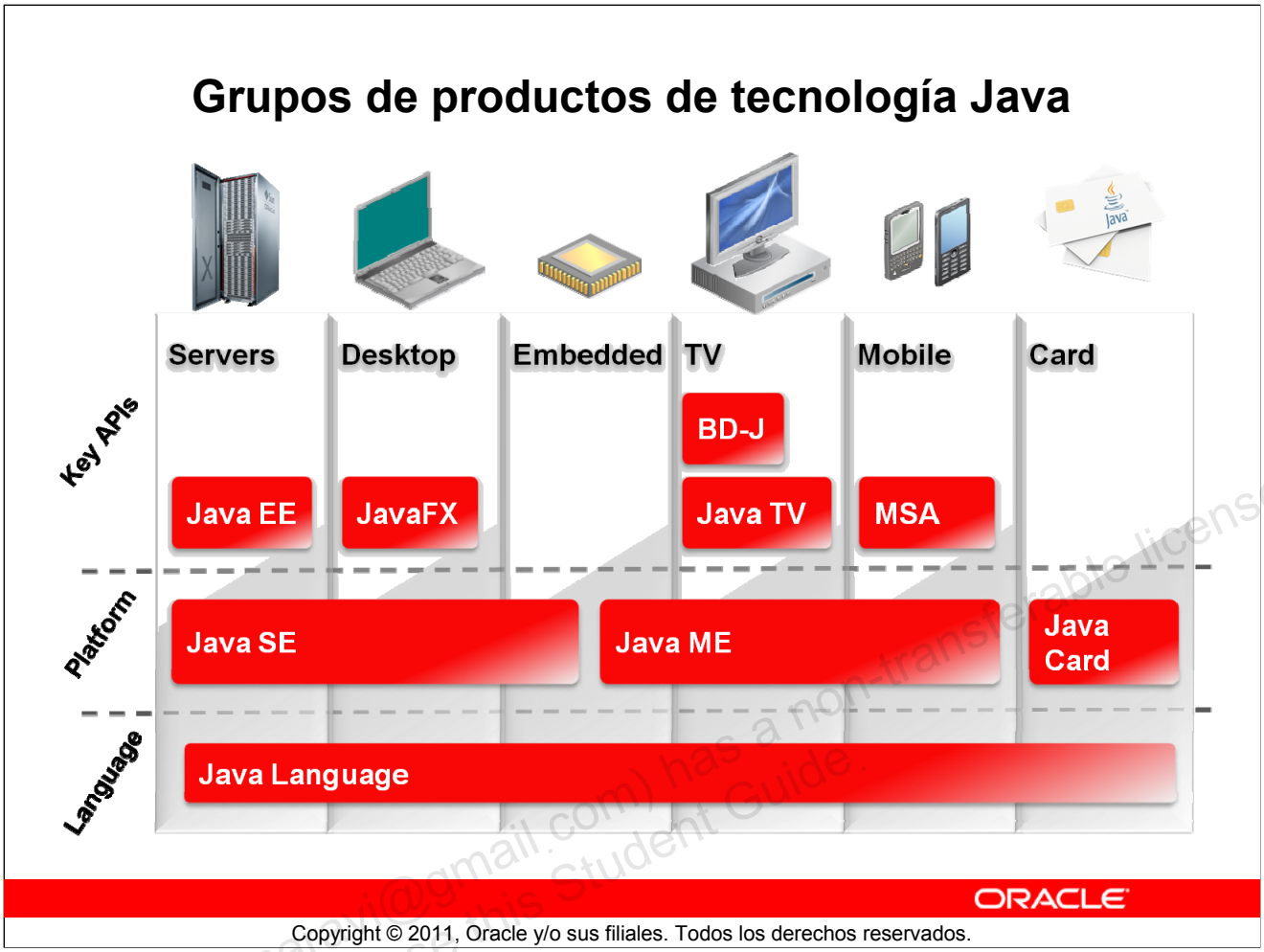
Los programas Java son independientes de la plataforma



ORACLE

Programas independientes de la plataforma

Las aplicaciones de tecnología Java se escriben en el lenguaje de programación Java y se compilan en código de byte de Java. El código de byte se ejecuta en la plataforma Java. El software que le proporciona una plataforma Java que se puede ejecutar se denomina Java Runtime Environment (JRE). Se usa un compilador, incluido en el Java SE Development Kit (JDK), para convertir el código fuente Java en código de byte de Java.



Identificación de grupos de tecnología Java

Oracle proporciona una línea completa de productos de tecnología Java, que va desde kits que crean programas de tecnología Java hasta entornos de emulación (prueba) para dispositivos de consumo, como teléfonos móviles. Como se indica en el gráfico, todos los productos de tecnología Java comparten la base del lenguaje Java. Las tecnologías Java, como Java Virtual Machine, se incluyen (de distintas formas) en tres grupos diferentes de productos, cada uno diseñado para cumplir las necesidades de un mercado objetivo concreto. En la figura se ilustran los tres grupos de productos de tecnología Java y sus tipos de dispositivo objetivo. Entre otras tecnologías Java, cada edición incluye un Software Development kit (SDK) que permite a los programadores crear, compilar y ejecutar programas de tecnología Java en una plataforma concreta:

- **Java Platform, Standard Edition (Java SE):** desarrolla applets y aplicaciones que se ejecutan en exploradores web y en computadoras de escritorio, respectivamente. Por ejemplo, puede utilizar Java SE Software Development Kit (SDK) para crear un programa de procesador de texto para una computadora personal. También puede usar Java SE para crear una aplicación que se ejecute en un explorador.

Nota: los applets y las aplicaciones se diferencian en varios aspectos. Principalmente, los applets se inician en un explorador web, mientras que las aplicaciones se inician en un sistema operativo.

Versiones de la plataforma Java SE

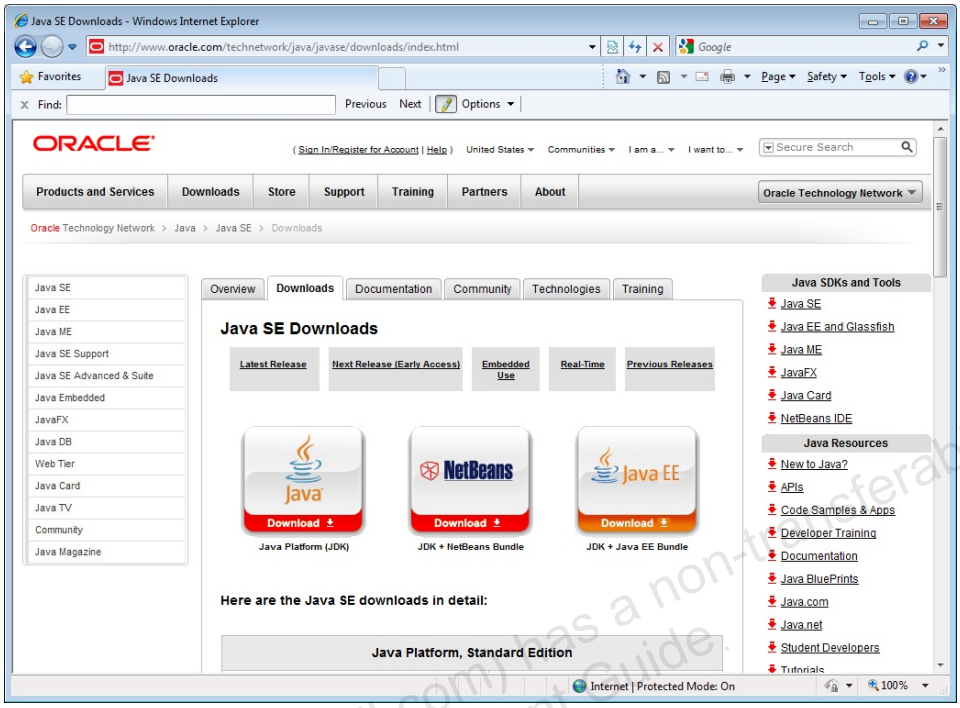
Año	Versión del desarrollador (JDK)	Plataforma
1996	1.0	1
1997	1.1	1
1998	1.2	2
2000	1.3	2
2002	1.4	2
2004	1.5	5
2006	1.6	6
2011	1.7	7

ORACLE

Cómo detectar la versión

Si tiene Java SE instalado en el sistema, puede detectar el número de versión ejecutando `java -version`. Tenga en cuenta que el comando `java` se ha incluido con Java Runtime Environment (JRE). Como desarrollador, también necesita un compilador Java, normalmente `javac`. El comando `javac` está incluido en Java SE Development Kit (JDK). Es posible que haya que actualizar la variable `PATH` del sistema operativo para que incluya la ubicación de `javac`.

Descarga e instalación del JDK



1. Vaya a http://www.oracle.com/technetwork/java/javase/downloads/index.html.
2. Seleccione el enlace Java Platform, Standard Edition (Java SE).
3. Descargue la versión correspondiente a su sistema operativo.
4. Siga las instrucciones de instalación.
5. Defina su valor de `PATH`.

Java en entornos de servidor

Java se suele usar en entornos de empresa:

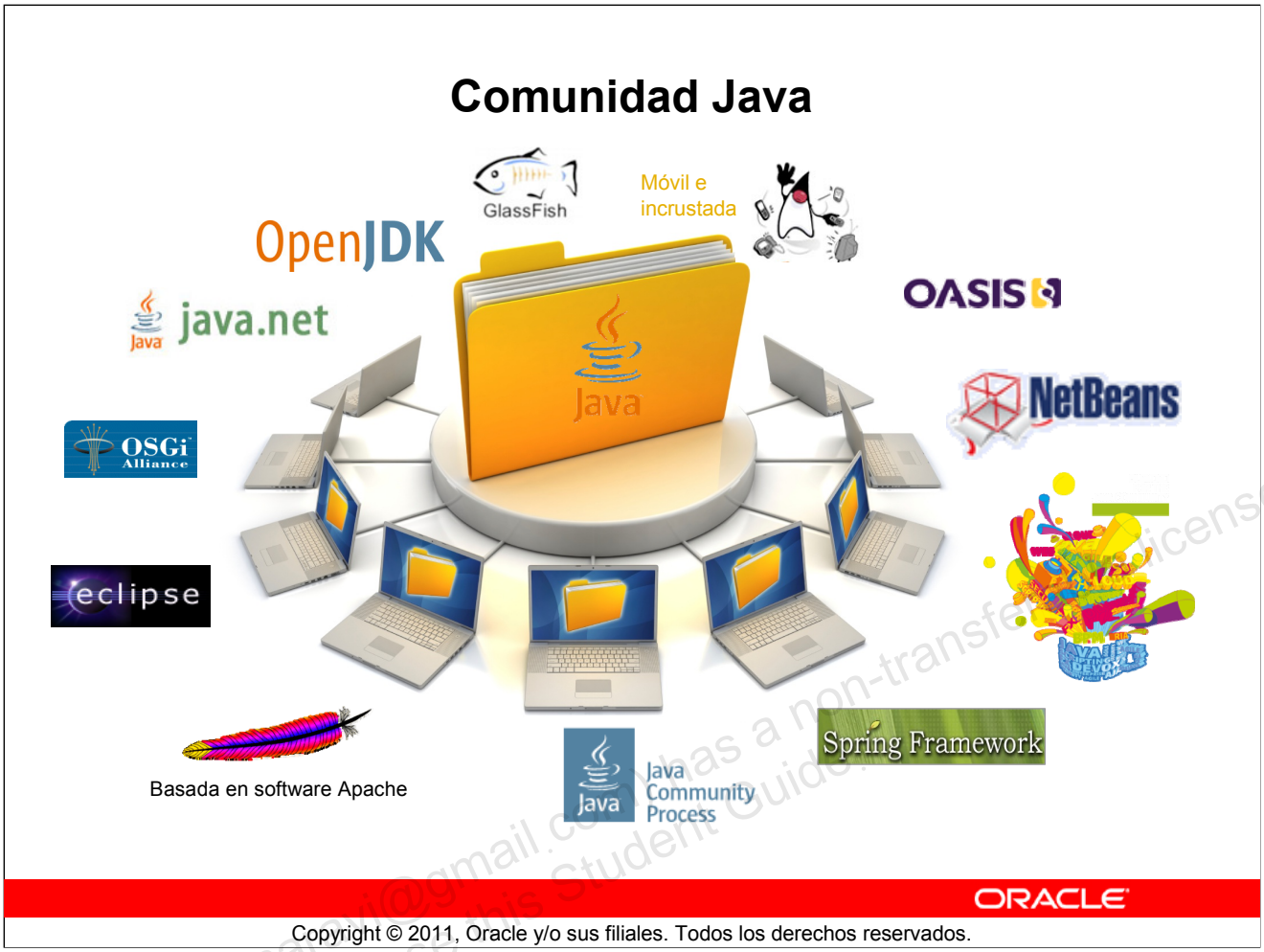
- Oracle Fusion Middleware
  - Servidores de aplicaciones Java
    - GlassFish
    - WebLogic
- Servidores de base de datos
  - MySQL
  - Oracle Database

ORACLE

Entornos de empresa

En este curso, desarrollará aplicaciones Java SE. Hay patrones estándar que se tienen que seguir al implantar aplicaciones Java SE, como crear siempre un método `main` que pueda variar al implantar aplicaciones de empresa. Java SE es el único punto de inicio en su trayectoria para convertirse en un desarrollador Java. En función de las necesidades de su organización, puede que tenga que desarrollar aplicaciones que se ejecuten dentro de servidores de aplicaciones Java EE o en otros tipos de middleware Java.

En ocasiones, también tendrá que trabajar con información almacenada en bases de datos relacionales como MySQL u Oracle Database. En este curso se le presentan los conceptos fundamentales de la programación de bases de datos.



¿Qué es la comunidad Java?

A un alto nivel, la *comunidad Java* es el término que se usa para referirse al gran número de personas y organizaciones que desarrollan, innovan y usan la tecnología Java. En esta comunidad se incluyen desarrolladores como personas, organizaciones, empresas y proyectos de código abierto.

En la comunidad Java es muy habitual que descargue y use bibliotecas Java de fuentes que no sean de Oracle. Por ejemplo, en este curso, utilizará una biblioteca JDBC desarrollada por Apache para acceder a una base de datos relacional.

Java Community Process (JCP)

JCP se usa para desarrollar nuevos estándares Java:

- http://jcp.org
- Descarga gratuita de todas las solicitudes de especificación Java (JSR)
- Acceso anticipado a las especificaciones
- Posibilidad de recibir comentarios y revisiones por parte de otros miembros
- Participación libre

ORACLE

JCP.next

JCP produce las solicitudes JSR que detallan los estándares de la plataforma Java. Mediante el proceso JSR también se define y se mejora el comportamiento del propio JCP. JCP está en plena evolución y sus mejoras se definen en JSR-348. JSR-348 presenta cambios en las áreas de la transparencia, la participación, la agilidad y el gobierno.

- **Transparencia:** en el pasado, algunos de los aspectos relacionados con el desarrollo de una JSR se producían de forma opaca. Sin embargo, el desarrollo transparente es ahora la práctica recomendada.
- **Participación:** se anima tanto a los usuarios como a los grupos de usuarios Java a formar parte de JCP.
- **Agilidad:** las JSR lentas ahora no se recomiendan.
- **Gobierno:** los grupos de expertos SE y ME se están fusionando en un solo cuerpo.

OpenJDK

OpenJDK es la implantación de código fuente de Java:

- http://openjdk.java.net/
- Proyecto de código abierto con licencia GPL
- Implantación de referencia JDK
- Donde se desarrollan nuevas funciones
- Permite contribuciones a la comunidad
- Base de Oracle JDK

ORACLE

Importancia de OpenJDK

Debido a que se trata de código abierto, OpenJDK permite a los usuarios trasladar Java a sistemas operativos y plataformas de hardware de su elección. Actualmente se están realizando traslados para muchas plataformas (además de las ya soportadas), incluidas FreeBSD, OpenBSD, NetBSD y MacOS X.

Soporte de Oracle Java SE

Java está disponible de forma gratuita. Sin embargo, Oracle proporciona soluciones Java de pago:

- Java SE Support Program ofrece actualizaciones para versiones Java con un fin de vida determinado.
- Oracle Java SE Advanced y Oracle Java SE Suite:
  - JRockit Mission Control
  - Detección de falta de memoria
  - Low Latency GC (Suite)
  - JRockit Virtual Edition (Suite)

ORACLE

Aún es gratuito

Java (Oracle JDK) se pone a libre disposición del público sin coste alguno. Oracle ofrece soluciones comerciales avanzadas sin ningún coste. El programa “Java for Business” que se ofrecía anteriormente se ha sustituido por Oracle Java SE Support, que permite acceso a Oracle Premier Support y a los binarios de Oracle Java SE Advanced y de Oracle Java SE Suite. Para obtener más información, visite http://www.oracle.com/us/technologies/java/java-se-suite-394230.html.

Recursos adicionales

Tema	Sitio web
Enseñanza y formación	http://education.oracle.com
Documentación de productos	http://www.oracle.com/technology/documentation
Descarga de productos	http://www.oracle.com/technology/software
Artículos sobre productos	http://www.oracle.com/technology/pub/articles
Soporte de productos	http://www.oracle.com/support
Foros sobre productos	http://forums.oracle.com
Tutoriales sobre productos	http://www.oracle.com/technetwork/tutorials/index.html
Código de ejemplo	https://www.samplecode.oracle.com
Oracle Technology Network para desarrolladores Java	http://www.oracle.com/technetwork/java/index.html
Oracle Learning Library	http://www.oracle.com/goto/oll

ORACLE

En la tabla de la diapositiva se muestran los distintos recursos web disponibles que permiten obtener más información sobre la programación Java SE.

Resumen

En esta lección debe haber aprendido lo siguiente:

- Los objetivos del curso
- Software usado en este curso
- Plataformas Java (ME, SE y EE)
- Números de versión de Java SE
- Obtención de un JDK
- La naturaleza abierta de Java y su comunidad
- Opciones de soporte comercial para Java SE

ORACLE

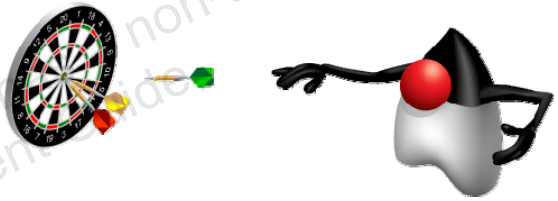
2

Sintaxis Java y revisión de clases

Objetivos

Al finalizar esta lección, debería estar capacitado para lo siguiente:

- Crear clases Java simples
  - Crear variables primitivas
  - Manipular cadenas
  - Usar las sentencias de bifurcación `if-else` y `switch`
  - Iterar con bucles
  - Crear matrices
- Usar campos, constructores y métodos Java
- Usar las sentencias `package` e `import`

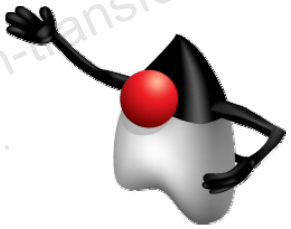


ORACLE

Revisión del lenguaje Java

En esta lección se repasan los conceptos fundamentales de Java y de la programación. Se supone que los alumnos conocen los siguientes conceptos:

- La estructura básica de una clase Java
- Los bloques y comentarios de un programa
- Variables
- Las construcciones de bifurcación básicas `if-else` y `switch`
- La iteración con bucles `for` y `while`



ORACLE

Estructura de la clase

```
package <package_name>;

import <other_packages>;

public class ClassName {
   <variables(also known as fields)>;

   <constructor method(s)>;

   <other methods>;
}
```


Se describe una clase Java en un archivo de texto con una extensión `.java`. En el ejemplo mostrado, las palabras clave Java aparecen resaltadas en negrita.

- La palabra clave `package` define si esta clase está relacionada con otras clases y se proporciona un nivel de control de acceso. Se usan modificadores de acceso (como `public` y `private`) más adelante en esta lección.
- La palabra clave `import` define otras clases o grupos de clases que está usando en la clase. La sentencia `import` permite restringir lo que el compilador tiene que buscar a la hora de resolver los nombres de clases usados en esta clase.
- La palabra clave `class` precede al nombre de esta clase. El nombre de la clase y el nombre de archivo deben coincidir cuando se declare la clase `public` (lo que es una buena práctica). Sin embargo, la palabra clave `public` situada delante de la palabra clave `class` es un modificador y no es necesaria.
- Las variables, o los datos asociados a los programas (como enteros, cadenas, matrices y referencias a otros objetos), se denominan *campos de instancia* (en ocasiones abreviado como *campos*).
- Los constructores son funciones que se llaman durante la creación (instanciación) de un objeto (representación en la memoria de una clase Java).
- Los métodos son las funciones que se pueden realizar en un objeto. También se conocen como *métodos de instancia*.

Clase simple

Una clase Java simple con un método main:

```
public class Simple{

    public static void main(String args[]){

    }
}
```


Para ejecutar un programa Java, debe definir un método `main`, como se muestra en la diapositiva. Al método `main` se le llama automáticamente cuando se llama a la clase desde la línea de comandos.
Los argumentos de la línea de comandos se transfieren al programa mediante la matriz `args[]`.

Nota: se llama a un método que se modifica con la palabra clave `static` sin una referencia a un objeto concreto. En su lugar, se usa el nombre de clase. A estos métodos se les conoce como *métodos de clase*. El método `main` es un método especial al que se llama cuando se ejecuta esta clase con Java Runtime.

Bloques de código

- Todas las declaraciones de clase se incluyen en un bloque de código.
- Las declaraciones de métodos se incluyen en bloques de código.
- El ámbito de los campos y los métodos Java es el bloque (o la clase).
- Los bloques de códigos se definen entre corchetes:

```
{  }
```

- Ejemplo:

```
public class SayHello {
    public static void main(String[] args) {
       System.out.println("Hello world");
    }
}
```

ORACLE

El ámbito de clase de los campos (variables) y los métodos Java se define mediante el corchete angular de apertura de la izquierda y el de cierre de la derecha.

El ámbito de clase permite a cualquier método de la clase llamar a cualquier otro método de la clase. El ámbito de clase también permite que cualquier método acceda a cualquier campo de la clase.

Los bloques de código siempre se definen con corchetes `{}`. Para ejecutar un bloque, se ejecuta cada una de las sentencias definidas en el bloque en el orden del primero al último (de izquierda a derecha).

El compilador Java ignora todos los espacios en blanco. No es necesario el sangrado de líneas, pero facilita la lectura del código. En este curso, el sangrado de la línea incluye cuatro espacios, que es el sangrado de línea por defecto que usa NetBeans IDE.

Tipos de datos primitivos

Entero	Coma flotante	Carácter	True False
byte short int long	float double	char	boolean
1, 2, 3, 42 07 0xff	3.0F .3337F 4.022E23	'a' '\u0061' '\n'	true false
0	0.0f	'\u0000'	false

Agregue las letras "`L`" o "`F`" en mayúsculas o minúsculas al número para especificar un número largo o uno flotante.

ORACLE

Entero

Java proporciona cuatro tipos de enteros diferentes para números de diferentes tamaños. Todos los tipos numéricos llevan signo, lo cual quiere decir que pueden incluir números positivos o negativos.

Los tipos enteros tienen los siguientes rangos:

- El rango de `byte` es de –128 a +127. Número de bits = 8.
- El rango de `short` es de –32 768 a +32 767. Número de bits = 16.
- El rango de `int` es de –2 147 483 648 a +2 147 483 647. El tipo de entero más común es `int`. Número de bits = 32.
- El rango de `long` es de –9 223 372 036 854 775 808 a +9 223 372 036 854 775 807. Número de bits = 64.

Coma flotante

Los tipos de coma flotante incluyen números con una parte fraccionaria conforme con el estándar IEEE 754. Hay dos tipos de comas flotantes: `float` y `double`.

`double` recibe este nombre porque proporciona el doble de precisión que `float`. Un elemento `float` usa 32 bits para almacenar datos, mientras que un elemento `double` utiliza 64 bits.

Carácter

El tipo `char` se utiliza en caracteres individuales, a diferencia de una cadena de caracteres (que se implanta como un objeto `String`). Java soporta Unicode, un estándar internacional para representar caracteres en cualquier idioma escrito del mundo en valores únicos de 16 bits. Los primeros 256 caracteres coinciden con el juego de caracteres ISO Latín 1, parte del cual es ASCII.

Booleano

El tipo `boolean` puede ser `true` o `false`.

Nota: `true` y `false` pueden parecer palabras clave, pero son técnicamente literales booleanos.

Valores por defecto

Si no se especifica ningún valor, se utiliza un valor por defecto. Los valores en rojo en la diapositiva son los valores utilizados por defecto. El valor `char` por defecto es `null` (representado como `'\u0000'`), mientras que el valor por defecto de `boolean` es `false`.

Nota: las variables locales (es decir, las variables declaradas en métodos) no tienen un valor por defecto. Si se intenta usar una variable local a la que no se le ha asignado un valor, se producirá un error del compilador. Es bueno incluir siempre un valor por defecto en cualquier variable.

Literales numéricas de Java SE 7

En Java SE 7 (y versiones posteriores), puede aparecer cualquier número de caracteres subrayados (_) entre dígitos en un campo numérico. Esto puede mejorar la lectura del código.

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi =  3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

ORACLE

Reglas para literales

Solo puede colocar caracteres de subrayado entre dígitos; no puede colocarlos en los siguientes lugares:

- Al principio o al final de un número
- Junto a una coma decimal en un literal de coma flotante
- Antes de un sufijo `F` o `L`
- En posiciones en las que se espere una cadena de dígitos

Nota: Java es sensible a mayúsculas y minúsculas. En Java, la variable `creditCardNumber` no es igual que `CREDITCARDNUMBER`. La convención indica que las variables y los nombres de métodos Java usan “formato CamelCase en minúsculas”, minúsculas para la primera letra del primer elemento de un nombre de variable y mayúsculas para la primera letra de los posteriores elementos.

Literales binarios de Java SE 7

En Java SE 7 (y versiones posteriores), los literales binarios también se pueden expresar con el sistema binario agregando los prefijos `0b` o `0B` al número:

```
// An 8-bit 'byte' value:
byte aByte = (byte)0b00100001;

// A 16-bit 'short' value:
short aShort = (short)0b1010_0001_0100_0101;

// Some 32-bit 'int' values:
int anInt1 = 0b1010_0001_0100_0101_1010_0001_0100_0101;
int anInt2 = 0b101;
int anInt3 = 0B101; // The B can be upper or lower case.
```

ORACLE

Los literales binarios son valores `int` de Java. Los valores `byte` y `short` de Java se deben convertir para evitar una advertencia de pérdida de precisión del compilador.

Operadores

- Operador de asignación simple
  - = Operador de asignación simple
- Operadores aritméticos
  - + Operador de suma (también se usa para la concatenación de cadenas)
  - – Operador de resta
  - * Operador de multiplicación
  - / Operador de división
  - % Operador de resto
- Operadores unarios
  - + Operador más unario; indica positivo
  - – Operador menos unario; niega una expresión
  - ++ Operador de aumento; aumenta un valor en 1
  - -- Operador de disminución; disminuye un valor en 1
  - ! Operador de complemento lógico; invierte el valor de un booleano

ORACLE

Como se han presentado los números, en la diapositiva se muestra una lista de operadores comunes. La mayoría son habituales de cualquier lenguaje de programación y se proporciona una descripción en la diapositiva.

Los operadores binarios y a nivel de bit se han omitido por brevedad. Para obtener detalles sobre ellos, consulte el tutorial de Java:

http://download.oracle.com/javase/tutorial/java/nutsandbolts/operators.html

Nota: los operadores tienen una prioridad definitiva. Para obtener la lista completa, consulte el enlace del tutorial de Java mencionado anteriormente. La prioridad se puede sustituir mediante el uso de paréntesis.

Cadenas

```
1 public class Strings {
2
3     public static void main(String args[]){
4
5         char letter = 'a';
6
7         String string1 = "Hello";
8         String string2 = "World";
9         String string3 = "";
10        String dontDoThis = new String ("Bad Practice");
11
12        string3 = string1 + string2; // Concatenate strings
13
14        System.out.println("Output: " + string3 + " " + letter);
15
16    }
17 }
```

Los literales de cadena se crean automáticamente como objetos String.

ORACLE

El código de la diapositiva muestra cómo se representan los caracteres de texto en Java. Los caracteres sencillos se pueden representar con el tipo `char`. Sin embargo, Java también incluye un tipo `String` para representar varios caracteres. Las cadenas se pueden definir como se muestra en la diapositiva y combinarse con el signo "`+`" como operador de concatenación.

La salida del código en la diapositiva es:

```
Output: HelloWorld a
```

Atención: las cadenas se deben inicializar siempre con el operador de asignación "`=`" y texto entre comillas, como se muestra en los ejemplos. No se recomienda usar `new` para inicializar un objeto `String`. El motivo es que "`Bad Practice`", que aparece en la línea 10, es un literal `String` del tipo `String`. El uso de la palabra clave `new` simplemente sirve para crear otra instancia idéntica desde el punto de vista funcional al literal. Si esta sentencia apareciera dentro de un bucle al que se llamara frecuentemente, se crearían muchas instancias de `String` innecesarias.

Operaciones de cadenas

```
1 public class StringOperations {
2     public static void main(String arg[]){
3         String string2 = "World";
4         String string3 = "";
5
6         string3 = "Hello".concat(string2);
7         System.out.println("string3: " + string3);
8
9         // Get length
10        System.out.println("Length: " + string1.length());
11
12        // Get SubString
13        System.out.println("Sub: " + string3.substring(0, 5));
14
15        // Uppercase
16        System.out.println("Upper: " + string3.toUpperCase());
17    }
18}
```

Los literales de cadena se crean automáticamente como objetos String.

ORACLE

En la diapositiva se muestran algunos métodos de cadena comunes, incluidos:

- `concat()`
- `length()`
- `substring()`
- `toUpperCase()`

Para ver qué otros métodos se pueden usar en un objeto `String`, consulte la documentación de la API.

La salida del programa es:

```
string3: HelloWorld
Length: 5
Sub: Hello
Upper: HELLOWORLD
```

Nota: `String` es una clase, no un tipo primitivo. Las instancias de la clase `String` representan secuencias de caracteres Unicode.

if else

```
1 public class IfElse {
2
3     public static void main(String args[]){
4         long a = 1;
5         long b = 2;
6
7         if (a == b){
8             System.out.println("True");
9         } else {
10            System.out.println("False");
11        }
12
13    }
14 }
```


En el ejemplo de la diapositiva se muestra la sintaxis de una sentencia `if-else` en Java.
La salida del código en la diapositiva es la siguiente:

```
False
```

Operadores lógicos

- Operadores de igualdad y relacionales
  - == Igual que
  - != Distinto de
  - > Mayor que
  - >= Mayor o igual que
  - < Menor que
  - <= Menor o igual que
- Operadores condicionales
  - && AND condicional
  - || OR condicional
  - ?: Ternario (versión abreviada de la sentencia `if-then-else`)
- Operador de comparación de tipos
  - `instanceof` Compara un objeto con un tipo especificado

ORACLE

En la diapositiva se muestra un resumen de los operadores lógicos y condicionales en Java.

Matrices y bucle `for-each`

```
1 public class ArrayOperations {
2     public static void main(String args[]){
3
4         String[] names = new String[3];
5
6         names[0] = "Blue Shirt";
7         names[1] = "Red Shirt";
8         names[2] = "Black Shirt";
9
10         int[] numbers = {100, 200, 300};
11
12         for (String name:names){
13             System.out.println("Name: " + name);
14         }
15
16         for (int number:numbers){
17             System.out.println("Number: " + number);
18         }
19     }
20 }
```

Las matrices son objetos. Los objetos de matriz tienen una longitud de campo final.

ORACLE

Esta clase demuestra cómo definir las matrices en Java. En el primer ejemplo se crea una matriz `String` y se inicializa cada uno de los elementos por separado. La segunda matriz `int` se define en una sola sentencia.

Cada matriz se itera con la construcción `for-each` de Java. El bucle define un elemento que representará cada elemento de la matriz y la matriz de la que se va realizar bucle. La salida de la clase se muestra aquí:

```
Name: Blue Shirt
Name: Red Shirt
Name: Black Shirt
Number: 100
Number: 200
Number: 300
```

Nota: por defecto, las matrices también son objetos. Todas las matrices soportan los métodos de la clase Object. Siempre puede obtener el tamaño de una matriz mediante su campo `length`.

Bucle `for`

```
1 public class ForLoop {
2
3     public static void main(String args[]){
4
5         for (int i = 0; i < 9;  i++ ){
6             System.out.println("i: " + i);
7         }
8
9     }
10 }
```


La diapositiva muestra el bucle `for` clásico. Se inicializa un contador y se incrementa con cada paso del bucle. Cuando se cumple la sentencia de la condición, el bucle sale. A continuación se muestra la salida de ejemplo de este programa.

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
```

Bucle `while`

```
1 public class WhileLoop {
2
3    public static void main(String args[]){
4
5        int i = 0;
6        int[] numbers = {100, 200, 300};
7
8        while (i < numbers.length ){
9            System.out.println("Number: " + numbers[i]);
10            i++;
11         }
12    }
13 }
```

ORACLE

El bucle `while` realiza una prueba y continúa si la expresión se evalúa en `true`. El bucle `while`, que se muestra aquí, se itera en una matriz mediante un contador. En esta diapositiva se muestra la salida del código:

```
Number: 100
Number: 200
Number: 300
```

Nota: también existe un bucle `do-while`, donde se ha ejecutado la prueba después de la expresión al menos una vez:

```
class DoWhileDemo {
     public static void main(String[] args){
          int count = 1;
          do {
               System.out.println("Count is: " + count);
               count++;
          } while (count <= 11);
     }
}
```

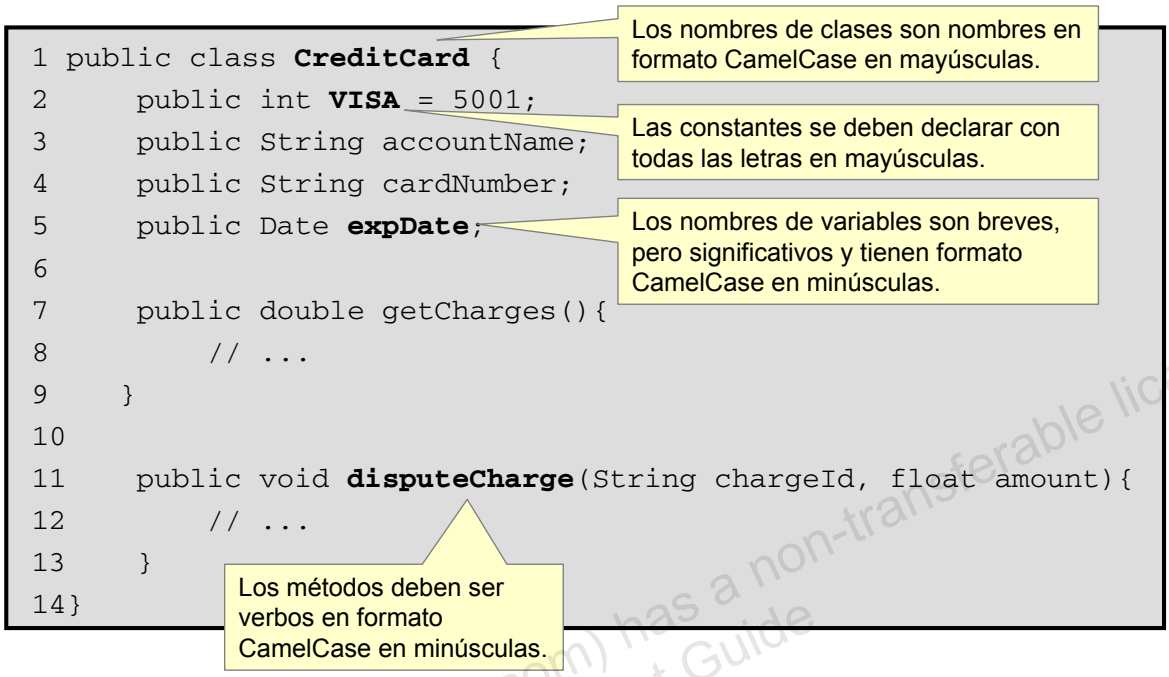
Sentencia `switch` de cadena

```
1 public class SwitchStringStatement {
2    public static void main(String args[]){
3
4        String color = "Blue";
5        String shirt = " Shirt";
6
7        switch (color){
8            case "Blue":
9                shirt = "Blue" + shirt;
10               break;
11           case "Red":
12               shirt = "Red" + shirt;
13               break;
14           default:
15               shirt = "White" + shirt;
16       }
17
18       System.out.println("Shirt type: " + shirt);
19   }
20 }
```

ORACLE

Este ejemplo muestra una sentencia `switch` en Java con un objeto `String`. Antes de la versión 7 de Java, solo se podían usar enumeraciones y los tipos de datos primitivos `byte`, `short`, `char` e `int` en una sentencia `switch`. Las enumeraciones aparecen en la lección titulada “Diseño de clases avanzadas”.

Convenciones de nomenclatura Java



- Los nombres de clases deben ser nombres con mayúsculas y minúsculas, con la primera letra mayúscula y la primera letra de cada palabra interna en mayúsculas. A este enfoque se le conoce como "CamelCase en mayúsculas".
- Los métodos deben ser verbos con mayúsculas y minúsculas, con la primera letra en minúscula y la primera letra de cada palabra interna en mayúsculas. A esto se le conoce como "CamelCase en minúsculas".
- Los nombres de variables deben ser cortos, pero significativos. La elección de un nombre de variable debe ser mnemotécnica: diseñada para indicar al observador casual su finalidad.
- Los nombres de variables de un carácter se deben evitar, excepto como variables "desechables" temporales.
- Las constantes se deben declarar con todas las letras en mayúsculas.

Para obtener el documento completo *Code Conventions for the Java Programming Language* (Convenciones de código para el lenguaje de programación Java), vaya a http://www.oracle.com/technetwork/java/codeconv-138413.html.

Una clase Java simple: `Employee`

Una clase Java se suele usar para representar un concepto.

```
1 package com.example.domain;
2 public class Employee {
3     public int empId;
4     public String name;
5     public String ssn;
6     public double salary;
7
8     public Employee () {
9     }
10
11     public int getEmpId () {
12       return empId;
13    }
14 }
```

declaración de clase

campos

un constructor

un método

ORACLE

La clase Java se suele usar para almacenar o representar datos para la construcción que representa la clase. Por ejemplo, podría crear un modelo (una representación programática) de un objeto Employee. Un objeto `Employee` definido mediante este modelo contendrá valores para `empId`, `name`, número de la Seguridad Social (`ssn`) y `salary`.

El método de constructor de esta clase crea una instancia de un objeto denominado `Employee`.

Un método constructor es exclusivo de Java, ya que el tipo de retorno del método es una instancia de la clase, por lo que los constructores siempre tienen el mismo nombre que la clase y no declaran un tipo de retorno. Puede declarar más de un constructor, como verá en la lección titulada “Diseño de clases Java”.

Métodos de la clase `Employee`

Cuando una clase tiene campos de datos, una práctica habitual consiste en proporcionar métodos para almacenar datos (métodos setter) y recuperar datos (métodos getter) de los campos.

```
1 package com.example.domain;
2 public class Employee {
3     public int empId;
4     // other fields...
5     public void setEmpId(int empId) {
6         this.empId = empId;
7     }
8     public int getEmpId() {
9         return empId;
10    }
11    // getter/setter methods for other fields...
12 }
```

A menudo un par de métodos para definir y obtener el valor del campo actual.

ORACLE

Adición de métodos a la clase `Employee`

Una práctica común es crear un juego de métodos que manipulen datos de campo: métodos que definen el valor de cada uno de los campos y métodos que obtienen el valor de cada campo. A estos métodos se les denomina *métodos de acceso* (getter) y *mutadores* (setter).

La convención consiste en usar `set` y `get` más el nombre del campo con la primera letra del nombre de campo en mayúsculas (CamelCase en minúsculas). La mayoría de los entornos de desarrollo integrados (IDE) modernos proporcionan un método sencillo para generar automáticamente los métodos de acceso (getter) y mutadores (setter).

Observe que los métodos set usan la palabra clave `this`. La palabra clave `this` permite al compilador distinguir entre el nombre de campo de la clase (`this`) y el nombre del parámetro que se está transfiriendo como argumento. Sin la palabra clave `this`, el compilador detectaría: "Assignment to self".

En este sencillo ejemplo, podría usar el método `setName` para cambiar el nombre de empleado y el método `setSalary` para cambiar el valor `salary` del empleado.

Creación de una instancia de un objeto

Para crear una instancia (objeto) de la clase `Employee`, utilice la palabra clave `new`.

```
/* In some other class, or a main method */
Employee emp = new Employee();
emp.empId = 101;    // legal if the field is public,
                    // but not good OO practice
emp.setEmpId(101); // use a method instead
emp.setName("John Smith");
emp.setSsn("011-22-3467");
emp.setSalary(120345.27);
```

- En este fragmento de código Java, crea una instancia de la clase `Employee` y asigna la referencia al nuevo objeto a una variable denominada `emp`.
- A continuación, asigna valores al objeto `Employee`.

ORACLE

Creación de una instancia de la clase Employee

Para utilizar la clase `Employee` para incluir información de un empleado, debe asignar memoria al objeto `Employee` y llamar a un método constructor de la clase. Se crea una instancia de un objeto al usar la palabra clave `new`. A todos los campos declarados en la clase se les proporciona espacio de memoria y se inicializan en sus valores por defecto. A continuación, se llama al método constructor. Si la asignación de memoria y el constructor son correctos, se devuelve como resultado una referencia al objeto. En el ejemplo de la diapositiva, la referencia se asigna a una variable denominada `emp`.

Para almacenar valores (datos) en la instancia de objeto `Employee`, simplemente podría asignar valores a cada uno de los campos. Sin embargo, esto no es recomendable y niega el principio de la encapsulación. En su lugar, debería usar métodos para definir el valor de cada uno de los campos de datos. Posteriormente en esta lección, examinará la restricción de acceso a los campos para fomentar la encapsulación.

Una vez que todos los campos de datos estén definidos con valores, tiene una instancia de un objeto `Employee` con un `empId` con un valor 101, `name` con la cadena John Smith, Social Security Number (`ssn`) definida en 011-22-3467 y `salary` con el valor 120,345.27.

Constructores

```
public class Employee {
    public Employee() {
    }
}
```

Un constructor sin argumentos (no-arg) simple.

```
Employee emp = new Employee();
```

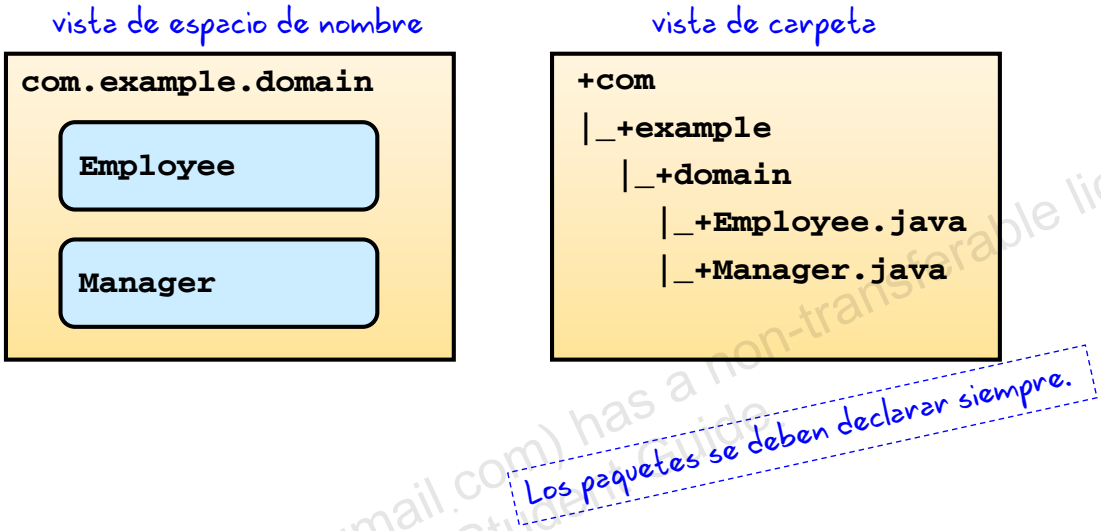
- El valor que devuelve el constructor es una referencia a un objeto Java del tipo creado.
- Los constructores aceptan el uso de parámetros.

Un constructor es un pseudométodo que crea un objeto. En el lenguaje de programación Java, los constructores son métodos con el mismo nombre que su clase, que se usan para crear una instancia de un proyecto. A los constructores se les llama mediante la palabra clave `new`.

Los constructores se tratan con más detalle en la lección titulada “Encapsulación y creación de subclases”.

Sentencia `package`

La palabra clave `package` se usa en Java para agrupar clases. Un paquete se implanta como carpeta y, al igual que una carpeta, proporciona un *espacio de nombre* a una clase.



ORACLE

Paquetes

En Java, un paquete es un grupo de tipos (de clase). Solo puede haber una declaración `package` para una clase.

Los paquetes van más allá de ser algo útil. Los paquetes crean un espacio de nombre, una recopilación lógica de cosas, como una jerarquía de directorios.

Es una buena práctica utilizar siempre una declaración `package`. La declaración `package` siempre está en la parte superior de la clase.

Sentencias `import`

La palabra clave `import` se usa para identificar a las clases a las que desea hacer referencia en la clase.

- La sentencia `import` ofrece un método práctico para identificar clases a las que desea hacer referencia en la clase.

```
import java.util.Date;
```

- Puede importar una única clase o un paquete completo:

```
import java.util.*;
```

- Puede incluir varias sentencias `import`:

```
import java.util.Date;
import java.util.Calendar;
```

- Se aconseja usar todo el paquete y el nombre de clase en lugar del carácter comodín `*` para evitar conflictos de nombres de clase.

ORACLE

Importaciones

Podría hacer referencia a una clase usando su espacio de nombre cualificado en las aplicaciones, como en el siguiente ejemplo:

```
java.util.Date date = new java.util.Date();
```

Sin embargo, esto le supondría tener que introducir muchos datos. En lugar de esto, Java ofrece la sentencia import para permitirle declarar que desea hacer referencia a una clase de otro paquete.

Nota: se recomienda usar el paquete específico totalmente cualificado y un nombre de clase para evitar la confusión cuando haya dos clases con el mismo nombre, como en el siguiente ejemplo: `java.sql.Date` y `java.util.Date`. La primera es una clase `Date`, que se usa para almacenar un tipo `Date` en una base de datos y `java.util.Date` es una clase `Date` de uso general. Resulta que `java.sql.Date` es una subclase de `java.util.Date`. Esto se trata con mayor detalle más adelante en el curso.

Nota: los IDE modernos, como NetBeans y Eclipse, buscan de forma automática y agregan sentencias import. En NetBeans, por ejemplo, use la secuencia de teclas Ctrl + Mayús + I para corregir las importaciones del código.

Más información sobre `import`

- Las sentencias import van después de la declaración del paquete y antes de la declaración de la clase.
- No es necesaria una sentencia import.
- Por defecto, su clase siempre importa `java.lang.*`
- No es necesario que importe clases que estén en el mismo paquete:

```
package com.example.domain;
import com.example.domain.Manager; // unused import
```

ORACLE

Los detalles sobre este paquete y sus clases se tratan más adelante en el curso.

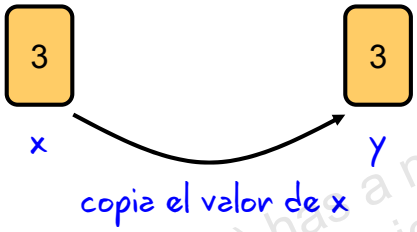
Java se transfiere por valor

El lenguaje Java (a diferencia de C++) usa la transferencia por valor para pasar todos los parámetros.

- Para visualizar esto con primitivos, tenga en cuenta lo siguiente:

```
int x = 3;
int y = x;
```

- El valor de `x` se copia y transfiere a `y`:



- Si se modifica `x` (por ejemplo, `x = 5;`), no se cambia el valor de `y`.

ORACLE

El lenguaje Java usa la transferencia por valor para todas las operaciones de asignación. Esto significa que se evalúa el argumento de la derecha del signo igual y que el valor del argumento se asigna a la izquierda del signo igual.

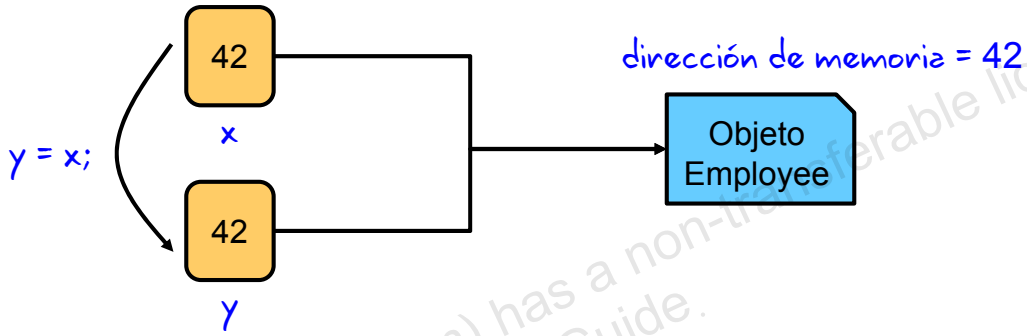
En el caso de primitivos Java, esto es muy sencillo. Java no transfiere una referencia a un primitivo (como un entero), sino una copia del valor.

Transferencia por valor para referencias de objetos

En el caso de objetos Java, el *valor* del lado derecho de una asignación es una referencia a la memoria que almacena un objeto Java.

```
Employee x = new Employee();
Employee y = x;
```

- La referencia es alguna dirección de la memoria.



- Tras la asignación, el valor de `y` es el mismo que el valor de `x`: una referencia al mismo objeto `Employee`.

ORACLE

En el caso de objetos Java, el valor de una referencia de objeto es el puntero de memoria a la instancia del objeto `Employee` creado.

Al asignar el valor de `x` a `y`, no está creando un nuevo objeto `Employee`, sino una copia del valor de la referencia.

Nota: un objeto es una instancia de clase o una matriz. Los valores de referencia (referencias) son punteros a esos objetos y una referencia `null` especial, que no hace referencia a ningún objeto.

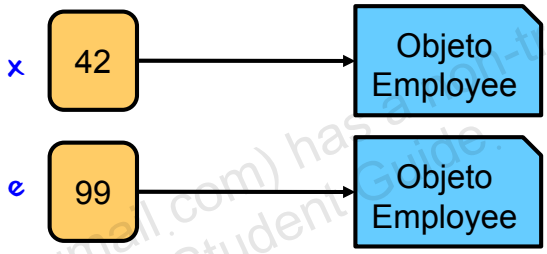
Objetos transferidos como parámetros

- Siempre que se cree un nuevo objeto, se crea una nueva referencia. Considere los siguientes fragmentos de código:

```
Employee x = new Employee();
foo(x);
```

```
public void foo(Employee e) {
    e = new Employee();
    e.setSalary (1_000_000_00);  // What happens to x here?
}
```

- El valor de `x` no cambia como resultado de la llamada al método `foo`:



ORACLE

Siempre que se crea un nuevo valor, como la sentencia `e = new Employee();` en el método `foo`, se crea un nuevo objeto y se cambia el valor del parámetro por el nuevo valor.

El valor de `x` simplemente se copia en el parámetro para el método, por lo que `x` no cambia durante y tras la ejecución del método `foo`.

¿Qué valor se devuelve de `x.getSalary();` tras la llamada a `foo(x);`?

Como se muestra, el valor de la referencia x no cambia y el valor devuelto sería el mismo que antes de la llamada. Esto se debe a que se ha usado la palabra clave `new` para crear un nuevo objeto y se ha asignado esa instancia de objeto a `e`. Sea cual sea el valor que e tuviera antes (la referencia a x) ahora se sobrescribe.

El método `setSalary` funciona en la referencia `e`, no en la referencia de objeto transferida al método.

Si fuera a eliminar la sentencia, `e = new Employee();`, el método `e.setSalary` estaría funcionando en la referencia `x` y el empleado representado por la variable `x` estaría ganando mucho dinero.

Nota: las ubicaciones de memoria 42 y 99 solo se muestran con fines ilustrativos.

Cómo compilar y ejecutar

Los archivos de clase Java se deben compilar antes de ejecutarse.
Para compilar un archivo de origen Java, utilice el compilador Java (`javac`).

```
javac –classpath <path to other classes> -d <complier
output path> <path to source>.java
```

- Puede utilizar la variable de entorno `CLASSPATH` al directorio superior a la ubicación de la jerarquía de paquetes.
- Tras compilar el archivo `.java` de origen, se genera un archivo `.class`.
- Para ejecutar la aplicación Java, ejecútela con el intérprete Java (`java`):

```
java –classpath <path to other classes> <package
name>.<classname>
```

ORACLE

CLASSPATH

La variable `CLASSPATH` la usa tanto el compilador Java como el intérprete Java (tiempo de ejecución).

`classpath` puede incluir:

- Una lista de nombres de directorio (separados por puntos y comas en Windows y dos puntos en UNIX)
  - Las clases están en un árbol de paquete en relación con los directorios de la lista.
  - `classpath` incluye el directorio de trabajo actual (.) por defecto.
- Un nombre de archivo `.zip` o `.jar` que está totalmente cualificado con su nombre de ruta de acceso
  - Las clases de estos archivos se deben comprimir con los nombres de ruta de acceso que se derivan de los directorios formados por sus nombres de paquete.

Nota: el directorio que contiene el nombre raíz de un árbol de paquete se debe agregar a `classpath`. Considere la acción de colocar la información de `classpath` en la ventana de comandos o incluso en el comando Java, en lugar de codificarla en el entorno.

Compilación y ejecución: ejemplo

- Suponga que la clase que aparece en las notas está en el directorio `D:\test\com\example`:

```
javac -d D:\test D:\test\com\example\HelloWorld.java
```

- Para ejecutar la aplicación, utilice el intérprete y el nombre de clase totalmente cualificado:

```
java -classpath D:\test com.example.HelloWorld
Hello World!

java -classpath D:\test com.example.HelloWorld Tom
Hello Tom!
```

- La ventaja de un IDE como NetBeans es que la gestión del classpath, la compilación y la ejecución de la aplicación Java se manejan mediante la herramienta.

ORACLE

Ejemplo

Tenga en cuenta la siguiente clase simple de un archivo denominado `HelloWorld.java` en el directorio `D:\test\com\example`:

```
package com.example;
public class HelloWorld {
    public static void main (String [] args) {
        if (args.length < 1) {
            System.out.println("Hello World!");
        } else {
            System.out.println("Hello " + args[0] + "!");
        }
    }
}
```

Cargador de clase Java

Durante la ejecución de un programa Java, Java Virtual Machine carga los archivos de clase Java compilados con una clase Java propia denominada el “cargador de clases” (`java.lang.ClassLoader`).

- Al instanciar un objeto, se llama al cargador de clases:

```
public class Test {
    public void someOperation() {
        Employee e = new Employee();
        //...
    }
}
```

Al cargador de clases se le llama para "cargar" esta clase en la memoria.

```
Test.class.getClassLoader().loadClass("Employee");
```


Normalmente, el uso del cargador de clases es totalmente invisible para el usuario. Puede ver los resultados del cargador de clases usando el indicador `-verbose` al ejecutar la aplicación. Por ejemplo:

```
java -verbose -classpath D:\test com.example.HelloWorld
[Loaded java.lang.Object from shared objects file]
[Loaded java.io.Serializable from shared objects file]
[Loaded java.lang.Comparable from shared objects file]
[Loaded java.lang.CharSequence from shared objects file]
[Loaded java.lang.String from shared objects file]
[Loaded java.lang.reflect.GenericDeclaration from shared objects file]
[Loaded java.lang.reflect.Type from shared objects file]
[Loaded java.lang.reflect.AnnotatedElement from shared objects file]
[Loaded java.lang.Class from shared objects file]
[Loaded java.lang.Cloneable from shared objects file]
[Loaded java.lang.ClassLoader from shared objects file]
... and many more
```

Recolección de basura

Cuando se crea una instancia de un objeto con la palabra clave `new`, se asigna memoria al objeto. El ámbito de una referencia de objeto depende de si se ha instanciado el objeto:

```
public void someMethod() {
    Employee e = new Employee();
    // operations on e
}
```

El ámbito del objeto `e` finaliza aquí.

- Cuando finaliza `someMethod`, ya no se puede acceder a la memoria a la que hace referencia `e`.
- El recolector de basura de Java reconoce cuándo ya no se puede acceder a una instancia y libera automáticamente esta memoria.

El esquema de recolección de basura de Java se puede ajustar en función del tipo de aplicación que esté creando. Para obtener más información, debería realizar el curso de Oracle University *Ajuste de rendimiento de Java y optimización* (D69518GC10).

Resumen

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

- Crear clases Java simples
  - Crear variables primitivas
  - Manipular cadenas
  - Usar las sentencias de bifurcación `if-else` y `switch`
  - Iterar con bucles
  - Crear matrices
- Usar campos, constructores y métodos Java
- Usar las sentencias `package` e `import`

ORACLE

Prueba

En el siguiente fragmento, ¿qué tres problemas puede identificar?

```
package com.oracle.test;
public class BrokenClass {
    public boolean valid = "false";
    public String s = new String ("A new string");
    public int i = 40_000.00;
    public BrokenClass() { }
}
```

a. Falta una sentencia `import`.
b. A `boolean valid` se le ha asignado un objeto String.
c. `String s` se ha creado mediante `new`.
d. Al método `BrokenClass` le falta una sentencia `return`.
e. Se necesita un método para crear un nuevo objeto `BrokenClass`.
f. Al valor entero `i` se le ha asignado un double.

ORACLE

Prueba

Con la clase `Employee` definida en esta lección, determine el valor de `e.name` en el siguiente fragmento:

```
public Employee changeName (Employee e, String name) {
    e.name = name;
    return (e);
}
//... in another class
Employee e = new Employee();
e.name = "Fred";
e = changeName("Bob", e);
System.out.println (e.getName());
```

a. Fred
b. Bob
c. `null`
d. Objeto String vacío

Prueba

En el siguiente fragmento, ¿cuál es el resultado?

```
public float average (int[] values) {
    float result = 0;
    for (int i = 1; i < values.length; i++)
        result += values;
    return (result/values.length);
}
int[] nums = {100, 200, 300};
System.out.prinln (average(avg));
```

a. 100
b. 150.33
c. 166.67
d. 200

ORACLE

Visión general de la práctica 2-1: Creación de clases Java

En esta práctica, se abordan los siguientes temas:

- Creación de una clase Java con NetBeans IDE
- Creación de una clase Java con un método `main`
- Escritura de código en el cuerpo del método `main` para crear una instancia del objeto `Employee` e imprimir valores de la clase en la consola
- Compilación y prueba de la aplicación mediante NetBeans IDE

ORACLE

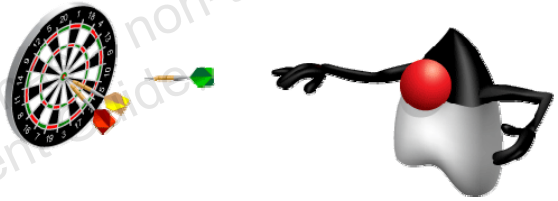
3

Encapsulación y creación de subclases

Objetivos

Al finalizar esta lección, debería estar capacitado para lo siguiente:

- Usar la encapsulación en el diseño de clases Java
- Modelar problemas de negocio con clases Java
- Convertir las clases en inmutables
- Crear y usar subclases Java
- Sobrecargar métodos
- Usar métodos de argumentos variables



ORACLE

Encapsulación

El término *encapsulación* significa incluir en una cápsula o envolver algo alrededor de un objeto para cubrirlo. En la programación orientada a objetos, la encapsulación cubre, o envuelve, el funcionamiento interno de un objeto Java.

- El usuario del objeto no puede ver las variables de datos o los campos.
- Los métodos, las funciones de Java, proporcionan un servicio explícito al usuario del objeto, pero ocultan la implantación.
- Mientras los servicios no cambien, la implantación se puede modificar sin que esto afecte al usuario.

ORACLE

El término *encapsulación* tiene la siguiente definición en el glosario de referencia de tecnología Java:

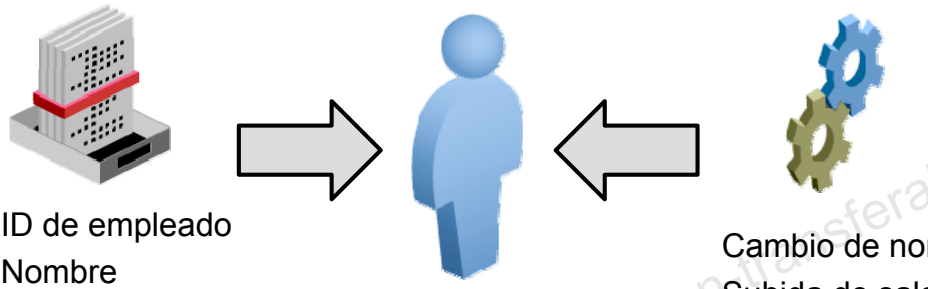
"La localización del conocimiento en un módulo. Debido a que los objetos encapsulan datos y la implantación, el usuario de un objeto puede verlo como una caja negra que proporciona servicios. Se pueden agregar, suprimir o cambiar métodos y variables de instancia, pero si los servicios que proporciona el objeto no varían, el código que usa el objeto puede seguir usándolo sin tenerse que volver a escribir."

Una analogía de la encapsulación sería el volante de un vehículo. Al conducir el vehículo, tanto si es el suyo, el de un amigo o uno de alquiler, normalmente no se plantearía cómo el volante implanta una función de giro a la derecha o a la izquierda. El volante podría estar conectado a las ruedas delanteras de varias formas: mecanismo de rótula, cremallera y piñón o mediante algún otro extraño juego de servomecanismos.

Siempre y cuando la dirección del vehículo sea la deseada al girar, el volante encapsulará las funciones necesarias. No tendrá que preocuparse por la implantación.

Encapsulación: ejemplo

¿Qué datos y operaciones encapsularía en un objeto que represente a un empleado?



ID de empleado
Nombre
Número de la Seguridad Social
Salario

Cambio de nombre
Subida de salario

ORACLE

Modelo simple

Suponga que se le ha pedido que cree un modelo de un empleado típico. ¿Qué datos desearía representar en un objeto que describiera a un empleado?

- **ID de empleado:** puede utilizarlo como identificador único del empleado.
- **Nombre:** humanizar a los empleados siempre es una buena idea.
- **Número de la Seguridad Social:** solo para empleados de EE. UU. Tal vez desee incluir algún otro tipo de identificación para empleados que no sean de EE. UU.
- **Salario:** siempre es bueno registrar cuánto gana el empleado.

¿Qué operaciones podría permitir en el objeto de empleado?

- **Cambio de nombre:** si el empleado se casa o se divorcia, se podría producir un cambio de nombre.
- **Subida de salario:** aumenta según los méritos.

Tras crear un objeto de empleado, es probable que no desee permitir cambios en los campos de ID de empleado o número de la Seguridad Social. Por tanto, necesitará un medio para crear un objeto de empleado sin modificaciones, excepto por los métodos permitidos.

Encapsulación: datos privados, métodos públicos

Una forma de ocultar los detalles de implantación es declarar todos los campos como `private`.

```
1 public class CheckingAccount {
2     private int custID;
3     private String name;
4     private double amount;
5     public CheckingAccount {
6     }
7     public void setAmount (double amount) {
8         this.amount = amount;
9     }
10     public double getAmount () {
11         return amount;
12     }
13     //... other public accessor and mutator methods
14 }
```

La declaración de campos como `private` evita que se pueda acceder directamente a estos datos desde una instancia de clase.

```
// illegal!
ca.amount =
1_000_000_000.00;
```

ORACLE

En este ejemplo, los campos `custID`, `name` y `amount` ahora están marcados como `private`, lo que hace que sean invisibles fuera de los métodos de la propia clase.

Modificadores de acceso públicos y privados

- La palabra clave `public`, que se aplica a campos y métodos, permite a cualquier clase de cualquier paquete acceder al campo o al método.
- La palabra clave `private`, que se aplica a campos y métodos, permite el acceso solo a otros métodos de la propia clase.

```
CheckingAccount chk = new CheckingAccount ();
chk.amount = 200; // Compiler error - amount is a private field
chk.setAmount (200); // OK
```

- La palabra clave `private` también se puede aplicar a un método para ocultar un detalle de implantación.

```
// Called when a withdrawal exceeds the available funds
private void applyOverdraftFee () {
    amount += fee;
}
```

ORACLE

Revisión de la clase Employee

La clase `Employee` utiliza actualmente el acceso de tipo `public` para todos sus campos. Para encapsular los datos, convierta los campos en `private`.

```
package come.example.model;
public class Employee {
    private int empId;
    private String name;
    private String ssn;
    private double salary;
    //... constructor and methods
}
```

Paso 1 de la encapsulación: ocultar los datos (campos).

Asignación de nombres de métodos: recomendaciones

Si bien los campos ahora están ocultos mediante el acceso `private`, hay algunos problemas con la clase `Employee` actual.

- Los métodos setter (actualmente acceso de tipo `public`) permiten a cualquier otra clase cambiar el ID, el SSN y el salario (aumentarlo o reducirlo).
- La clase actual no representa realmente las operaciones definidas en el diseño de la clase `Employee` original.
- Dos recomendaciones para los métodos:
  - Oculte todos los detalles de implantación que pueda.
  - Asigne al método un nombre que identifique claramente su uso o funcionalidad.
- En el modelo original de la clase `Employee` se han realizado las operaciones de cambio de nombre y subida de salario.

ORACLE

Selección de métodos bienintencionados

Al igual que los campos deben definir de forma clara el tipo de datos que almacenan, los métodos deben identificar claramente las operaciones que realizan. Una de las formas más sencillas de mejorar la lectura del código (código Java o de cualquier otro tipo) es escribir nombres de métodos que identifiquen claramente su función.

Clase `Employee` refinada

```
1 package com.example.domain;
2 public class Employee {
3     // private fields ...
4     public Employee () {
5     }
6     // Remove all of the other setters
7     public void setName(String newName) {
8          if (newName != null) {
9              this.name = newName;
10          }
11     }
12
13     public void raiseSalary(double increase) {
14         this.salary += increase;
15     }
16 }
```

Paso 2 de la encapsulación: estos nombres de métodos tienen sentido en el contexto de una clase Employee.

ORACLE

Los métodos setter actuales de la clase permiten a cualquier clase que use una instancia de `Employee` modificar los campos ID, Salary y SSN del objeto. Desde un punto de vista empresarial, no desearía que estas operaciones se realizasen en un objeto de empleado. Una vez creado el empleado, estos campos deben ser inmutables (no se permiten cambios).

El modelo `Employee`, según la definición de la diapositiva titulada “Encapsulación: ejemplo” solo incluía dos operaciones: una para cambiar el nombre de un empleado (como resultado de un matrimonio o un divorcio) y otra para aumentar el salario de un empleado.

Para refinar la clase `Employee`, el primer paso es eliminar los métodos setter y crear métodos que identifiquen claramente su finalidad. Aquí hay dos métodos, uno para cambiar el nombre de un empleado (`setName`) y el otro para aumentar el salario de un empleado (`raiseSalary`).

Tenga en cuenta que la implantación del método `setName` prueba el parámetro de cadena transferido para asegurarse de que la cadena no es un valor null. El método puede realizar más comprobaciones si es necesario.

Haga que las clases sean lo más inmutables posibles

```
1 package com.example.domain;
2 public class Employee {
3     // private fields ...
4     // Create an employee object
5     public Employee (int empId, String name,
6                      String ssn, double salary) {
7         this.empId = empId;
8         this.name = name;
9         this.ssn = ssn;
10        this.salary = salary;
11     }
12
13     public void setName(String newName) { ... }
14
15     public void raiseSalary(double increase) { ... }
16 }
```

Paso 3 de la encapsulación: elimine el constructor por defecto; implante un constructor para definir el valor de todos los campos.

ORACLE

Recomendación: inmutabilidad

Por último, dado que la clase ya no tiene métodos setter, necesita un modo de definir el valor inicial de los campos. La respuesta es transferir el valor de cada campo en la construcción del objeto. Al crear un constructor que tome todos los campos como argumentos, puede garantizar que una instancia de `Employee` se rellene totalmente con datos *antes* de convertirse en un objeto de empleado válido. Este constructor *sustituye* al constructor sin argumentos.

Con los permisos adecuados, el usuario de su clase podría transferir valores null y, por ello, debería determinar si desea comprobar esos valores en el constructor. En lecciones posteriores se abordarán las estrategias para manejar esos tipos de situaciones.

La eliminación de los métodos setter y la sustitución del constructor sin argumentos también garantiza que los campos Employee ID y Social Security Number (SSN) de la instancia de `Employee` sean inmutables.

Creación de subclases

Ha creado una clase Java para modelar los datos y las operaciones de un objeto `Employee`. Ahora suponga que deseara especializar los datos y las operaciones para describir un objeto `Manager`.

```
1  package com.example.domain;
2  public class Manager {
3      private int empId;
4      private String name;
5      private String ssn;
6      private double salary;
7      private String deptName;
8      public Manager () {  }
9      // access and mutator methods...
10 }
```

un momento...
este código resulta muy familiar....

ORACLE

Especialización con las subclases Java

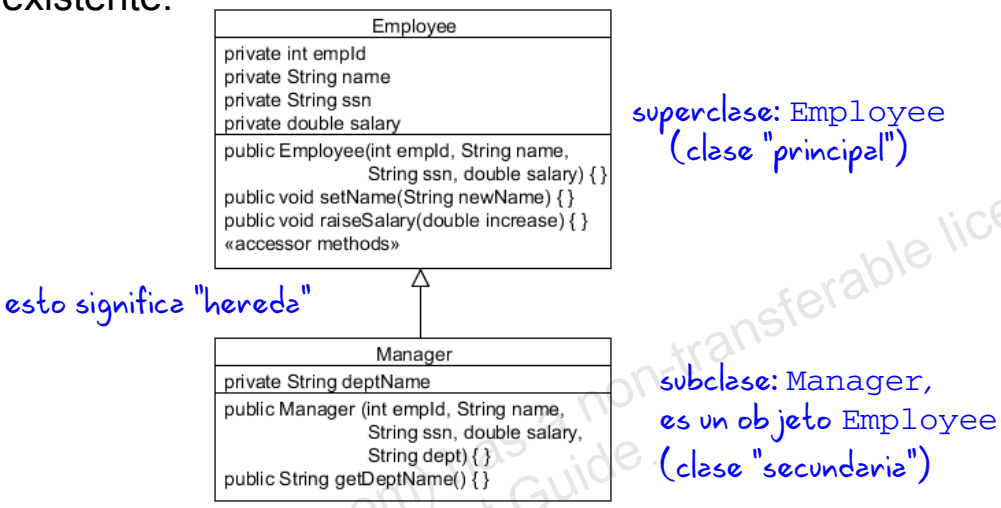
La clase `Manager` mostrada aquí se parece mucho a la clase `Employee`, pero con cierta especialización. Un objeto `Manager` también incluye un departamento, con su correspondiente nombre. Como resultado, también puede que haya más operaciones.

Lo que esto demuestra es que un objeto `Manager` es de tipo `Employee`, pero un objeto `Employee` con más funciones.

Sin embargo, si debiéramos definir clases Java de esta forma, habría mucho código redundante.

Subclases

En un lenguaje orientado a los objetos como Java, las subclases se usan para definir una nueva clase en relación con una existente.



ORACLE

Un programa Java simple

Cuando una clase existente tiene subclases, la nueva clase creada se dice que hereda las características de la otra clase. A esta nueva clase se la denomina *subclase* y es una especialización de la superclase. Todos los campos y los métodos no privados de la superclase forman parte de la subclase.

Por tanto, en este diagrama, una clase `Manager` obtiene `empId`, `name`, `SSN`, `salary` y los demás métodos públicos de `Employee`.

Es importante tener en cuenta que, si bien `Manager` especializa `Employee`, un objeto `Manager` sigue siendo un objeto `Employee`.

Nota: el término *subclase* es un poco confuso. La mayoría de la gente considera que el prefijo "*sub*-" denota "menos". Sin embargo, una subclase Java es la suma de sí misma y de su principal. Al crear una instancia de una subclase, la estructura de memoria resultante contiene todos los códigos de la clase principal, la clase principal anterior, y así sucesivamente en sentido ascendente en la jerarquía de clases hasta que alcanza la clase `Object`.

Subclase `Manager`

```
1 package com.example.domain;
2 public class Manager extends Employee {
3     private String deptName;
4     public Manager (int empId, String name,
5                     String ssn, double salary, String dept) {
6         super (empId, name, ssn, salary);
7         this.deptName = dept;
8     }
9
10     public String getDeptName () {
11         return deptName;
12    }
13    // Manager also gets all of Employee's public methods!
14 }
```

La palabra clave `super` se usa para llamar al constructor de la clase principal. Debe ser la primera sentencia del constructor.

ORACLE

Sintaxis Java para las subclases

La palabra clave `extends` se usa para crear una subclase.

La clase `Manager`, al ampliar la clase `Employee`, hereda todos los métodos y los campos de datos que no sean privados de `Employee`. Después de todo, si un superior también es un empleado, lo lógico es que `Manager` tenga los mismos atributos y operaciones que `Employee`.

Observe que la clase `Manager` declara su propio constructor. Los constructores *no* se heredan de la clase principal. En la siguiente diapositiva se ofrecen más detalles sobre este aspecto.

El constructor que `Manager` declara en la línea 4 llama al constructor de su clase principal, `Employee`, mediante la palabra clave `super`. Esto define el valor de todos los campos `Employee`: `id`, `name`, `ssn` y `salary`. `Manager` es una especialización de `Employee`, por lo que para crear un elemento `Manager` se necesita un nombre de departamento, que se asigna al campo `deptName` de la línea 7.

¿Qué otros métodos podría desear en un modelo de Manager? Tal vez desee un método que agregue un objeto `Employee` a este elemento `Manager`. Puede utilizar una matriz o una clase especial denominada *recopilación* para realizar un seguimiento de los empleados a los que supervisa el superior. Para obtener información sobre las recopilaciones, consulte la lección titulada “Elementos genéricos y recopilaciones”.

Los constructores no se heredan

Si bien una subclase hereda todos los métodos y campos de una clase principal, no hereda los constructores. Hay dos formas de obtener un constructor:

- Escribir su propio constructor.
- Usar el constructor por defecto.
  - Si no declara un constructor, se le proporcionará un constructor sin argumentos por defecto.
  - Si declara su propio constructor, el constructor por defecto ya no se proporcionará.

ORACLE

Constructores en subclases

Cada una de las subclases hereda los métodos y los campos no privados de su principal (superclase). Sin embargo, la subclase no hereda el constructor de su principal. Debe proporciona un constructor.

La *especificación de lenguaje Java* incluye la siguiente descripción:

“Las declaraciones de constructores no son miembros. Nunca se heredan y, por tanto, no están sujetas a ocultación o sustitución.”

Uso de `super`

Para crear una instancia de una subclase, normalmente resulta más fácil llamar al constructor de la clase principal.

- En su constructor, `Manager` llama al constructor de `Employee`.

```
super (empId, name, ssn, salary);
```

- La palabra clave `super` se usa para llamar al constructor de un principal.
- Debe ser la primera sentencia del constructor.
- Si no se proporciona, se inserta una llamada por defecto a `super()`.
- La palabra clave `super` también se puede usar para llamar al método de un principal o para acceder a un campo (no privado) de un principal.

ORACLE

La clase `Manager` declara su propio constructor y llama al constructor de la clase principal con la palabra clave `super`.

Nota: la llamada `super` del constructor del principal debe aparecer primero en el constructor.

La palabra clave `super` también se puede usar para llamar de forma explícita a los métodos de los campos de acceso o de la clase principal.

Creación de un objeto `Manager`

La operación de creación de un objeto `Manager` es similar a la de creación de un objeto `Employee`:

```
Manager mgr = new Manager (102, "Barbara Jones",
                  "107-99-9078", 109345.67, "Marketing");
```

- Todos los métodos `Employee` están disponibles para `Manager`:

```
mgr.raiseSalary (10000.00);
```

- La clase `Manager` define un nuevo método para obtener el valor Department Name:

```
String dept = mgr.getDeptName();
```

ORACLE

A pesar de que el archivo `Manager.java` no contiene todos los métodos de la clase `Employee.java` (explícitamente), se incluyen en la definición del objeto. Por tanto, tras crear una instancia de un objeto `Manager`, puede usar los métodos declarados en `Employee`.

También puede llamar a métodos que sean específicos de la clase `Manager`.

¿Qué es el polimorfismo?

El término *polimorfismo*, en su definición estricta, significa "muchas formas".

```
Employee emp = new Manager();
```

- Esta asignación es perfectamente válida. Un empleado puede ser un superior.
- Sin embargo, el siguiente código no se compila:

```
emp.setDeptName ("Marketing"); // compiler error!
```

- El compilador Java reconoce la variable `emp` solo como un objeto `Employee`. Debido a que la clase `Employee` no tiene un método `setDeptName`, muestra un error.

En lenguajes de programación orientados a objetos como Java, *polimorfismo* es la capacidad de hacer referencia a un objeto con su formato real o con un formato principal.

Esto resulta particularmente útil al crear un método de negocio de uso general. Por ejemplo, puede aumentar el salario de cualquier objeto `Employee` (principal o secundario) con solo transferir la referencia de objeto a un método de negocio de uso general que acepte un objeto `Employee` como argumento.

Sobrecarga de métodos

Su diseño puede llamar a varios métodos de la misma clase con el mismo nombre, pero con distintos argumentos.

```
public void print (int i)
public void print (float f)
public void print (String s)
```

- Java le permite reutilizar un nombre de método para más de un método.
- Se aplican dos reglas a los métodos sobrecargados:
  - Las listas de argumentos *deben* ser distintas.
  - Los tipos de retorno *pueden* variar.
- Por tanto, el siguiente ejemplo no es válido:

```
public void print (int i)
public String print (int i)
```

ORACLE

Puede que desee diseñar métodos con la misma finalidad (nombre de método), como `print`, para imprimir varios tipos distintos. Podría diseñar un método para cada tipo:

```
printInt(int i)
printFloat(float f)
printString(String s)
```

Sin embargo, esto sería tedioso y no estaría muy orientado a los objetos. En su lugar, puede crear un nombre de método reutilizable y simplemente cambiar la lista de argumentos. Este proceso se denomina *sobrecarga*.

Con los métodos de sobrecarga, las listas de argumentos deben ser distintas: en orden, número o tipo. Asimismo, los tipos de retorno pueden ser distintos. Sin embargo, no se permiten dos métodos con la misma lista de argumentos que solo difieran en el tipo de retorno.

Métodos con argumentos variables

Una variación a la sobrecarga de métodos es cuando necesita un método que tome cualquier número de argumentos del mismo tipo:

```
public class Statistics {
    public float average (int x1, int x2) {}
    public float average (int x1, int x2, int x3) {}
    public float average (int x1, int x2, int x3, int x4) {}
}
```

- Estos tres métodos sobrecargados comparten la misma funcionalidad. Estaría bien reducir estos métodos a uno solo.

```
Statistics stats = new Statistics ();
float avg1 = stats.average(100, 200);
float avg2 = stats.average(100, 200, 300);
float avg3 = stats.average(100, 200, 300, 400);
```

ORACLE

Métodos con un número de variable del mismo tipo

Un ejemplo de sobrecarga es cuando necesita proporcionar un juego de métodos sobrecargados que difieran en el número de argumentos del mismo tipo. Por ejemplo, suponga que desea contar con métodos para calcular una media. Tal vez desee calcular las medias para 2, 3 o 4 (o más) enteros.

Cada uno de estos métodos realiza un tipo similar de cálculo, la media de argumentos transferidos, como en este ejemplo:

```
public class Statistics {
    public float average(int x1, int x2) { return (x1 + x2) / 2; }
    public float average(int x1, int x2, int x3) {
        return (x1 + x2 + x3) / 3;
    }
    public float average(int x1, int x2, int x3, int x4) {
        return (x1 + x2 + x3 + x4) / 4;
    }
}
```

Java proporciona una sintaxis útil para reducir estos tres métodos a solo uno y proporcionar cualquier cantidad de argumentos.

Métodos con argumentos variables

- Java proporciona una función denominada *varargs* o *argumentos variables*.

La notación varargs considera al parámetro `nums` como una matriz.

```
1 public class Statistics {
2     public float average(int... nums) {
3         int sum = 0;
4         for (int x : nums) {  // iterate int array nums
5             sum += x;
6         }
7         return ((float) sum / nums.length);
8     }
9 }
```

- Tenga en cuenta que el argumento `nums` es realmente un objeto de matriz de tipo `int[]`. Esto permite al método iterarse y permitir cualquier cantidad de elementos.

ORACLE

Uso de argumentos variables

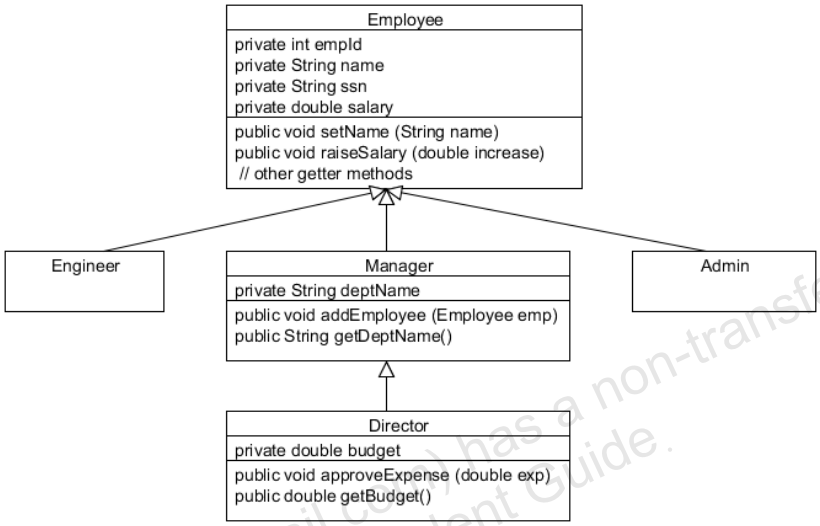
El método `average` mostrado en la diapositiva toma cualquier serie de argumentos enteros. La notación `(int... nums)` convierte la lista de argumentos transferidos al método `average` en un objeto de matriz del tipo `int`.

Nota: los métodos que usan varargs tampoco pueden tomar parámetros, si bien sí se pueden llamar a `average()`. En la API de NIO.2, en la lección titulada “E/S de archivos Java”, los varargs aparecerán como parámetros opcionales. Para tener esto en cuenta, podría reescribir el método `average` en la diapositiva de la siguiente forma:

```
public float average(int... nums) {
    int sum = 0; float result = 0;
        if (nums.length > 0) {
            for (int x : nums)    // iterate int array nums
                sum += x;
            result = (float) sum / nums.length;
        }
        return (result);
    }
}
```

Herencia única

El lenguaje de programación Java permite que una clase solo amplíe otra clase. A esto se le denomina *herencia única*.



Si bien Java no permite más de una clase en una subclase, el lenguaje proporciona funciones que permiten a varias clases implantar las funciones de otras clases. Verá este concepto en la lección sobre herencia.

La herencia única no impide el refinamiento continuado y la especialización de las clases mostrados anteriormente.

En el diagrama que se muestra en la diapositiva, un superior puede tener empleados y un director tiene un presupuesto y puede aprobar gastos.

Resumen

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

- Crear clases Java simples
- Usar la encapsulación en el diseño de clases Java
- Modelar problemas de negocio con clases Java
- Convertir las clases en inmutables
- Crear y usar subclases Java
- Sobrecargar métodos
- Usar métodos de argumentos variables

ORACLE

Prueba

Teniendo en cuenta el diagrama de la diapositiva titulada "Herencia única" y las siguientes sentencias Java, ¿cuál es la sentencia que *no* se compila?

```
Employee e = new Director();
Manager m = new Director();
Admin a = new Admin();
```

a. `e.addEmployee (a);`
b. `m.addEmployee(a);`
c. `m.approveExpense(100000.00);`
d. Ninguna de ellas se compila.

Prueba

Tenga en cuenta las siguientes clases que no se compilan:

```
public class Account {
    private double balance;
    public Account(double balance) { this.balance = balance; }
    //... getter and setter for balance
}
public class Savings extends Account {
    private double interestRate;
    public Savings(double rate) { interestRate = rate; }
}
```

¿Con qué corrección se podrían compilar estas clases?

a. Agregar un constructor sin argumentos a `Savings`.
b. Llamar al método `setBalance` de `Account` desde `Savings`.
c. Cambiar el acceso de `interestRate` a `public`.
d. Agregar un constructor a `Savings` que llame al constructor de `Account` con `super`.

ORACLE

Prueba

¿En cuáles de las siguientes declaraciones se demuestra la aplicación de unas convenciones de nomenclatura Java correctas?

a. `public class repeat { }`
b. `public void Screencoord (int x, int y){}`
c. `private int XCOORD;`
d. `public int calcOffset (int xCoord, int yCoord) { }`

ORACLE

Prueba

¿Qué cambios realizaría para convertir a esta clase en inmutable? (Seleccione todas las respuestas posibles).

```
public class Stock {
    public String symbol;
    public double price;
    public int shares;
    public double getStockValue() { }
    public void setSymbol(String symbol) { }
    public void setPrice(double price) { }
    public void setShares(int number) { }
}
```

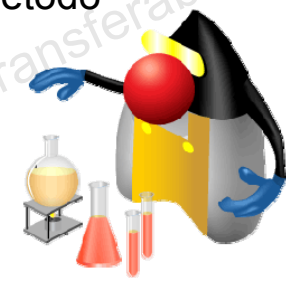
a. Convertir los campos `symbol`, `shares` y `price` en `private`.
b. Eliminar `setSymbol`, `setPrice` y `setShares`.
c. Convertir el método `getStockValue` en `private`.
d. Agregar un constructor que tome `symbol`, `shares` y `price` como argumentos.

ORACLE

Visión general de la práctica 3-1: Creación de subclases

En esta práctica, se abordan los siguientes temas:

- Aplicación de los principios de encapsulación a la clase `Employee` que ha creado en la práctica anterior
- Creación de subclases de `Employee`, incluidas `Manager`, `Engineer` y Administrative assistant (`Admin`)
- Creación de una subclase de `Manager` denominada `Director`
- Creación de una clase de prueba con un método `main` para probar las clases nuevas



ORACLE

(Opcional) Visión general de la práctica 3-2: Adición de una clase Staff a una clase Manager

En esta práctica, se abordan los siguientes temas:

- Creación de una matriz de `Employees` denominada `staff`
- Creación de un método para agregar un empleado al elemento del superior `staff`
- Creación de un método para eliminar un empleado del elemento del superior `staff`

ORACLE

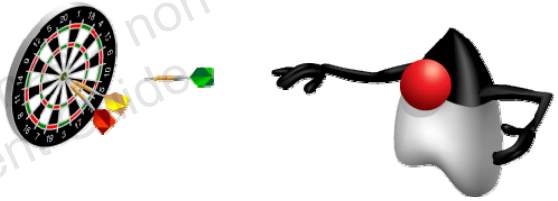
4

Diseño de clases Java

Objetivos

Al finalizar esta lección, debería estar capacitado para lo siguiente:

- Usar niveles de acceso: `private`, `protected`, el nivel por defecto y `public`.
- Sustituir métodos
- Sobrecargar constructores y otros métodos de la forma adecuada
- Usar el operador `instanceof` para comparar tipos de objeto
- Usar la llamada al método virtual
- Usar conversiones ascendentes y descendentes
- Sustituir métodos de la clase `Object` para mejorar la funcionalidad de la clase



ORACLE

Uso del control de acceso

Ha visto las palabras clave `public` y `private`. Hay cuatro niveles de acceso que se pueden aplicar a los métodos y los campos de datos. En la siguiente tabla se muestra el acceso a un campo o método marcado con el modificador de acceso en la columna izquierda.

Modificador (palabra clave)	Misma clase	Mismo paquete	Subclase de otro paquete	Universo
`private`	Sí			
`por defecto`	Sí	Sí		
`protected`	Sí	Sí	Sí *	
`public`	Sí	Sí	Sí	Sí

Las clases pueden ser por defecto (sin modificador) o de tipo `public`.

ORACLE

Las palabras clave de modificador de acceso de esta tabla son `private`, `protected` y `public`. Cuando falta una palabra clave, se aplica el modificador de acceso *por defecto*.

La palabra clave `private` permite el máximo control de acceso a los campos y métodos. Con `private`, solo se puede acceder a un campo de datos o método de la misma clase Java.

La palabra clave `public` permite el máximo acceso a campos y métodos, haciendo que se pueda acceder a ellos en cualquier parte: en la clase, paquete, subclases y cualquier otra clase.

La palabra clave `protected` sea aplica para mantener el acceso en el paquete y la subclase. Los campos y métodos que usan `protected` se dice que “permiten subclases”.

***Nota:** el tipo de acceso `protected` se ha ampliado a las subclases que residen en un paquete distinto de la clase propietaria de la función protegida. Como resultado, los campos o los métodos protegidos permiten realmente un mayor acceso que los marcados con un control de acceso por defecto. Debe usar el tipo de acceso de tipo `protected` cuando sea adecuado para una subclase de una clase, pero no las clases no relacionadas.

Control de acceso protegido: ejemplo

```
1 package demo;
2 public class Foo {
3     protected int result = 20;
4     int other = 25;
5 }
```

declaración que permite subclases

```
1 package test;
2 import demo.Foo;
3 public class Bar extends Foo {
4      private int sum = 10;
5      public void reportSum () {
6          sum += result;
7          sum += other;
8      }
9 }
```

error del compilador

ORACLE

En este ejemplo, hay dos clases en dos paquetes. La clase `Foo` está en el paquete `demo` y declara un campo de datos denominado `result` con un modificador de acceso `protected`.

En la clase `Bar`, que amplía `Foo`, hay un método, `reportSum`, que agrega el valor de `result` a `sum`. A continuación, el método intenta agregar el valor de `other` a sum. El campo `other` se declara mediante el modificador por defecto y esto genera un error de compilador. ¿Por qué?

Respuesta: el campo `result`, declarado como campo `protected`, está disponible para todas las subclases, incluso aunque estén en otro paquete. El campo `other` se declara como que usa el acceso por defecto y solo está disponible para clases y subclases declaradas en el mismo paquete.

Este ejemplo se ha extraído del proyecto `JavaAccessExample`.

Sombra de campos: ejemplo

```
1 package demo;
2 public class Foo2 {
3     protected int result = 20;
4 }
```

```
1 package test;
2 import demo.Foo2;
3 public class Bar2 extends Foo2 {
4     private int sum = 10;
5     private int result = 30;
6     public void reportSum () {
7         sum += result;
8     }
9 }
```

El campo `result` es una sombra del campo principal.

ORACLE

En este ejemplo, la clase `Foo2` declara el campo `result`. Sin embargo, la clase `Bar2` declara su propio campo `result`. La consecuencia es que el campo `result` de la clase `Foo2` tiene una sombra en el campo `result` de la clase `Bar2`. ¿Qué es `sum` en este ejemplo? `sum` ahora es 40 (10 + 30). Los IDE modernos (como NetBeans) detectan las sombras y generan una advertencia.

De los métodos con el mismo nombre no se crea una sombra, sino que se sustituyen. El proceso de sustitución se explica más adelante en esta misma lección.

Control de acceso: recomendación

Una buena práctica al trabajar con campos es hacer que sean tan poco accesibles como sea posible y especificar claramente el uso de los campos en los métodos.

```
1 package demo;
2 public class Foo3 {
3    private int result = 20;
4    protected int getResult() { result = value; }
5 }
```

```
1 package test;
2 import demo.Foo3;
3 public class Bar3 extends Foo3 {
4     private int sum = 10;
5     public void reportSum() {
6         sum += getResult();
7     }
8 }
```

ORACLE

En la diapositiva se muestra una versión ligeramente modificada del ejemplo con la palabra clave `protected`. Si la idea es limitar el acceso al resultado del campo a las clases del paquete y las subclases (protegido por paquete), debe asegurarse de que el acceso es explícito mediante la definición de un método escrito expresamente para el acceso a nivel de paquetes y subclases.

Sustitución de métodos

Considere un requisito para proporcionar una cadena que represente algunos detalles sobre los campos de la clase `Employee`.

```
1 public class Employee {
2     private int empId;
3     private String name;
4     // ... other fields and methods
5     public String getDetails () {
6         return "Employee id:  " + empId +
7                "  Employee name:" + name;
8     }
9 }
```

ORACLE

Si bien la clase `Employee` tiene getters para devolver valores para una sentencia print, estaría bien contar con un método de utilidad para obtener los detalles específicos sobre el empleado. Piense en un método que se agrega a la clase `Employee` para imprimir detalles sobre el objeto `Employee`.

Además de agregar campos o métodos a una subclase, también puede modificar o cambiar el comportamiento existente de un método del principal (superclase).

Tal vez desee especializar este método para describir un objeto `Manager`.

Sustitución de métodos

En la clase `Manager`, mediante la creación de un método con la misma firma que el método de la clase `Employee`, está *sustituyendo* el método `getDetails`:

```
1 public class Manager extends Employee {
2     private String deptName;
3     // ... other fields and methods
4     public String getDetails () {
5         return super.getDetails () +
6                 " Department: " + deptName;
7     }
8 }
```

Una subclase puede llamar a un método principal usando la palabra clave `super`.

Cuando se sustituye un método, sustituye el método de la clase (principal) de la superclase.

A este método se le llama para cualquier instancia de `Manager`.

Una llamada con el formato `super.getDetails()` llama al método `getDetails` de la clase principal.

Nota: si, por ejemplo, una clase declara dos métodos públicos con el mismo nombre y una subclase sustituye uno de ellos, la subclase seguirá heredando el otro método.

Llamada a un método sustituido

- Con los ejemplos anteriores de `Employee` y `Manager`:

```
Employee e = new Employee (101, "Jim Smith", "011-12-2345",
100_000.00);
Manager m = new Manager (102, "Joan Kern", "012-23-4567",
110_450.54, "Marketing");
System.out.println (e.getDetails());
System.out.println (m.getDetails());
```

- Se llama al método `getDetails` correcto de cada clase:

```
Employee id: 101 Employee name: Jim Smith
Employee id: 102 Employee name: Joan Kern Department: Marketing
```


Durante el tiempo de ejecución, Java Virtual Machine llama al método `getDetails` de la clase adecuada. Si comenta el método `getDetails` de la clase `Manager` que se muestra en la diapositiva anterior, ¿qué sucede cuando se llama a `m.getDetails()`?

Respuesta: recuerde que los métodos se heredan de la clase principal. Por tanto, en tiempo de ejecución, se ejecuta el método `getDetails` de la clase principal (`Employee`).

Llamada al método virtual

- ¿Qué sucede si tiene lo siguiente?

```
Employee e = new Manager (102, "Joan Kern", "012-23-4567",
110_450.54, "Marketing");
System.out.println (e.getDetails());
```

- Durante la ejecución, se determina que el tipo de tiempo de ejecución del objeto es un objeto `Manager`:

```
Employee id: 102 Employee name: Joan Kern Department: Marketing
```

- El compilador no tiene fallos porque la clase `Employee` tiene un método `getDetails` y, en tiempo de ejecución, al método que se ejecuta se le hace referencia desde un objeto `Manager`.
- Se trata de un aspecto de polimorfismo denominado *llamada al método virtual*.

ORACLE

Comportamiento del compilador frente al comportamiento en tiempo de ejecución

El aspecto importante que se debe recordar es la diferencia entre el compilador (que comprueba que se pueda acceder a cada uno de los métodos y campos según la definición estricta de la clase) y el comportamiento asociado a un objeto determinado en tiempo de ejecución.

Esta distinción es un aspecto importante y potente del polimorfismo: el comportamiento de un objeto viene determinado por su referencia de tiempo de ejecución.

Debido a que el objeto que ha creado es un objeto `Manager`, en tiempo de ejecución, cuando se llama al método `getDetails`, la referencia de tiempo de ejecución es al método `getDetails` de una clase `Manager`, incluso aunque la variable `e` sea del tipo `Employee`.

A este comportamiento se le conoce como *llamada al método virtual*.

Nota: si es un programador de C++, obtiene este comportamiento en C++ solo si marca el método con la palabra clave de C++ `virtual`.

Accesibilidad de los métodos sustituidos

Un método sustituido no puede ser menos accesible que el método de la clase principal.

```
public class Employee {
    //... other fields and methods
    public String getDetails() { ... }
}
```

```
public class Manager extends Employee {
    //... other fields and methods
    private String getDetails() { //... }
}
```

```
public class OverriddingTest {
    Employee e = new Manager(102, "Joan Kern", "012-23-4567",
110_450.54, "Marketing");
        e.getDetails();  // illegal - method is private
    }
}
```

ORACLE

Para sustituir un método, el nombre y el orden de los argumentos deben ser idénticos.

Al cambiar el acceso del método `getDetails` de Manager a `private`, solo esa clase podrá ejecutar el método. Sin embargo, la semántica del lenguaje para ejecutar `e.getDetails` determina que se ejecuta el método `getDetails` de Manager. El resultado es un error de tiempo de ejecución.

¿Qué sucede si hace el método `Employee` de `getDetails` privado y el método `Manager` público?

Respuesta: debido a que el compilador comprueba los tipos, indica que está intentando acceder a un método privado en `Employee`.

Aplicación de polimorfismo

Suponga que se le solicita que cree una nueva clase que calcule las acciones otorgadas a los empleados según su salario y su rol (superior, ingeniero o administrador):

```
1 public class EmployeeStockPlan {
2    public int grantStock (Manager m) {
3       // perform a calculation for a Manager
4    }
5    public int grantStock (Engineer e) {
6       // perform a calculation for an Engineer
7    }
8    public int grantStock (Admin a) {
9       // perform a calculation for an Admin
10   }
11   //... one method per employee type
12}
```

no muy orientado a objetos.

ORACLE

Problema de diseño

¿Cuál es el problema en el ejemplo de la diapositiva? Cada uno de los métodos realiza el cálculo en función del tipo de empleado transferido y devuelve el número de acciones.

Tenga en cuenta lo que sucede si agrega dos o tres tipos de empleados adicionales. Necesitaría agregar tres métodos más y, posiblemente, replicar el código según la lógica de negocio necesaria para calcular las acciones.

Es obvio que no es una buena forma de solucionar este problema. Si bien el código funcionará, no es fácil de leer y es probable que cree mucho código duplicado.

Aplicación de polimorfismo

Una buena práctica consiste en transferir parámetros y escribir métodos que usen el formato más genérico del objeto posible.

```
public class EmployeeStockPlan {
   public int grantStock (Employee e) {
      // perform a calculation based on Employee data
   }
}
```

```
// In the application class
EmployeeStockPlan esp = new EmployeeStockPlan ():
Manager m = new Manager ();
int stocksGranted = grantStock (m);
...
```


Uso del formato más genérico

Una buena práctica consiste en diseñar y escribir métodos que adopten el formato más genérico del objeto posible.

En este caso, `Employee` es una buena clase base que tomar como punto de partida. Pero ¿cómo sabe el tipo de objeto transferido? La respuesta aparecerá en la siguiente diapositiva.

Uso de la palabra clave `instanceof`

El lenguaje Java proporciona la palabra clave `instanceof` para determinar un tipo de clase de objeto en tiempo de ejecución.

```
1 public class EmployeeStockPlan {
2    public int grantStock (Employee e) {
3       // perform a calculation based on Employee data
4       if (e instanceof Manager) {
5          // process Manager stock grant
6       } else if (e instanceof Engineer) {
7          // process Engineer stock grant
8       } else if (e instanceof Admin) {
9          // process Admin stock grant
10      } else {
11         // perhaps an error - a generic Employee?
12      }
13   }
14}
```

ORACLE

En este elemento `EmployeeStockPlan` modificado, el método `grantStock` utiliza `instanceof` para determinar el tipo de `Employee` transferido al método.

Otra perspectiva para este problema es usar la nueva función en JDK 7 para activar las cadenas:

```
String type = emp.getClass().getSimpleName();
switch (type) {
    case "Engineer":
        // process Engineer grant
    case "Admin":
        // process Admin grant
    case "Manager":
        // process Manager grant
    case "Director":
        // process Director grant
    default:
        // error result
}
```

Conversión de referencias de objetos

Después de usar el operador `instanceof` para verificar que el objeto recibido como argumento es una subclase, puede acceder a toda la funcionalidad del objeto convirtiendo la referencia:

```
1 public void modifyDeptForManager (Employee e, String dept) {
2    if (e instanceof Manager) {
3       Manager m = (Manager) e;
4       m.setDeptName (dept);
5    }
6 }
```

Sin la conversión a `Manager`, el método `setDeptName` no se compilaría.

ORACLE

Si bien una referencia a superclase genérica es útil para transferir objetos de un lado a otro, puede que tenga que usar un método de la subclase.

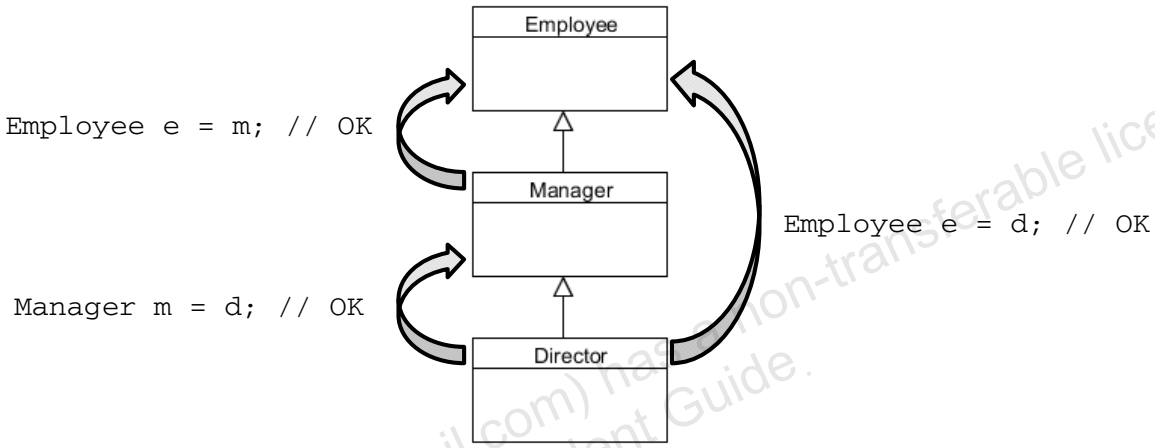
Por ejemplo, en la diapositiva, necesita el método `setDeptName` de la clase `Manager`. Para que el compilador no tenga fallos, puede convertir una referencia de la superclase genérica a la clase específica.

Sin embargo, hay reglas para convertir las referencias. Esto aparece en la siguiente diapositiva.

Conversión de reglas

Las conversiones ascendentes siempre están permitidas y en ellas no se necesita un operador cast.

```
Director d = new Director();
Manager m = new Manager();
```

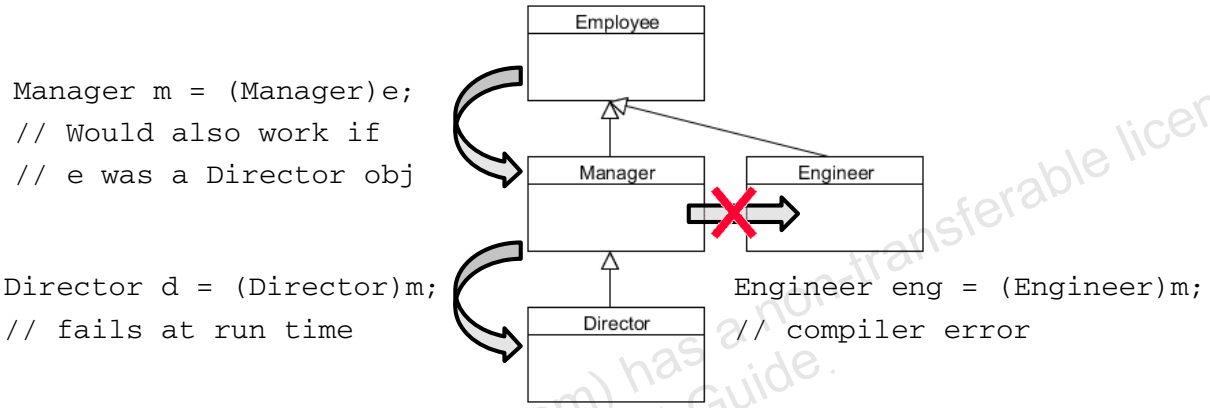


ORACLE

Conversión de reglas

En el caso de las conversiones descendentes, el compilador debe aceptar que la conversión es, al menos, posible.

```
Employee e = new Manager();
Manager m = new Manager();
```



ORACLE

Conversiones descendentes

Con una conversión descendente, el compilador simplemente determina si se puede realizar la conversión; si el destino de la conversión descendente es una subclase, puede que la conversión se pueda realizar correctamente.

Observe que, en tiempo de ejecución, la conversión produce una excepción `java.lang.ClassCastException` si la referencia del objeto es de una superclase y no del tipo de clase o una subclase.

La conversión de la variable `e` a una referencia `m` de Manager hace que el compilador no falle, porque `Manager` y `Employee` están en la misma jerarquía de clases, por lo que la conversión probablemente sea correcta. Esta conversión también funciona en tiempo de ejecución, porque resulta que la variable `e` es realmente un objeto `Manager`. Esta conversión también funcionaría en tiempo de ejecución si `e` apuntara a una instancia de un objeto `Director`.

La conversión de la variable `m` en una instancia `Director` hace que el compilador no falle, pero porque `m` es realmente una instancia de `Manager`; esta conversión falla en tiempo de ejecución con una excepción `ClassCastException`.

Por último, cualquier conversión que esté fuera de la jerarquía de clases fallará, como la conversión de una instancia de `Manager` en una instancia de `Engineer`. `Manager` y `Engineer` son ambos empleados, pero `Manager` no es un objeto `Engineer`.

Sustitución de métodos de objeto

Una de las ventajas de la herencia única es que cada una de las clases tiene un objeto principal por defecto. La clase raíz de cada clase Java es `java.lang.Object`.

- No es necesario que declare que la clase amplía `Object`. El compilador se encarga de esa tarea.

```
public class Employee { //... }
```

es equivalente a:

```
public class Employee extends Object { //... }
```

- La clase raíz contiene varios métodos que no son finales, pero hay tres que son importantes para pensar en la sustitución:
  - `toString`, `equals` y `hashCode`

Recomendación: sobrecargar métodos de objeto

La clase `java.lang.Object` es la clase raíz de todas las clases en el lenguaje de programación Java. Todas las clases incluirán por defecto la subclase `Object`.

`Object` define varios métodos que no son finales diseñados para que los sustituya la clase. Son: `equals`, `hashCode`, `toString`, `clone` y `finalize`. De ellos, hay tres métodos que debería intentar sustituir.

Método `Object toString`

Al método `toString` se le llama siempre que se transfiera una instancia de la clase a un método que tome un objeto `String`, como `println`:

```
Employee e = new Employee (101, "Jim Kern", ...)
System.out.println (e);
```

- Puede utilizar `toString` para proporcionar información de la instancia:

```
public String toString () {
    return "Employee id:   " + empId + "\n"
           "Employee name:" + name;
}
```

- Este enfoque para obtener detalles sobre la clase es mejor que crear su propio método `getDetails`.

ORACLE

Nota: en ocasiones puede que desee imprimir el nombre de la clase que está ejecutando un método. El método `getClass()` es un método `Object` que se usa para devolver la instancia de objeto `Class` y el método `getName()` proporciona el nombre totalmente cualificado de la clase de tiempo de ejecución.

```
getClass().getName(); // returns the name of this class instance
```

Estos métodos están en la clase `Object`.

Método `Object equals`

El método `Object equals` solo compara referencias de objetos.

- Si hay dos objetos x e y en cualquier clase, x es igual a y si y solo si x e y hacen referencia al mismo objeto.
- Ejemplo:

```
Employee x = new Employee (1,"Sue","111-11-1111",10.0);
Employee y = x;
x.equals (y); // true
Employee z = new Employee (1,"Sue","111-11-1111",10.0);
x.equals (z); // false!
```

- Ya que lo que realmente se desea es probar el contenido del objeto Employee, es necesario sustituir el método `equals`:

```
public boolean equals (Object o) { ... }
```

ORACLE

El método `equals` de `Object` determina (por defecto) solo si los valores de dos referencias de objetos apuntan al mismo objeto. Básicamente, la prueba en la clase `Object` es simplemente la siguiente:

Si x == y, devolver true.

En el caso de un objeto (como el objeto `Employee`) que contiene valores, esta comparación no es suficiente, particularmente si deseamos asegurarnos de que solo hay un empleado con un ID concreto.

Sustitución de `equals` en `Employee`

Un ejemplo de sustitución del método `equals` en la clase `Employee` compara todos los campos para ver si tienen igualdad:

```
1 public boolean equals (Object o) {
2     boolean result = false;
3     if ((o != null) && (o instanceof Employee)) {
4         Employee e = (Employee)o;
5         if ((e.empId == this.empId) &&
6             (e.name.equals(this.name)) &&
7             (e.ssn.equals(this.ssn)) &&
8             (e.salary == this.salary)) {
9                result = true;
10          }
11     }
12     return result;
13 }
```

ORACLE

Esta simple prueba `equals` comprueba en primer lugar que el objeto transferido no es nulo y, a continuación, prueba para asegurarse de que se trata de una instancia de una clase `Employee` (todas las subclases también son empleados, por lo que funciona). A continuación, el elemento `Object` se convierte en `Employee` y cada uno de los campos de `Employee` se comprueban para ver si existe igualdad.

Nota: para los tipos `String`, debe utilizar el método `equals` para probar la igualdad de las cadenas carácter a carácter.

Sustitución de `Object` `hashCode`

El contrato general de `Object` indica que si dos objetos se consideran iguales (con el método `equals`), el código hash del entero devuelto para los dos objetos también debe ser igual.

```
1 // Generated by NetBeans
2 public int hashCode() {
3     int hash = 7;
4     hash = 83 * hash + this.empId;
5     hash = 83 * hash + Objects.hashCode(this.name);
6     hash = 83 * hash + Objects.hashCode(this.ssn);
7     hash = 83 * hash + (int)
(Double.doubleToLongBits(this.salary) ^
(Double.doubleToLongBits(this.salary) >>> 32));
8     return hash;
9 }
```

ORACLE

Sustitución de `hashCode`

La documentación Java para la clase `Object` indica:

"... Normalmente es necesario sustituir el método `hashCode` siempre que se sustituya este método [`equals`], para mantener el contrato general del método `hashCode`, que indica que los objetos iguales deben tener códigos hash iguales."

El método `hashCode` se usa junto con el método `equals` en recopilaciones basadas en hash, como `HashMap`, `HashSet` y `Hashtable`.

Este método se puede malinterpretar fácilmente, por lo que debe tener cuidado. Lo bueno es que los IDE, como NetBeans, pueden generar `hashCode`.

Para crear su propia función hash, los siguientes pasos le permitirán aproximar un valor hash razonable para instancias iguales y distintas:

1) Empezar con una constante de entero que no sea cero. Con los números primos se producen menos colisiones de hashcode.
2) Para cada campo usado en el método `equals`, calcule un código hash `int` para el campo. Observe que para el elemento `Strings`, puede utilizar el elemento `hashCode` del objeto `String`.
3) Agregar los códigos hash calculados juntos.
4) Devolver el resultado.

Resumen

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

- Usar niveles de acceso: `private`, `protected`, el nivel por defecto y `public`
- Sustituir métodos
- Sobrecargar constructores y otros métodos de la forma adecuada
- Usar el operador `instanceof` para comparar tipos de objeto
- Usar la llamada al método virtual
- Usar conversiones ascendentes y descendentes
- Sustituir métodos de la clase `Object` para mejorar la funcionalidad de la clase

ORACLE

Prueba

Suponga que tiene una clase `Account` con un método `withdraw()` y una clase `Checking` que amplía `Account`, que declara su propio método `withdraw()`. ¿Cuál es el resultado del fragmento de código siguiente?

```
1  Account acct = new Checking();
2  acct.withdraw(100);
```

a. El compilador avisa sobre un fallo en la línea 1.
b. El compilador avisa sobre un fallo en la línea 2.
c. Error de tiempo de ejecución: asignación incompatible (línea 1).
d. Se ejecuta el método `Account.withdraw()`.
e. Se ejecuta el método `Checking.withdraw()`.

ORACLE

Prueba

Suponga que tiene una clase `Account` y una clase `Checking` que amplía `Account`. Se ejecutará el cuerpo de la sentencia `if` en la línea 2.

```
1  Account acct = new Checking();
2  if (acct instanceof Checking) { // will this block run? }
```

a. Verdadero
b. Falso

Prueba

Suponga que tiene una clase `Account` y una clase `Checking` que amplía `Account`. También tiene una clase `Savings`, que amplía `Account`. ¿Cuál es el resultado del código siguiente?

```
1  Account acct1 = new Checking();
2  Account acct2 = new Savings();
3  Savings acct3 = (Savings)acct1;
```

a. `acct3` contiene la referencia a `acct1`.
b. Se produce una excepción `ClassCastException` de tiempo de ejecución.
c. El compilador avisa sobre un fallo en la línea 2.
d. El compilador avisa sobre un fallo en la conversión de la línea 3.

ORACLE

Prueba

```
1 package com.bank;
2 public class Account {
3     double balance;
4 }
```

```
10 package com.bank.type;
11 import com.bank.Account;
12 public class Savings extends Account {
13     private double interest;
14     Account acct = new Account();
15     public double getBalance (){ return (interest + balance); }
16 }
```

¿Qué cambio haría que se compilase este código?

a. Declarar `balance` como `private` en la línea 3.
b. Declarar `balance` como `protected` en la línea 3.
c. Sustituir `balance` por `acct.balance` en la línea 15.
d. Sustituir `balance` por `Account.balance` en la línea 15.

ORACLE

Visión general de la práctica 4-1: Sustitución de métodos y aplicación de polimorfismo

En esta práctica, se abordan los siguientes temas:

- Modificación de las clases `Employee`, `Manager` y `Director`; sustitución del método `toString()`
- Creación de una clase `EmployeeStockPlan` con un método de otorgamiento de acciones que usa la palabra clave `instanceof`

ORACLE

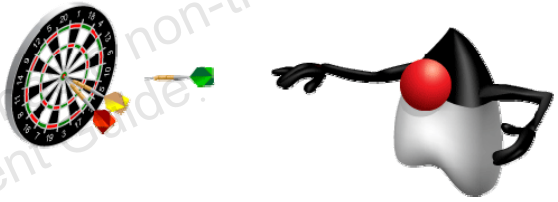
5

Diseño de clases avanzadas

Objetivos

Al finalizar esta lección, debería estar capacitado para lo siguiente:

- Diseñar clases base de uso general mediante clases abstractas
- Crear clases y subclases Java abstractas
- Modelar problemas de negocio mediante las palabras clave `static` y `final`
- Implantar el patrón de diseño singleton
- Distinguir entre clases de nivel superior y anidadas



ORACLE

Modelación de problemas de negocio con clases

La herencia (o creación de subclases) es una función esencial del lenguaje de programación Java. La herencia permite la reutilización de código mediante:

- Herencia de métodos: las subclases evitan la duplicación del código al heredar las implantaciones de métodos.
- Generalización: el código que está diseñado para basarse en el tipo más genérico posible es más fácil de mantener.

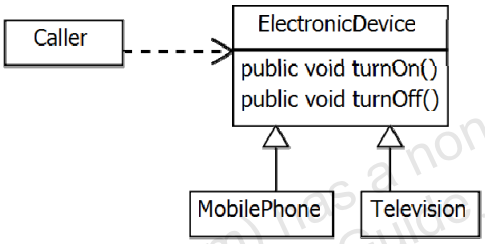


Diagrama de herencia de clases

Herencia de clases

Al diseñar una solución orientada a objetos, debe intentar evitar la duplicación del código. Una técnica para evitar la duplicación consiste en crear métodos y clases de biblioteca. Las bibliotecas funcionan como un punto central para incluir código de uso común. Otra técnica para evitar la duplicación de código es usar la herencia de clases. Cuando se identifique un tipo base compartido entre dos clases, cualquier código compartido se puede colocar en una clase principal.

Cuando sea posible, utilice referencias de objetos del tipo base más genérico posible. En Java, la generalización y la especialización permiten la reutilización mediante la herencia de métodos y la llamada de métodos virtuales (VMI). VMI, en ocasiones denominado “enlace tardío”, permite a un emisor de llamada llamar de forma dinámica a un método siempre que este se haya declarado en un tipo base genérico.

Activación de la generalización

La codificación en un tipo base común permite introducir nuevas subclases con pocas modificaciones o ninguna de cualquier código que dependa del tipo base más genérico.

```
ElectronicDevice dev = new Television();
dev.turnOn(); // all ElectronicDevices can be turned on
```

Use siempre el tipo de referencia más genérico posible.

Codificación de la generalización

Use siempre el tipo de referencia más genérico posible. Los IDE Java pueden contener herramientas de refactorización que facilitan el cambio de referencias existentes por un tipo base más genérico.

Identificación de la necesidad de clases abstractas

Las subclases no tienen que heredar una implantación de método si el método está especializado.

```
public class Television extends ElectronicDevice {

    public void turnOn() {
        changeChannel(1);
        initializeScreen();
    }
    public void turnOff() {}

    public void changeChannel(int channel) {}
    public void initializeScreen() {}

}
```

ORACLE

Implantaciones de métodos

Cuando las clases hermanas tienen un método común, este se suele colocar en una clase principal. En algunas circunstancias, sin embargo, la implantación de la clase principal siempre se tendrá que sobrescribir con una implantación especializada.

En estos casos, la inclusión del método en una clase principal tiene tanto ventajas como desventajas. Permite el uso de tipos de referencias genéricas, pero los desarrolladores pueden fácilmente olvidar especificar la implantación especializada de las subclases.

Definición de clases abstractas

Una clase se puede declarar como abstracta mediante el modificador de nivel de clase `abstract`.

```
public abstract class ElectronicDevice { }
```

- Una clase abstracta puede tener una subclase.

```
public class Television extends ElectronicDevice { }
```

- Una clase abstracta no se puede instanciar.

```
ElectronicDevice dev = new ElectronicDevice(); // error
```


La declaración de una clase como abstracta evita que se cree cualquier instancia de dicha clase. Instanciar una clase abstracta es un error de tiempo de compilación. Una clase abstracta se ampliará normalmente mediante una clase secundaria y se puede usar como tipo de referencia.

Definición de métodos abstractos

Un método se puede declarar como abstracto mediante el modificador de nivel de método `abstract`.

```
public abstract class ElectronicDevice {

   public abstract void turnOn();
   public abstract void turnOff();
}
```

Sin corchetes

Un método abstracto:

- No puede tener un cuerpo de método.
- Se debe declarar en una clase abstracta.
- Se sobrescribe en subclases.

Herencia de métodos abstractos

Cuando una clase secundaria hereda un método abstracto, está heredando una firma de método, pero sin implantación. Por este motivo, no se permiten corchetes al definir los métodos abstractos.

Un método abstracto es una forma de garantizar que cualquier clase secundaria contendrá un método con una firma coincidente.

Validación de clases abstractas

Las siguientes reglas adicionales se aplican cuando se usan las clases y los métodos abstractos:

- Una clase abstracta puede tener cualquier número de métodos abstractos y no abstractos.
- Al heredar de una clase abstracta, debe realizar una de las siguientes acciones:
  - Declarar la clase secundaria como abstracta.
  - Sustituya todos los métodos abstractos heredados de la clase principal. De lo contrario, se producirá un error de tiempo de compilación.

```
error: Television is not abstract and does not override
abstract method turnOn() in ElectronicDevice
```

ORACLE

Uso de las clases abstractas

Si bien se puede evitar implantar un método abstracto declarando clases secundarias como abstractas, esto solo sirve para retrasar lo inevitable. Las aplicaciones necesitan métodos no abstractos para crear objetos. Utilice métodos abstractos para describir la funcionalidad que se necesita en las clases secundarias.

Prueba

Para que la compilación se realice correctamente, un método abstracto no debe tener:

a. Un valor de retorno
b. Una implantación de método
c. Parámetros del método
d. Acceso `private`

Palabra clave `static`

El modificador `static` se usa para declarar campos y métodos como recursos de nivel de clase. Los miembros de clase estáticos:

- Se pueden usar sin instancias de objeto.
- Se usan cuando un problema se soluciona mejor sin objetos.
- Se usan cuando hay objetos del mismo tipo que deben compartir campos.
- *No* se deben usar para no usar las funciones orientadas a objetos de Java a menos que haya un motivo justificado.

ORACLE

Java: orientado a objetos por diseño

El lenguaje de programación Java se ha diseñado como un lenguaje orientado a objetos, a diferencia de lenguajes como Objective-C y C++, que han heredado el diseño de procedimientos de C. Al desarrollar en Java, siempre debe intentar diseñar una solución orientada a objetos.

Métodos estáticos

Los métodos estáticos son métodos que se pueden llamar incluso si la clase en la que se hayan declarado no se ha instanciado. Los métodos estáticos:

- Se denominan métodos de clase.
- Son útiles para las API que no están orientadas a objetos.
  - `java.lang.Math` contiene muchos métodos estáticos.
- Se suelen usar en lugar de los constructores para realizar tares relacionadas con la inicialización de objetos.
- No pueden acceder a miembros no estáticos de la misma clase.
- Se pueden ocultar en subclases, pero no se pueden sustituir.
  - Sin llamada a método virtual.

ORACLE

Métodos de fábrica

En lugar de llamar directamente a constructores, a menudo usará métodos estáticos para recuperar referencias a objetos. A menos que se espere que pase algo inesperado, se crea un nuevo objeto siempre que se llama a un constructor. Un método de fábrica estático podría mantener una caché de objetos para reutilizarlos o crear nuevas instancias si se agotara la caché. Un método de fábrica también puede producir un objeto que subclasifique el tipo de retorno del método.

Ejemplo:

```
NumberFormat nf = NumberFormat.getInstance();
```

Implantación de métodos estáticos

```
public class StaticErrorClass {
    private int x;

    public static void staticMethod() {
        x = 1; // compile error
        instanceMethod(); // compile error
    }

    public void instanceMethod() {
        x = 2;
    }
}
```


Limitaciones de métodos estáticos

Los métodos estáticos se pueden usar antes de crear cualquier instancia de su clase delimitadora. Desde el punto de vista cronológico, esto significa que en una Java Virtual Machine en ejecución puede que no haya incidencias de las variables de instancias de las clases contenedoras. Los métodos estáticos no pueden nunca acceder a las variables de instancias de sus clases delimitadoras ni llamar a sus métodos no estáticos.

Llamada a métodos estáticos

```
double d = Math.random();
StaticUtilityClass.printMessage();
StaticUtilityClass uc = new StaticUtilityClass();
uc.printMessage(); // works but misleading
sameClassMethod();
```

Al llamar a los métodos estáticos, debería:

- Cualificar la ubicación del método con un nombre de clase si el método se encuentra en otra clase distinta a la del emisor de la llamada
  - No es necesario para métodos de la misma clase
- Evitar el uso de una referencia de objeto para llamar a un método estático

ORACLE

Variables estáticas

Las variables estáticas son variables a las que se puede acceder incluso aunque la clase en la que se hayan declarado no se haya instanciado. Las variables estáticas:

- Se denominan variables de clase.
- Se limitan a una sola copia por JVM.
- Son útiles para contener datos compartidos.
  - Los métodos estáticos almacenan datos en variables estáticas.
  - Todas las instancias de objetos comparten una sola copia de cualquier variable estática.
- Se inicializan cuando la clase contenedora se carga por primera vez.

ORACLE

Carga de clases

Las clases suministradas por el desarrollador de aplicaciones se suelen cargar a petición (primer uso). Las variables estáticas se inicializan cuando se carga su clase delimitadora. Intentar acceder a un miembro de clase estática puede disparar la carga de una clase.

Definición de variables estáticas

```
public class StaticCounter {
    private static int counter = 0;

    public StaticCounter() {
        counter++;
    }

    public static int getCount() {
        return counter;
    }
}
```

Solo una copia en la memoria

Persistencia de variables estáticas

Muchas de las tecnologías que se usan para mantener el estado de la aplicación en Java solo guardan variables de instancias. El mantenimiento de un solo objeto que realiza un seguimiento del estado “compartido” se puede usar como alternativa a las variables estáticas.

Uso de variables estáticas

```
double p = Math.PI;
```

```
new StaticCounter();
new StaticCounter();
System.out.println("count: " + StaticCounter.getCount());
```

Al acceder a las variables estáticas, debería:

- Cualificar la ubicación de la variable con un nombre de clase si la variable se encuentra en otra clase distinta a la del emisor de la llamada
  - No es necesario para variables de la misma clase
- Evitar el uso de una referencia de objeto para acceder a una variable estática

ORACLE

Referencias de objetos a miembros estáticos

Al igual que se debe evitar el uso de referencias de objetos a método estáticos, también debería evitar el uso de referencias de objetos para acceder a variables estáticas. Si todos los miembros de una clase son estáticos, procure usar un constructor privado para evitar la instanciación de objetos.

Importaciones estáticas

Una sentencia de importación estática hace que los miembros estáticos de una clase estén disponibles con su nombre simple.

- Con cualquiera de las siguientes líneas:

```
import static java.lang.Math.random;
import static java.lang.Math.*;
```

- La llamada al método `Math.random()` se podría escribir como:

```
public class StaticImport {
    public static void main(String[] args) {
        double d = random();
    }
}
```


El uso excesivo de la importación estática puede afectar negativamente a la lectura del código. Procure no agregar varias importaciones estáticas a una clase.

Prueba

El número de instancias de una variable estática está relacionado con el número de objetos que se han creado.

a. Verdadero
b. Falso

Métodos finales

Un método se puede declarar como `final`. Los métodos finales no se pueden sobrescribir.

```
public class MethodParentClass {
    public final void printMessage() {
        System.out.println("This is a final method");
    }
}
```

```
public class MethodChildClass extends MethodParentClass {
    // compile-time error
    public void printMessage() {
        System.out.println("Cannot override method");
    }
}
```

ORACLE

Mitos sobre el rendimiento

La declaración de un método como final a penas afecta al rendimiento. Los métodos se deben declarar como finales solo para desactivar la sustitución de métodos.

Clases finales

Una clase se puede declarar como `final`. Las clases finales no se pueden ampliar.

```
public final class FinalParentClass { }
```

```
// compile-time error
public class ChildClass extends FinalParentClass { }
```

Variables finales

El modificador `final` se puede aplicar a las variables. Las variables finales no pueden cambiar sus valores una vez inicializadas. Las variables finales pueden ser:

- Campos de clase
  - Los campos finales con expresiones de constantes de tiempo de compilación son variables de constantes.
  - Los campos estáticos se pueden combinar con los finales para crear una variable siempre disponible y que nunca cambia.
- Parámetros del método
- Variables locales

Nota: las referencias finales siempre deben hacer referencia al mismo objeto, pero el contenido de ese objeto se puede modificar.

Ventajas y desventajas de las variables finales

Prevención de bugs

Los valores de las variables finales no se pueden nunca modificar una vez inicializadas. Este comportamiento funciona como un mecanismo de prevención de bugs.

Protección de threads

La naturaleza inmutable de las variables finales elimina cualquiera de las preocupaciones que conlleva el acceso simultáneo mediante varios threads.

Referencia final a los objetos

Una referencia de objeto `final` solo sirve para evitar que una referencia apunte a otro objeto. Si está diseñando objetos inmutables, debe evitar que los campos del objeto se modifiquen. Las referencias finales también le evitan asignar un valor `null` a la referencia. El mantenimiento de referencias de un objeto evita que ese objeto esté disponible para la recolección de basura.

Declaración de variables finales

```
public class VariableExampleClass {
    private final int field;
    private final int forgottenField;
    private final Date date = new Date();
    public static final int JAVA_CONSTANT = 10;

    public VariableExampleClass() {
        field = 100;
    }

    public void changeValues(final int param) {
        param = 1; // compile-time error
        date.setTime(0); // allowed
        date = new Date(); // compile-time error
        final int localVar;
        localVar = 42;
        localVar = 43; // compile-time error
    }
}
```

ORACLE

Campos finales

Inicialización

Los campos finales (variables de instancia) deben ajustarse a uno de los siguientes supuestos:

- Tener asignado un valor al declararse
- Tener asignado un valor en cada uno de los constructores

Estáticos y finales

Un campo que es tanto estático como final se considera una constante. Por convención, los campos constantes usan identificadores que solo están formados por letras mayúsculas y caracteres de subrayado.

Prueba

A un campo final (variable de instancia) se le puede asignar un valor cuando se declara o en todos los constructores.

a. Verdadero
b. Falso

Cuándo evitar las constantes

Las variables `public`, `static` y `final` pueden ser muy útiles, pero hay un patrón de uso concreto que se debería evitar. Las constantes pueden proporcionar una falsa sensación de validación de los datos introducidos o de comprobación del rango de valores.

- Piense en un método que solo deba recibir uno de los tres valores posibles:

```
Computer comp = new Computer();
comp.setState(Computer.POWER_SUSPEND);
```

Se trata de una constante `int` que es igual que `2`.

- Las siguientes líneas de código se seguirían compilando:

```
Computer comp = new Computer();
comp.setState(42);
```


Comprobación de rangos en tiempo de ejecución

En el ejemplo de la diapositiva, debe realizar una comprobación de rango en tiempo de ejecución al usar un valor `int` para representar el estado. En el método `setState`, se puede usar una sentencia `if` para validar que solo se acepten los valores `0`, `1` o `2`. Este tipo de comprobación se realiza cada vez que se llama al método `setState`, lo que provoca una sobrecarga adicional.

Enumeraciones Typesafe

Java 5 ha incluido una enumeración typesafe al lenguaje. Las enumeraciones:

- Se crean con una variación de una clase Java
- Proporcionan una comprobación de rangos en tiempo de compilación

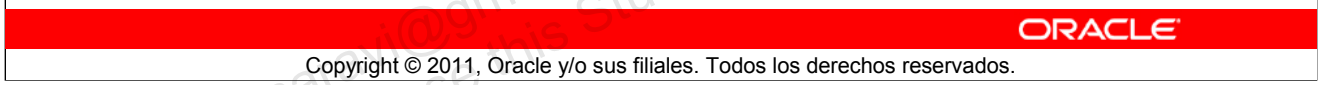
```
public enum PowerState {
    OFF,
    ON,
    SUSPEND;
}
```

Estas son las referencias a los tres únicos objetos `PowerState` que pueden existir.

Una enumeración se puede utilizar de la siguiente forma:

```
Computer comp = new Computer();
comp.setState(PowerState.SUSPEND);
```

Este método toma una referencia `PowerState`.



Comprobación de rangos en tiempo de compilación

En el ejemplo de la diapositiva, el compilador realiza una comprobación de tiempo de compilación para garantizar que solo se transfieran las instancias válidas de `PowerState` al método `setState`. En tiempo de ejecución no se produce ninguna sobrecarga en la comprobación de los rangos.

Uso de enumeraciones

Las referencias a enumeraciones se pueden importar de forma estática.

```
import static com.example.PowerState.*;

public class Computer extends ElectronicDevice {
    private PowerState powerState = OFF;
    //...
}
```

PowerState.OFF

Las enumeraciones se pueden usar como expresión en una sentencia switch.

```
public void setState(PowerState state) {
    switch(state) {
        case OFF:
        //...
    }
}
```

Importadas de forma estática

Enumeraciones complejas

Las enumeraciones pueden tener campos, métodos y constructores privados.

```
public enum PowerState {
    OFF("The power is off"),
    ON("The usage power is high"),
    SUSPEND("The power usage is low");

    private String description;
    private PowerState(String d) {
        description = d;
    }
    public String getDescription() {
        return description;
    }
}
```

Llame a un constructor `PowerState` para inicializar la referencia `public static final OFF`.

El constructor no puede ser del tipo `public` ni `protected`.

ORACLE

Constructores de enumeración

Una instancia de enumeración no se puede instanciar con `new`.

Prueba

Una enumeración puede tener un constructor con los siguientes niveles de acceso:

a. public
b. protected
c. por defecto (nivel de acceso no declarado)
d. private

Patrones de diseño

Los patrones de diseño:

- Son soluciones reutilizables a problemas de desarrollo de software comunes.
- Están documentados en catálogos de patrones.
  - *Design Patterns: Elements of Reusable Object-Oriented Software* (Patrones de diseño: elementos del software reutilizable orientado a objetos), de Erich Gamma et al. (conocido como “Gang of Four”, la banda de los cuatro, por sus cuatro autores)
- Forman un vocabulario para hablar sobre el diseño.

Catálogos de patrones de diseño

Hay disponibles catálogos de patrones para muchos lenguajes de programación. La mayoría de los patrones de diseño tradicionales se aplican a cualquier lenguaje de programación orientado a objetos. Una de las obras más conocidas, *Design Patterns: Elements of Reusable Object-Oriented Software*, usa una combinación de C++, Smalltalk y diagramas para mostrar las posibles implantaciones de patrones. Son muchos los desarrolladores Java que aún hacen referencia a esta obra, ya que los conceptos se pueden extrapolar a cualquier lenguaje orientado a objetos.

Obtendrá más información sobre los patrones de diseño y otras recomendaciones de Java en el curso *Patrones de diseño Java*.

Patrón singleton

El patrón de diseño singleton detalla una implantación de clase que solo se puede instanciar una vez.

```
public class SingletonClass {
1   private static final SingletonClass instance =
            new SingletonClass();

2   private SingletonClass() {}

3   public static SingletonClass getInstance() {
        return instance;
    }
}
```


Implantación de un patrón singleton

El patrón de diseño singleton es uno de los patrones de diseño de creación que se clasifican en *Design Patterns: Elements of Reusable Object-Oriented Software*.

Para implantar el patrón de diseño singleton:

1. Utilice una referencia estática para apuntar a la instancia única. Declarar la referencia como final garantiza que nunca se hará referencia a otra instancia.
2. Agregue un solo constructor privado a la clase singleton. El modificador privado solo permite acceso de la "misma clase", lo que prohíbe cualquier intento de instanciar la clase singleton, excepto en el caso del intento en el paso 1.
3. Un método de fábrica público devuelve una copia de la referencia a singleton. Este método se declara como estático para acceder al campo estático declarado en el paso 1. El paso 1 podría usar una variable pública, lo que hace que no sea necesario el método de fábrica. Los métodos de fábrica ofrecen una mayor flexibilidad (por ejemplo, implantar una solución singleton por thread) y se suelen usar en la mayoría de las implantaciones de singleton.

Para obtener una referencia de singleton, llame al método `getInstance`:

```
SingletonClass ref = SingletonClass.getInstance();
```

Clases anidadas

Una clase anidada es una clase declarada dentro del cuerpo de otra clase. Las clases anidadas:

- Tienen varias categorías.
  - Clases internas
    - Clases de miembros
    - Clases locales
    - Clases anónimas
  - Clases anidadas estáticas
- Se suelen usar en aplicaciones con elementos de interfaz gráfica de usuario (GUI).
- Pueden limitar el uso de una "clase helper" a la clase delimitadora de nivel superior.

ORACLE

Motivos para usar clases anidadas

La siguiente información se ha obtenido de http://download.oracle.com/javase/tutorial/java/javaOO/nested.html.

Agrupación lógica de clases

Si una clase resulta útil solo para otra clase, resulta lógico embeberla en esa clase y conservar las dos juntas. Anidar estas "clases helper" hace que su paquete sea más racionalizado.

Encapsulación aumentada

Piense en dos clases de nivel superior, A y B, donde B tiene que acceder a miembros de A que, de otra forma, estarían declarados como private. Al ocultar la clase B en la clase A, los miembros de la clase A se pueden declarar como privados y B puede acceder a ellos. Además, B también se puede ocultar al mundo exterior.

Código más legible y fácil de mantener

La anidación de clases pequeñas en clases de nivel superior acerca el código al lugar donde se usa.

Clase interna: ejemplo

```
public class Car {
    private boolean running = false;
    private Engine engine = new Engine();

    private class Engine {
        public void start() {
            running = true;
        }
    }

    public void start() {
        engine.start();
    }
}
```


Clases internas frente a clases anidadas estáticas

Una clase interna se considera parte de la clase externa y hereda el acceso a todos los miembros privados de la clase externa. En el ejemplo de la diapositiva se muestra una clase interna, que es una clase de miembro. Las clases internas también se declaran dentro de un bloque de métodos (clases locales).

Una clase anidada estática no es una clase interna, pero su declaración parece similar con un modificador `static` adicional en la clase anidada. Las clases anidadas estáticas se pueden instanciar antes de la clase externa delimitadora y, por tanto, se deniega el acceso a todos los miembros no estáticos de la clase delimitadora.

Clases internas anónimas

Una clase anónima se usa para definir una clase sin nombre.

```
public class AnonymousExampleClass {
    public Object o = new Object() {
        @Override
        public String toString() {
            return "In an anonymous class method";
        }
    };
}
```


Clase sin nombre

En el ejemplo de la diapositiva, la clase `java.lang.Object` tiene una subclase y es esa subclase la que se está instanciando. Al compilar una aplicación con clases anónimas, se generará un archivo de clase independiente con una convención de nomenclatura `Outer$1.class`, donde `1` es el número de índice de las clases anónimas en una clase delimitadora y `Outer` es el nombre de la clase delimitadora.

Las clases internas anónimas también pueden ser clases locales.

Prueba

¿Cuáles de los siguientes tipos de clases anidadas son clases internas?

a. anonymous
b. local
c. static
d. member

Resumen

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

- Diseñar clases base de uso general mediante clases abstractas
- Crear clases y subclases Java abstractas
- Modelar problemas de negocio mediante las palabras clave `static` y `final`
- Implantar el patrón de diseño singleton
- Distinguir entre clases de nivel superior y anidadas

Visión general de la práctica 5-1: Aplicación de la palabra clave abstract

En esta práctica, se abordan los siguientes temas:

- Identificación de posibles problemas que se pueden solucionar con clases abstractas
- Refactorización de una aplicación Java existente para que use clases y métodos abstractos

ORACLE

Visión general de la práctica 5-2: Aplicación del patrón de diseño singleton

En esta práctica se aborda el uso de las palabras clave static y final, y la refactorización de una aplicación existente para implantar el patrón de diseño singleton.

Visión general de la práctica 5-3: (Opcional) Uso de enumeraciones Java

En esta práctica se abarca el uso de una aplicación existente y la refactorización del código para utilizar una enumeración.

(Opcional) Visión general de la práctica 5-4: Reconocimiento de clases anidadas

En esta práctica se abarca el análisis de una aplicación Java existente y la identificación del número y los tipos de clases presentes.

ORACLE

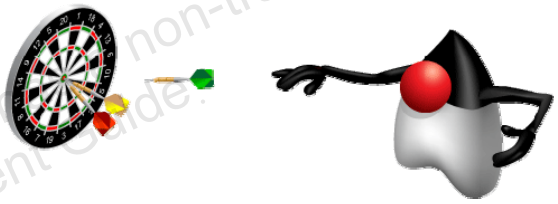
6

Herencia con interfaces Java

Objetivos

Al finalizar esta lección, debería estar capacitado para lo siguiente:

- Modelar problemas de negocio mediante interfaces
- Definir una interfaz Java
- Seleccionar entre herencia de interfaz y herencia de clase
- Ampliar una interfaz
- Refactorizar código para implantar el patrón DAO



ORACLE

Implantación de sustitución

La capacidad de describir tipos abstractos es una potente función de Java. La abstracción permite:

- Facilidad de mantenimiento
  - Clases con errores lógicos que se pueden sustituir con clases nuevas y mejoradas.
- Implantación de sustitución
  - El paquete `java.sql` describe los métodos que usan los desarrolladores para comunicarse con las bases de datos, pero la implantación es específica del proveedor.
- División del trabajo
  - Describir la API de negocio que necesita la interfaz de usuario de una aplicación permite a dicha interfaz de usuario y a la lógica de negocio desarrollarse de forma conjunta.

ORACLE

Abstracción

Acaba de aprender a definir tipos abstractos mediante el uso de clases. Hay dos formas de definir la abstracción de tipos en Java: clases e interfaces abstractas. Al escribir código para que haga referencia a tipos abstractos, ya no dependerá de las clases de implantación específicas. La definición de estos tipos abstractos puede parecer un trabajo adicional al principio, pero puede reducir la refactorización posteriormente si se usa bien.

Interfaces Java

Las interfaces Java se usan para definir tipos abstractos. Las interfaces:

- Son similares a las clases abstractas que solo contienen métodos abstractos públicos.
- Describen los métodos que debe implantar una clase.
  - Los métodos no deben tener una implantación {corchetes}.
- Pueden contener campos constantes.
- Se pueden usar como tipo de referencia.
- Son un componente esencial de muchos patrones de diseño.

ORACLE

En Java, una interfaz describe un contrato para una clase. El contrato definido por una interfaz dicta los métodos que se deben implantar en una clase. Las clases que implantan el contrato deben cumplir todo el contrato o declararse como tipo `abstract`.

Desarrollo de interfaces Java

Las interfaces públicas de nivel superior se declaran en su propio archivo `.java`. Las interfaces se implantan en lugar de ampliarse.

```
public interface ElectronicDevice {
    public void turnOn();
    public void turnOff();
}
```

```
public class Television implements ElectronicDevice {
    public void turnOn() { }
    public void turnOff() { }
    public void changeChannel(int channel) {}
    private void initializeScreen() {}
}
```

ORACLE

Reglas para interfaces

Modificadores de acceso

Todos los métodos de una interfaz son de tipo `public`, incluso aunque olvide declararlos como tipo `public`. No puede declarar los métodos como `private` o `protected` en una interfaz. El contrato que detalla una interfaz es una API pública que una clase debe proporcionar.

Modificador abstracto

Debido a que todos los métodos son de tipo `abstract` de forma implícita, es redundante (si bien está permitido) declarar un método como `abstract`. Debido a que todos los métodos de interfaz son abstractos, no puede proporcionar ninguna implantación de método, ni siquiera un juego vacío de corchetes.

Implantaciones y ampliaciones

Una clase puede ampliar una clase principal y, a continuación, implantar una lista separada por comas de interfaces.

Campos constantes

Las interfaces pueden tener campos constantes.

```
public interface ElectronicDevice {
    public static final String WARNING =
        "Do not open, shock hazard";
    public void turnOn();
    public void turnOff();
}
```


Solo se permiten campos constantes en una interfaz. Al declarar un campo en una interfaz, de forma implícita es de tipo `public`, `static` y `final`. Puede especificar de forma redundante estos modificadores. Procure no agrupar todos los valores constantes de una aplicación en una sola interfaz; un diseño correcto es el que distribuye los valores constantes de una aplicación en varias clases e interfaces. La creación de clases monolíticas o interfaces que contengan grandes agrupaciones de código no relacionado no respeta las recomendaciones del diseño orientado a objetos.

Referencias a la interfaz

Puede utilizar una interfaz como tipo de referencia. Al usar un tipo de referencia de interfaz, debe usar solo los métodos señalados en la interfaz.

```
ElectronicDevice ed = new Television();
ed.turnOn();
ed.turnOff();
ed.changeChannel(2); // fails to compile
String s = ed.toString();
```

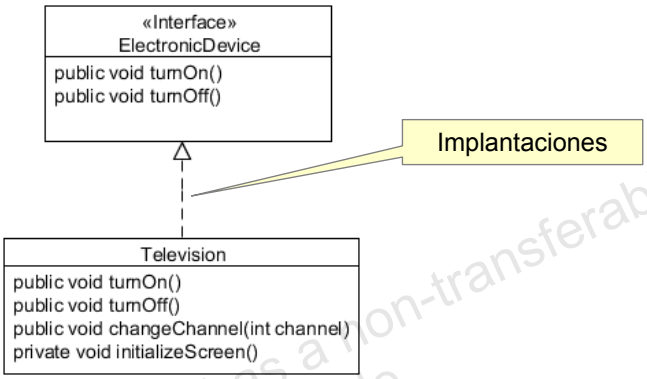
ORACLE

Una referencia introducida como interfaz solo se puede usar para hacer referencia a un objeto que implanta esa interfaz. Si el objeto tiene todos los métodos detallados en la interfaz, pero no la implanta, no se podrá usar la interfaz como tipo de referencia para ese objeto. Las interfaces incluyen de forma implícita todos los métodos de `java.lang.Object`.

Operador `instanceof`

Puede usar `instanceof` con las interfaces.

```
Television t = new Television();
if (t instanceof ElectronicDevice) { }
```



Television es una instancia de un objeto ElectronicDevice.

ORACLE

Anteriormente, ha usado `instanceof` en los tipos de clase. Cualquier tipo que se pueda usar como referencia se podrá usar como operando para el operador `instanceof`. En la diapositiva, `Television` implanta `ElectronicDevice`. Por tanto, un elemento `Television` es una instancia de `Television`, un elemento `ElectronicDevice` y un elemento `java.lang.Object`.

Interfaces de marcador

- Las interfaces de marcador definen un tipo, pero no señalan ningún método que deba implantar una clase.

```
public class Person implements java.io.Serializable { }
```

- El objetivo de estos tipos de interfaces solo es comprobar los tipos.

```
Person p = new Person();
if (p instanceof Serializable) {

}
```

ORACLE

`java.io.Serializable` es una interfaz de marcador que usa una biblioteca de E/S de Java para determinar si se puede serializar el estado de un objeto. Al implantar `Serializable`, no tendrá que proporcionar implantaciones de métodos. La prueba (como operador `instanceof`) de la capacidad de serializar un objeto va incorporada en las bibliotecas de E/S estándar. Esta interfaz se utiliza en la lección titulada “Conceptos fundamentales de E/S en Java”.

Conversión en tipos de interfaz

Puede realizar la conversión en un tipo de interfaz.

```
public static void turnObjectOn(Object o) {
    if (o instanceof ElectronicDevice) {
        ElectronicDevice e = (ElectronicDevice)o;
        e.turnOn();
    }
}
```

ORACLE

Instrucciones de conversión

Al igual que se hace al convertir en tipos de clase, si convierte en un tipo que no es válido para ese objeto, la aplicación generará una excepción e incluso se podría bloquear. Para verificar que una conversión se realizará correctamente, debe usar una prueba `instanceof`.

En el ejemplo de la diapositiva se muestra un diseño deficiente, ya que el método `turnObjectOn()` solo funciona en elementos `ElectronicDevice`. Usar `instanceof` y realizar una conversión agrega sobrecarga en tiempo de ejecución. Cuando sea posible, utilice una prueba en tiempo de compilación reescribiendo el método como:

```
public static void turnObjectOn(ElectronicDevice e) {
    e.turnOn();
}
```

Uso de tipos de referencia genéricos

- Utilice el tipo de referencia más genérico siempre que sea posible:

```
EmployeeDAO dao = new EmployeeDAOMemoryImpl();
dao.delete(1);
```

`EmployeeDAOMemoryImpl` implanta `EmployeeDAO`.

- Al usar un tipo de referencia de interfaz, puede usar una clase de implantación distinta sin correr el riesgo de interrumpir las siguientes líneas de código:

```
EmployeeDAOMemoryImpl dao = new EmployeeDAOMemoryImpl();
dao.delete(1);
```

Es posible que solo use métodos `EmployeeDAOMemoryImpl` aquí.

ORACLE

Al crear referencias, debe usar el tipo más genérico posible. Esto significa que, para el objeto que está instanciando, debe declarar la referencia para que sea de un tipo de interfaz o de un tipo de clase principal. Al hacer esto, todo el uso de la referencia no está ligado a una clase de implantación concreta y, si fuera necesario, podría usar otra clase de implantación. Al usar una interfaz que implanta varias clases como tipo de referencia, tendrá libertad para cambiar la implantación sin que el código se vea afectado. Se podría usar una referencia de tipo `EmployeeDAOMemoryImpl` para llamar a un método que solo aparezca en la clase `EmployeeDAOMemoryImpl`.

Las referencias especificadas en una clase concreta harán que el código esté totalmente acoplado a esa clase y podrían provocar una mayor refactorización del código cuando se cambien las implantaciones.

Implantación y ampliación

- Las clases pueden ampliar una clase principal e implantar una interfaz:

```
public class AmphibiousCar extends BasicCar implements
MotorizedBoat { }
```

- También puede implantar varias interfaces:

```
public class AmphibiousCar extends BasicCar implements
MotorizedBoat, java.io.Serializable { }
```

Utilice una coma para separar la lista de interfaces.

ORACLE

En primer lugar Extends

Si utiliza `extends` y `implements`, `extends` debe ir primero.

Ampliación de interfaces

- Las interfaces pueden ampliar otras interfaces:

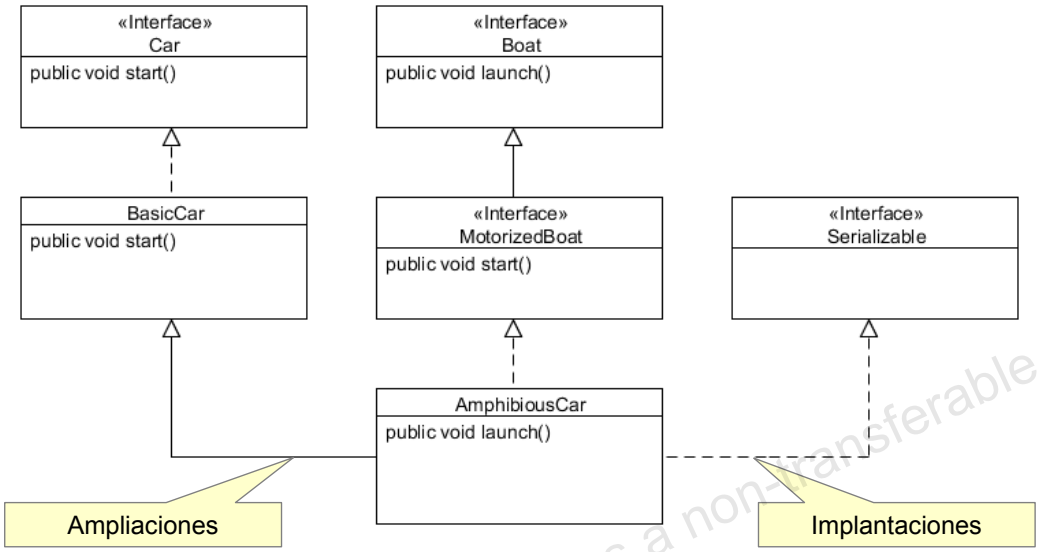
```
public interface Boat { }
```

```
public interface MotorizedBoat extends Boat { }
```

- Al implantar `MotorizedBoat`, la clase `AmphibiousCar` debe cumplir el contrato señalado tanto por `MotorizedBoat` como por `Boat`:

```
public class AmphibiousCar extends BasicCar implements
MotorizedBoat, java.io.Serializable { }
```

Interfaces en jerarquías de herencia



Herencias de interfaces

Las interfaces se usan para una clase de herencia que se conoce como *herencia de interfaces*. Java permite la herencia de varias interfaces, pero solo la herencia de una clase.

Ampliación de una clase de implantación

Si escribe una clase que amplía una clase que implanta una interfaz, la clase creada también implanta la interfaz. Por ejemplo, `AmphibiousCar` amplía `BasicCar`. `BasicCar` implanta `Car`; por tanto, `AmphibiousCar` también implanta `Car`.

Interfaces que amplían interfaces

Una interfaz puede ampliar otra interfaz. Por ejemplo, la interfaz `MotorizedBoat` puede ampliar la interfaz `Boat`. Si la clase `AmphibiousCar` implanta `MotorizedBoat`, debe implantar todos los métodos de `Boat` y `MotorizedBoat`.

Métodos duplicados

Cuando cuenta con una clase que implanta varias interfaces, directa o indirectamente, puede aparecer la misma firma de método en distintas interfaces implantadas. Si las firmas son las mismas, no se produce ningún conflicto y solo se necesita una implantación.

Prueba

Una clase puede implantar varias interfaces.

a. Verdadero
b. Falso

Diseño de patrones e interfaces

- Uno de los principios del diseño orientado a objetos es:
  "Programe para una interfaz, no para una implantación."
- Se trata de un tema común en muchos patrones de diseño. Este principio desempeña un rol en:
  - El patrón de diseño DAO
  - El patrón de diseño de fábrica

ORACLE

Principios del diseño orientado a objetos

"Programe para una interfaz, no para una implantación" es una práctica que se popularizó gracias a la obra *Design Patterns: Elements of Reusable Object-Oriented Software* (Patrones de diseño: elementos del software reutilizable orientado a objetos).

Puede obtener más información sobre los principios del diseño orientado a objetos y los patrones de diseño en el curso *Patrones de diseño Java*.

Patrón DAO

El patrón de objeto de acceso a datos (DAO) se usa al crear una aplicación que debe mantener información. El patrón DAO:

- Separa el dominio de problemas del mecanismo de persistencia.
- Usa una interfaz para definir los métodos usados para la persistencia. Una interfaz permite sustituir la implantación de la persistencia por:
  - DAO basados en memoria como solución temporal
  - DAO basados en archivos para una versión inicial
  - DAO basados en JDBC para soportar la persistencia de la base de datos
  - DAO basados en la API de persistencia Java (JPA) para soportar la persistencia de la base de datos

ORACLE

¿Por qué se debe separar el código de persistencia del de negocio?

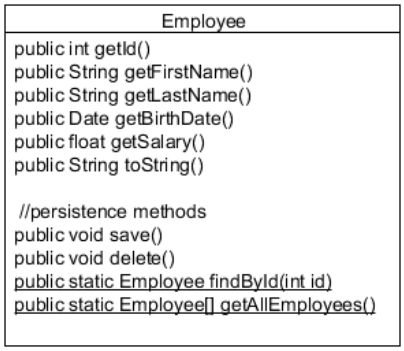
Al igual que la funcionalidad necesaria de una aplicación influirá en el diseño de las clases, así lo harán otros aspectos. En el diseño también influyen aspectos como el deseo de que el mantenimiento sea sencillo y la capacidad de mejorar una aplicación. El código modular separado por la funcionalidad es más fácil de actualizar y mantener.

Al separar la lógica de negocio y de persistencia, las aplicaciones son más fáciles de implantar y mantener a costa de clases e interfaces adicionales. A menudo estos dos tipos de lógica tienen distintos ciclos de mantenimiento. Por ejemplo, la lógica de persistencia se tendría que modificar si la base de datos que usa la aplicación se hubiera migrado de MySQL a Oracle 11*g*.

Si crea interfaces para las clases que contienen la lógica de persistencia, resulta más fácil sustituir la implantación de persistencia.

Antes del patrón DAO

Observe la combinación de métodos de persistencia y métodos de negocio.



Antes del patrón DAO

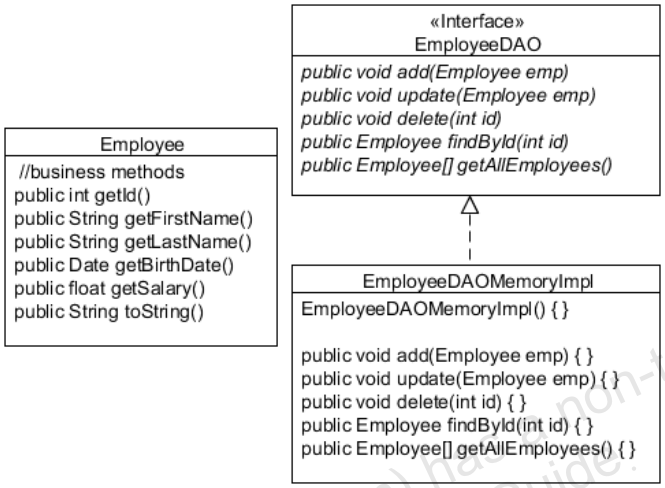
ORACLE

Principio de responsabilidad única

La clase `Employee` que aparece en la diapositiva tiene métodos que se centran en dos principios o aspectos distintos. Un juego de métodos se centra en la manipulación de la representación de una persona, mientras que el otro se centra en el mantenimiento de objetos `Employee`. Debido a que estos dos juegos de responsabilidades se pueden modificar en puntos distintos de la vida de las aplicaciones, tiene sentido separarlos en distintas clases.

Después del patrón DAO

El patrón DAO extrae la lógica de persistencia de las clases de dominios y las traslada a clases distintas.



Después de la refactorización del patrón DAO

ORACLE

Implantaciones de DAO

Si cree que necesitará cambiar la implantación de DAO posteriormente para que use otro mecanismo de persistencia, es mejor usar una interfaz para definir el contrato que deben cumplir las implantaciones de DAO.

Las interfaces DAO señalan métodos para crear, leer, actualizar y suprimir datos, si bien los nombres de métodos pueden variar. Al implantar por primera vez el patrón DAO, no verá la ventaja de forma inmediata. La verá posteriormente, cuando empiece a modificar o a sustituir el código. En la lección titulada “Creación de aplicaciones de base de datos con JDBC” se trata la sustitución del DAO basado en memoria por DAO de archivos y bases de datos.

La necesidad del patrón de fábrica

El patrón DAO depende del uso de interfaces para definir una abstracción. El uso de un constructor de implantación DAO le ata a una implantación concreta.

```
EmployeeDAO dao = new EmployeeDAOMemoryImpl();
```

Al usar un tipo de interfaz, las posteriores líneas no estarán ligadas a una sola implantación.

Esta llamada al constructor está ligada a una implantación y aparecerá en muchos lugares de una aplicación.

Uso del patrón de fábrica

Al usar una fábrica, se evita que la aplicación tenga que estar totalmente acoplada a una implantación de DAO concreta.

```
EmployeeDAOFactory factory = new EmployeeDAOFactory();
EmployeeDAO dao = factory.createEmployeeDAO();
```

La implantación `EmployeeDAO` está oculta.

Este patrón elimina las llamadas directas al constructor a favor de la llamada a un método. Las fábricas se suelen usar al implantar el patrón DAO.

En el ejemplo de la diapositiva, desconoce el tipo de mecanismo de persistencia que usa `EmployeeDAO` porque se trata solo de una interfaz. La fábrica podría devolver una implantación de DAO que usara archivos o una base de datos para almacenar y recuperar datos. Como desarrollador, desea saber el tipo de persistencia que se está usando, porque influye en el rendimiento y la fiabilidad de la aplicación. Sin embargo, no desea que la mayoría del código que ha escrito esté totalmente acoplado al tipo de persistencia.

Fábrica

La implantación de la fábrica es el único punto de la aplicación que debe depender de clases DAO concretas.

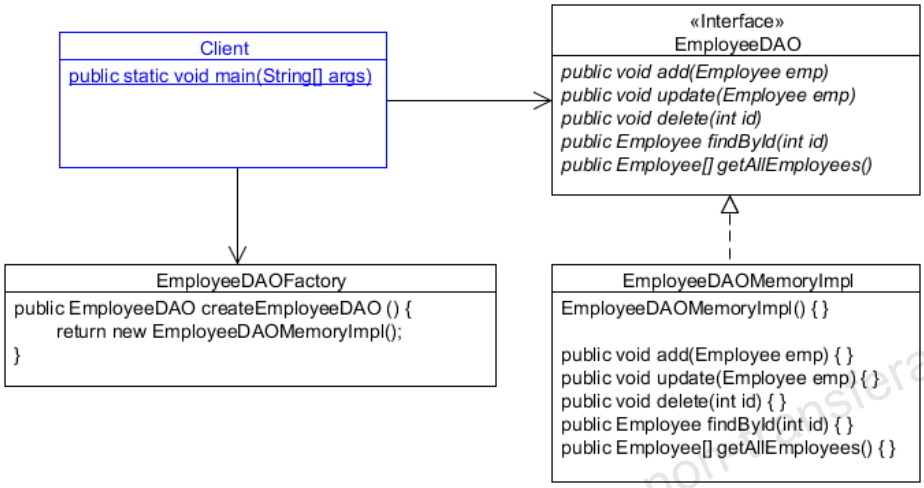
```
public class EmployeeDAOFactory {

    public EmployeeDAO createEmployeeDAO() {
        return new EmployeeDAOMemoryImpl();
    }
}
```

Devuelve una referencia escrita como interfaz

Para una mayor simplicidad, esta fábrica codifica el nombre de una clase concreta que instanciar. Podría mejorar esta fábrica colocando el nombre de la clase en un origen externo, como un archivo de texto y utilizar la clase `java.lang.Class` para instanciar la subclase concreta. Un ejemplo básico del uso de `java.lang.Class` es el siguiente:

```
String name = "com.example.dao.EmployeeDAOMemoryImpl";
Class clazz = Class.forName(name);
EmployeeDAO dao = (EmployeeDAO)clazz.newInstance();
```

Combinación de DAO y fábrica



Los clientes dependen solo de DAO abstractos

ORACLE

Prueba

Una implantación de singleton típica contiene un método de fábrica.

a. Verdadero
b. Falso

Reutilización del código

La duplicación del código (copiar y pegar) puede conllevar problemas de mantenimiento. No desea corregir el mismo bug una y otra vez.

- "No se repita ." (principio DRY, del inglés Don't Repeat Yourself)
- Reutilice el código de la forma correcta:
  - Refactorice rutinas de uso común en bibliotecas.
  - Mueva el comportamiento que comparten las clases hermanas a su clase principal.
  - Cree nuevas combinaciones de comportamientos combinando varios tipos de objetos (composición).

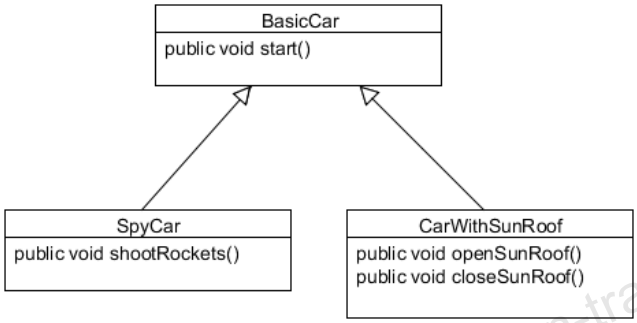
ORACLE

Copiar y pegar código *no* es algo que siempre se deba evitar. Si el código duplicado sirve como punto de partida y se realizan muchas modificaciones, esto podría ser un comportamiento correcto para copiar y pegar líneas de código. Debe ser consciente de qué cantidad de copiado y pegado se produce en un proyecto. Además de realizar auditorías de código manuales, hay herramientas que se pueden usar para detectar el código duplicado. Para consultar un ejemplo de esto, consulte http://pmd.sourceforge.net/cpd.html.

Dificultades en el diseño

La herencia de clases permite reutilizar código, pero no es algo muy modular.

- ¿Cómo se crea un elemento SpyCarWithSunRoof?



Implantaciones de métodos situadas en distintas clases

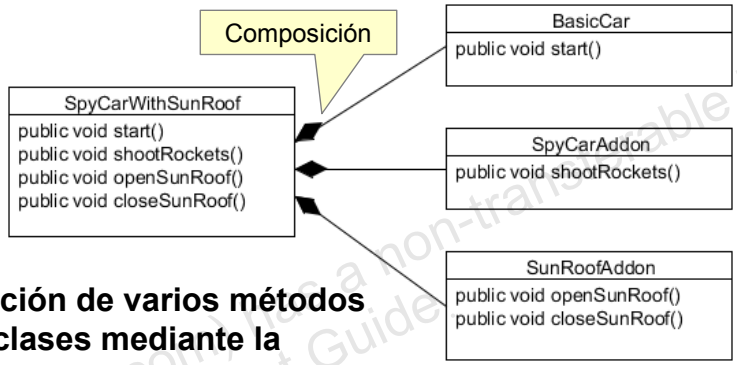
Limitaciones de la herencia

Java solo soporta la herencia de una sola clase, lo que impide que se puedan heredar distintas implantaciones de un método con la misma firma. La herencia de varias interfaces no plantea el mismo problema que la herencia de clases, ya que no puede haber implantaciones de métodos en conflicto en las interfaces.

Composición

La composición de objetos permite crear objetos más complejos. Para implantar la composición:

1. Cree una clase con referencias a otras clases.
2. Agregue los mismos métodos de firma que se reenvían a los objetos a los que se hace referencia.



Combinación de varios métodos de clases mediante la composición y el reenvío

Delegación

La delegación y el reenvío de métodos son dos términos que se suelen usar indistintamente. El reenvío de métodos se produce cuando se escribe un método que no hace nada, excepto transferir la ejecución a otro método. En algunos casos, la delegación puede implicar algo más que el simple reenvío. Para obtener más información sobre la diferencia entre los dos conceptos, consulte la página 20 de la obra *Design Patterns: Elements of Reusable Object-Oriented Software*.

Implantación de la composición

```
public class SpyCarWithSunRoof {
    private BasicCar car = new BasicCar();
    private SpyCarAddon spyAddon = new SpyCarAddon();
    private SunRoofAddon roofAddon = new SunRoofAddon();

    public void start() {
        car.start();
    }

    // other forwarded methods
}
```

Reenvío de métodos

Los asistentes de IDE facilitan la implantación de la composición

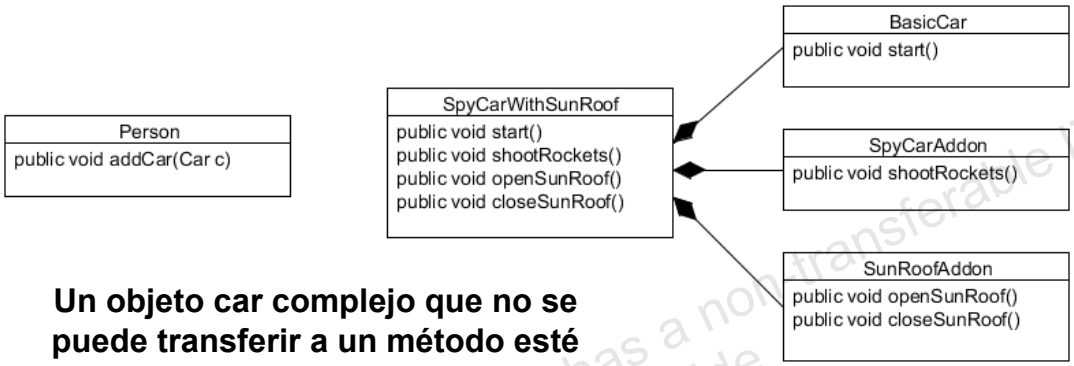
Para implantar la composición con NetBeans IDE, utilice la herramienta Insert Code de la siguiente forma:

1. Haga clic con el botón derecho en los corchetes de la clase compleja y seleccione "Insert Code”.
2. Seleccione "Delegate Method”.
   Aparece el cuadro de diálogo Generate Delegate Methods.
3. Seleccione las llamadas de métodos que desea reenviar.
   Los métodos se insertan.

Repita estos pasos para cada una de las clases delegadas.

Polimorfismo y composición

El polimorfismo nos debe permitir transferir cualquier tipo de elemento Car al método `addCar`. La composición no permite el polimorfismo, a menos que…



Un objeto car complejo que no se puede transferir a un método esté esperando un objeto car simple

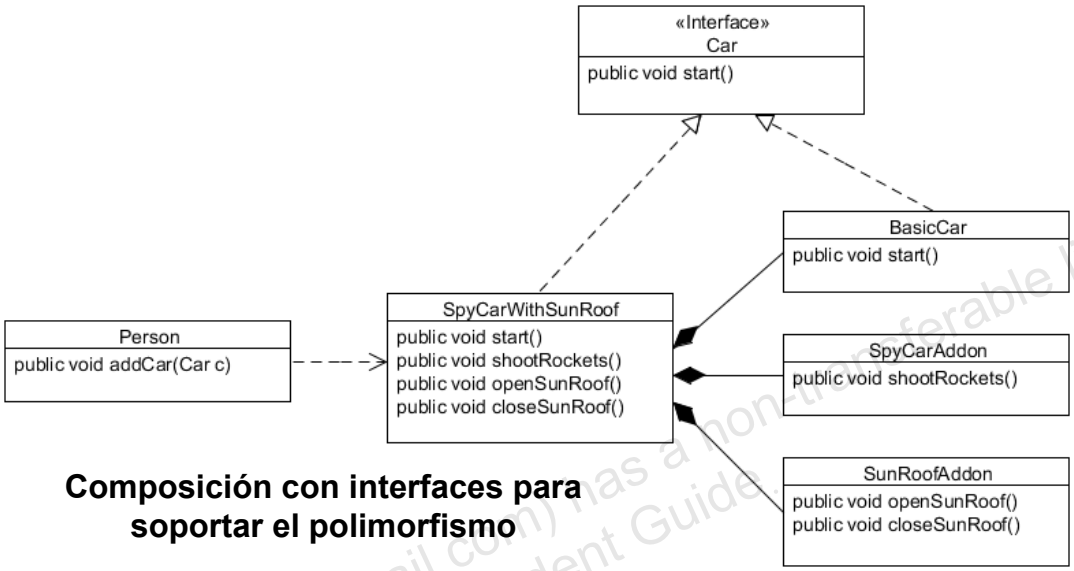
Reutilización del código

La capacidad de usar el método `addCar` para cualquier tipo de objeto Car, con independencia de su complejidad, es otra forma de reutilizar el código. No podemos afirmar actualmente lo siguiente:

```
addCar(new SpyCarWithSunRoof());
```

Polimorfismo y composición

Utilice interfaces para que todas las clases delegadas soporten el polimorfismo.



Composición con interfaces para soportar el polimorfismo

Cada clase delegada que use en una composición debe tener definida una interfaz. Al crear la clase de composición, declara que implanta todos los tipos de interfaz delegada.
Al realizar esta acción, crea un objeto que es una composición de otros objetos y que tiene muchos tipos.

Ahora, podemos afirmar:

```
addCar(new SpyCarWithSunRoof());
```

Prueba

La delegación de métodos es necesaria para crear objetos complejos mediante:

a. Polimorfismo
b. Composición

Resumen

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

- Modelar problemas de negocio mediante interfaces
- Definir una interfaz Java
- Seleccionar entre herencia de interfaz y herencia de clase
- Ampliar una interfaz
- Refactorizar código para implantar el patrón DAO

ORACLE

Visión general de la práctica 6-1: Implantación de una interfaz

En esta práctica, se abordan los siguientes temas:

- Escritura de una interfaz
- Implantación de una interfaz
- Creación de referencias de un tipo de interfaz
- Conversión en tipos de interfaz

ORACLE

Visión general de la práctica 6-2: Aplicación del patrón DAO

En esta práctica, se abordan los siguientes temas:

- Reescritura de un objeto de dominio existente con una implantación de persistencia basada en memoria con el patrón DAO
- Uso de una fábrica abstracta para evitar la dependencia de implantaciones concretas

ORACLE

(Opcional) Visión general de la práctica 6-3: Implantación de la composición

En esta práctica, se abordan los siguientes temas:

- Reescritura de una aplicación existentes para soportar mejor la reutilización de código en la composición
- Uso de interfaces para permitir el polimorfismo

ORACLE

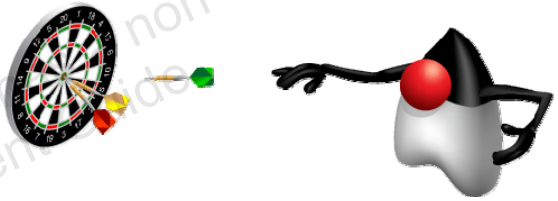
7

Genéricos y recopilaciones

Objetivos

Al finalizar esta lección, debería estar capacitado para:

- Crear una clase genérica personalizada
- Utilizar el diamante de inferencia de tipo para crear un objeto
- Crear una recopilación sin utilizar genéricos
- Crear una recopilación mediante el uso de genéricos
- Implantar una `ArrayList`
- Implantar una interfaz `Set`
- Implantar un `HashMap`
- Implantar una pila mediante el uso de Deque
- Utilizar tipos enumerados



ORACLE

Genéricos

- Proporcionan una seguridad flexible al código.
- Mueven muchos errores comunes de tiempo de ejecución a tiempo de compilación.
- Proporcionan un código más limpio y fácil de escribir.
- Reducen la necesidad de conversión de recopilaciones.
- Se utilizan frecuentemente en la API Collections de Java.

ORACLE

Clase de caché simple sin genéricos

```
public class CacheString {
    private String message = "";

    public void add(String message){
        this.message = message;
    }

    public String get(){
        return this.message;
    }
}
```

```
public class CacheShirt {
    private Shirt shirt;

    public void add(Shirt shirt){
        this.shirt = shirt;
    }

    public Shirt get(){
        return this.shirt;
    }
}
```


Los dos ejemplos de la diapositiva muestran clases de caché muy simples. Aunque las clases son muy simples, es necesario utilizar una clase independiente para cada tipo de objeto.

Clase de caché genérica

```
1 public class CacheAny <T>{
2
3    private T t;
4
5    public void add(T t){
6        this.t = t;
7    }
8
9    public T get(){
10        return this.t;
11    }
12 }
```


Para crear una versión genérica de la clase `CacheAny`, se agrega una variable denominada `T` a la definición de clase que aparece entre los paréntesis angulares. En este caso, `T` se refiere al "tipo" y puede representar cualquier tipo. Como muestra el ejemplo, se ha cambiado el código para utilizar `t` en lugar de información de tipo específica. El cambio permite que la clase `CacheAny` almacene cualquier tipo de objeto.

`T` no se ha elegido de forma casual, sino por convención. Con los genéricos se suele utilizar una serie de letras.

Nota: puede utilizar el identificador que desee. Únicamente se recomienda el uso de los valores que se muestran a continuación.

Las convenciones son las siguientes:

- `T`: tipo
- `E`: elemento
- `K`: clave
- `V`: valor
- `S`, `U`: se utilizan si hay un segundo tipo, un tercer tipo o más

Funcionamiento de los genéricos

Compare los objetos de tipo restringido con las alternativas genéricas.

```
1 public static void main(String args[]){
2    CacheString myMessage = new CacheString(); // Type
3    CacheShirt myShirt = new CacheShirt();     // Type
4
5    //Generics
6    CacheAny<String> myGenericMessage = new CacheAny<String>();
7    CacheAny<Shirt> myGenericShirt = new CacheAny<Shirt>();
8
9    myMessage.add("Save this for me"); // Type
10    myGenericMessage.add("Save this for me"); // Generic
11
12 }
```

ORACLE

Observe cómo la versión genérica de la clase puede sustituir un número cualquiera de clases de caché específicas de tipo. Las funciones `add()` y `get()` funcionan exactamente de la misma forma. De hecho, si la declaración `myMessage` pasa a ser genérica, no es necesario realizar cambios en el resto del código.

El código de ejemplo se encuentra en el proyecto Generics del archivo `TestCacheAny.java`.

Genéricos con diamante de inferencia de tipo

- Sintaxis.
  - No es necesario repetir los tipos en la parte derecha de la sentencia.
  - Los paréntesis angulares indican el reflejo de los parámetros de tipo.
- Simplifica las declaraciones genéricas.
- Ahorra la introducción de datos.

```
//Generics
CacheAny<String> myMessage = new CacheAny<>();
}
```

ORACLE

El diamante de inferencia de tipo es una nueva función de JDK 7. En el código genérico, observe cómo la definición de tipo de la derecha equivale siempre a la definición de tipo de la izquierda. En JDK 7, puede utilizar el diamante para indicar que la definición de tipo de la derecha equivale a la de la izquierda. Esto ayuda a evitar la introducción de información redundante una y otra vez.

Ejemplo: `TestCacheAnyDiamond.java`

Nota: de algún modo, funciona al contrario que una asignación de tipo Java “normal”. Por ejemplo, `Employee emp = new Manager();` convierte el objeto `emp` en una instancia de `Manager`.

Sin embargo, en el caso de los genéricos:

```
ArrayList<Manager>  managementTeam = new ArrayList<>();
```

es la parte izquierda de la expresión (en lugar de la derecha) la que determina el tipo.

Prueba

¿Cuál de las siguientes opciones *no* es una abreviatura convencional en el uso de los genéricos?

a. T: tabla
b. E: elemento
c. K: clave
d. V: valor

Recopilaciones

- Una recopilación es un objeto único diseñado para gestionar un grupo de objetos.
  - Los objetos de una recopilación se denominan *elementos.*
  - *No se permite el uso de primitivos en una recopilación.*
- Distintos tipos de recopilaciones implantan varias estructuras de datos comunes:
  - Pilas, colas, matrices dinámicas o elementos hash.
- La API Collections se basa, en gran medida, en los genéricos para su implantación.

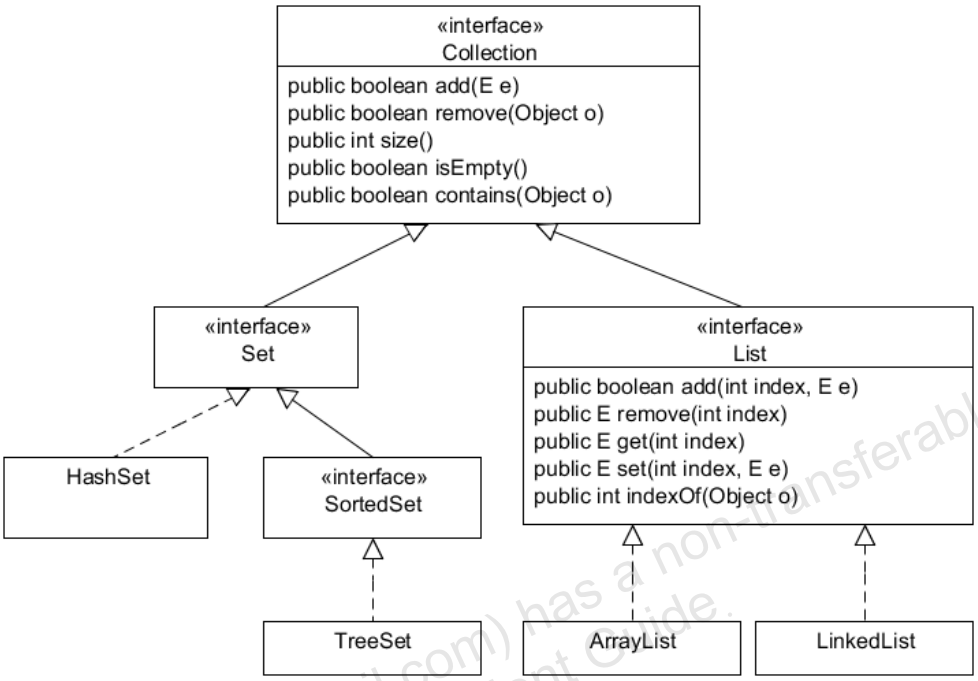
ORACLE

Una recopilación es un objeto único que gestiona un grupo de objetos. Los objetos de la recopilación se denominan *elementos*. Distintos tipos de recopilaciones implantan estructuras de datos estándar, entre las que se incluyen pilas, colas, matrices dinámicas y elementos hash. Todos los objetos de recopilación se han optimizado para su uso en aplicaciones Java.

Nota: las clases Collections se almacenan en el paquete `java.util`. Las sentencias `import` no se muestran en los siguientes ejemplos, pero son necesarias para todos los tipos de recopilaciones:

- `import java.util.List;`
- `import java.util.ArrayList;`
- `import java.util.Map;`

Tipos de recopilaciones



El diagrama de la diapositiva muestra todos los tipos de recopilaciones que descienden de `Collection`. Algunos métodos de ejemplo se proporcionan para `Collection` y `List`. Observe el uso de los genéricos.

Características de las clases de implantación

- `HashSet`: recopilación de elementos que no contiene elementos duplicados.
- `TreeSet`: recopilación de elementos ordenada que no contiene elementos duplicados.
- `ArrayList`: implantación de matriz dinámica.
- `Deque`: recopilación que se puede utilizar para implantar una pila o cola.

Nota: la interfaz `Map` es un árbol de herencia independiente y se detalla más adelante en esta misma lección.

Interfaz `List`

- `List` es una interfaz que define el comportamiento de una lista genérica.
  - Recopilación ordenada de elementos
- `List` incluye los siguientes comportamientos:
  - Adición de elementos en un índice específico
  - Adición de elementos al final de la lista
  - Obtención de un elemento basado en un índice
  - Eliminación de un elemento basado en un índice
  - Sobrescritura de un elemento basado en un índice
  - Obtención del tamaño de la lista
- Utilice `List` como un objeto de instancia para ocultar los detalles de la implantación.

ORACLE

La interfaz `List` es la base de todas las clases Collections que muestran el comportamiento de una lista.

Clase de implantación `ArrayList`

- Se trata de una matriz aumentable de forma dinámica.
  - La lista crece automáticamente si los elementos exceden el tamaño inicial.
- Tiene un índice numérico.
  - El índice accede a los elementos.
  - Los elementos se pueden insertar según el índice.
  - Los elementos se pueden sobrescribir.
- Permite los elementos duplicados.

ORACLE

Una `ArrayList` implanta una recopilación `List`. La implantación muestra las características de una matriz aumentable de forma dinámica. Una aplicación de lista de tareas es un buen ejemplo de aplicación que puede aprovechar una `ArrayList`.

ArrayList sin genéricos

```
1 public class OldStyleArrayList {
2  public static void main(String args[]){
3    List partList = new ArrayList(3);
4
5    partList.add(new Integer(1111));
6    partList.add(new Integer(2222));
7    partList.add(new Integer(3333));
8    partList.add("Oops a string!");
9
10    Iterator elements = partList.iterator();
11    while (elements.hasNext()) {
12      Integer partNumberObject = (Integer) (elements.next()); // error?
13      int partNumber = (int) partNumberObject.intValue();
14
15      System.out.println("Part number: " + partNumber);
16    }
17  }
18 }
```

ORACLE

En el ejemplo de la diapositiva, se crea una lista de números de artículos mediante el uso de una `ArrayList`. Con el uso de la sintaxis anterior a la versión 1.5 de Java, no existe la definición de tipo. Por tanto, los tipos se pueden agregar a la lista tal y como se muestra en la línea 8. Depende del programador saber cuáles son los objetos que están en la lista y su orden. Si la lista fuera solo para objetos `Integer`, se produciría un error de tiempo de ejecución en la línea 12.

En las líneas de la 10 a la 16, con una recopilación no genérica, se utiliza un elemento `Iterator` para iterar con la lista de elementos. Observe que es necesario realizar un gran número de conversiones para volver a extraer los objetos de la lista con el fin de imprimir los datos.

Al final, hay una gran cantidad de “azúcar sintáctica” (código adicional) innecesaria que funciona con las recopilaciones de esta forma.

Si se comenta la línea que agrega el elemento `String` a la `ArrayList`, el programa produce la siguiente salida:

```
Part number: 1111
Part number: 2222
Part number: 3333
```

ArrayList genérica

```
1 import java.util.ArrayList;
2 import java.util.List;
3
4 public class GenericArrayList {
5 public static void main(String args[]){
6
7    List<Integer> partList = new ArrayList<>(3);
8
9    partList.add(new Integer(1111));
10    partList.add(new Integer(2222));
11    partList.add(new Integer(3333));
12    partList.add(new Integer(4444)); // ArrayList auto grows
13
14    System.out.println("First Part: " + partList.get(0)); // First item
15    partList.add(0, new Integer(5555)); // Insert an item by index
16
17    // partList.add("Bad Data");  // compile error now
```

ORACLE

Con los genéricos, todo es más sencillo. Cuando la `ArrayList` se inicializa en la línea 6, cualquier intento de agregar un valor no válido (línea 15) da como resultado un error de tiempo de compilación.

En el ejemplo de la diapositiva se muestra una serie de funciones de `ArrayList`.

- La línea 12 muestra cómo una `ArrayList` crece automáticamente al agregar un elemento con un tamaño superior al original.
- La línea 14 muestra cómo puede acceder el índice a los elementos.
- La línea 15 muestra cómo se pueden insertar los elementos en la lista según el índice.

Nota: en la línea 7, la `ArrayList` se asigna a un tipo `List`. El uso de este estilo permite intercambiar la implantación de `List` sin cambiar otro tipo de código.

`ArrayList` genérica: Iteración y empaquetado

```
for (Integer partNumberObj:partList){
      int partNumber = partNumberObj; // Demos auto unboxing
      System.out.println("Part number: " + partNumber);
}
```

- El bucle `for` mejorado o el bucle `for-each` proporcionan un código más limpio.
- La conversión no se realiza debido al empaquetado automático y el desempaquetado.

El uso del bucle `for-each` es más sencillo y proporciona un código más limpio. No se realizan las conversiones debido a la función de desempaquetado automático de Java.

Empaquetado automático y desempaquetado

- Simplifica la sintaxis.
- Produce un código más limpio y fácil de leer.

```
1 public class AutoBox {
2    public static void main(String[] args){
3        Integer intObject = new Integer(1);
4        int intPrimitive = 2;
5
6        Integer tempInteger;
7        int tempPrimitive;
8
9        tempInteger = new Integer(intPrimitive);
10        tempPrimitive = (int) intObject.intValue();
11
12        tempInteger = intPrimitive; // Auto box
13        tempPrimitive = intObject; // Auto unbox
```

ORACLE

Las líneas 9 y 10 muestran un método tradicional para el movimiento entre objetos y primitivos. Las líneas 12 y 13 muestran el empaquetado y desempaquetado.

Empaquetado automático y desempaquetado

El empaquetado automático y el desempaquetado son funciones del lenguaje Java que le permiten realizar asignaciones razonables sin el uso de la sintaxis de conversión formal. Java ofrece las conversiones en el momento de la compilación.

Nota: tenga cuidado a la hora de utilizar el empaquetado automático en un bucle. El uso de esta función implica un coste de rendimiento.

Prueba

¿Cuál de las siguientes opciones declara una `Integer ArrayList` con tres elementos?

a. `List<Integer> partList = new ArrayList<>(three);`
b. `List<Integer> partList = new ArrayList<>[3];`
c. `List<Integer> partList = new ArrayList<>(three);`
d. `List<Integer> partList = new ArrayList<>(3);`

Interfaz `Set`

- Set es una lista que contiene solo elementos únicos.
- Una interfaz Set no tiene índice.
- No se permite el uso de elementos duplicados en una interfaz Set.
- Puede iterar con elementos para acceder a ellos.
- `TreeSet` ofrece una implantación ordenada.

ORACLE

Por ejemplo, se puede utilizar un elemento Set para realizar un seguimiento de una lista de números de artículo únicos.

Interfaz `Set`: ejemplo

Una interfaz Set es una recopilación de elementos únicos.

```
1 public class SetExample {
2    public static void main(String[] args){
3        Set<String> set = new TreeSet<>();
4
5        set.add("one");
6        set.add("two");
7        set.add("three");
8        set.add("three"); // not added, only unique
9
10        for (String item:set){
11            System.out.println("Item: " + item);
12        }
13    }
14 }
```

ORACLE

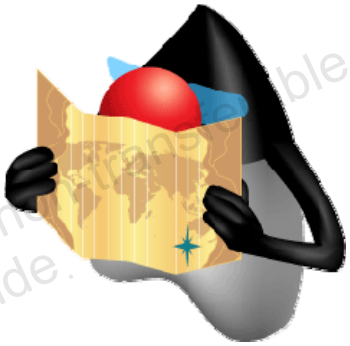
Una interfaz Set es una recopilación de elementos únicos. Este ejemplo utiliza un elemento `TreeSet`, que ordena los elementos en la interfaz Set. Si se ejecuta el programa, la salida es la siguiente:

```
Item: one
Item: three
Item: two
```

Interfaz `Map`

- Recopilación que almacena varios pares clave-valor.
  - Clave: identificador único de cada elemento de una recopilación
  - Valor: valor almacenado en el elemento asociado a la clave
- Se denomina “matriz asociativa” en otros lenguajes.

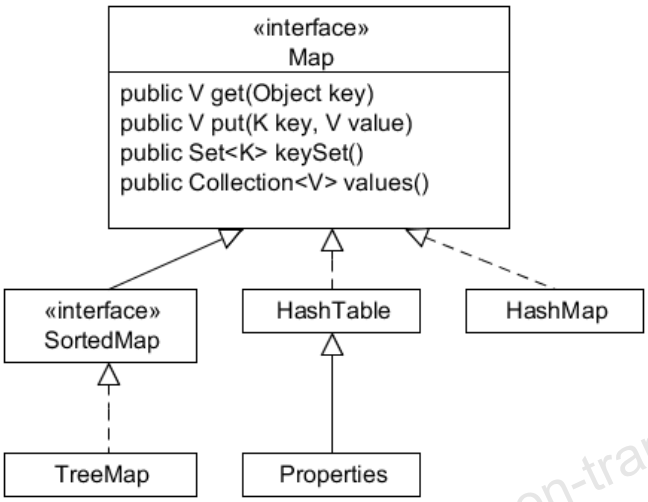
Clave	Valor
101	Blue Shirt
102	Black Shirt
103	Gray Shirt



ORACLE

`Map` resulta útil para realizar un seguimiento de elementos, como listas de artículos y sus descripciones (tal y como se muestra en la diapositiva).

Tipos de `Map`



La interfaz `Map` no amplía la interfaz `Collection` debido a que representa asignaciones en lugar de una recopilación de objetos. Entre las principales clases de implantación se incluyen:

- `TreeMap`: asignación en la que las claves se ordenan automáticamente.
- `HashTable`: implantación de matriz asociativa clásica con claves y valores. `HashTable` está sincronizada.
- `HashMap`: implantación similar a `HashTable` excepto en que acepta claves y valores nulos. Además, no está sincronizada.

Interfaz `Map`: ejemplo

```
• public class MapExample {
•    public static void main(String[] args){
•        Map <String, String> partList = new TreeMap<>();
•        partList.put("S001", "Blue Polo Shirt");
•        partList.put("S002", "Black Polo Shirt");
•        partList.put("H001", "Duke Hat");
•
•        partList.put("S002", "Black T-Shirt"); // Overwrite value
•        Set<String> keys = partList.keySet();
•
•        System.out.println("=== Part List ===");
•        for (String key:keys){
•            System.out.println("Part#: " + key + " " +
partList.get(key));
•        }
•    }
• }
```

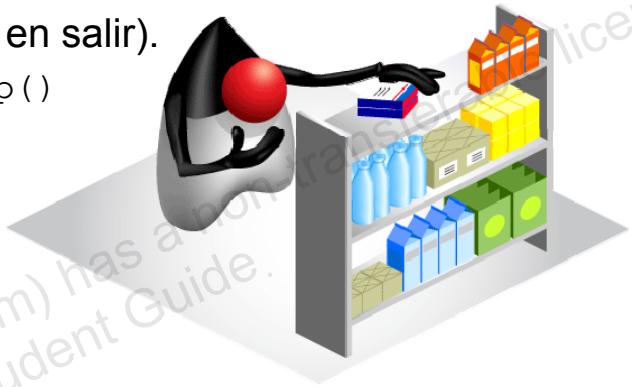

El ejemplo muestra cómo crear una interfaz `Map` y realizar operaciones estándar en ella. La salida del programa es:

```
=== Part List ===
Part#: 111111 Blue Polo Shirt
Part#: 222222 Black T-Shirt
Part#: 333333 Duke Hat
```

Interfaz `Deque`

Recopilación que se puede utilizar como pila o cola.

- Significa “cola de dos extremos” (y se pronuncia “deck”).
- Una cola proporciona operaciones FIFO (primero en entrar, primero en salir).
  - Métodos `add(e)` y `remove()`
- Una pila proporciona operaciones LIFO (último en entrar, primero en salir).
  - Métodos `push(e)` y `pop()`



ORACLE

`Deque` es una interfaz secundaria de `Collection` (al igual que `Set` y `List`).

Una cola se suele utilizar para realizar el seguimiento de solicitudes de mensaje asíncronas con el fin de procesarlas de forma ordenada. Puede ser muy útil para recorrer un árbol de directorios o estructuras similares.

Pila con Deque: ejemplo

```
1 public class TestStack {
2    public static void main(String[] args){
3        Deque<String> stack = new ArrayDeque<>();
4        stack.push("one");
5        stack.push("two");
6        stack.push("three");
7
8        int size = stack.size() - 1;
9        while (size >= 0 ) {
10           System.out.println(stack.pop());
11           size--;
12        }
13    }
14 }
```

ORACLE

Deque (pronunciado “deck”) es una “cola de dos extremos”. Esto significa fundamentalmente que Deque se puede utilizar como cola (operaciones FIFO [primero en entrar, primero en salir]) o como pila (operaciones LIFO [último en entrar, primero en salir]).

Ordenación de recopilaciones

- Las interfaces `Comparable` y `Comparator` se utilizan para ordenar recopilaciones.
  - Las dos se implantan mediante el uso de genéricos.
- Uso de la interfaz `Comparable`:
  - Sustituye al método `compareTo`.
  - Proporciona una única opción de ordenación.
- Uso de la interfaz `Comparator`:
  - Se implanta mediante el uso del método `compare`.
  - Permite crear varias clases `Comparator`.
  - Permite crear y utilizar distintas opciones de ordenación.

La API Collections ofrece dos interfaces para la ordenación de elementos: `Comparable` y `Comparator`.

La interfaz `Comparable` se implanta en una clase y proporciona una única opción de ordenación para la clase.

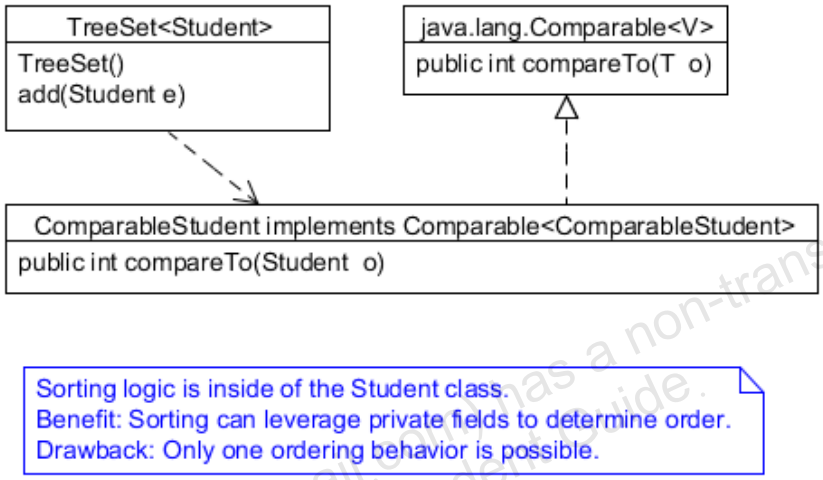
La interfaz `Comparator` permite crear varias opciones de ordenación. Puede conectar la opción diseñada cuando lo desee.

Las dos interfaces se pueden utilizar con recopilaciones ordenadas, como `TreeSet` y `TreeMap`.

Interfaz `Comparable`

Uso de la interfaz `Comparable`:

- Sustituye al método `compareTo`.
- Proporciona una única opción de ordenación.



La diapositiva muestra cómo la clase `ComparableStudent` está relacionada con la interfaz `Comparable` y `TreeSet`.

Comparable: ejemplo

```
1 public class ComparableStudent implements Comparable<ComparableStudent>{
2    private String name; private long id = 0; private double gpa = 0.0;
3
4    public ComparableStudent(String name, long id, double gpa){
5       // Additional code here
6    }
7    public String getName(){ return this.name; }
8       // Additional code here
9
10    public int compareTo(ComparableStudent s){
11        int result = this.name.compareTo(s.getName());
12        if (result > 0) { return 1; }
13        else if (result < 0){ return -1; }
14        else { return 0; }
15    }
16 }
```


El ejemplo de la diapositiva implanta la interfaz `Comparable` y el método `compareTo`. Observe que, puesto que la interfaz se ha diseñado mediante el uso de genéricos, los paréntesis angulares definen el tipo de clase transferida al método `compareTo`. Las sentencias `if` se incluyen para mostrar las comparaciones que se llevan a cabo. También puede únicamente devolver un resultado.

Los números devueltos tienen los siguientes significados.

- **Número negativo:** `s` es anterior al elemento actual.
- **Número positivo:** `s` es posterior al elemento actual.
- **Cero:** `s` es igual que el elemento actual.

En los casos en que la recopilación contenga valores equivalentes, sustituya el código que devuelve cero con código adicional que devuelva un número negativo o positivo.

Prueba de `Comparable`: ejemplo

```
1  public class TestComparable {
2    public static void main(String[] args){
3        Set<ComparableStudent> studentList = new TreeSet<>();
4
5        studentList.add(new ComparableStudent("Thomas Jefferson", 1111,
3.8));
6        studentList.add(new ComparableStudent("John Adams", 2222, 3.9));
7        studentList.add(new ComparableStudent("George Washington", 3333,
3.4));
8
9        for(ComparableStudent student:studentList){
10           System.out.println(student);
11        }
12    }
13 }
```


En el ejemplo de la diapositiva, se crea una `ArrayList` de elementos `ComparableStudent`. Una vez que se inicializa la lista, se ordena mediante la interfaz `Comparable`. La salida del programa es la siguiente:

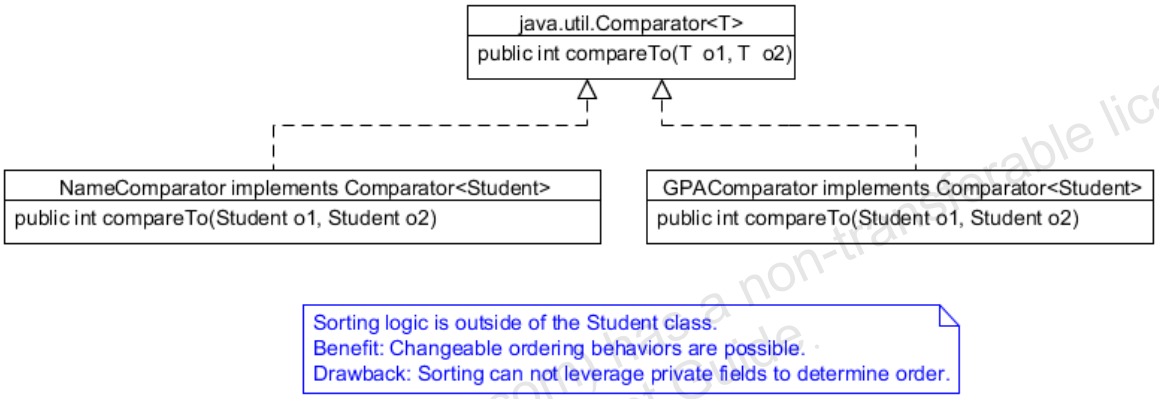
```
Name: George Washington  ID: 3333  GPA:3.4
Name: John Adams  ID: 2222  GPA:3.9
Name: Thomas Jefferson  ID: 1111  GPA:3.8
```

Nota: la clase `ComparableStudent` ha sustituido el método `toString()`.

Interfaz `Comparator`

Uso de la interfaz `Comparator`:

- Se implanta mediante el uso del método `compare`.
- Permite crear varias clases `Comparator`.
- Permite crear y utilizar distintas opciones de ordenación.



ORACLE

La diapositiva muestra dos clases `Comparator` que se pueden utilizar con la clase `Student`. El ejemplo de la diapositiva siguiente muestra cómo utilizar `Comparator` con una interfaz sin ordenar como `ArrayList` mediante el uso de la clase de utilidad `Collections`.

Comparator: ejemplo

```
public class StudentSortName implements Comparator<Student>{
    public int compare(Student s1, Student s2){
        int result = s1.getName().compareTo(s2.getName());
        if (result != 0) { return result; }
        else {
            return 0;  // Or do more comparing
        }
    }
}
```

```
public class StudentSortGpa implements Comparator<Student>{
    public int compare(Student s1, Student s2){
        if (s1.getGpa() < s2.getGpa()) { return 1; }
        else if (s1.getGpa() > s2.getGpa()) { return -1; }
        else { return 0; }
    }
}
```

ORACLE

El ejemplo de la diapositiva muestra las clases `Comparator` creadas para la ordenación según Name y GPA. Para la comparación de nombres, se han simplificado las sentencias `if`.

Prueba de `Comparator`: ejemplo

```
1  public class TestComparator {
2    public static void main(String[] args){
3        List<Student> studentList = new ArrayList<>(3);
4        Comparator<Student> sortName = new StudentSortName();
5        Comparator<Student> sortGpa = new StudentSortGpa();
6
7        // Initialize list here
8
9        Collections.sort(studentList, sortName);
10        for(Student student:studentList){
11            System.out.println(student);
12        }
13
14        Collections.sort(studentList, sortGpa);
15        for(Student student:studentList){
16            System.out.println(student);
17        }
18    }
19 }
```

ORACLE

El ejemplo de la diapositiva muestra cómo los dos objetos `Comparator` se utilizan con una recopilación.

Nota: se ha comentado código para ahorrar espacio.

Observe cómo los objetos `Comparator` se inicializan en las líneas 4 y 5. Una vez creadas las variables `sortName` y `sortGpa`, se pueden transferir al método `sort()` por el nombre. La ejecución del programa produce la siguiente salida.

```
Name: George Washington  ID: 3333  GPA:3.4
Name: John Adams  ID: 2222  GPA:3.9
Name: Thomas Jefferson  ID: 1111  GPA:3.8
Name: John Adams  ID: 2222  GPA:3.9
Name: Thomas Jefferson  ID: 1111  GPA:3.8
Name: George Washington  ID: 3333  GPA:3.4
```

Notas

- La clase de utilidad `Collections` proporciona una serie de métodos útiles para distintas recopilaciones. Entre los métodos se incluyen `min()`, `max()`, `copy()` y `sort()`.
- La clase `Student` ha sustituido el método `toString()`.

Prueba

¿Qué interfaz utilizaría para crear varias opciones de ordenación para una recopilación?

a. `Comparable`
b. `Comparison`
c. `Comparator`
d. `Comparinator`

Resumen

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

- Crear una clase genérica personalizada
- Utilizar el diamante de inferencia de tipo para crear un objeto
- Crear una recopilación sin utilizar genéricos
- Crear una recopilación mediante el uso de genéricos
- Implantar una `ArrayList`
- Implantar una interfaz `Set`
- Implantar un `HashMap`
- Implantar una pila mediante el uso de Deque
- Utilizar tipos enumerados

ORACLE

Visión general de la práctica 7-1: Recuento de números de artículo mediante el uso de un `HashMap`

En esta práctica, se abordan los siguientes temas:

- Creación de una asignación para almacenar un número de artículo y el recuento
- Creación de una asignación para almacenar un número de artículo y la descripción
- Procesamiento de la lista de artículos y generación de un informe

ORACLE

Visión general de la práctica 7-2: Coincidencia de paréntesis mediante Deque

En esta práctica se aborda el procesamiento de las sentencias de programación para asegurar que coincide el número de paréntesis.

ORACLE

Visión general de la práctica 7-3: Recuento de inventario y ordenación con elementos `Comparator`

En esta práctica se aborda el procesamiento de transacciones de inventario que generan dos informes ordenados de forma distinta mediante el uso de elementos `Comparator`.

ORACLE

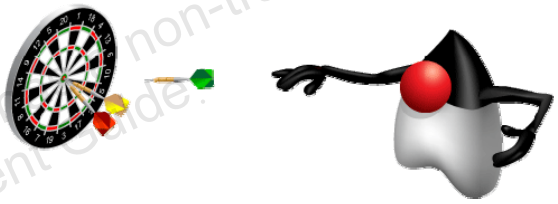
8

Procesamiento de cadenas

Objetivos

Al finalizar esta lección, debería estar capacitado para:

- Leer datos de la línea de comandos
- Buscar cadenas
- Analizar cadenas
- Crear cadenas mediante `StringBuilder`
- Buscar cadenas mediante el uso de expresiones regulares
- Analizar cadenas mediante el uso de expresiones regulares
- Sustituir cadenas mediante el uso de expresiones regulares



ORACLE

Argumentos de línea de comandos

- Todas las aplicaciones de tecnología Java pueden utilizar argumentos de línea de comandos.
- Estos argumentos de cadena se colocan en la línea de comandos para iniciar el intérprete de Java después del nombre de la clase:

```
java TestArgs arg1 arg2 "another arg"
```

- Cada argumento de la línea de comandos se coloca en la matriz args que se transfiere al método main estático:

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


Al iniciar un programa Java en una ventana de terminal, puede proporcionar el programa con cero o más argumentos de línea de comandos.

Los argumentos de línea de comandos permiten al usuario especificar la información de configuración de la aplicación. Estos argumentos son cadenas: pueden ser tokens autónomos (como arg1) o cadenas entre comillas (como "another arg").

Argumentos de línea de comandos

```
public class TestArgs {
   public static void main(String[] args) {
     for ( int i = 0; i < args.length; i++ ) {
        System.out.println("args[" + i + "] is '" +
                           args[i] + "'");
     }
   }
}
```

Ejecución de ejemplo:

```
java TestArgs "Ted Baxter" 45 100.25
args[0] is 'Ted Baxter'
args[1] is '45'
args[2] is '100.25'
```

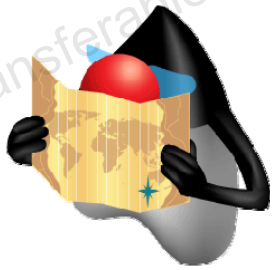

Los argumentos de línea de comandos se transfieren siempre al método main como cadenas, independientemente del tipo de destino. Si una aplicación requiere argumentos de línea de comandos que no sean del tipo `String` (por ejemplo, valores numéricos), la aplicación debe convertir los argumentos de cadena en sus tipos correspondientes mediante el uso de las clases envoltorio, como el método `Integer.parseInt`, que se puede utilizar para convertir el argumento de cadena que representa al número entero en el tipo `int`.

Propiedades

- La clase `java.util.Properties` se utiliza para cargar y guardar pares clave-valor en Java.
- Se pueden almacenar en un archivo de texto simple:

```
hostName = www.example.com
userName = user
password = pass
```

- El nombre del archivo termina en `.properties`.
- El archivo puede estar en cualquier ubicación en la que el compilador pueda encontrarlo.



ORACLE

La ventaja de un archivo de propiedades es la capacidad de definir valores para la aplicación de forma externa. El archivo de propiedades se suele leer al inicio de la aplicación y se utiliza para los valores por defecto. Sin embargo, el archivo de propiedades puede ser también una parte integral de un esquema de localización en el que almacena los valores de las etiquetas de menú y el texto para los distintos idiomas que puede soportar la aplicación.

La regla de nomenclatura para el archivo de propiedades es `<nombre_archivo>.properties`, aunque el archivo puede tener la extensión que desee. El archivo puede estar en cualquier ubicación en la que la aplicación pueda encontrarlo.

Carga y uso de un archivo de propiedades

```
1   public static void main(String[] args) {
2     Properties myProps = new Properties();
3     try {
4       FileInputStream fis = new FileInputStream("ServerInfo.properties");
5       myProps.load(fis);
6     } catch (IOException e) {
7       System.out.println("Error: " + e.getMessage());
8     }
9
10     // Print Values
11     System.out.println("Server: " + myProps.getProperty("hostName"));
12     System.out.println("User: " + myProps.getProperty("userName"));
13     System.out.println("Password: " + myProps.getProperty("password"));
14  }
```

ORACLE

En el fragmento de código, se crea un objeto `Properties`. A continuación, mediante una sentencia `try`, se abre un archivo relativo a los archivos de origen en el proyecto de NetBeans. Al cargarlo, los pares nombre-valor están disponibles para su uso en la aplicación.

Los archivos de propiedades permiten inyectar fácilmente información de la configuración u otros datos de la aplicación en la misma.

Carga de propiedades desde la línea de comandos

- La información sobre propiedades también se puede transferir en la línea de comandos.
- Utilice la opción –D para transferir pares clave-valor:

```
java -Dpropertyname=value -Dpropertyname=value myApp
```

- Por ejemplo, transfiera uno de los valores anteriores:

```
java -Dusername=user myApp
```

- Obtenga los datos de `Properties` del objeto `System`:

```
String userName = System.getProperty("username");
```


La información sobre propiedades también se puede transferir en la línea de comandos. La ventaja de la transferencia de propiedades desde la línea de comandos es la simplicidad, ya que no tiene que abrir un archivo y leerlo. Sin embargo, si tiene más de unos pocos parámetros, es preferible utilizar un archivo de propiedades.

`PrintWriter` y la consola

Si ya no desea utilizar `System.out.println()` para imprimir texto en la consola, existe una alternativa.

```
import java.io.PrintWriter;

public class PrintWriterExample {
    public static void main(String[] args){
        PrintWriter pw = new PrintWriter(System.out, true);
        pw.println("This is some output.");

    }
}
```

ORACLE

Si no desea introducir `System.out.whatever` una y otra vez, puede utilizar `PrintWriter` para guardar la información. En el ejemplo de la diapositiva se muestra cómo crear el objeto. La opción `true` es necesaria para forzar que `PrintWriter` vacíe las líneas impresas en la consola.

Formato `printf`

Java proporciona varias opciones para aplicar formato a las cadenas:

- `printf` y `String.format`

```
public class PrintfExample {
    public static void main(String[] args){
        PrintWriter pw = new PrintWriter(System.out, true);
        double price = 24.99; int quantity = 2; String color = "Blue";
        System.out.printf("We have %03d %s Polo shirts that cost
    $%3.2f.\n", quantity, color, price);
        System.out.format("We have %03d %s Polo shirts that cost
    $%3.2f.\n", quantity, color, price);
        String out = String.format("We have %03d %s Polo shirts that cost
    $%3.2f.", quantity, color, price);
        System.out.println(out);
        pw.printf("We have %03d %s Polo shirts that cost $%3.2f.\n",
    quantity, color, price);
    }
}
```

ORACLE

Puede aplicar el formato `printf` mediante el uso de la clase String y cualquier flujo de salida. En la diapositiva se muestran varios ejemplos de aplicación de formato a cadenas. Consulte la documentación de la API de Java para obtener más información sobre todas las opciones.

- **`%s`:** cadena
- **`%d`:** decimal
- **`%f`:** flotante

La salida del programa es la siguiente:

```
We have 002 Blue Polo shirts that cost $24.99.
We have 002 Blue Polo shirts that cost $24.99.
We have 002 Blue Polo shirts that cost $24.99.
We have 002 Blue Polo shirts that cost $24.99.
```

Prueba

¿Cuáles son las dos sentencias de impresión con formato válidas?

a. `System.out.printf("%s Polo shirts cost $%3.2f.\n", "Red", "35.00");`

b. `System.out.format("%s Polo shirts cost $%3.2f.\n", "Red", "35.00");`

c. `System.out.println("Red Polo shirts cost $35.00.\n");`

d. `System.out.print("Red Polo shirts cost $35.00.\n");`

ORACLE

Procesamiento de cadenas

- `StringBuilder` para construir la cadena
- Métodos de cadena incorporados
  - Búsqueda
  - Análisis
  - Extracción de subcadenas
- Análisis con `StringTokenizer`

ORACLE

La primera parte de esta sección abarca las funciones de cadena que no son expresiones regulares. Para realizar manipulaciones de cadenas simples, existe una serie de métodos incorporados muy útiles.

StringBuilder y StringBuffer

- `StringBuilder` y `StringBuffer` son las herramientas preferidas cuando la concatenación de cadenas no es trivial.
  - Más eficaces que "+"
- Simultaneidad
  - `StringBuilder` (sin protección de thread)
  - `StringBuffer` (con protección de thread)
- Definición de la capacidad con el tamaño realmente necesario.
  - El cambio continuo de tamaño del buffer también puede producir problemas de rendimiento.

Las clases `StringBuilder` y `StringBuffer` son la forma recomendable de concatenar cadenas.

StringBuilder: ejemplo

```
public class StringBuilding {
    public static void main(String[] args){
        StringBuilder sb = new StringBuilder(500);

        sb.append(", the lightning flashed and the thunder
    rumbled.\n");
        sb.insert(0, "It was a dark and stormy night");

        sb.append("The lightning struck...\n").append("[  ");
        for(int i = 1; i < 11; i++){
            sb.append(i).append("  ");
        }
        sb.append("] times");

        System.out.println(sb.toString());
    }
}
```


El ejemplo de la diapositiva muestra algunos métodos `StringBuilder` comunes. Puede utilizar `StringBuilder` para insertar texto en su posición. Se recomienda el encadenamiento de llamadas de adición para la creación de cadenas.

La salida del programa es la siguiente:

```
It was a dark and stormy night, the lightning flashed and the thunder
rumbled.
The lightning struck...
[  1  2  3  4  5  6  7  8  9  10 ] times
```

Métodos de cadena de ejemplo

```
1   public class StringMethodsExample {
2       public static void main(String[] args){
3           PrintWriter pw = new PrintWriter(System.out, true);
4           String tc01 = "It was the best of times";
5           String tc02 = "It was the worst of times";
6
7           if (tc01.equals(tc02)){
8               pw.println("Strings match..."); }
9           if (tc01.contains("It was")){
10              pw.println("It was found"); }
11          String temp = tc02.replace("w", "zw");
12          pw.println(temp);
13          pw.println(tc02.substring(5, 12));
14      }
15  }
```


El código de la diapositiva muestra algunos de los métodos de cadena más útiles de la clase String.

- **`equals()`:** prueba la igualdad del contenido de dos cadenas. Se prefiere a `==`, que prueba si dos objetos apuntan a la misma referencia.
- **`contains()`:** busca una cadena para ver si contiene la cadena proporcionada.
- **`replace()`:** busca la cadena determinada y sustituye las instancias con la cadena de destino proporcionada. Hay un método `replaceFirst()` para sustituir solo la primera instancia.
- **`substring()`:** devuelve una cadena en función de su posición en la cadena.

La ejecución de los programas que se muestran en la diapositiva devuelve la siguiente salida:

```
It was found
It zwas the zworst of times
s the w
```

Uso del método `split()`

```
1   public class StringSplit {
2       public static void main(String[] args){
3           String shirts = "Blue Shirt, Red Shirt, Black
    Shirt, Maroon Shirt";
4
5           String[] results = shirts.split(", ");
6           for(String shirtStr:results){
7               System.out.println(shirtStr);
8           }
9       }
10  }
```


La forma más sencilla de analizar una cadena es utilizar el método `split()`. Llame al método con el carácter (o los caracteres) que dividirán la cadena. El resultado se captura en una matriz.

Nota: el delimitador se puede definir mediante el uso de expresiones regulares.

La salida del programa de la diapositiva es la siguiente:

```
Blue Shirt
Red Shirt
Black Shirt
Maroon Shirt
```

Análisis con `StringTokenizer`

```
1   public class StringTokenizerExample {
2       public static void main(String[] args){
3           String shirts = "Blue Shirt, Red Shirt, Black Shirt, Maroon
    Shirt";
4
5           StringTokenizer st = new StringTokenizer(shirts, ", ");
6
7           while(st.hasMoreTokens()){
8               System.out.println(st.nextToken());
9           }
10      }
11  }
```


La clase `StringTokenizer` realiza la misma función que `split()` pero con un enfoque diferente. Debe iterar los tokens para acceder a ellos. Tenga en cuenta también que el delimitador `", "` en este caso indica el uso de comas y espacios como delimitadores. Por tanto, el resultado del análisis es el siguiente:

```
Blue
Shirt
Red
Shirt
Black
Shirt
Maroon
Shirt
```

Scanner

Una clase `Scanner` puede convertir en un token una cadena o un flujo.

```
1       public static void main(String[] args) {
2           Scanner s = null;
3           StringBuilder sb = new StringBuilder(64);
4           String line01 = "1.1, 2.2, 3.3";
5           float fsum = 0.0f;
6
7           s = new Scanner(line01).useDelimiter(", ");
8           try {
9               while (s.hasNextFloat()) {
10                  float f = s.nextFloat();
11                  fsum += f;
12                  sb.append(f).append(" ");
13              }
14              System.out.println("Values found: " + sb.toString());
15              System.out.println("FSum: " + fsum);
16          } catch (Exception e) {
17              System.out.println(e.getMessage());
18          }
```

ORACLE

Una clase `Scanner` se puede utilizar para convertir en un token un flujo de entrada o una cadena. Además, la clase `Scanner` se puede utilizar para convertir números en tokens y en cualquier tipo de número primitivo. Observe cómo se define `Scanner` en la línea 7. El objeto resultante se puede iterar en función de un tipo específico. En este caso, se utiliza un tipo `float`.

La salida de este segmento de código es la siguiente:

```
Values found: 1.1 2.2 3.3
FSum: 6.6000004
```

Expresiones regulares

- Lenguaje para coincidencias de cadenas de texto
  - Vocabulario muy detallado
  - Búsqueda, extracción o búsqueda y sustitución
- Con Java, el uso de la barra invertida (\) es importante.
- Objetos Java
  - `Pattern`
  - `Matcher`
  - `PatternSyntaxException`
  - `java.util.regex`

`Pattern` y `Matcher`

- `Pattern`: define una expresión regular
- `Matcher`: especifica una cadena de búsqueda

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class PatternExample {
    public static void main(String[] args){
        String t = "It was the best of times";

        Pattern pattern = Pattern.compile("the");
        Matcher matcher = pattern.matcher(t);

        if (matcher.find()) { System.out.println("Found match!"); }
    }
}
```


Los objetos `Pattern` y `Matcher` funcionan de forma conjunta para proporcionar una solución completa.

El objeto `Pattern` define la expresión regular que se utilizará para la búsqueda. Como se muestra en el ejemplo, una expresión regular puede ser tan sencilla como una palabra o frase.

El objeto `Matcher` se utiliza para seleccionar la cadena de destino que se va a buscar. Hay disponible una serie de métodos para el objeto matcher. Estos métodos se tratan en las siguientes diapositivas.

Al ejecutarlo, el ejemplo produce la siguiente salida:

```
Found match!
```

Clases de caracteres

Carácter	Descripción
`.`	Coincide con cualquier carácter único (letra, dígito o carácter especial), salvo marcadores de final de línea.
`[abc]`	Coincidiría con “a”, “b” o “c” en esa posición.
`[^abc]`	Coincidiría con cualquier carácter que no fuera “a”, “b” o “c” en esa posición.
`[a-c]`	Rango de caracteres (en este caso, “a”, “b” y “c”).
|	Alternancia; básicamente un indicador “or”.

ORACLE

Las clases de caracteres le permiten buscar coincidencias con un carácter de varias formas.

Clase de caracteres: ejemplos

Cadena de destino	It was the best of times

Patrón	Descripción	Texto de coincidencia
`w.s`	Cualquier secuencia que empiece por “w” seguida de cualquier carácter seguido de “s”.	It was the best of times
`w[abc]s`	Cualquier secuencia que empiece por “w” seguida de “a”, “b” o “c” y, a continuación, “s”.	It was the best of times
`t[^aeo]mes`	Cualquier secuencia que empiece por “t” seguida de cualquier carácter que no sea “a”, “e” u “o” seguida de “mes”.	It was the best of times

ORACLE

Puede encontrar el código de este ejemplo en el proyecto `StringExamples` en el archivo `CustomCharClassExamples.java`.

Código de clase de caracteres: ejemplos

```
1   public class CustomCharClassExamples {
2       public static void main(String[] args) {
3           String t = "It was the best of times";
4
5           Pattern p1 = Pattern.compile("w.s");
6           Matcher m1 = p1.matcher(t);
7           if (m1.find()) { System.out.println("Found: " + m1.group());
8           }
9
10          Pattern p2 = Pattern.compile("w[abc]s");
11          Matcher m2 = p2.matcher(t);
12          if (m2.find()) { System.out.println("Found: " + m2.group());
13          }
14
15          Pattern p3 = Pattern.compile("t[^eou]mes");
16          Matcher m3 = p3.matcher(t);
17          if (m3.find()) { System.out.println("Found: " + m3.group());
18          }
```

ORACLE

En el ejemplo de la diapositiva se muestran dos formas de buscar “`was`” y una forma de buscar “`times`”.

Para que esto suceda, en Java:

1. Cree un objeto `Pattern` para almacenar la expresión regular con la que desea realizar la búsqueda.
2. Cree un objeto `Matcher` mediante la transferencia del texto que va a buscar al objeto `Pattern` y la devolución de un objeto `Matcher`.
3. Llame a `Matcher.find()` para buscar el texto con el objeto `Pattern` definido.
4. Llame a `Matcher.group()` para mostrar los caracteres que coinciden con el patrón.

Clases de caracteres predefinidas

Carácter predefinido	Clase de caracteres	Carácter negado	Clase negada
\d (dígito)	[0-9]	\D	[^0-9]
\w (carácter alfanumérico)	[a-zA-Z0-9_]	\W	[^a-zA-Z0-9_]
\s (espacio en blanco)	[\r\t\n\f\0XB]	\S	[^ \r\t\n\f\0XB]

ORACLE

Hay varias clases de caracteres que se utilizan de forma repetida. Estas clases se convierten en clases de caracteres predefinidas. Las clases existen para identificar dígitos, caracteres alfanuméricos y espacios en blanco.

Caracteres de espacio en blanco

\t: carácter de tabulación

\n: carácter de nueva línea

\r: retorno de carro

\f: avance de página

\x0B: tabulador vertical

Clases de caracteres predefinidas: ejemplos

Cadena de destino	Jo told me 20 ways to San Jose in 15 minutes.

Patrón	Descripción	Texto de coincidencia
`\\d\\d`	Buscar dos dígitos.**	Jo told me 20 ways to San Jose in 15 minutes.
`\\sin\\s`	Buscar “in” entre dos espacios y, a continuación, los tres caracteres siguientes.	Jo told me 20 ways to San Jose in 15 minutes.
`\\Sin\\S`	Buscar “in” entre dos caracteres que no sean de espacio y, a continuación, los tres caracteres siguientes.	Jo told me 20 ways to San Jose in 15 minutes.

ORACLE

** Si hay más coincidencias en la línea actual, las llamadas adicionales a `find()` devolverán la siguiente coincidencia en la misma línea.

Ejemplo:

```
Pattern p1 = Pattern.compile("\\d\\d");
        Matcher m1 = p1.matcher(t);
        while (m1.find()){
            System.out.println("Found: " + m1.group());
        }
```

Produce:

```
Found: 20
Found: 15
```

Puede encontrar el código de este ejemplo en el proyecto `StringExamples` en el archivo `PredefinedCharClassExample.java`.

Cuantificadores

Cuantificador	Descripción
`*`	El carácter precedente se repite cero o más veces.
`+`	El carácter precedente se repite una o más veces.
`?`	El carácter precedente debe aparecer una vez o ninguna.
`{n}`	El carácter precedente aparece exactamente *n* veces.
`{m,n}`	El carácter precedente aparece de *m* a *n* veces.
`{m,}`	El carácter precedente aparece *m* o más veces.
`(xx){n}`	Este grupo de caracteres se repite *n* veces.

ORACLE

Los cuantificadores le permiten seleccionar fácilmente un rango de caracteres en las consultas.

Cuantificador: ejemplos

Cadena de destino	Longlonglong ago, in a galaxy far far away

Patrón	Descripción	Texto de coincidencia
ago.*	Buscar “ago” y 0 o todos los caracteres restantes en la línea.	Longlonglong ago, in a galaxy far far away
gal.{3}	Coincidir con “gal” y los tres caracteres siguientes. Esto sustituye a “…” como se utiliza en un ejemplo anterior.	Longlonglong ago, in a galaxy far far away
(long){2}	Buscar “long” repetido dos veces.	Longlonglong ago, in a galaxy far far away

ORACLE

Puede encontrar el código de este ejemplo en el proyecto `StringExamples` en el archivo `QuantifierExample.java`.

Voracidad

- Una expresión regular intenta recuperar siempre tantos caracteres como sea posible.
- Utilice el operador `?` para limitar la búsqueda al menor número de coincidencias posible.

Cadena de destino	Longlonglong ago, in a galaxy far far away.

Patrón	Descripción	Texto de coincidencia
`ago.*far`	Una expresión regular recupera siempre el mayor número de caracteres posible.	Longlonglong ago, in a galaxy far far away.
`ago.*?far`	El carácter “?” básicamente desactiva la voracidad.	Longlonglong ago, in a galaxy far far away.

ORACLE

Una expresión regular intenta siempre buscar coincidencias con caracteres que devuelvan el máximo de caracteres. Esto se conoce como “principio de voracidad”. Utilice el operador `?` para limitar el resultado al menor número de caracteres necesarios para coincidir con el patrón.

Puede encontrar el código de este ejemplo en el proyecto `StringExamples` en el archivo `GreedinessExample.java`.

Prueba

¿Qué símbolo significa que el carácter se repite una o más veces?

a. *
b. +
c. .
d. ?

Coincidencias de límite

Fijación	Descripción
`^`	Coincide con el principio de una línea.
`$`	Coincide con el final de una línea.
`\b`	Coincide con el inicio o el final de una palabra.
`\B`	*No* coincide con el principio o el final de una palabra.

ORACLE

Los caracteres de límite se pueden utilizar para la coincidencia con distintas partes de una línea.

Límite: ejemplos

Cadena de destino	it was the best of times or it was the worst of times

Patrón	Descripción	Texto de coincidencia
`^it.*?times`	Secuencia que empieza una línea con “it” seguido de algunos caracteres y “times”, con la voracidad desactivada.	It was the best of times or it was the worst of times
`\\sit.*times$`	Secuencia que empieza con “it” seguido de algunos caracteres y termina la línea con “times”.	It was the best of times or it was the worst of times
`\\bor\\b.{3}`	Buscar “or” entre límites de palabras y los tres caracteres siguientes.	It was the best of times or it was the worst of times

ORACLE

Puede encontrar el código de este ejemplo en el proyecto `StringExamples` en el archivo `BoundaryCharExample.java`.

Prueba

¿Qué símbolo coincide con el final de línea?

a. *
b. +
c. $
d. ^

Coincidencia y grupos

Cadena de destino	**george.washington@example.com**
Coincidencia de 3 partes	**(george).(washington)@(example.com)**
Números de grupo	**(1).(2)@(3)**
Patrón	**(\\S+?)\\.(\\S+?)\\@(\\S+)**

Con las expresiones regulares, puede utilizar los paréntesis para identificar partes de una cadena para la coincidencia. En el ejemplo, se coincide con las distintas partes de una dirección de correo electrónico. Observe cómo se ha numerado cada par de paréntesis. En una expresión regular, `group(0)` coincide con todas las coincidencias de texto del uso de grupos. A continuación, se muestra el código fuente del ejemplo:

```
public class MatchingExample {
    public static void main(String[] args){
        String email = "george.washington@example.com";

        Pattern p1 = Pattern.compile("(\\S+?)\\.(\\S+?)\\@(\\S+)");
        Matcher m1 = p1.matcher(email);
        if (m1.find()){
            System.out.println("First: " + m1.group(1));
            System.out.println("Last: " + m1.group(2));
            System.out.println("Domain: " + m1.group(3));
            System.out.println("Everything Matched: " + m1.group(0));
        }
    }
}
```

Uso del método `replaceAll`

Con el método `replaceAll` puede buscar y sustituir elementos.

```
public class ReplacingExample {
    public static void main(String[] args){
        String header = "<h1>This is an H1</h1>";

        Pattern p1 = Pattern.compile("h1");
        Matcher m1 = p1.matcher(header);
        if (m1.find()){
            header = m1.replaceAll("p");
            System.out.println(header);
        }
    }
}
```

ORACLE

Puede buscar y sustituir elementos mediante el uso del método `replaceAll` después de realizar una búsqueda.

La salida del programa es la siguiente:

```
<p>This is an H1</p>
```

Resumen

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

- Leer datos de la línea de comandos
- Buscar cadenas
- Analizar cadenas
- Crear cadenas mediante `StringBuilder`
- Buscar cadenas mediante el uso de expresiones regulares
- Analizar cadenas mediante el uso de expresiones regulares
- Sustituir cadenas mediante el uso de expresiones regulares

ORACLE

Visión general de la práctica 8-1: Análisis de texto con `split()`

En esta práctica se aborda el uso del método `String.split()` para analizar texto.

ORACLE

Visión general de la práctica 8-2: Creación de un programa de búsqueda de expresiones regulares

En esta práctica se aborda la creación de un programa que busque un archivo de texto mediante una expresión regular.

ORACLE

Visión general de la práctica 8-3: Transformación de HTML mediante expresiones regulares

Esta práctica aborda la transformación del código HTML de un archivo mediante el uso de varias expresiones regulares.

ORACLE