

CAPÍTULO 2

1 / 1

» Orientación a Objetos Encapsulación

La encapsulación es un principio fundamental de la programación orientada a objetos y consiste en ocultar el estado interno del objeto y obligar a que toda interacción se realice a través de los métodos del objeto.

Mantiene las variables de instancia protegidas.

Crea métodos de acceso público para modificar los valores a la variable de instancia.

Herencia

La herencia es un mecanismo que permite la definición de una clase a partir de la definición de otra ya existente. Permite compartir automáticamente métodos y datos entre clases, subclases y objetos.

Hay dos tipos de herencia: Simple y Multiple. La primera indica que se pueden definir nuevas clases solamente a partir de una clase inicial mientras que la segunda indica que se pueden definir nuevas clases a partir de dos o más clases iniciales. Java solo permite herencia simple.

Tres razones de uso más comunes para el uso de herencia son:

- Para promover el uso del código
- Para el uso de polimorfismos

Términos de Orientado a Objetos para la herencia son los siguientes:

Vehículo es la superclase de carro.

Carro es la subclase de vehículo.

Carro es la superclase de subaru

Subaru es la subclase de vehículo.

Carro hereda de vehículo.

Subaru hereda de vehículo y carro

Subaru es derivado de carro

Carro es derivado de vehículo

Subaru es derivado de vehículo

Relación IS-A. las siguientes afirmaciones son ciertas

Carro se extiende de vehículo significa carro IS-A vehículo. Subaru se extiende de carro significa subaru IS-A CARRO.

Vehículo



Carro se extiende de vehículo



Subaru se extiende de carro.

HAS-A

Las relaciones se basan en el uso, el lugar de herencia, se refiere a la instancia de la clase.

Polimorfismo

Es la capacidad que tienen los objetos de una clase en ofrecer respuesta distinta e independiente en función de los parámetros (diferentes implementaciones) utilizados durante su invocación. Dicho de otro modo el objeto como entidad puede contener valores de diferentes tipos durante la ejecución del programa.

- * Una variable de referencia puede ser de un solo tipo, y una vez declarada, ese tipo nunca podrá cambiarse (aunque el objeto al que hace referencia puede cambiar).
- * Una referencia es una variable, por lo que se puede reasignar a otros objetos (a menos que la referencia se declare definitiva).
- * El tipo de una variable de referencia determina los métodos que se pueden invocar en el objeto al que hace referencia la variable.
- * Una variable de referencia puede referirse a cualquier objeto del mismo tipo que la referencia declarada o, este es el más grande y puede referirse a cualquier subtipo del tipo declarado.
- * Una variable de referencia se puede declarar como un tipo de clase o un tipo de interfaz si la variable se declara como un tipo de interfaz, puede hacer referencia a cualquier clase que implemente la interfaz.

Java soporta únicamente de una sola herencia, esto quiere decir que una clase puede tener una sola inmediata superclase, no más de uno.

En algunos lenguajes (como C++) permiten que una clase se extienda a más de una clase, esta capacidad se le conoce como "múltiples herencias". La razón por la que el creador de Java no permitió las "herencias múltiples" fue porque servía muy enmarañado.

Método polimórfico se aplica la invocación solamente para los métodos de instancia

» Sobreescripción / Sobre carga

► Métodos Sobreescritos

Reglas para Métodos Sobreescritos

- * La lista de argumentos debe coincidir exactamente para el método sobreescrito. Si no coinciden, tu puedes terminarla como una sobre carga del método que no pretendías.
- * El tipo de retorno debe ser el mismo que, o un subtipo de el tipo de retorno declarado en el método original anulado en la superclase.
- * Los niveles de acceso no pueden ser más restrictivos que de los métodos sobreescritos.
- * Los niveles de acceso pueden ser menos restrictivos que los métodos sobreescritos.
- * Los métodos de instancia pueden ser sobreescritos solo si son heredados por la subclase. Una subclase con el mismo paquete como la superclase de instancia puede sobreescribirse cualquier método de clase que no sea marcado private o final. Una subclase que no sea del mismo paquete solamente puede sobreescribir aquellos métodos públicos o protegidos que no sean final.
- * El método sobreescrito puede lanzar cualquier excepción sin marcar en tiempo de ejecución independientemente de si el método sobreescrito declara la excepción.

- El método sobrescrito no debe lanzar excepciones marcadas que sean nuevas o más amplias que las declaradas por el método anulado.
- El método sobrescrito puede lanzar más o menos excepciones. Esto porque un método sobrescrito toma riesgos no significa que la excepción de subclase sobrescrita tome los mismos riesgos.
- Tu no puedes sobreescribir un método como marca final.
- Tu no puedes sobreescribir un método con marca static.
- Si un método no puede ser heredado, no es posible ser sobrescrito.

Invocando una versión supervclase de un método sobrescrito.

El super funciona para invocar un método sobreescrito sólo aplicando para la instancia del método.

» Métodos Sobrecargados

Sobrecarga:

Son aquellos métodos que reusan el mismo nombre en una clase, pero con diferentes argumentos.

- Los métodos sobrecargados deben cambiar el tipo de retorno.
- Los métodos sobrecargados pueden cambiar su tipo de retorno.

- los métodos sobrecargados pueden cambiar los modificadores de acceso.
- los métodos sobrecargados pueden declarar nuevas o viejas excepciones.

» Invocando Métodos Sobrecargados

Diciendo cuales de los métodos repetidos para invocar es basándose en los argumentos del mismo. Si se pasa el método con argumentos string, la versión sobrecargada toma la string. Y si se invoca al mismo método pero con argumentos de tipo float, toma la versión sobrecargada que toma el float.

» Polimorfismo en Métodos sobrecargados y sobreescritos

Código de Invocación Método → Resultado

Animal a = new Animal(); Generic Animal
a.eat(); Eating Generic

Horse h = new Horse(); Horse eating hay
h.eat();

Animal ah = new Horse(); Horse eating hay
ah.eat();

El polimorfismo funciona: el tipo de objeto real (caballo) no el tipo de referencia (Animal) se utiliza para determinar a que se llama eat().

Horse he = new Horse(); Horse eating apples
he.eat("Apples");

El método sobrecargado eat (Strings) es invocado.

Animal az = new Animal (); Error de compilador
az.eat("treats");

El compilador ve que la clase Animal un método eat que tome un String.

Animal ahz = new Horse (); Error de compilador
ahz.eat ("carrots");

El compilador todavía ve solo en la referencia, y ve que Animal no tiene un método eat que tome un string. El compilador no le importa que el objeto actual podría ser un caballo en tiempo de ejecución.

» Diferencias entre métodos de sobrecarga y Sobreescripción.

	MÉTODOS SOBRECARGADOS	MÉTODOS SOBREESCRITOS
ARGUMENTOS	Debe cambiarse	No debe cambiarse
TIPO DE REFERENCIA	Puede cambiarse	No puede cambiarse, excepción para retornos o variantes.
EXCEPCIONES	Puede cambiarse	Puede reducirse o eliminarse
ACCESOS	Puede cambiarse	No debe ser más restrictivo (puede ser menos restrictivo.)
INVOCACIÓN	Tipo de referencia que determina cual versión palabram, el tipo de la	Tipo de objeto (en otras

de sobrecarga (basado en la instancia actual en el en los tipos de argumentos) determina los declarados) es seleccionado. El método actualizado en la ejecución. que es invocado es todavía un método virtual de invocación que pasa en el tiempo de ejecución, pero el compilador ya sabrá la firma del método a invocar. Aunque esté en tiempo de ejecución, el argumento de coincidencia ya habrá sido clavado, simplemente no en la clase en la que vive el método.

» Casteando Variables de Referencia

Hemos visto como es posible y comúnmente utilizar tipos de variables de referencia genéricos para referirse a tipos de objetos más específicos. Esto es el corazón del polimorfismo.

No se puede castear objetos que no sean de la misma herencia.

») Implementando una Interface

Cuando se implementa una interfaz, acepta adherirse al contrato definido en la interfaz. Eso significa que estás aceptando proporcionar implementaciones legales para cada método definido en la interfaz, y que cualquiera que sepa cómo se ven los métodos de la interfaz puede estar seguro de que puede invocar esos métodos en una instancia de su clase de implementación.

Por ejemplo, si crea una clase que implementa la interfaz Runnable (para que su código pueda ser ejecutado por un hilo específico), debe proporcionar el método public void run(). De lo contrario, podría decirse al subprocesso deficiente que ejecute el código del objeto Runnable y, sorprendentemente, el subprocesso descubriría que el objeto no tiene un método run().

Si la clase dice que está implementando una interfaz, servir mejor tener una implementación para cada método en la interfaz.

Clases de implementación debe adherir las mismas reglas para el método de implementación como una clase extendida en una clase abstracta.

Para ser una clase de implementación legal en orden, una clase de implementación no abstracta debe de seguir lo siguiente:

- Proporcionar implementaciones concretas (no abstractas) para todos los métodos desde la interfaz declarada.
- Siga todas las reglas para sobreescritura legales.
- No declarar excepciones comprobadas en los métodos de implementación distintos a los declarados por el método de interfaz, o subclases de los declarados por el método de interfaz.
- Mantener la firma del método de interfaz y mantener el mismo tipo de retorno (o un subtipo).

Dos reglas que más necesitas saber y luego podemos poner este tema en repaso.

1. Una clase implementa más que una interface. Esto es perfectamente legal decir, por ejemplo lo siguiente:
public class Ball implements Bounceable, Serializable, Runnable { ... }
Tu puedes extender una sola clase, pero implementando múltiples interfaces.
2. Una interface puede extenderse con otra interface, pero nunca implementar cualquiera.
El siguiente código es perfectamente legal:
public interface Bounceable extends Moveable { }