Resumen para presentar el examen 1Z0-804 Oracle Certified Profesional Java SE 7

Libro de S G Ganesh y Tushar Sharma

Capítulo 3: Diseño de Clases en Java

Tópicos del examen:

* Uso de modificadores de acceso: private, protected y public
* Override method (Métodos sobre-escritos)
* Overload constructors and methods (Sobre-cargar constructores y métodos)
* Uso del operador instanceof y casting
* Virtual method invocation (invocación virtual de métodos)
* Sobre-escribir métodos de la clase Object para mejorar la funcionalidad
* Usar las sentencias package e import

Java es un lenguaje de programación orientada a objetos OOP. Este paradigma de programación ayuda al desarrollador a lograr un sistema modular, extensible, mantenible y reusable. Para escribir programas de buena calidad, el programador deberá tener un buen aprendizaje de los conceptos de OOP.

Encapsulación y abstracción

Encapsulación combina los datos (campos) de la clase y las operaciones lógicamente relacionadas (métodos). La abstracción esconde los detalles del nivel interno de implementación y expone solo los detalles relevantes de la clase a los usuarios. La abstracción se logra a través de la encapsulación.

Herencia

La herencia es el mecanismo de reusabilidad, con el cual se agrupan las propiedades comunes de varias clases de objetos en una sola super-clase, también denominada clase base o clase padre, la cual se irá especializando, extendiendo su funcionalidad a través de las subclases que hereden de ella; también denominadas clases hija.

Polimorfismo

El polimorfismo hace referencia a las múltiples formas que puede tomar una entidad-objeto dependiendo del contexto en el que se aplique o utilice.

Fundamentales de clases

La clase es la entidad de abstracción fundamental en la OOP. Una clase encapsula el estado (datos-campos) y el comportamiento (operaciones-métodos) de una entidad.

Creación de un objeto

Debes programar clases de objetos para usarlas en tus programas. Puedes crear una instancia de la clase. En resumen estos son los tres pasos para crear un objeto a partir de una clase.

1. Declaración: Proveer la declaración de un nombre de variable que consiste en el nombre de la clase que se quiere utilizar y el nombre que identificará (apuntará) al objeto ya creado. Por ejemplo, la siguiente sentencia declara un objeto de tipo Circle y lo identifica con el nombre de variable circleObj: Circle circleObj;
2. Instanciación: usando la palabra “new” para crear una instancia; esta palabra clave reserva y aloja la memoria requerida por el objeto. En este ejemplo, instanciamos el objeto circleObj usando la siguiente sentencia: Circle circleObj = new Circle();
3. Inicialización: un método especial llamado constructor es llamado automáticamente. El constructor inicializa el nuevo objeto.

Constructores

Cada vez que creamos un objeto, un constructor de esa clase es llamado. Puedes hacer uso del constructor para inicializar el nuevo objeto creado, ajustando el estado inicial del objeto e incluso utilizar algunos recursos como manejadores de archivos etc. La principal regla de los métodos constructores es que tengan el mismo nombre de la clase. Una clase puede tener más de un constructor.

Todas las clases tienen al menos un constructor. Si no definimos explícitamente un método constructor para nuestra clase, java agrega uno sin parámetros a la clase al momento de compilar el código fuente.

Imagina que estás implementando la Clase Circle en la aplicación FunPaint. Un círculo debe guardar su centro y su radio, por lo que esta clase deberá tener tres campos: xPos, yPos y radius, entonces el código se vería de la siguiente forma:

class Circle {

int xPos, yPos, radius;

}

¿Qué pasa cuando creas un nuevo objeto Circle (círculo)? Los valores de xPos, yPos y radius serán inicializados a cero por default, ya que pertenecen al tipo entero (int). Sin embargo, esto no es deseable para crear un objeto círculo propiamente. Entonces tendrás que definir valores por default para las variables mencionadas en el constructor por default:

class Circle {

int xPos, yPos, radius;

public Circle() {

xPos = 20;

yPos = 20;

radius = 10;

}

}

Como podrás ver en el ejemplo anterior, el método constructor tiene el mismo nombre de la clase y no tiene ningún tipo de retorno. El constructor por default (como el del ejemplo anterior) no tiene ningún argumento. Un constructor por default es invocado cuando creas un nuevo objeto sin pasar argumentos a su constructor.

Ahora revisa cuándo es invocado este constructor por default o cuándo no cuando instancies un objeto de la Clase Circle. Para esto implementarás el método toString() para imprimir los valores de los campos del objeto.

class Circle {

int xPos, yPos, radius;

// Constructor por default

public Circle() {

xPos = 20;

yPos = 20;

radius = 10;

}

// Sobre-escribir (overridden) el método toString()

public String toString() {

return “center = (” + xPos + “,” + “yPos” + “) and radius = ” + radius;

}

// Método maestro de inicio de la aplicación

public static void main(String[] args) {

System.out.println(new Circle());

}

}

Este programa imprimirá en pantalla: center = (20,20) and radius = 10

Con esto se demuestra que el constructor por default funciona bien.

Modificadores de acceso

El examen incluye preguntas directas e indirectas a cerca de los modificadores de acceso que requieren un buen dominio del tema. Por esto es importante entender la variedad de modificadores de acceso soportados en Java.

Los modificadores de acceso determinan el nivel de visibilidad (y por lo tanto de acceso) para una entidad de Java, la cual puede ser una clase, un método o un campo. Los modificadores de acceso permiten reforzar una efectiva encapsulación. Si todos los miembros de una clase pueden ser accedidos desde cualquier lado, no hay ninguna razón para poner esos miembros dentro de la clase y por lo tanto no se logra una buena encapsulación.

Java soporta cuatro tipos de modificadores de acceso:

* Public: Este modificador permite el acceso a una entidad de java desde cualquier parte
* Private: Este es el modificador más restrictivo y permite la visibilidad y el acceso a la entidad, exclusivamente por entidades dentro de la misma clase: campos y métodos. Cualquier intento por acceder a una entidad marcada como private desde fuera de la clase contenedora, resultará en un problema de compilación.
* Protected: Este modificador restringe la visibilidad y el acceso a la entidad, solo por entidades dentro de las subclases que hereden de la clase contenedora.
* Default (cuando no se especifica explícitamente): Es como el modificador anterior, con la restricción de uso a clases dentro del mismo package.

En esta tabla se describen los modificadores de acceso y su visibilidad

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Modificador de acceso | En la misma clase | Subclase en el mismo package | Subclase afuera del package | Otra clase dentro del package | Otra clase fuera del package |
| Public | Sí | Sí | Sí | Sí | Sí |
| Private | Sí | No | No | No | No |
| Protected | Sí | Sí | Sí | Sí | No |
| Default | Sí | Sí | No | Sí | No |

Sobrecarga de métodos

Los métodos en java se pueden sobrecargar, de tal manera que con un mismo nombre de método se puedan ejecutar instrucciones de diferente forma, de acuerdo a los parámetros que este reciba. Un ejemplo podría ser un método para calcular el área de un triángulo, el cual puede recibir dos parámetros la base y la altura, o, en el caso de triángulos rectángulos puede recibir tres, el valor del cateto adyacente, el cateto opuesto, así como el valor del ángulo y la operación se ejecuta de manera distinta, aunque el método se llame igual. El compilador de java enlazará cada método invocado en el código, de acuerdo al tipo de datos que se pasen como parámetros al mismo.

Para sobrecargar un método es necesario tener las siguientes consideraciones:

* El método debe tener el mismo nombre
* Se debe diferenciar, ya sea en el número de parámetros o en el tipo de datos que reciben o en el orden que están dispuestos
* Un método no se puede sobrecargar únicamente cambiando el tipo de dato que retorna
* Un método no se puede sobrecargar únicamente cambiando los modificadores de acceso y no-acceso

A continuación vamos a sobrecargar el método miMetodo:

void miMetodo() { // sin parámetros }

int miMetodo(int a) { // con un parámetro int }

int miMetodo(int a, int b) { // recibe dos parámetros int }

String miMetodo(String s) { // recibe un parámetro String}

Sobrecarga de constructores

Así como puedes sobrecargar un método, también se puede sobrecargar un constructor, siguiendo las mismas reglas mencionadas para los métodos, sin olvidar que los constructores no devuelven ningún valor.

Analiza la siguiente clase y observa cómo se sobrecarga su constructor.

public class Foo {

public Foo (){

System.out.println(“Constructor por default”)

}

private Foo (int bar) {

System.out.println(“Constructor con un parámetro int”)

}

}

El compilador de java enlazará en tiempo de compilación, el tipo de constructor o método que está siendo invocado para lo cual lleva a cabo un proceso que denominamos “Overload Resolution” (Resolución de sobrecarga).

Resolución de sobrecarga

Cuando defines métodos o constructores sobrecargados ¿Cómo le hace el compilador para saber cuál método está siendo invocado? Analiza el siguiente código y determina cuál podría ser su salida.

class OverResol {

public static void aMethod(int val) { System.out.println(“int”); }

public static void aMethod(short val) { System.out.println(“short”); }

public static void aMethod(Object val) { System.out.println(“Object”); }

public static void aMethod(String val) { System.out.println(“String”); }

public static void main(String[] args) {

// distintas llamadas al método sobrecargado

byte b = 9;

aMethod(b); // primera llamada

aMethod(9); // segunda llamada

Integer i = 9;

aMethod(i); // tercera llamada

aMethod(“9”); // cuarta llamada

}

}

// imprimirá en consola

short

int

Object

String

Aquí se explica como el compilador resuelve estas llamadas al método sobrecargado

* En la primera llamada, la sentencia es aMethod(b) donde la variable “b” es de tipo byte. No hay ningún aMethod definido que reciba byte como argumento. El tipo de dato más cercano (en tamaño) es short y no int, por lo tanto el compilador resuelve esta llamada a la definición del método aMethod(short val).
* En la segunda llamada, la sentencia es aMethod(9). El valor constante 9 es (por default) int. Por lo tanto el compilador resuelve la llamada a la definición del método aMethod(int val).
* La tercera llamada al método es aMethod(i), donde la variable “i” es de tipo Integer. Ningúna definición del método sobrecargado recibe un Integer como argumento. El tipo de dato más cercano es aMethod(Object val) y es llamado. ¿Por qué el compilador no llama al método que recibe un int como argumento? Para encontrar el método que más se ajuste al tipo de dato del argumento, el compilador permite upcast implícito, no downcast, por lo tanto aMethod(int val) no es considerado.
* La última llamada es aMethod(“9”). El argumento es de tipo String por lo que se ajusta exactamente a la definición aMethod(String val).

Este proceso, donde el compilador intenta resolver el las llamadas a métodos sobrecargados, se denomina overload resolution (resolución de sobrecarga). Para resolver la llamada al método primero busca el ajuste exacto entre la cantidad de argumentos y el tipo de dato de los mismos. Si no encuentra un ajuste exacto, busca el ajuste más cercano usando upcast. Si no encuentra ningún método que se ajuste a la llamada obtendrás un error de compilación. También se obtendrá un error de compilación si el compilador encuentra más de un método que se ajuste a la llamada.

Aquí veremos algunas interesantes reglas que rigen la sobrecarga de métodos y que es indispensable manejarlas para el examen.

* El proceso de Overload Resolution toma lugar completamente en tiempo de compilación. En la siguiente sección conocerás el runtime polymorphism (polimorfismo en tiempo de ejecución) donde la llamada a un método se resuelve en runtime.
* No se pueden sobrecargar métodos únicamente con diferenciar el tipo de dato de retorno.
* No se pueden sobrecargar métodos únicamente diferenciando el tipo de excepciones que arrojan.
* Para que el proceso Overload Resolution tenga éxito, necesitas definir métodos en los que que el compilador pueda encontrar un ajusto exacto o por upcast, sin ser ambiguo. De lo contrario obtendrás un error de compilación

Nota.- la firma de un método está compuesta por el nombre del método, el número de argumentos y su tipo de dato. Puedes sobrecargar un método con el mismo nombre pero diferentes firmas. El tipo de retorno y el tipo de excepción lanzada no son parte de la firma del método.

Herencia

Una importante característica soportada por la programación orientada a objetos es la clasificación jerárquica conocida como HERENCIA. Modelos jerárquicos son fáciles de entender. Por ejemplo, puedes, de manera lógica, categorizar los vehículos como de dos ruedas, tres ruedas, cuatro ruedas etc. En la categoría de cuatro ruedas hay coches, vans camiones, etc. En la categoría de carros, hay sedan, deportivos, de lujo, etc. Cuando categorizas jerárquicamente, se vuelve fácil entender, modelar y escribir programas.

Veamos la siguiente jerarquía de objetos de la clase Number que, a su vez heredan de la clase Object:

La clase Number tiene varios métodos comunes que son heredados por sus clases derivadas. Las clases derivadas no tienen que implementar los métodos comunes implementados por la clase Number. En cambio, puedes suplir un tipo derivado en donde se espere un tipo de la clase base. Para una instancia, un Byte es un Number, lo que significa que tú puedes proveer un objeto Byte en donde se espere un objeto Number. Puedes escribir métodos de propósito general (o algoritmos) cuando escribes métodos para una clase base. En seguida un ejemplo donde se muestra un array de tipo Number que se inicializa con objetos de sus clases derivadas.

class TestNumber {

public static double sum(Number[] nums) {

doublé sum = 0.0;

for(Number num: nums) {

sum += num.doubleValue();

}

return sum;

}

public static void main(String… args) {

Number[] nums = new Number[4];

nums[0] = new Byte((byte)10);

nums[1] = new Integer(10);

nums[2] = new Float(10.0f);

nums[3] = new Double(10.0f);

System.out.println(“La suma de los números es: “ + sum(nums));

}

}

El código anterior imprimirá “La suma de los números es: 40.0”

En el método main() puedes declarar la variable nums de tipo Number[]. Una referencia Number puede almacenar cualquier objeto de sus clases derivadas. Estas creando objetos de tipo Byte, Integer, Float y Double, con un valor inicial de 10; el arreglo nums puede almacenar estos elementos.

El método sum recibe como argumento un objeto de la clase Number[] y retorna la suma de los elementos Number contenidos en el arreglo. El tipo Double puede almacenar el tipo de dato con el más alto rango de los valores, por lo tanto tendrás que utilizar este tipo de dato para retornar la suma de los números.

Polimorfismo en Runtime (Runtime Polymorphism)

Has aprendido como la referencia a una clase base puede utilizarse para referirse a todas sus clases derivadas. Puedes invocar métodos de la clase base; sin embargo, la invocación actual del método depende del tipo dinámico del objeto apuntado por la referencia a la clase base. El tipo de la referencia a la clase base es conocido como “tipo estático” del objeto y objeto que actualmente está siendo apuntado por la referencia en runtime es conocido como “tipo dinámico” del objeto.

Cuando el compilador encuentra una invocación a un método de una clase base, y el método es sobre-escribible (no-estático y no final), el compilador no puede determinar el método exacto que se va a utilizar en el runtime (enlazado tardío). En runtime, basado en el tipo dinámico del objeto, el método apropiado es invocado. Este mecanismo es conocido como “dynamic method resolution” o “dynamic method invocation”.

Como ejemplo, considera que tienes un método area() en la clase base Shape. Depende de la clase derivada (Circle o Square por ejemplo) el método area() será implementado diferente como se muestra en el siguiente código.

// Shape.java

class Shape {

public double area() {return 0;}

}

// Circle.java

public class Circle extends Shape{

private int radius;

public Circle(int r) {radius = r;}

public double area() {return Math.PI \* radius \* radius;

}

// Square.java

public class Square extends Shape {

private int side;

public Square(int a) {side = a;}

public double area() {return side \* side;]

}

Ahora testeamos nuestro código.

class TestShape {

public static void main(String[] args) {

Shape shape1 = new Circle(10);

System.out.println(shape1.area());

Shape shape2 = new Square(10);

System.out.println(shape2.area());

}

}

// este programa imprimirá

// 314,15926535…

// 100.0

Este programa ilustra como el método area() es llamado en base al tipo dinámico de la clase Shape.

Ahora hare preguntas más fundamentales: ¿por qué necesitas sobre-escribir métodos? En la programación orientada a objetos, la idea fundamental en la herencia es proveer funcionalidad común o por default en una clase base; se espera que las clases derivadas provean una funcionalidad más específica. En la clase base Shape y en las clases derivadas Circle y Square, la clase Shape provee una implementación por default del método area(). Las clases derivadas Circle y Square definen su propia versión del método area() que sobre-escribe el método area() de la clase base. Por lo tanto, depende del tipo de la clase derivada del objeto creado, cuál método area() será llamado.

Para sobre-escribir un método hay que tener en cuenta los siguientes puntos:

* Debe tener la misma lista de argumentos y sus tipos (o tipos compatibles) como en la clase base
* Debe tener el mismo tipo de retorno
  + A partir de java 5 se puede modificar el tipo de retorno por una de sus clases derivadas y es conocido como subclass-covariant return type
* No puede tener un modificador de acceso más restrictivo que la versión de la clase base
  + Puede tener un modificador de acceso menos restrictivo
* No debe lanzar nuevas o más amplias excepciones checked
  + Puede lanzar excepciones checked derivadas o cualquier excepción unchecked
* Debe tener el mismo nombre de método
* No se pueden sobre-escribir métodos privados de la clase base, ya que estos no son heredados
* No se pueden sobre-escribir métodos marcados como final

Covariant Return Types

Ahora sabes que el tipo de retorno de los métodos debe ajustarse exactamente. Sin embargo, con la característica covariant return type introducida en java 5, puedes proveer una clase derivada del tipo de retorno en el método sobre-escrito. Bueno, esto está bien, pero ¿por qué necesitas esta característica? Analiza estos métodos sobre-escritos con el mismo tipo de retorno:

abstract class Shape {

public abstract Shape copy();

}

class Circle extends Shape {

private int x;

private int y;

private int radius;

public Circle(int x, int y, int radius) {

this.x = x;

this.y = y;

this.radius = radius;

}

public Shape copy() {

return new Circle(x, y, radius);

}

}

class Test {

public static void main(String[] args {

Circle c1 = new Circle(10,20,30);

Circle c2 = c1.copy();

}

}

El código anterior tendrá un error de compilación, ya que no se puede asignar a una variable de tipo Circle un objeto de tipo Shape ya que es un downcast y para eso se requeriría un casteo explícito en la asignación de c2. Para evitar el casteo, podemos usar la característica covariant return type com sigue:

public Circle copy() {return new Circle(x, y, radius);} // Circle en lugar de Shape

Analiza un ejemplo más, para que entiendas con mayor profundidad, cómo funciona la sobre-escritura de métodos.

En el siguiente código sobre-escribiremos el método equals() de la clase Point. Antes de esto, la firma del método equals es la siguiente:

public boolean equals(Object obj)

El método equals() definido en la clase Object es un método sobre-escribible que toma un objeto de tipo Object. El método checa si el contenido del objeto actual y el objeto pasado como argumento son iguales. Si lo son, el método retorna true, de lo contrario retorna false.

Ahora analiza la clase Point que sobre-escribe el método equals() correctamente en el siguiente código:

public class Point {

private int xPos, yPos;

Public Point(int x, int y) {

xPos = x;

yPos = y;

}

// En este caso usamos Point como argumento en lugar de Object, por esta razón

// el compilador no entenderá que se trata de sobre-escritura

public boolean equals(Point other) {

if(other = null) return false;

if((xPos == other.xPos) && (yPos == other.yPos)) return true;

else return false;

}

}

En el código anterior no se está sobre-escribiendo el método equals() de la clase Object, en cambio se está definiendo un método nuevo con el mismo nombre equals(), el cual oculta solamente el método equals de la clase Object, por esta razón, cuando la referencia al tipo estático del objeto es de la clase Object, la versión utilizada es la definida en esta misma clase (Object) ya que el compilador no considera el método equals() en la clase Point como sobre-escrito. Hay que recordar que cuando se sobre-escribe un método, los tipos de datos de los parámetros deben coincidir exactamente, y en este caso, se está violando esta premisa al incluir en los parámetros, un objeto de tipo Point y no de tipo Object.

Este tipo de errores pueden ocasionar bugs difíciles de detectar, ya que, hasta cierto punto, el programa parece estar bien escrito, incluso hasta puede ejecutarse con “normalidad”. Sin embargo, en algunas situaciones específicas puede tener un comportamiento erróneo. Para evitar esto, se le puede avisar al compilador que se está tratando de sobre-escribir un método de la clase base con la anotación @Override. Con esta anotación, el compilador verifica si el método está correctamente sobre-escrito y, de no ser así, falla la compilación, obligando al programador a corregir el error.

A continuación está la correcta manera de implementar el método equals() en la clase Point

@Override

public boolean equals(Object other) {

if(other == null) return false;

if(other instanceof Point) {

Point otherPoint = (Point) other;

If((xPos == otherPoint.xPos) && (yPos == otherPoint.xPos)) return true;

}

return false;

}

Invocar métodos de la superclase

Es muy útil usar los métodos definidos en la clase base dentro del método sobre-escrito. Para hacer esto puedes usar la palabra clave “super”. Esta llamada a través de la palabra súper debe ser la primera sentencia en un constructor, para poder llamar al constructor de la clase base. También puedes usar “super” para referirte a los miembros de la clase base. En estos casos no es necesario que sea la primera sentencia en el cuerpo del método. Vamos a ver un ejemplo. Implementas una clase Point que es un Point-2D: tiene una posición x, así como una posición y. También puedes implementar una clase Point-3D con las posiciones x, y, z. Para lograr esto, no es necesario implementar esta clase desde cero: puedes extender el Point-2D y añadir la posición z en la clase Point-3D. Primero renombra la implementación simple de la clase Point a Point2D, después crea una clase Point3D que extienda a la clase Point2D

// Point2D.java

public class Point2D {

private int xPos, yPos;

Public Point(int x, int y) {

xPos = x;

yPos = y;

}

@Override

public boolean equals(Object other) {

if(other == null) return false;

if(other instanceof Point) {

Point otherPoint = (Point) other;

If((xPos == otherPoint.xPos) && (yPos == otherPoint.xPos)) return true;

}

return false;

}

@Override

public String toString() {

return “Point: xPos = “ + xPos + “, yPos = “ + yPos;

}

}

// Point3D.java

public class Point3D extends Point2D {

private zPos;

public Point3D(int x, int y, int z) {

super(x, y);

zPos = z;

}

public String toString(){

return super.toString() + “, zPos = “ + zPos;

}

public static void main(String[] args){

System.out.println(new Point3D(100, 200, 300);

}

}

Conversiones entre tipos

Java es un lenguaje de programación fuertemente tipado: el compilador realiza un estricto chequeo de los tipos para estar seguro que estás haciendo conversiones válidas. Si ejecutas casteos evidentemente inválidos obtendrás un error de compilación. Si el compilador no cacha algún casteo inválido, si hubiera, resultará en un problema en el runtime o en una excepción. Por lo tanto debes de ser muy cuidadoso a la hora de realizar conversiones entre tipos.

Upcast y Downcast

Puedes asignar objetos de una clase derivada a una referencia de su clase base sin realizar ningún casteo explícito: esto es una conversión Upcasting, si necesitas regresar el objeto a la clase derivada, necesitas realizar un casteo explícito: esto es downcasting. Vamos a examinar estos dos tipos de casteo en detalle usando ejemplos simples.

En java, toda clase deriva de la clase Object. Por lo tanto puedes poner cualquier tipo de objeto en una referencia a la clase base Object y nunca fallará.

String str1 = “Hello world”;

Object obj = str1; // no necesita casteo explícito: upcasting

Pero si conviertes de una referencia a la clase Object a cualquier tipo derivado (en este caso String) puede fallar. ¿Por qué? Porque, en general, una referencia de tipo Object puede almacenar un objeto de cualquier tipo, por lo tanto puede no ser el tipo al que estás convirtiendo con downcasting.

String str2 = obj; // se obtiene un error de compilación de tipos incompatibles

// la forma correcta de implementar este casteo sería haciéndolo explícito:

String str2 = (String) obj;

Cuando realizas downcast con casteos explícitos, es tu responsabilidad asegurar que el downcast es válido. De otra manera se obtendrá una runtime exception. Observa el siguiente programa:

class Downcast {

public static void main(String[] args) {

Integer i = new Integer(10);

// upcast, está bien… siempre funciona

Object obj = i;

// downcast – qué pasará si falla

String str = (String) obj;

// este programa fallará con una runtime exception

// específicamente ClassCastException

}

}

Casteo entre tipos incompatibles

Las clases String y StringBuffer (y todas las clases existentes) heredan de la clase Object. Pero no puedes castear directamente de String a StringBuffer y viceversa. Por ejemplo, alguien escribe el siguiente código por error:

Object obj = new StringBuffer(“Hello”);

String str2 = (String) obj; // obliga al compilador a castear a String

// la compilación es exitosa pero falla en runtime

// por ClassCastException

En este caso, primero pusiste un objeto de la clase StringBuffer en una referencia de tipo Object, después intentaste castear hacia String. ¿Qué pasa si intento convertir directamente de StringBuffer a String? ¿Obtendré un error de compilación, una advertencia o una excepción en runtime?

String str = (String) new StringBuffer(“Hello”);

// se obtiene un error de compilación porque no es posible castear de StringBuffer a String

StringBuffer str = (String) new StringBuffer(“Hello”);

// se obtiene el mismo error de compilación

Derivado de estos dos ejemplos, nos surge una interesante pregunta ¿Cómo saber si un casteo inválido generará un error de compilación o una excepción runtime?

Si el compilador puede usar únicamente la información del tipo estático del objeto origen y el destino, puede inferir que es un casteo inválido, por lo que se obtiene un error de compilación. Si el éxito/falla del casteo depende del tipo dinámico del objeto, el compilador no puede predecir el resultado del casteo, por lo que resulta en una excepción Runtime.

Para asegurarnos de que se realiza un casteo válido puedes recurrir al operador “instanceof” que verifica si el objeto a castear es del tipo requerido.

Paquetes en java

Cuando el tamaño de tu aplicación crece, necesitas un mecanismo efectivo para manejar tus archivos. Java maneja el concepto de package, que tiene como objetivo organizar tus clases y proveer un buen manejo del espacio de nombres. Todas las clases relacionadas pueden ser puestas juntas en una única entidad: un package (paquete). Un paquete no solo reduce la complejidad de una gran aplicación, también provee protección de acceso.

En esencia, estas son las ventajas de usar paquetes:

* Reducen la complejidad, facilitando la categorización de clases similares
* Proveen manejo del espacio de nombres. Por ejemplo, dos desarrolladores pueden definir el mismo nombre de tipo sin problema cuando están en diferentes paquetes
* Ofrece protección de acceso

Se puede usar la palabra clave “import” para importar clases en otros paquetes. Se puede usar la palabra clave “import static” para importar los miembros estáticos de una clase o de un paquete.

Convenciones para el uso de nombres para paquetes en java

* Una estructura jerárquica
* Procurar escribir el nombre del paquete en minúsculas
* Si se tiene un dominio en internet usarlo en reversa: company.com sería com.company
* Si el nombre tiene varias palabras, se pueden usar guiones bajos

Capítulo 4

Diseño avanzado de clases

Tópicos del examen

* Identificar cómo y cuándo utilizar clases abstractas
* Construir clases abstractas y subclases
* Usar las palabras clave static y final
* Crear clases en primer nivel y clases anidadas
* Uso de tipos enumerated

Clases abstractas

En muchas situaciones de programación quieres especificar una abstracción sin especificar los detalles de su implementación. En algunos casos, puedes usar tanto clases abstractas, como interfaces. Las clases abstractas son usadas en casos donde quieres definir una abstracción con una funcionalidad común. Tú no puedes crear instancias de clases abstractas. Aquí algunas razones para utilizarlas:

* Las clases abstractas definen una abstracción y dejan que las subclases definan la implementación concreta. Un tipo abstracto puede tener varias versiones y diferentes implementaciones; las clases abstractas proveen la plataforma correcta para que las subclases sean logradas con éxito. Si un tipo abstracto y sus subclases (la implementación concreta) están bien definidas, entonces los usuarios de estas clases pueden emplear el Runtime Polimorphism. Esto es deseable ya que desacopla la dependencia de los objetos
* Las clases abstractas tienen la habilidad de ofrecer una implementación por default; las subclases de esta pueden especializar esta implementación por default, si fuese necesario
* Las clases abstractas son el perfecto lugar para programar funcionalidad común que se requiere en todas sus subclases

Vamos a recordar la aplicación FunPaint que has venido desarrollando. La clase Shape provee una abstracción de las diferentes formas que se pueden dibujar en la aplicación. Puedes crear objetos de Shape como Square (cuadrado) y Circle (círculo), pero no tiene sentido crear un objeto de la clase Shape, ya que no existe ninguna forma “Shape”. Esta es una abstracción de propósito general y puede ser reemplazada por las formas Circle y Square. La clase Shape puede escribirse como una clase abstracta como lo vimos en los ejemplos del capítulo anterior. Se tiene que anteponer la palabra clave “abstract” antes de la palabra clave “class”.

Abstract class Shape {

public double area() {return 0;} // implementación por default

}

// el siguiente trozo de código genera un error de compilación

Shape shape = new Shape();

Este código genera un error de compilación ya que Shape es abstracta; no puede ser instaciada. En la definición de la clase Shape hay un método llamado area() el cual retorna el área de una figura en particular. Este método es aplicable para todas las figuras posibles y esta es la razón por la que se define en la clase base Shape. Sin embargo, ¿Qué esperamos que pase con la implementación del método area() en la clase Shape? No puedes proveer una implementación por default, la implementación que retorna 0 es una mala solución, aunque el compilador la acepta. La mejor solución es declarar el método area() abstracto:

public abstract doublé area(); // observa que no hay implementación

Similar a la declaración de una clase abstracta, puedes declarar métodos abstractos, anteponiendo la palabra clave “abstract” como se ve en el ejemplo anterior. La diferencia entre un método normal y uno abstracto es que el segundo no tiene un cuerpo de método; si se intenta poner un cuerpo a un método abstracto, genera un error de compilación.

public abstract doublé area() {return 0;} // error de compilación

En el código anterior obtendrás un error de compilación, entonces ¿Cuál es lo significativo de un método abstracto? Una clase abstracta promete cierta funcionalidad a través de los métodos abstractos para todos sus clientes (los usuarios de esta abstracción). Un método abstracto obliga a todas las subclases a proveer una implementación del mismo. Si una clase derivada no implementa todos los métodos abstractos definidos en la clase base, la subclase también deberá ser declarada como abstracta:

// implementación incorrecta de la clase abstracta Shape: error de compilación

class Rectangle extends Shape { } // no implementa el método abstracto area()

// implementación correcta

class Rectangle extends Shape {

private int length, height;

public doublé area() {return length \* height;} // implementa el método abstracto

}

Ahora discutiremos otro escenario donde se necesita que una funcionalidad común sea soportada por todos los objetos de tipo Shape. Por ejemplo, cada objeto puede ser dibujado dentro de otro objeto Shape, lo que significa que cada objeto Shape puede contener como objeto padre a otros. Como vimos antes, una clase abstracta es la mejor opción para poner funcionalidad común.

Ejemplo:

public abstract class Shape {

abstract double area();

private Shape parentShape;

public void setParentShape(Shape shape) {

parentShape = shape;

}

public Shape getParentShape() {

return parentShape;

}

}

Esta clase abstracta define un atributo y dos métodos que estarán disponibles para todas sus subclases.

Puntos a recordar

* La palabra clave “abstract” puede ser utilizada en la declaración de una clase o método pero no en un campo
* Una clase abstracta no puede ser instanciada. Puedes, en cambio, crear variables referenciando a una clase abstracta. De hecho, puedes crear objetos de las clases derivadas de la clase abstracta y hacer referencias a la clase abstracta para crear objetos de sus clases derivadas
* Una clase abstracta puede extender a otra clase abstracta o puede implementar una interface
* Una clase abstracta puede derivar de una clase concreta. Aunque el lenguaje lo permite, no es una buena idea hacerlo.
* Una clase abstracta no necesita declarar métodos abstractos, pero si declaras un método abstracto en una clase, esta debe forzosamente ser declarada como abstracta.
* Una subclase de una clase abstracta debe implementar todos sus métodos abstractos, de otra manera, la subclase también deberá declararse como abstracta
* Una clase abstracta puede tener métodos y campos declarados como static

Uso de la palabra clave “final”

Clases “final”

Una clase “final” no puede ser heredada, lo que significa que si declaras una clase como final, no puedes crear subclases de la misma. En general, la programación orientada a objetos sugiere que una clase debe estar abierta para extensión pero cerrada para modificación (principio open/close). Sin embargo, en algunos casos no querrás permitir que una clase pueda ser heredada. Aquí hay dos razones importantes para analizar:

* Para prevenir que se modifique el comportamiento por las subclases. En algunos casos puedes pensar que la implementación de la clase está completa y no debe cambiar. Si la sobre-escritura está permitida, el comportamiento de los métodos puede ser modificado. Tú sabes que un objeto derivado puede usarse como su clase base si es requerido, y en algunos casos preferirás no permitirlo. Para hacer una clase final, los usuarios asumen que el comportamiento de la misma no debe cambiar
* Mejora el rendimiento. Todas las llamadas a los métodos de una clase final pueden resolverse en tiempo de compilación. Como aquí no hay posibilidad de sobre-escritura de métodos, no es necesario resolver la llamada en el runtime para las clases final, lo cual se traduce en una mejora del rendimiento. TODO: (traducir la última preposición del párrafo)

En la librería de java tiene muchas clases declaradas como final; por ejemplo

* java.lang.String
* java.lang.System

Estas clases son ampliamente usadas en muchos programas de java. Por ejemplo, si tusas el método System.out.println(), estás usando la clase System y la clase String. Si estas dos clases no están declaradas como final, es posible que alguien cambie el comportamiento de las mismas mediante herencia y sobre-escritura, entonces el programa podría tener un comportamiento diferente. Para evitar estos problemas, clases como estas o las clases wrapper como Integer están declaradas como final.

La mejora en el rendimiento haciendo una clase final es modesta; la clave para usar una clase final es apropiada.

En la aplicación de ejemplo FunPaint, puedes tener un lienzo para arrastrar y soltar figuras para crear dibujos. Asume que tienes una clase Canvas (lienzo) para implementar esta funcionalidad. Tienes que asegurarte que el comportamiento de la clase Canvas no sea modificado, por lo que tendrás que declararla como final.

final class Canvas { }

Métodos y variables “final”

En una clase, puedes declarar un método como fina. Un método final no puede ser sobre-escrito. Por lo tanto, si has declarado un método como final en una clase no-final, puedes extender a esta clase pero no puedes sobre-escribir este método.

En la aplicación FunPaint, en la clase Shape, por ejemplo, un método es final (setParetShape()) y otro método no es final (getParentShape()) por lo tanto puedes sobre-escribir solo el segundo.

Las variables final, son como los CDs, ya que escribiste en ellos no puedes cambiarlos.

Puntos a recordar

* El modificador final puede aplicarse a una clase, método o variable. Todos los métodos de una clase final son implícitamente final
* Una variable final puede ser asignada una sola vez. Si una declaración de variable como final no está inicializada, esta está referenciada como blank-final. Necesitas inicializar una variable blank-final en todos los constructores de la clase; de otra manera el compilador fallará
* La palabra clave final puede también ser aplicada a parámetros en métodos. El valor de un parámetro final no puede ser cambiado ya que ha sido asignado. Aquí, es importante notar que el valor es implícito para valores primitivos. Sin embargo, el valor de una referencia a un objeto, no es el estado del mismo. Por lo tanto, puedes cambiar el estado interno del objeto pasado como parámetro pero no puedes cambiar la referencia al mismo

Usar la palabra clave “Static”

Supón que quieres escribir una clase que cuente el número de objetos de esa clase, creados en un programa, como sigue:

public class Counter {

private int count;

public Counter() {

cout++;

}

public void printCount() {

System.out.println(“El número de instancias creadas es: “ + count”);

}

public static void main(String[] args) {

Counter anInstance = new Counter();

anInstance.printCount();

Counter anotherInstance = new Counter();

anotherInstance.printCount();

}

}

// Imprimirá:

// El número de instancias creadas es 1

// El número de instancias creadas es 1

Como te podrás dar cuenta, la clase no da seguimiento al número de objetos creados ¿Qué pasa?

Estás usando una variable de instancia “count” para llevar la cuenta del número de objetos creados a partir de esta clase. Cada instancia de la clase tiene su propio valor “count”, por lo que siempre imprimirá 1. Lo que necesitas es una variable que sea compartida con todas las instancias. Esto puede lograse declarando la variable como “static” (estática). Una variable estática está asociada con la clase, en lugar de con la instancia; por esta razón se conoce como variable de clase. Una variable estática es inicializada solo una vez cuando la ejecución del programa empieza. Una variable estática comparte su estado con todas las instancias de la clase. Puedes accesar a una variable estática usando el propio nombre clase (en lugar del nombre de una instancia específica). El siguiente código muestra la correcta implementación de la clase “Counter” para que imprima el número de instancias creadas en el programa:

Counter.java

public class Counter {

// cambia a variable estática

private static int count;

public Counter() {

cout++;

}

public void printCount() {

System.out.println(“El número de instancias creadas es: “ + count”);

}

public static void main(String[] args) {

Counter.printCount();

Counter anInstance = new Counter();

anInstance.printCount();

Counter anotherInstance = new Counter();

anotherInstance.printCount();

Counter.printCount();

}

}

// Imprimirá:

// El número de instancias creadas es 0

// El número de instancias creadas es 1

// El número de instancias creadas es 2

// El número de instancias creadas es 2

Aquí, la variable estática “count” es inicializada cuando comienza la ejecución del programa. En el momento que se crea un objeto, la variable estática “count” es incrementada a uno. Similarmente, cuando el segundo objeto es creado, el valor de “count” se incrementa a 2. Como muestra la salida del programa, ambos objetos actualizan la misma copia de la variable “count”.

Nota como cambiamos la llamada al método printCount() para usar el nombre de la clase “Counter”. El compilador aceptará las llamadas del programa anterior (como nombre de instancia) y no habrá diferencia semántica. Se puede llamar a un método estático usando el nombre de la clase o el nombre de una instancia específica, sin embargo lo segundo no es recomendado.

Un método estático solo tiene acceso a variables estáticas y solo puede llamar a otros métodos estáticos; en contraste con los métodos de instancia (no-estáticos) que pueden llamar métodos estáticos y, a su vez, tienen acceso a variables estáticas.

Bloque estático

Aparte de las variables y métodos estáticos, también puedes definir un bloque estático un tu definición de la clase. Este bloque estático será ejecutado por la Máquina Virtual de Java cuando se cargue la clase en la memoria. Por ejemplo, en el ejemplo anterior, puedes definir un bloque estático para inicializar la variable “count” para que su valor inicial sea 1, en lugar de 0, como veremos a continuación:

Counter.java

public class Counter {

// cambia a variable estática

private static int count;

static {

count = 1;

}

public Counter() {

cout++;

}

public void printCount() {

System.out.println(“El número de instancias creadas es: “ + count”);

}

public static void main(String[] args) {

Counter.printCount();

Counter anInstance = new Counter();

anInstance.printCount();

Counter anotherInstance = new Counter();

anotherInstance.printCount();

Counter.printCount();

}

}

// Imprimirá:

// El número de instancias creadas es 1

// El número de instancias creadas es 2

// El número de instancias creadas es 3

// El número de instancias creadas es 3

No confundas un bloque estático con un constructor. Un constructor es invocado cuando una instancia de la clase es creada, mientras el bloque estático solo se invoca cuando el programa empieza.

Puntos a recordar

* El método main(), donde comienza la ejecución principal del programa, es siempre declarado como “static” ¿Por qué? Si fuera un método de instancia, sería imposible invocarlo ya que primero tendrías que crear una instancia de la clase y en el momento que el programa empieza no existe ninguna.
* No puedes sobre-escribir un método estático que se encuentre en una clase base ¿Por qué? Basado en el tipo de instancia, la llamada al método es resuelta con polimorfismo en runtime, mientras los métodos estáticos son asociados a la clase (más no a la instancia), no puedes sobre-escribir métodos estáticos y el polimorfismo en runtime no es posible con métodos estáticos.
* Un método estático no puede usar la palabra clave “this” en su cuerpo ¿Por qué? Recuerda que los métodos estáticos están asociados a la clase y no a la instancia; “this” hace referencia a la instancia donde es usada.
* Los métodos estáticos no tienen acceso a las variables de instancia ya que estos están asociados a la clase y no a una instancia específica. Estos métodos casi siempre son desarrollados con fines utilitarios.
* Los métodos estáticos no pueden usar la palabra clave “super” ya que esta última hace referencia a la superclase de una instancia de la clase.
* La llamada a un método estático es considerada ligeramente más eficiente que la llamada a un método de instancia. Esto es porque el compilador no necesita pasar implícitamente el objeto “this” cuando se llama a un método estático.

Sabores de las clases anidadas “nested classes”

Las clases definidas dentro del cuerpo de otras clases (o interfaces) son conocidas como clases anidadas. Normalmente defines una clase, cuando es una clase en el nivel más alto, seguida de la declaración del paquete. En contraste, las clases anidadas son clases contenidas dentro de otra clase o interface.

¿Cuál es el beneficio de crear clases dentro de otra clase o interface? Hay muchos beneficios. Primero, puedes poner juntas a las clases relacionadas entre sí, como un único grupo lógico. Segundo, las clases anidadas tienen acceso a todos los miembros de la clase contenedora, lo cual es muy útil en ciertos casos. Tercero, las clases anidadas son, en ocasiones, muy útiles para propósitos específicos. Por ejemplo, una clase anónima interna es útil para simplificar el manejo de eventos con AWT/Swing (AWT/Swing no vienen en el examen). Por ahora, considera que las clases anidadas son útiles en algunas situaciones, por lo que es importante que aprendas acerca de ellas.

Hay cuatro tipos de clases anidadas:

1. Static nested (anidada estática)
2. Inner (interna)
3. Local inner (local interna)
4. Anonymous inner (anónima interna)

La distinción de estos cuatro tipos, no es evidente a primera vista y no es una ayuda de importancia los términos que se han adoptado para ellas. Para clarificar la confusión, te presentamos los cuatro tipos de clases anidadas, de manera esquemática, en la figura siguiente:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Estática | No-estática | Anónima |
| No-local | Anidada-estática | Interna | n/a |
| Local | n/a | Local-interna | Anónima-interna |

Estudia este esquema. Una clase local es definida dentro de un bloque de código (puede ser un método, un constructor o un bloque de inicialización), mientras una clase no local es definida dentro de la clase. Una clase estática es cualificada usando la palabra clave “static”, mientras una clase no-estática no usa esta palabra clave. En una clase anidada anónima no provees el nombre de la clase; solo defines su cuerpo.

Como puedes ver en el esquema anterior, las clases anidadas estáticas son estáticas y no-local, mientras las clases internas son no-estáticas y no-locales. Una clase anidada no-estática y local es una clase local interna, y una clase anidada local y anónima es una clase anónima interna. Regresa a este esquema para recordar los cuatro tipos de clases anidadas.

Ahora vamos a analizar estos cuatro tipos con más detalle.

Clases (o interfaces) anidadas estáticas

Puedes definir una clase o interface como un miembro estático dentro de otra clase o interface. Puesto que el tipo externo puede ser una clase o interface y el tipo interno puede ser, también, una clase o interface, hay cuatro combinaciones que se muestran en el siguiente código.

class Outer { // una clase externa con una clase anidada estática

static class Inner { }

}

interface Outer { // una interface externa con una clase anidada estática

static class Inner { }

}

class Outer { // una clase externa con una interface anidada estática

static interface Inner { }

}

interface Outer { // una interface externa con una interface anidada estática

static interface Inner { }

}

No tienes que usar explícitamente la palabra clave “static” con una interface anidada, ya que esta es implícitamente estática. Ahora revisa el siguiente ejemplo que utiliza clases anidadas estáticas.

Recuerda la aplicación “FunPaint” del capítulo 3 que implementa la clase “Color” cuyos campos son m\_red, m\_green y m\_blue (int). Ya que todas las figuras pueden ser coloreadas, puedes definir la clase “Color” dentro de la clase “Shape” como se muestra en el siguiente código.

abstract class Shape {

public static class Color {

int m\_red, m\_green, m\_blue;

public Color() {

this(0,0,0);

}

// sigue en la otra página

public Color(int red, int green, int blue) {

m\_red = red;

m\_green = green;

m\_blue = blue;

}

public String toString () {

return “ red = ” + m\_red + “, green = ” + m\_green + “, blue = ” + m\_blue;

}

}

}

// Clase para testear

public class TestColor {

public static void main(String[] args) {

// ya que la clase Color es estática y anidada

// tenemos acceso a ella usando el nombre de la clase contenedora

// NO tienes (ni puedes) que instanciar la clase Shape para utilizar

// la clase Color

Shape.Color white = new Shape.Color(255,255,255);

System.out.println(“White color has values: ” + white);

}

}

Como podrás observar, la clase “Shape” es abstracta. Puedes ver que la clase “Color” está definida como pública y estática dentro de la clase “Shape”. La clase “TestColor” usa la sintaxis “Shape.Color” para referirse a esta clase. Sin otra menor diferencia, la clase “Color” no se ve diferente que la misma clase definida afuera de la clase “Shape”, pero vamos a sintetizarlo para que quede claro: una clase anidada estática es tan buena como una clase definida afuera, con una diferencia.- está físicamente definida adentro de otra clase.

Puntos a recordar

* La accesibilidad (public, protected, etc.) de una clase anidada estática está definida por su clase contenedora
* El nombre de la clase anidada estática es expresada con la sintaxis OuterClass.NestedClass
* Cuando defines una clase anidada (o interface) dentro de una interface, la clase anidada es declarada implícitamente pública y estática. Este punto es fácil de recordar: cualquier campo dentro de una interface es implícitamente público y estático, y las clases anidadas tienen este mismo comportamiento
* Las clases anidadas estáticas pueden ser declaradas como abstractas y final
* Las clases anidadas estáticas pueden extender a otra clase o pueden ser usadas como clase base
* Las clases anidadas estáticas pueden contener miembros estáticos (como podrás ver más adelante, esto no aplica a otro tipo de clases anidadas)
* Las clases anidadas estáticas tienen acceso a los miembros estáticos de la clase contenedora
* La clase contenedora también tiene acceso a los miembros (incluso los privados) de su clase anidada, a través de un objeto de la clase anidada. Si no declaras una instancia de la clase anidada, la clase contenedora no tiene acceso a los miembros de la clase anidada directamente

Clases internas (inner clases)

Puedes definir una clase (o interface) como un miembro no-estático dentro de otra clase. ¿Cómo se puede declarar una clase o una interface dentro de una interface? Como observaste en el tercer viñeta de la lista anterior, cuando defines una clase o interface dentro de una interface, esta se vuelve implícitamente estática, por lo cual es imposible declarar una clase interna no-estática dentro de una interface, lo cual deja solo dos posibilidades:

// una clase que contiene una clase interna

class Outer {

class Inner {

// cuerpo de la clase

}

}

// una clase que contiene una interface interna

class Outer {

interface Inner {

// cuerpo de la interface

}

}

Vamos a crear una clase “Point” para implementar el centro del Círculo en la clase “Circle”. Ya que queremos asociar cada círculo con un punto centro, es una buena idea hacer la clase “Point” una clase interna dentro de la clase “Circle”, como se muestra a continuación.

public class Circle {

// definimos la clase Point como interna

class Point {

private int xPos;

private int yPos;

// puedes proveer un constructor para la clase interna como este

public Point(int x, int y) {

xPos = x;

yPos = y;

}

// las clases internas son como cualquier otra clase, puedes sobre-escribir //métodos

public String toString() {

return “(” + xPos + “,” + yPos + “)”;

}

} // continua en la siguiente página

// vamos a usar la clase interna para declarar un campo

private Point center;

private int radius;

public Circle(int x, int y, int r) {

// NOTA la forma de instanciar la clase interna para usarla

center = this.new Point(x, y);

radius = r;

}

public String toString() {

return “punto medio = ” + center + “ y radio = ” + radius;

}

// método main

public static void main(String[] s) {

System.out.println(new Circle(10,10,20));

}

}

Esta implementación de las clases “Circle” y “Point” es muy similar a lo que viste antes, con una mayor diferencia que es la definición del campo de tipo “Point” como privado para la clase “Circle”. Estarás sorprendido del por qué no usar la palabra clave “new” como normalmente se usa: Necesitas anteponer la referencia al objeto de la clase contenedora para crear una instancia de la clase interna. En este caso esta referencia se realiza con la palabra clave “this”.

NOTA: toda clase interna está asociada con la instancia de su clase contenedora. En otras palabras, una clase interna siempre se asocia a su objeto contenedor.

Las clases externas e internas comparten una relación especial, como amigos o miembros de una familia. El acceso a miembros es válido bilateralmente, sin embargo, hay una diferencia: tienes acceso a los miembros de la clase externa en la clase interna sin crear una instancia; pero este no es el caso con la clase externa.- primero tienes que crear una instancia de la clase interna para tener acceso a sus miembros.

Una limitación de las clases internas es que no puedes declarar miembros estáticos como este:

class Outer {

class Inner {

// no es posible y genera un error de compilación

static int i = 10;

}

}

Puntos a recordar

* La accesibilidad (public, protected, etc) de las clases internas está definida por la clase contenedora
* Como las clases en el primer nivel, una clase interna puede extender a una clase o implementar interfaces. Similarmente, una clase interna puede ser extendida por otra clase y una interface interna puede ser implementada o extendida por otras clases o interfaces
* Una clase interna puede ser declarada como final o abstracta
* Las clases internas pueden tener, a su vez, clases internas, pero esto puede dificultar la lectura y comprensión del código, por lo que es mejor evitar esta práctica

Clases local-interna (local-inner clases)

Una clase local-interna es definida en un bloque de código (método, constructor o bloque de inicialización). A diferencia de las clases anidadas estáticas y las clases internas, las clases local-internas no son miembros de la clase contenedora; estas son únicamente locales para el método o el código donde están definidas.

Aquí veremos un ejemplo de la sintaxis general de una clase local-interna

class SomeClass {

void someMethod() {

class Local {

// cuerpo de la clase local-interna

}

}

}

Como puedes ver en este código, “Local” es una clase definida dentro de someMethod(). Esta no está disponible fuera de este método, tampoco para los miembros de la clase “SomeClass”. Ya que no puedes declarar una variable local como estática, tampoco puedes declarar una clase local como estática.

Como no puedes definir métodos dentro de interfaces, tampoco puedes tener clases locales o interfaces dentro de una interface, tampoco puedes crear interfaces locales. En otras palabras, no puedes definir interfaces dentro de métodos, constructores o bloques de inicialización.

Ahora para que entiendas la sintaxis vamos a analizar un ejemplo práctico. En la aplicación “FunPaint” implementaste la clase “Color” como una clase anidada estática. El método toString() muestra una representación textual de la clase “Color”. Imagina que necesitas mostrar mensajes de ayuda en la parte debajo de la pantalla en la aplicación. Para esto necesitas mensajes descriptivos. Mostrar mensajes en este formato cifrado no es de mucha ayuda para el lector. Ahora quieres mostrar la clase “Color” con el siguiente formato: “Seleccionaste el color con los valores RGB siguientes red = 0 green = 0 blue = 0”. Para esto tienes que definir un método llamado getDescriptiveColor() en la clase “StatusReporter. En el método getDescriptiveColor() tienes que crear una clase derivada de la clase “Shape.Color” donde el método toString() retorne este mensaje descriptivo como se muestra en el siguiente código.

class StatusReporter {

// es importante notar que el argumento “color” es declarado como final

// de otra manera, la clase local-interna no estará habilitada para usarlo

static Shape.Color getDescriptiveColor(final Shape.Color color) {

// la clase local DescriptiveColor extiende a la clase Shape.Color

class DescriptiveColor extends Shape.Color {

// sobre-escribe el método toString

return “Has seleccionado un color con los valores RGB “ + color;

}

return new DescriptiveColor();

}

// main

public static void main(String[] args) {

Shape.Color descriptiveColor = StatusReporter.getDescriptiveColor

(new Shape.Color(0,0,0));

System.out.println(descriptiveColor);

}

}

Ahora analicemos como está definida la clase local. El método getDescriptiveColor() toma el objeto plano de la clase “Shape.Color” y retorna un objeto de la clase “Shape.Color”. Dentro del método getDescriptiveColor() está definida la clase “DescriptiveColor” la cual es local para este método. La clase “DescriptiveColor” es derivada de la clase “Shape.Color”. Dentro de la clase “DescriptiveColor” el único método definido es toString(), el cual sobre-escribe el contenido de la clase base “Shape.Color”. Después de la definición de la clase “DescriptiveColor”, el método getDescriptiveColor() crea un objeto de la clase “DescriptiveColor” y lo retorna.

En el método main() observas que llama al método StatusReporter.getDescriptiveColor() y almacena el resultado en una referencia de tipo “Shape.Color”. Te habrás dado cuenta que el método getDescriptiveColor() retorna un objeto de la clase “DescriptiveColor”, el cual deriva de la clase “Shape.Color”, por lo tanto, la inicialización de la variable “descriptiveColor” es correcta. En el método println(), el tipo dinámico de la variable “descriptiveColor” es un objeto de la clase “DescriptiveColor”, por lo tanto, el método toString() llamado es el de esta última clase.

Te habrás dado cuenta también de otra característica del método getDescriptiveColor(). Su argumento está declarado como final. ¿Qué pasa si no fuera así? Se obtiene un error de compilación ya que PUEDES PASAR SOLO VARIABLES FINALES A UNA CLASE LOCAL-INTERNA.

Puntos a recordar

* Puedes crear una clase no-estática dentro de un bloque de código. Interfaces no tienen clases locales y no puedes crear interfaces locales
* Las clases locales son accesibles solo para el bloque de código donde están definidas. Son inaccesibles fuera del bloque de código
* Puedes extender a una clase o implementar interfaces cuando defines una clase local
* Una clase local tiene acceso a todas las variables dentro del bloque de código donde está definida. Puedes pasar solo variables finales a una clase local-interna

Clases anónimas-internas (Anonymous inner clases)

Como su nombre lo dice, una clase anónima-interna no tiene nombre. La declaración de la clase deriva automáticamente en una expresión de instanciación. Esto es conocido simplemente como clases anónimas.

Una clase anónima es útil en casi todas las situaciones donde utilices clases local-internas. Una clase local-interna tiene un nombre, mientras una clase anónima no y esta es la diferencia principal. Una diferencia adicional es que una clase anónima-interna no tiene constructores explícitos. Un constructor es nombrado después del nombre de la clase, pero una clase anónima no tiene nombre, tampoco se puede definir un constructor. Antes de seguir mencionaré que NO HAY interfaces anónimas.

El siguiente es un ejemplo para entender la sintaxis de una clase local.

class SomeClass {

void someMethod() {

new Object() { };

}

}

Este código se ve enigmático. ¿Qué está pasando aquí? En la sentencia “new Object() { };” estas declarando una clase derivada de la clase “Object” directamente, usando la palabra clave “new”. No se define código que retorne la instancia derivada. El objeto creado no es usado en ninguna parte, por lo tanto es ignorado. La expresión “new” invoca al constructor por default, pero también puedes invocar cualquier otro constructor con argumentos de la clase base con solo pasar los argumentos a la expresión. Cuando definimos una clase anónima, esta extiende implícitamente a la clase base (en este ejemplo es “Object”).

No te preocupes si no entiendes este ejemplo. Ahora analizaremos un ejemplo más práctico y el uso de clases anónimas será más claro.

Previamente observaste la clase “DescriptiveColor” en el método getDescriptiveColor() dentro de la clase “StatusReporter”. Puedes simplificar el código convirtiendo la clase local en una clase anónima como se muestra en el código siguiente.

class StatusReporter {

static Shape.Color getDescriptiveColor(final Shape.Color color) {

return new Shape.Color() {

public String toString(){

return “Has seleccionado el color RGB ” + color;

}

};

}

// continúa

public static void main(String[] p) {

Shape.Color descriptiveColor =

StatusReporter.getDescriptiveColor(new Shape.Color(0,0,0));

System.out.println(descriptiveColor);

}

}

Puntos a recordar

* Las clases anónimas son definidas en la expresión “new” por sí mismas, pero no puedes crear múltiples objetos de una clase anónima
* No puedes extender explícitamente a una clase o implementar una interface cuando defines una clase anónima

Tipos ENUM

Hay muchas situaciones donde quieres restringir al usuario la cantidad de entradas a un proceso de una lista determinada. Por ejemplo, es posible que quieras que el usuario elija una serie de opciones de impresoras:

public static final int DOTMATRIX = 1;

public static final int INKJET = 2;

public static final int LASER = 3;

La solución funciona. En este caso, sin embargo, tú puedes pasar cualquier otro entero (por ejemplo 10), y el compilador pasará desapercibido y lo tomará. Por lo tanto, esta solución no es de type-safe (tipo seguro).

Para evitar esta condición puedes definir tu propia clase (ejemplo “PrinterType”) y permitir únicamente valores legitimados (seguros). Sin embargo, necesitarás definir los atributos manualmente. Es aquí donde utilizamos los tipos “enum”.

En el siguiente código se define una clase “enum” (sí, enum son clases especiales) para el ejemplo anterior.

// definimos un enum para clasificar los tipos de impresoras

enum PrinterType {

DOTMATRIX, INKJET, LASER

}

// los tipos enum se definen como las clases, pero en lugar de la palabra clave “class” se

// escribe la palabra clave “enum”

Ahora escribiremos un código para probar nuestro enum

public class EnumTest {

PrinterType printerType;

public EnumTest(PrinterType pType) {

printerType = pType;

}

public void feature() {

// al switch le podemos pasar un enum como opción

switch(printerType) {

case DOTMATRIX:

System.out.println(“Dot-matrix”);

break;

case INKJET:

System.out.println(“Inkjet”);

break;

case LASER:

System.out.println(“Laser”);

break;

}

}

// main

public static void main(String[] a) {

// le pasamos al constructor un enum de tipo LASER

EnumTest enumTest = new EnumTest(PriterType.LASER);

enumTest.feature();

}

}

Vamos a analizar el ejemplo anterior con más detalle

* En una sentencia switch no necesitas proveer el nombre cualificado completo de los elementos enum. Esto es porque el switch toma una instancia del tipo enum y, por lo tanto, entiende el contexto (tipo) en donde especificas los elementos enum.
* No puedes proveer ninguna entrada que no esté especificada en la definición enum. Esto hace a enum type-safe

Nota que puedes declarar un tipo enum (PrinterType en este caso) en un archivo separado, como si se tratara de otra clase.

Ahora que ya entendiste el concepto básico de los tipos enum, echemos un vistaso a un ejemplo más detallado, en donde defines atributos miembro y métodos dentro de un tipo enum. Sí, puedes definir métodos y campos en la definición de un tipo enum, como se muestra en el código siguiente.

public enum PrinterType {

DOTMATRIX(5), INKJET(10), LASER(50);

private int pagePrintCapacity;

private PrinterType(int pagePrintCapacity) {

this.pagePrintCapacity = pagePrintCapacity;

}

public int getPrintPageCapacity() {

return pagePrintCapacity;

}

}

// EnumTest.java

public class EnumTest {

PrinterType printerType;

public EnumTest(PrinterType pType) {

printerType = pType;

}

public void feature() {

switch (printerType) {

case DOTMATRIX:

System.out.println(“dot-matrix”);

break;

case INKJET:

System.out.println(“inkjet”);

break;

case LASER:

System.out.println(“laser”);

break;

}

System.out.println(“capacidad de impresión por minuto: ” +

printerType.getPrintPageCapacity());

}

public static void main(String[] args) {

EnumTest enumTest1 = new EnumTest(PrinterType.LASER);

enumTest1.feature();

EnumTest enumTest2 = new EnumTest(PrinterType.INKJET);

enumTest2.feature();

}

}

Bueno, como puedes observar en esta nueva versión del ejemplo, puedes definir atributos, constructores y métodos para los tipos enum. El atributo pagePrintCapacity es fijado por los valores iniciales especificados con los elementos enum, dentro de los paréntesis, los cuales llaman al constructor del tipo enum, sin embargo, las clases enum no pueden tener un constructor público; SOLO PUEDEN TENER UN CONSTRUCTOR PRIVADO.

Puntos a recordar

* Los tipos enum son implícitamente públicos, estáticos y finales, por lo que no puedes extenderlos
* Cuando defines los elementos de un enum, estos heredan implícitamente de java.lang.Enum. Internamente, los tipos enum son convertidos en clases. Además, las constantes enum son instancias de la clase enum donde fueron declaradas.
* Puedes utilizar los métodos valueOf() y name() a un elemento enum para retornar el nombre del elemento
* Si declaras un enum dentro de una clase, este es implícitamente estático
* No puedes usar el operador “new” para los tipos enum, incluso dentro de la misma clase enum
* Puedes comparar dos instancias enum con el operador “==”
* Cuando a una constante enum se le invoca el método toString() imprime el nombre de la constante
* El método estático values() en la clase java.lang.Enum retorna un array de las constantes enum cuando se llama a un tipo enum
* Las constantes enum no pueden clonarse, si lo intentas generará una excepción “CloneNotSupportedException”
* Si las constantes enum son de dos diferentes enums el método equals() no retorna “true”
* Debes definir los elementos del enum antes de cualquier otro miembro de la clase
* No puedes definir un tipo enum como local-inner

Los tipos enum mejoran la lectura y comprensión del código. También, son construcciones type-safe. Por lo tanto debes usarlos cuando sea aplicable.

Resumen

Clases abstractas

* Una abstracción especifica la funcionalidad soportada sin adentrarse en un gran nivel de detalle
* No puedes crear instancias de clases abstractas
* Las clases abstractas habilitan el polimorfismo en runtime

Palabra clave final

* Una clase final no es heredable
* Un método final no se puede sobre-escribir
* Todos los métodos de una clase final son implícitamente final
* Una variable final solo puede ser asignada una sola vez

Palabra clave static

* Hay dos tipos de variables miembro: variables de clase y variables de instancia. Todas las variables que requieren una instancia de la clase para tener acceso a ellas se conocen como “variables de instancia”. Todas las variables que se comparten con todas las instancias de una clase y son asociadas con la clase misma en lugar de con un objeto son conocidas como “variables de clase”. Para declarar una “variable de clase” se necesita anteponer la palabra clave “static”
* Todos los miembros estáticos no requieren una instancia para ser llamados o tener acceso a ellos. Puedes llamarlos o tener acceso a ellos usando el nombre propio de la clase
* Los miembros estáticos solo pueden utilizar/llamar a otros miembros estáticos

Tipos de clases anidadas

* Java soporta cuatro tipos de clases anidadas:
  + static-nested: estática-anidada
  + inner: interna
  + local-inner: local-interna
  + anonymous-inner: anónima-interna o solo anónima
* Las clases de tipo estática-anidada puede tener miembros estáticos mientras los otros tipos de clases anidadas no
* Las clases de tipo estática-anidada e interna pueden tener acceso a los miembros de su clase contenedora, incluso los privados. Sin embargo, las clases estáticas solo tienen acceso a los miembros estáticos
* Las clases locales tanto local-interna como anónima, tienen acceso a todas las variables dentro del ámbito donde están declaradas, ya sea un método, un constructor o un bloque de inicialización

Enums

* Los tipos enum son el camino type-safe para restringir la entrada de datos del usuario
* No puedes usar el operador “new” con los enums, incluso dentro de la definición
* Los tipos enum son por default final
* Todas las clases enum heredan implícitamente de java.lang.Enum

Capítulo 5

Principios de diseño orientado a objetos

Tópicos del examen:

* Escribir código que declare, implemente y/o extienda interfaces
* Escoger entre herencia a través de interfaces o herencia a través de clases
* Desarrollar código que implemente las relaciones is-a y/o has-a
* Aplicar principios de composición de objetos
* Diseñar una clase usando el patrón Singleton
* Escribir código para implementar el patrón DAO
* Diseñar y crear objetos usando una Factory y usar Factorías de la API de java

Interfaces

En general, una interface se refiere a un común nodo de interconexión entre dos entidades (lo cual puede ser artículos humanos, sistemas, conceptos, máquinas, etc.). Por ejemplo, un teclado en una computadora provee una interface entre el humano y la computadora. Un lenguaje natural como el inglés es una interface entre dos o más humanos que permite intercambiar sus ideas.

En java, una interface es un conjunto de métodos abstractos, los cuales definen un protocolo. Las clases que implementan una interface deben implementar los métodos especificados en la interface. Una interface define un protocolo y la clase que implementa la interface se adapta al protocolo. En otras palabras, una interface promete cierta funcionalidad a sus clientes por medio de la definición de una abstracción. Todas las clases que implementen la interface proveen su propia implementación para la funcionalidad prometida en la interface.

Vamos a darnos cuenta de la funcionalidad de una interface con un ejemplo. Considera la interface java.lang.Comparable que especifica el siguiente protocolo:

public interface Comparable {

public int compareTo(Object o);

// intenta comparar este objeto con el objeto especificado

// el tipo de retorno es un entero

// retorna negativo cuando el objeto es menor

// retorna cero cuando los objetos son iguales

// retorna positivo cuando el objeto es mayor

}

Clases no relacionadas pueden proveer su propia implementación de este método, cuando implementan la interface. Estas clases no relacionadas tienen un aspecto en común: siguen las especificaciones proporcionadas por la interface Comparable y esta deja que las clases implementen el método compareTo de acuerdo a sus necesidades. Las estructuras de datos contenidas en la api java.util implementan esta interface. Si quieres utilizar los algoritmos provistos en esta librería necesitas implementar esta interface. Por ejemplo, considera la implementación de ejemplo del método max() en la clase java.util.Collections que sirve para encontrar el elemento máximo de una colección, este usa la interface Comparable y los elementos en la colección deben proveer la implementación para el método compareTo().

Los algoritmos (por ejemplo los clientes de la interface) ignoran por completo cómo está implementado el método compareTo(). Pero los clientes conocen el contrato (protocolo) que insinúa la disponibilidad del mismo, por lo tanto los clientes pueden usarlo. Esto resulta en una gran ventaja: cuando un método toma una interface como argumento puedes pasar cualquier objeto que implemente esta interface (polimorfismo runtime).

Conceptualmente, una clase y una interface son dos construcciones diferentes usadas para diferentes propósitos. Una clase combina el estado y el comportamiento de un objeto real, mientras que una interface especifica el comportamiento de una entidad abstracta.

Declarar y usar interfaces

Ahora es tiempo de implementar tu propia interface para los objetos shape. Algunos objetos semi/circulares como el círculo o la elipse, pueden ser rotados a un ángulo dado. Puedes crear la interface “Rollable” y declarar el método roll():

interface Rollable {

void roll(float degree);

}

Como puedes ver, defines una interface usando la palabra clave “interface”. Puedes declarar métodos en esta interface; en este caso el método roll(). El método toma un argumento: el ángulo a rotar. Ahora implementa la interface para la clase “Circle”, que es Rollable.

class Circle implements Rollable {

public void roll(float degree) {

// implementar la funcionalidad para rotar aquí

}

}

Debes usar la palabra clave “implements” para implementar una interface. Nota que el nombre del método, su argumento y el tipo de retorno en la implementación, son exactamente iguales a los de la interface, de lo contrario no se considera una implementación de esta interface.

Si estás implementando una interface en una clase abstracta, la clase abstracta no necesita definir el método.

interface Rollable {

void roll(float degree);

}

abstract class CircularShape implements Rollable extends Shape { }

En este caso, “CircularShape” implementa la interface “Rollable” y extiende a la clase abstracta Shape que se estudió en los capítulos anteriores. Ahora, las clases concretas “Circle” y “Elipse” pueden extender a esta clase abstracta y definir el método roll().

La interface “Rollable” solo tiene un método (roll()). Sin embargo, es muy común para las interfaces tener múltiples métodos. Por ejemplo, java.util define la interface Iterable como sigue:

public interface Iterator<E> {

boolean hasNext();

E next();

void remove();

}

Esta interface está diseñada para recorrer una colección y, como puedes ver, esta declara tres métodos.

De hecho, una clase puede implementar múltiples interfaces al mismo tiempo, directa o indirectamente a través de su clase base. Por ejemplo, la clase “Circle” puede también implementar la interface standard “Cloneable” (para crear copias de objetos Circle) y la interface “Serializable” (para almacenar un objeto y después recuperarlo para usarlo) como sigue:

class Circle extends CircularShape implements Cloneable, Serializable {

// definición de la clase aquí

}

Puntos a recordar

* Una interface no puede ser instanciada
* Una interface puede extender otras interfaces usando la palabra clave “extends”
* Las interfaces no pueden contener variables de instancia. Si tú declaras un miembro dentro de una interface, este debe ser inicializado y será tratado implícitamente como “public static final”
* Una interface no puede declarar métodos estáticos. Solo puede declarar métodos de instancia
* No puedes declarar miembros como “protected” o “private”. Solo el modificador de acceso “public” está permitido para los miembros de una interface
* Todos los métodos declarados en una interface son implícitamente abstractos. Si tú quieres, puedes hacerlo explícito con “abstract”
* Puedes declarar métodos en una interface, mas no definir su implementación
* Una interface puede ser declarada con un cuerpo vacío, como las interfaces conocidas como etiquetas, marcas. Estas interfaces son útiles para definir un padre común y, con esto poder utilizar el polimorfismo en runtime. Por ejemplo, java.util define la interface “EventListener” sin cuerpo
* Una interface puede ser declarada dentro de otra interface o clase; estas interfaces son conocidas como interfaces anidadas
* A diferencia de las interfaces en primer nivel que solo pueden ser declaradas como “public” o “default”, las interfaces anidadas pueden ser declaradas como “public”, “protected” o “private”

Clases abstractas contra Interfaces

Las clases abstractas y las interfaces tienen mucho en común. Por ejemplo, ambas pueden declarar métodos (sin implementar) que las clases derivadas pueden definir. Tampoco pueden ser instanciadas.

Entonces, ¿Cuál es la diferencia entre las clases abstractas y las interfaces? La siguiente tabla muestra algunas diferencias sintácticas y la tabla posterior muestra algunas diferencias semánticas y de uso.

Diferencias sintácticas

|  |  |  |
| --- | --- | --- |
|  | Clases abstractas | Interfaces |
| Palabra(s) clave | abstract class | Interface |
| Palabra clave para la clase que implementa | extends | Implements |
| Implementación por default | Puede definir una implementación por default de los métodos | No es posible, ya que solo puedes declarar métodos, mas no implementarlos |
| Campos | Puede definir campos estáticos y no-estáticos | No puedes tener campos (variables de instancia) en una interface |
| Constantes | Puede tener tanto estáticas como no-estáticas | Solo puede tener estáticas. Si tú declaras un campo, este debe ser inicializado. Todos los campos son considerados como públicos, estáticos y finales |
| Constructores | Puedes definir un constructor en una clase abstracta (puede ser útil para inicializar campos, por ejemplo) | No es posible |
| Modificadores de acceso | Puedes tener miembros privados y protegidos | No puedes tener ningún miembro privado o protegido; todos son públicos por default |
| Herencia | Una clase solo puede heredar una sola clase, sea abstracta o no | Una clase puede implementar cualquier número de interfaces |

Diferencias semánticas y de uso

|  |  |  |
| --- | --- | --- |
|  | Clases Abstractas | Interfaces |
| Relación es-un (is-a) contra seguir un protocolo | Una clase base abstracta provee un protocolo; en suma, esta sirve como una clase base en una relación es-un | Una interface provee solo un protocolo. Este especifica la funcionalidad que debe ser implementada por las clases que implementen la interface |
| Implementación por default de un método | Posible | No es posible. Todas las clases que implementen la interface deben de definir su propia implementación de los métodos |
| Dificultad para hacer cambios | Es posible hacer cambios a la implementación de una clase abstracta. Por ejemplo, puedes agregar un método con una implementación por default y la clase derivada no se romperá | Si hay muchas clases que implementen una interface, no puedes fácilmente cambiarla. Por ejemplo, si declaras un método nuevo, todas las clases que la implementen dejarán de compilar mientras el nuevo método no sea implementado en cada una de ellas |

Escoger entre una clase abstracta y una interface

Ahora vamos a comparar las clases abstractas y las interfaces. ¿Cuándo escoger entre una clase abstracta y cuándo entre una interface?

* Si tú estás identificando una clase base que abstrae funcionalidad común de un conjunto de clases relacionadas, puedes usar una clase abstracta. Si estás proveyendo métodos comunes o protocolos que puedan ser implementados incluso por clases no relacionadas, es mejor definir una interface
* Si quieres capturar las similitudes de las clases (incluso clases no relacionadas) sin forzar a una relación entre clases, debes usar interfaces. Por otro lado, si hay una relación es-un entre las clases y la nueva entidad, debes declarar la nueva entidad como abstracta

Vamos a ver un ejemplo de escoger entre clases abstractas e interfaces en la aplicación “FunPaint”. Puedes tener a “Shape” como clase abstracta para todas las formas (Circle, Square, etc); esto es un ejemplo de una relación es-un. También, las implementaciones comunes, como una forma paterna, pueden ser puestas en la clase “Shape”. Por lo tanto, “Shape” como clase abstracta es la mejor opción.

En la aplicación “FunPaint”, el usuario puede ejecutar varias acciones sobre los objetos “Shape”. Por ejemplo, unas formas pueden ser rotadas, y otras pueden ser enrolladas. Una forma como el cuadrado puede ser Rotada y otra forma como el Círculo puede ser enrollada. Por esto, no tiene sentido tener los métodos rotate() o roll() en la clase “Shape”. La implementación de rotate() o roll() difiere según la forma, por lo que una implementación por default no debe ser provista. En este caso es mejor usar interfaces en lugar de clases abstractas. Puedes crear las interfaces “Rotatable” y “Rollable” que especificarán el protocolo para rotate() y roll() respectivamente, como se muestra en el siguiente código:

// Shape.java

// Shape es la clase base para todos los objetos de formas; las formas están asociadas

// con una forma padre que es recordada en el campo parentShape field

public abstract class Shape {

abstract double area();

private Shape parentShape;

public void setParentShape(Shape shape) {

parentShape = shape;

}

public Shape getParentShape() {

return parentShape;

}

}

// Rollable.java

// la interface Rollable puede ser implementada por formas circulares como el círculo

// o la elipse

public interface Rollable {

void roll(float degree);

}

// Rotatable.java

// Interface que puede ser implementada por los cuadrados, rectángulos y rombos

public interface Rotatable {

void rotate(float degree);

}

// Circle.java que es una clase concreta que es-un suptipo de la clase Shape; puedes

// enrollarla por lo que debes implementar la interface Rollable

public class Circle extends Shape implements Rollable {

private int xPos, yPos, radius;

public Circle(int x, int y, int r) {

xPos = x;

yPos = y;

radius = r;

}

public double area() {return Math.PI \* radius \* radius;}

@Override

public void roll(float degree) {/\*implementación de este método\*/}

public static void main(String[] args) {

Circle circle = new Circle(10,10,12);

Circle.roll(45);

}

}

// Rectangle.java es una clase concreta y es-un Shape; puede ser rotado por lo que puedes

// implementar la interface Rotatable

public class Rectangle extends Shape implements Rotatable {

private int length, height;

public Rectangle(int l, int h) {

length = l;

height = h;

}

public double area() {return length \* height;}

@Override

public void rotate(float degree) {

// implementación del método

}

}

Composición de objetos

Hasta aquí has aprendido como definir abstracciones en la formación de clases concretas, clases abstractas e interfaces. Abstracciones individuales ofrecen ciertas funcionalidades que necesitas ser combinadas con otros objetos para representar una abstracción más grande: un objeto compuesto está hecho con objetos más pequeños. Necesitas hacer objetos compuestos para resolver problemas de la vida real. En muchos casos, los objetos compuestos comparten una relación tiene-un (has-a) con los objetos contenidos y este concepto expuesto se refiere como “composición de objetos”.

Como una analogía, una computadora es un objeto compuesto que contiene otros objetos como el CPU, memoria, disco duro, ect. En otras palabras, una computadora comparte una relación tiene-un con otros objetos.

Vamos a recolectar la aplicación “FunPaint” donde definiste la clase “Circle”.

public class Circle {

private int xPos;

private int yPos;

private int radius;

public Circle(int x, int y, int r) {

xPos = x;

yPos = y;

radius = r;

}

public String toString() {

return “mid point = (” + xPos + “,” + yPos + “) and radius = ” + radius;

}

}

En esta simple implementación, puedes ver “xPos” y “yPos” para definir el centro de un objeto Circle. En lugar de definir estas variables como miembros de la clase “Circle” vamos a definir la clase “Point”, la cual podrá ser usada para definir el centro del círculo. Checa la definición de esta clase:

// Point.java

class Point {

private int xPos;

private int yPos;

public Point(int x, int y) {

xPos = x;

yPos = y;

}

public String toString() {

return “(” + xPos + “,” + yPos + “)”;

}

}

// Circle.java

public class Circle {

private Point center;

private int radius;

public Circle(int x, int y, int r) {

center = new Point(x, y);

radius = r;

}

public String toString() {

return “center = ” + center + “ and radius = ” + radius;

}

public static void main (String[] a) {

System.out.println(new Circle(10,23,43));

}

}

Esta solución es mejor que teniendo los miembros enteros por separado. ¿Por qué? Puedes reusar la funcionalidad provista por la clase “Point”. Como en este caso, el método toString() de la clase “Circle” es más simple.

Aquí, el uso de la variable “center” se expande a center.toString() como un objeto “Point”. En otras palabras, las clases “Circle” y “Point” comparten una relación tiene-un (has-a); que en otras palabras, “Circle” es un objeto compuesto que contiene un objeto “Point”.

Composición vs Herencia

Ahora ya cuentas con el conocimiento necesario de “composición” así como el de “herencia”. En algunas situaciones, es difícil escoger entre estos dos. Es importante recordar que ninguno es la panacea; no puedes resolver todos los problemas con una sola construcción. Necesitas analizar cada situación detenidamente y decidir cuál es la mejor opción.

Una regla para usar las frases “es-un” (is-a) y “tiene-un” (has-a):

* Una computadora tiene-un CPU
* Un “Circle” es-un “Shape”
* Un “Circle” tiene-un “Point”
* Una laptop es-una computadora
* Un “Vector” es- una “List”

Esta regla puede ser útil para identificar relaciones incorrectas. Por ejemplo, la relación carro es-una llanta es completamente incorrecta, lo que significa que no puedes tener una relación de herencia entre un carro y una llanta. Sin embargo, la relación carro tiene-una llanta es correcta.

|  |  |
| --- | --- |
| Herencia | Es-un (is-a) |
| Implementación de interfaces | Es-como-un (is-like-a) |
| Composición | Tiene-un (has-a) |

En escenarios reales, las distinciones entre relaciones pueden ser no triviales. Has aprendido como hacer una clase base y colocar funcionalidad común en muchas clases dentro de ella. Sin embargo, mucha gente ignora una gran precaución en esta práctica: siempre checa cuando la relación es-un existe entre las clases derivadas y la clase base. Si la relación es-un no funciona, es mejor usar la composición en lugar de la herencia.

Por ejemplo, toma un conjunto de clases: “DynamicDataSet” y “SnapShotDataSet” las cuales requieren funcionalidad común, digamos ordenar “sorting”. Ahora, una puede derivar a estas clases para la implementación de “sorting” como se muestra en el siguiente código:

// Sorting.java

import java.awt.List;

public class Sorting {

public List sort(List list) {

// implementación sort

return list;

}

}

class DynamicDataSet extends Sorting {

// implementación

}

class SnapshotDataSet extends Sorting {

// implementación

}

Esta no es una buena solución por las siguientes razones:

* La regla de relación no se cumple. “DynamicDataSet” no es un tipo “Sorting”. Si cometes muchos errores en el diseño de clases, puede ser muy costoso y difícil de componer si se ha acumulado mucho código que hace incorrecta la relación de herencia. Por ejemplo, “Stack” extiende a “Vector” en la librería de java, puedes agregar o borrar elementos en cualquier parte del contenedor, porque la clase base es “Vector”.
* ¿Qué si estos dos tipos de juego de datos tuvieran una clase base genuina “DataSet”? En ese caso, o “Sorting” será la clase base de “DataSet” o uno podría poner la clase “Sorting” entre “DataSet” y los dos tipos de data-sets mencionados. Ambas soluciones son erróneas
* Aquí hay otro reto: ¿Qué pasa si una clase “DataSet” quiere usar un algoritmo de ordenamiento (digamos MergeSort) y la otra quiere usar un algoritmo diferente (digamos Quicksort)? ¿Si heredas puedes implementar dos tipos diferentes de algoritmos de ordenamiento? Primero, en java no hay herencia múltiple. Segundo, tampoco es buena solución heredar esta funcionalidad encadenada. Es un mal diseño de clases.

En este caso, lo mejor es usar composición. En otras palabras, usa la relación tiene-un en lugar de la relación es-un. El código quedaría así:

// Sorting.java

import java.awt.List;

interface Sorting {

List sort(List list);

}

class MergeSort implements Sorting {

public List sort(List list) {

// algoritmo

return list;

}

}

class QuickSort implements Sorting {

public List sort(List list) {

// algoritmo

return list;

}

}

class DynamicDataSet {

Sorting sorting;

public DynamicDataSet() {

sorting = new MergeSort();

}

// implementación

}

class SnapshotDataSet {

Sorting sorting;

public SnapshotDataSet() {

sorting = new QuickSort();

}

// implementación

}

Usa la herencia cuando una subclase hace más específica a su clase base y con esto explotar el polimorfismo dinámico. En otros casos, usa composición para tener un código fácil de cambiar.

Puntos a recordar

* Agrega aquí a los principios de diseño orientado a objetos “mejor la composición sobre la herencia”. La composición facilita que puedas seguir otro útil principio de diseño: “programar a una interface, no a la implementación”. Este segundo mandato significa que la funcionalidad de una clase puede depender solo de la interface de otra abstracción y no de la implementación específica de esta abstracción. En otras palabras, la implementación de una clase no debe depender de los aspectos de implementación internos de otra clase. Cuando sea posible, usa composición en lugar de herencia
* En la programación orientada a objetos hay muchos términos relacionados a la composición, como asociación, y agregación. Asociación es la forma más general de relación entre dos objetos, mientras que la composición y agregación son formas especiales de asociación. En general, los términos “agregación” y “composición” se usan indistintamente. Aunque estos dos términos son muy similares, tienen diferencias. En la relación de composición, el tiempo de vida del objeto contenido y el objeto contenedor es la misma, mientras que en la agregación no. Por ejemplo, una computadora y un CPU comparten una relación de composición, mientras una librería y un libro comparten una relación de agregación (esquematizar o ilustrar esto)

|  |  |
| --- | --- |
| Herencia | Es-un |
| Asociación-agregación | Tiene-un: Un objeto contiene a otro de manera temporal. El tiempo de vida del objeto contenedor y el objeto contenido son diferentes |
| Asociación-composición | Tiene-un: Un objeto contiene a otro de manera permanente. El tiempo de vida del objeto contenedor y el objeto contenido es el mismo. Uno depende del otro para vivir |

Patrones de diseño

En el mundo orientado a objetos, los conceptos detrás de los patrones de diseño están bien establecidos.

El significado literal de los patrones de diseño en programación es una solución repetida aplicable para resolver un problema de diseño genérico. Los programadores experimentados y diseñadores aprenden de su experiencia y las soluciones formuladas. Los patrones de diseño capturan y replican la experiencia de diseñadores de software experimentados.

CITA: Los patrones de diseño son descripciones de objetos comunicándose que están estandarizadas para resolver un problema de diseño en un contexto particular.

Una clarificación antes de adentrarnos en algunos patrones de diseño: los patrones de diseño son soluciones de diseño. No son soluciones listas para su uso como el código en una librería. Los patrones de diseño proveen una plantilla de solución que necesita ser trabajada según el contexto.

Considera un ejemplo de un patrón de diseño para tener una primera vista de la importancia de estos. En la aplicación “FunPaint”, asume que una clase (digamos ShapeArchiver) es responsable de archivar la información acerca de todas las figuras dibujadas. Similarmente, otra clase (digamos Canvas) es responsable de mostrar en pantalla todas las figuras dibujadas. Cuando cualquiera de las formas tuviera un cambio, estas dos clases tendrían que ser informadas con los cambios realizados. Por lo tanto, ¿Cómo implementarías esta notificación? El siguiente código muestra una posible solución:

// Circle.java

// Esta clase debe notificar a las clases Canvas y ShapeArchiver cuando cambie

// llamando al método update de las dos clases

public class Circle {

private Point center;

public void setCenter(Point center) {

this.center = center;

canvas.update(this);

shapeArchiver.update(this);

}

public void setRadius(int radius) {

this.radius = radius;

canvas.update(this);

shapeArchiver.update(this);

}

private ShapeArchiver shapeArchiver;

public void setShapeArchiver(ShapeArchiver shapeArchiver) {

this.shapeArchiver = shapeArchiver;

}

protected Canvas canvas;

public void setCanvas(Canvas canvas) {

this.canvas = canvas;

}

private int radius;

public Circle(int x, int y, int r) {

center = new Point(x, y);

radius = r;

}

public String toString() {

return “center = ” + center + “ and radius = ” + radius;

}

}

// Point.java

class Point {

private int xPos, yPos;

public Point(int x, int y) {

xPos = x; yPos = y;

}

public String toString() { return “(” + xPos + “,” + yPos + “)”;}

}

Bueno, esta implementación funciona como se intenta, pero hay un problema. Hay una completa dependencia entre el sujeto (clase Circle) y los dos observadores (las clases ShapeArchiver y Canvas). Aquí hay varias consecuencias de este tipo de diseño:

* La clase sujeto “Circle” conoce las clases observadoras. Como resultado, si cambias las clases observadoras necesitas también cambiar la clase sujeto. Esto no es nada bueno
* Si quisieras agregar o remover un observador, tendrías que cambiar también al sujeto
* No puedes reusar el sujeto o las clases observadoras por separado

En la implementación anterior hay varios problemas. Observa el siguiente código ¿Será que se eliminan los problemas mencionados?:

// Circle.java

import java.util.Observable;

public class Circle extends Observable {

private Point center;

public void setCenter(Point center){

this.center = center;

setChanged();

notifyObservers();

}

private int radius;

public void setRadius(int radius) {

this.radius = radius;

setChanged();

notifyObservers();

}

public Circle(int x, int y, int r) {

center = new Point(x, y);

radius = r;

}

public String toString() {

return “center = ” + center + “ and radius = ” + radius;

}

}

// Point.java

class Point {

private int xPos;

private int yPos;

public Point(int x, int y) {

xPos = x;

yPos = y;

}

public String toString() {

return “(” + xPos + “,” + yPos + “)”;

}

}

// Canvas.java

import java.util.Observable;

import java.util.Observer;

public class Canvas implements Observer {

@Override

public void update(Observable arg0, Object arg1) {

System.out.println(“Canvas update ”);

}

}

// ShapeArchiver.java

import java.util.Observable;

import java.util.Observer;

public class ShapeArchiver implements Observer {

@Override

public void update(Observable arg0, Object arg1) {

System.out.println(“ShapeArchiver update”);

}

}

// Test.java

public class Test {

public static void main(String[] a) {

Circle circle = new Circle(10,10,20);

System.out.println(circle);

circle.addObserver(new Canvas());

circle.addObserver(new ShapeArchiver());

circle.setRadius(50);

System.out.println(circle);

}

}

Bueno, la salida será la misma que el programa anterior. ¿Puedes lograr algo mejor? Sí puedes. Esta nueva implementación esta desacoplada. El sujeto “Circle” no sabe acerca de los observadores y los observadores tampoco conocen al sujeto. Consecuentemente, ambos, el observador y los observadores pueden ahora ser usados independientemente y cambiar independientemente. Además de que pues agregar o quitar observadores del sujeto sin cambiar la clase sujeto.

Este ejemplo es una implementación del patrón de diseño “Observer”. Este patrón de diseño es muy útil en casos donde tienes un sujeto (clase) que tiene que ser monitoreada por un conjunto de observadores. Estos observadores necesitan ser informados cuando el sujeto tiene cambios. Este patrón de diseño crea una forma desacoplada entre el sujeto y los observadores.

Java soporta una clase abstracta con el nombre “Observable” y una interface con el nombre “Observer” (ambas provistas por el paquete java.util) para implementar este patrón. La clase “Circle” es una extensión de la clase abstracta “Observable” que etiqueta a la clase “Circle” como sujeto. La clase “Canvas” y “ShapeArchiver” implementan la interface “Observer” y por lo tanto implementan el método update(). Cuando el estado del sujeto cambie, llamas al método setChanged() seguido por el método notifyObservers(), que están implementados en la clase “Observable”. El método notifyObservers() llama a todos los observadores registrados antes en un sujeto.

Esperamos que este ejemplo te dé una buena práctica en patrones de diseño. En el resto de este capítulo exploraremos en detalle los patrones de diseño más importantes.

Los patrones de diseño se pueden clasificar en las siguientes tres categorías:

1. Creacionales: ofrecen la flexibilidad para decidir quién es responsable de la creación de un objeto, cómo el objeto es creado y cuándo es creado el objeto. En esencia, este patrón ofrece una abstracción de la instanciación de un objeto

Ejemplos: Sigleton, Factory, Abstract Factory y Prototype

1. Estructurales: están centrados en cómo se relacionan las clases (y objetos), así como su composición entre ellos, en una estructura grande

Ejemplos: Composite, Decorator, Proxy y Facade

1. De Comportamiento: define la comunicación entre los objetos y controla el flujo entre los objetos participantes

Ejemplos: Mediator, Chain of Responsibility, Observer, State y Strategy

El patrón de diseño SINGLETON

Hay muchas situaciones en donde quieres que una única instancia esté presente para una clase en particular. Por ejemplo, asume que defines una clase que modifica el registro, o tu implementas una clase que maneje la bandeja de una impresora, o implementas el mánager de un pool de hilos. En todos estos casos, necesitas evitar que se generen bugs difíciles de buscar instanciando un único objeto de estas clases. En este caso puedes usar este patrón de diseño.

El SINGLETON está diseñando para asegurar que solo una instancia de la clase es creada. La implementación de este patrón provee una única entrada de acceso a la clase. Este patrón es Creacional, lo que significa que controla la instanciación del objeto. En Java SDK, el patrón es usado en muchos lugares, como java.lang.Runtime.

La siguiente figura muestra el diagrama de clase del patrón SIGLETON.

|  |
| --- |
| Singleton |
| -mySingleton : Singleton |
| -Singleton() +getSingleton() : Singleton |

El patrón SINGLETON ofrece dos cosas: una única instancia de la clase un único punto de acceso a este objeto.

Asume que la aplicación “FunPaint” requiere un logger (monitorea el funcionamiento de la clase así como de sus errores) que quieres que se implemente como SINGLETON

// La clase Logger tiene que ser instanciada solo una vez en la aplicación, para asegurar que

// la aplicación use la misma instancia

public class Logger {

// declara el constructor privado para prevenir que los clientes puedan instanciar

// un objeto de esta clase directamente

private Logger() { }

public static Logger myInstance; // por default este campo se inicializa con null

// el método estático será usado por los clientes para obtener una instancia de la

// clase

public static Logger getInstance() {

if(myInstance == null) {

// el método entra a este if, solo si es la primera vez que

// se invoca, ya que la variable myInstance es null

myInstance = new Logger();

}

return myInstance; // retornará la misma instancia

}

public void log(String s) {

// una implementación trivial de log que solo pasa a la consola el mensaje

System.err.println(s);

}

}

Observa la implementación del Singleton de la clase “Logger”. El constructor de la clase es declarado como privado, por lo tanto, no puedes crear simplemente una instancia nueva de la clase “Logger” usando el operador new. El único camino para obtener una instancia de esta clase es llamar al método estático, miembro de la clase getInstance(). Este método checa si la variable estática Logger myInstance está creada o no. Si no, crea una instancia y la asigna a la variable estática myInstance. Por lo tanto, siempre que llames al método getInstance(), siempre retornará el mismo objeto de la clase “Logger”. En el código anterior, se usa la inicialización perezosa. Puedes utilizar la inicialización temprana si tu implementación Singleton no utiliza muchos recursos de la computadora.

Asegurar que el Sigleton es en efecto un Singleton

Puede que te preguntes de qué estamos hablando, pero es realmente importante (como difícil) asegurar que tu patrón Singleton permite solo una instancia de la clase.

Por ejemplo, la implementación anterior trabaja únicamente si tu aplicación es de un solo hilo (single-threaded). En el caso de múltiples hilos, intentar obtener un objeto Singleton puede resultar en la creación de múltiples objetos, lo que anula el propósito de la implementación Singleton.

El siguiente código muestra una versión de la clase Logger que implementa el patrón Singleton en un entorno multi-hilos.

public class Logger {

private Logger() {// constructor privado para evitar la instanciación directa }

public static Logger myInstance;

public static synchronized Logger getInstance() {

if(myInstance == null)

myInstance = new Logger();

return myInstance;

}

public void log(String s) { System.err.Println(s); }

}

Nota el uso de la palabra clave “synchronized” en esta implementación. Esta palabra clave es el mecanismo de Java para permitir a un solo hilo acceder al ámbito marcado. Aprenderás más acerca de esta palabra clave en el capítulo 13.

Entonces, ahora has sincronizado todo el método para que sea accedido por un solo hilo a la vez. Esto es una buena solución, pero hay un problema: performance pobre. Necesitas hacer este método sincronizado solo cuando sea llamado por primera vez, ya que las veces subsecuentes harán que se haga un cuello de botella.

Ok, ¿Qué pasa si sincronizas solo la sentencia new? Observa el siguiente código.

public class Logger {

private Logger() { }

public static Logger myInstance;

public static Logger getInstance() {

if(myInstance == null) {

synchronized(Logger.class) {

myInstance = new Logger();

}

}

return myInstance;

}

public void log(String s) {

// implementación

System.err.println(s);

}

}

Es un buen intento, pero esta solución tampoco funciona. La sincronización no previene la creación accidental de dos objetos Singleton, ya que esta sincronización está después del if. Ahora, implementemos la famosa double-checked locking (chequeo de doble verificación) en el siguiente código.

public class Logger {

private Logger() { }

public static Logger myInstance;

public static Logger getInstance() {

if(myInstance == null) {

synchronized(Logger.class) {

if(myInstance == null) {

myInstance = new Logger();

}

}

}

return myInstance;

}

public void log(String s) {

System.err.println(s);

}

}

Bueno, esta implementación tampoco es la solución a toda prueba para aplicaciones multi-hilo. Puede crear problemas con escrituras erróneas fuera de orden permitidas por el modelo de memoria de Java. Aunque el problema de memoria ha sido corregido en Java 3, nosotros no recomendamos usar esta solución.

En el siguiente código se muestra otra implementación de la clase Logger que está basada en el estilo “inicialización en la demanda soporte”. Este estilo usa una clase interna y no usa ningún constructo sincronizado. Se aprovecha del hecho de que las clases internas no son cargadas mientras no sean referenciadas.

public class Logger {

private Logger() { }

public static Logger myInstance;

public static class LoggerHolder {

public static Logger logger = new Logger();

}

public static Logger getInstance() {

return LoggerHolder.logger;

}

public void log(String s) {

System.err.println(s);

}

}

El patrón de diseño FACTORY (fábrica)

En la vida real, las fábricas son unidades de manufactura que producen múltiples instancias de un producto (o sabores del mismo). Por ejemplo, una fábrica de carros produce tipos específicos y modelos. La mayor responsabilidad de la fábrica es seguir produciendo carros del tipo requerido y el modelo. Aquí, una cosa importante a observar es que un carro puede tener variantes y la fábrica debe ser capaz de manufacturar bajo demanda las variantes requeridas del mismo carro.

Similarmente, tú puedes implementar una fábrica que retorne el tipo requerido del objeto (u objetos) bajo demanda en la OOP. En este caso, la fábrica decide qué clases instanciar para crear el objeto requerido y cómo será creado exactamente.

El siguiente diagrama de clases muestra la representación UML del patrón FACTORY. “Client” invoca a “ProductFactory” para obtener un objeto apropiado. “ProductFactory” crea uno de los productos de la jerarquía de productos basado en la información provista. “Client” usa el producto sin saber cuál es el producto actual que está usando, ni cómo “ProductFactory” lo creó.



Vamos a considerar un ejemplo. En la aplicación “FunPaint” hay diferentes tipos de formas, como círculo y rectángulo. Ahora, el objeto “Canvas” podría no querer saber acerca de la forma concreta que se creó. Asume que el objeto “Canvas” recibe el identificador de la forma del usuario, basado en cómo la forma correspondiente necesita ser creada. Aquí, puedes tener una “ShapeFactory” (Fábrica de formas) que crea la forma requerida y la manda al “Canvas”. Observa la implementación de este patrón en el siguiente código.

// Shape.java

public interface Shape {

public void draw();

public void fillColor();

}

// Circle.java

public class Circle implements Shape {

private int xPos, yPos, radius;

public Circle(int x, int y, int r) {

xPos = x;

yPos = y;

radius = r;

System.out.println(“Circle constructor”);

}

@Override

public void draw() { System.out.println(“Circle draw()”);

@Override

public void fillColor() { // implementación }

}

// Rectangle.java

public class Rectangle implements Shape {

public Rectangle(int length, int height) {

this.length = length;

this.height = height;

System.out.println(“Rectangle constructor”);

}

private int length, height;

@Override

public void draw() {

System.out.println(“Rectangle draw()”);

}

@Override

public void fillColor() {

// implementación

}

}

// ShapeFactory.java

public class ShapeFactory {

public static Shape getShape(String sourceType) {

switch(sourceType) {

case “Circle”:

return new Circle(10,10,20);

case “Rectangle”:

return new Rectangle(10,20);

}

return null;

}

}

// Canvas.java

import java.util.ArrayList;

import java.util.Iterator;

public class Canvas {

private ArrayList<Shape> shapeList = new ArrayList<Shape>();

public void addNewShape(String shapeType) {

Shape shape = ShapeFactory.getShape(shapeType);

shapeList.add(shape);

}

public void redraw() {

Iterator<Shape> itr = shapeList.Iterator();

while(itr.hasNext()) {

Shape shape = itr.next();

Shape.draw();

}

}

}

A continuación la clase principal de prueba.

public class Test {

public static void main(String… args) {

Canvas canvas = new Canvas();

canvas.addNewShape(“Circle”);

canvas.addNewShape(“Rectangle”);

canvas.redraw();

}

}

Vamos a analizar esta implementación. Tú defines la interface “Shape” que define dos métodos públicos: draw() y fillColor(). Las clases “Circle” y “Rectangle” implementan esta interface y proveen la implementación de los métodos abstractos. La clase “Canvas” mantiene una lista de las formas dibujadas en ella y ofrece el método addNewShape() para permitir que el usuario pueda crear nuevas instancias de la figura solicitada. A partir del método main() invocas el método addNewShape() de la clase “Canvas”. En su momento, este método llama el método getShape() de la clase “ShapeFactory”. El método getShape() examina el tipo de forma solicitada, crea una nueva instancia de la misma y la retorna al canvas.

Las siguientes ideas pueden ser percibidas del ejemplo anterior:

* La clase “Canvas” no necesita conocer cómo se crean las figuras concretas. Esta transparencia se vuelve muy útil en caso de que la creación de objetos concretos sea costosa y complicada
* La clase “Canvas” no necesita saber el tipo concreto de las instancias “Shape”. Puedes observar de la implementación del “Canvas” que solo es consciente de la interface “Shape”. Por lo tanto, si agregas otra clase concreta de “Shape”, no necesitas hacer ningún cambio en la implementación de “Canvas”

El SDK define muchas fábricas. Por ejemplo, java.util.Calendar es una implementación del patrón Factory. El siguiente código es una implementación de la clase java.util.Calendar.

//mainClass.java

public class MainClass {

public static void main(String[] args) {

Calendar calendar = Calendar.getInstance();

System.out.println(calendar);

}

}

Este programa imprime lo siguente

java.util.GregorianCalendar […]

La salida del código anterior contiene todos los campos del objeto calendar.

Hay muchos otros ejemplos en el SDK de Java donde se utiliza el patrón Factory, incluyendo los siguientes:

* createStatement() de la interface java.sql.Connection, que crea una nueva declaración para comunicarse con la base de datos
* createSocket() de la interface java.rmi.server.RmiClientSocketFactory, la cual retorna un nuevo Socket cliente

Diferencia entre los patrones de diseño Factory y Abstract Factory

Ambos, Factory y Abstract Factory existen dentro de la categoría de los patrones de diseño creacionales. Como se explica en la sección anterior, el patrón de diseño Factory crea el tipo de objeto solicitado, bajo demanda. En contraste, el patrón Abstract Factory es, básicamente, una fábrica de fábricas. En otras palabras, el diseño Abstract Factory introduce una o más formas para crear un objeto específico. Un cliente de una Fábrica Abstracta primero solicita la fábrica específica de la fábrica abstracta y después solicita el objeto apropiado de la fábrica específica.

Otra importancia importante entre estos dos patrones es el ámbito de aplicación: cuando tienes solo un tipo de objeto para ser creado, puedes usar el patrón Factory; cuando tienes una familia de objetos para ser creados, puedes usar el patrón Abstrac Factory.

Vamos a explorar el patrón Abstract Factory haciendo referencia a su diagrama de clases. Asume que hay dos jerarquías de productos, ProductA y ProductB a través de sus clases de productos concretos. Quieres crear cualquiera de los dos: ConcreteProductA1 y ConcreteProductB1 (como grupo) o ConcreteProductA2 y ConcreteProductB2. En esta situación defines una ProductFactory abstracta con dos subclases, ProductFactory1 y ProductFactory2. ProductFactory1 crea un ConcreteProductA1 y ConcreteProductB1; ProductFactory2 crea ConcreteProductA2 y ConcreteProductB2. Por lo tanto, basado en el requerimiento, creas un objeto fábrica; la fábrica seleccionada te da los productos concretos requeridos.



Vamos a trabajar en un ejemplo. Ahora asume que las formas pueden ser de dos tipos: DisplayFriendly y PrinterFriendly. Por lo tanto, ahora hay dos sabores disponibles de tu clase “Circle” (como para todas las clases derivadas de “Shape”): DisplayFriendlyCircle y PrinterFriendlyCircle. Ahora, oviamente quieres crear solo un tipo de objetos: cualquiera de los dos. Bueno, has establecido que cuando quieres crear una familia de objetos, debes usar el patrón Abstract-Factory, como se muestra de ejemplo en el siguiente código.

// Shape.java

public interface Shape {

public void draw();

}

// PrinterFriendlyShape.java

public interface PrinterFriendlyShape extends Shape {

}

// DisplayFriendlyShape.java

public interface DisplayFriendlyShape extends Shape {

}

// DisplayFriendlyCircle.java

public class DisplayFriendlyCircle implements DisplayFriendlyShape {

private int xPos, yPos, radius;

public DisplayFriendlyCircle(int x, int y, int r) {

xPos = x;

yPos = y;

radius = r;

System.out.println(“DisplayFriendlyCircle constructor”);

}

@Override

public void draw() {

System.out.println(“DisplayFriendlyCircle draw()”);

}

}

// DisplayFriendlyRectangle.java

public class DisplayFriendlyRectangle implements DisplayFriendlyShape {

private int length;

private int height;

public DirsplayFriendlyRectangle(int length, int height) {

this.length = length;

this.height = height;

System.out.println(“DisplayFriendlyRectangle constructor”);

}

@Override

public void draw() {

System.out.println(“DisplayFriendlyRectangle draw()”);

}

}

// PrinterFriendlyCircle.java

public class PrinterFriendlyCircle implements PrinterFriendlyShape {

private int xPos, yPos, radius;

public PrinterFriendlyCircle(int x, int y, int r) {

xPos = x;

yPos = y;

radius = r;

System.out.println(“PrinterFriendlyCircle constructor”);

}

@Override

public void draw() {

System.out.println(“PrinterFriendlyCircle draw()”);

}

}

// PrinterFriendlyRectangle.java

public class PrinterFriendlyRectangle implements PrinterFriendlyShape {

private int length;

private int height;

public PrinterFriendlyRectangle(int length, int height) {

this.length = length;

this.height = height;

System.out.println(“PrinterFriendlyRectangle constructor”);

}

@Override

public void draw() {

System.out.println(“PrinterFriendlyRectangle draw()”);

}

}

// ShapeFactory.java

public interface ShapeFactory {

public Shape getShape(String sourceType);

}

// DisplayFriendlyFactory.java

public class DisplayFriendlyFactory implements ShapeFactory {

@Override

public Shape getShape(String sourceType) {

switch(sourceType) {

case “Circle”:

return new DisplayFriendlyCircle(10,10,20);

case “Rectangle”:

return new DisplayFriendlyRectangle(10,30);

}

return null;

}

}

// PrinterFriendlyFactory.java

public class PrinterFriendlyFactory implements ShapeFactory {

@Override

public shape getShape(String sourceType) {

switch(sourceType) {

case “Circle”:

return new PrinterFriendlyCircle(10,10,20);

case “Rectangle”:

return new PrinterFriendlyRectangle(10,20);

}

return null;

}

}

Estas son las características que debes aprender:

* Jerarquía de productos: “Shape” es la interface base que es extendida por las interfaces “DisplayFriendlyShape” y “PrinterFriendlyShape”. Dos tipos de la clase “Circle” y “Rectangle” son definidas por cada interface: DisplayFriendly y PrinterFriendly
* La implementación de Abstract Factory: Está hecha por “ShapeFactory” como interface base, y “PrinterFriendlyFactory”, así como “DisplayFriendlyFactory” son las fábricas concretas de esta interface.

El SDK de java emplea el patrón Abstract Factory en muchos lugares, como por ejemplo:

* javax.xml.transform.TransformerFactory
* javax.xml.xpath.XPathFactory

Fuera del ámbito del examen, pero otra forma notable de implementar el patrón Abstract Factory es usando la Api Reflection del SDK. La principal ventaja de usar reflexión es que puedes extender tu jerarquía de productos sin cambiar la implementación de la fábrica o comprometer la habilidad de crear objetos de las nuevas clases agregadas.

El patrón de diseño DAO (Data Access Object)

Supón que estás en una feria multicultural con tu familia y decides tomar un paseo en el carrusel. Ves un gran panel con muchos botones para operar el carrusel. Le pides al operador iniciar el carrusel, ajustar la velocidad y parar. El operador, que conoce como se usa el panel, sigue tus instrucciones. El te está proporcionando una abstracción del complicado panel de control. De hecho, si vas a un carrusel diferente, el operador seguirá las mismas instrucciones, aunque el panel sea diferente del primer carrusel. En esencia, tu familia puede subirse a cualquier carrusel sin necesidad de entender cómo opera, porque este conocimiento debe ser dominado por el operador. El patrón de diseño DAO provee una abstracción análoga a este ejemplo.

En proyectos de la vida real, encontrarás situaciones donde querrás hacer los datos de tu aplicación persistentes. Podrías utilizar archivos planos, xml, oodbms, rdbms, etc. En tales situaciones puedes usar el patrón de diseño DAO. Este patrón de diseño abstrae los detalles del mecanismo de persistencia subyacente y te ofrece una interface fácil de usar parar implementar la persistencia en tu aplicación. El patrón DAO esconde los detalles de implementación de la fuente de datos a sus cluentes, así como introduce el acoplamiento flexible entre tu lógica de negocio principal y tu mecanismo de persistencia. Esta flexibilidad de acoplamiento te permite migrar a otro tipo de mecanismo de persistencia sin un gran cambio.

Este patrón, en esencia, separa tu lógica de negocio principal con tu lógica de persistencia.

Vamos a examinar la estructura del patrón, que se muestra en la siguiente figura. “DataSource” representa una implementación concreta de persistencia como RDBMS, XML, base de datos o cualquier otro sistema/repositorio. DAO provee una abstracción del “DataSource”, esconde los detalles específicos de implementación y provee una única interface para todos los diferentes tipos de orígenes de datos. Un “Client” es un usuario del DAO que usa el “DataSource” a través del DAO y “TransferObject” es un objeto de transferencia de datos usado como medio para transferir los objetos principales.



Aparte de los participantes mencionados anteriormente, puede haber un participante más en este patrón: DAOFactory. Puedes tener múltiples objetos DAO, correspondientes a todos los diferentes tipos de objetos que quieres almacenar. Puedes definir una fábrica que puede tener un método por cada objeto DAO. No te preocupes… vas a implementar una fábrica en el siguiente ejemplo.

Asume que quieres implementar un DAO en la aplicación FunPaint. Aquí, tú tienes la clase “Circle” que quieres almacenar en un almacén de datos:

// Circle.java

public class Circle {

private int xPos, yPos;

private int radius;

public Circle(int x, int y, int r) {

xPos = x;

yPos = y;

radius = r;

}

public String toString() {

return “center = (” + xPos + “,” + yPos + “) and radius” + radius;

}

public CircleTransfer getCircleTransferObject() {

CircleTransfer circleTransfer = new CircleTransfer();

circleTransfer.setRadius(radius);

circleTransfer.setxPos(xPos);

circleTransfer.setyPos(yPos);

return circleTransfer;

}

}

// CircleDAO.java

public interface CircleDAO {

public void insertCircle(CircleTransfer circle);

public CircleTransfer findCircle(int id);

public void deleteCircle(int id);

}

// RDBMSDAO.java

public class RDBMSDAO implements CircleDAO {

@Override

public void insertCircle(CircleTransfer circle) {

// trivial implementation

System.out.println(insertCircle);

}

@Override

public CircleTransfer findCircle(int d) {

System.out.println(“findCircle() implementation”);

return null;

}

@Override

public void deleteCircle(int id) {

System.out.println(“deleteCircle() implementation”);

}

}

// DAOFactory.java

public class DAOFactroy {

public static CircleDAO getCircleDAO(String sourceType) {

// Este es un simple ejemplo, solo hemos listado a “RDBMS” como

// el único origen de datos.

// En el mundo real puedes proveer más tipos de origen de datos

switch(sourceType) {

case “RDBMS”:

return new RDBMSDAO();

}

return null;

}

}

// CircleTransfer.java

import java.io.Serializable;

public class CircleTransfer implements Serializable {

private int xPos;

private int yPos;

private int radius;

public void setxPos(int xPos){

this.xPos = xPos;

}

public int getxPos() {

return xPos;

}

public void setyPos(int yPos) {

this.yPos = yPos;

}

public int getyPos() {

return yPos;

}

public void setRadius(int radius) {

this.radius = radius;

}

}

// Test.java

public class Test {

public static void main (Stirng[] args) {

Circle circle = new Circle(10,10,20);

System.out.println(circle);

CircleTransfer circleTransfer = circle.getCircleTransferObject();

CircleDAO circleDAO = DAOFactory.getCircleDAO(“RDBMS”);

circleDAO.insertCircle(circleTransfer);

}

}

Bueno, esto es un gran programa, vamos a revisarlo paso por paso. La clase “Circle” pertenece a tu lógica de negocio principal, aparte de los otros miembros usuales, la clase “Circle” contiene un método getCircleTransferObject() que retorna un objeto de la clase “CircleTransfer” con los datos requeridos. Defines la interface “CircleDAO” con una implementación concreta para accesar al sistema manejador de bases de datos. El objeto CircleTransfer juega el rol de soporte de datos entre el método main() (cliente) y la implementación del DAO, en este caso la clase “RDBMS”.

Una característica más del código anterior, es el uso del patrón Factory. En una aplicación real puede haber muchos objetos DAO. Por ejemplo, en la aplicación “FunPaint” puedees tener las clases “CircleDAO”, “RectangleDAO”, “SquareDAO”, etc. Puedes definir los métodos getter para obtener el DAO correspondiente, en una única fábrica de DAO’s. En cada método, debes retornar el objeto DAO apropiado basado en el tipo provisto como parámetro al switch.

Aquí veremos los beneficios del patrón de diseño DAO

* El patrón presenta una abstracción: el DAO oculta los detalles de la implementación de la fuente de datos actual a la lógica de negocio principal. La lógica de negocio no necesita saber el meollo de la cuestión del origen de datos, que resulta en un código fácil de entender y menos complicado
* El patrón separa el mecanismo de persistencia del resto del código de la aplicación y pone juntos en una clase específica para un origen de datos. Esta centralización te permite un fácil mantenimiento y búsqueda de bugs
* Es muy fácil extender soporte para otros orígenes de datos usando este patrón. Por ejemplo, si quieres proveer soporte para un repositorio basado en XML, en la aplicación “FunPaint”, esto puede ser logrado definiendo una nueva clase “XMLDAO”. Esta nueva clase implementará la interface CircleDAO, por lo que no tendrás que cambiar la forma a la que tu accedes al origen de datos. La única cosa que necesitas cambiar es el parámetro que pasas a la clase DAOFactory para crear el DAO

Puntos a recordar

* Has visto que el patrón de diseño Factory en el patrón DAO. También puedes usar el patrón Abstract Factory si tienes muchos objetos DAO y muchos mecanismos de persistencia
* Nota que tu declaras un TransferObject (ejemplo CircleTransfer) como serializable para poder transferir el objeto entre dos JVM diferentes
* En la Programación Orientada a Objetos, es muy útil e importante el principio de la separación de responsabilidades. Este principio establece que las responsabilidades (o características) deben de separarse (para obtener un mínimo de overlap) con el fin de superar la complejidad inherente al diseño de software. El patrón DAO te ayuda a cumplir con este principio de diseño. Si no usas DAO, tu lógica de negocio estará expuesta a una implementación concreta de un mecanismo de persistencia específico, lo cual es indeseable. El uso del patrón DAO te asegura la separación entre el mecanismo de persistencia y la lógica de negocio.

Resumen

Interfaces

* Una interface es un conjunto de métodos abstractos que definen un protocolo
* Una interface no puede ser instanciada; sin embargo, una interface puede extender a otras interfaces
* Todos los métodos declarados en una interface son implícitamente abstractos
* Una clase abstracta y una interface son conceptos similares. Sin embargo, tienes que ser cuidadoso para usar el constructo asociado basado en el contexto

Composición

* La herencia implica una relación es-un, una interface implica una relación es-como-un y la composición implica una relación tiene-un
* Usa la composición en lugar de la herencia cuando sea posible
* Programa a una interface, no a la implementación

Patrones de diseño

* Los patrones de diseño son soluciones reusables para problemas de diseño recurrentes
* El patrón Observador mejora el acoplamiento flexible entre el sujeto observado y los observadores
* El patrón Singleton asegura que solo una instancia de la clase es creada en la aplicación
* Asegurar que la implementación del singleton sea realmente singleton es una tarea no-trivial, especialmente en entornos multi-hilo
* El patrón Factory manufactura el tipo de objeto requerido
* Puedes considerar usar el patrón Abstract Factory cuando tienes una familia de objetos para ser creados
* El patrón DAO separa tu lógica de negocio de tu lógica de persistencia

Capítulo 6

Colecciones y genéricos

Tópicos del examen

* Crear una clase genérica
* Usar la sintaxis de diamante para crear una colección
* Analizar la interoperabilidad de las colecciones usando los tipos raw y los tipos genéricos
* Usar las clases “envoltorio” y el autoboxing
* Crear u usar un List, un Set y un Deque
* Crear y usar un Map
* Usar java.util.Comparatory java.lang.Comparable
* Ordenar y buscar en arrays y lists

Todas las aplicaciones no triviales en java, están hechas usando estructuras de datos y algoritmos. El framework Collections de Java provee un amplio juego de estructuras de datos y algoritmos de uso general, listos para usarse. Estas estructuras de datos y algoritmos pueden ser usados con cualquier tipo de datos en una manera type-safe; esto se logra a través del uso de una característica del lenguaje llamada Generics.

Desde que las estructuras de datos y los algoritmos son implementados usando Generics, estos dos temas complementarios son combinados juntos como un único tema en el programa de estudios del examen 1zo-804.

Generics (genéricos)

Generics son una característica del lenguaje java introducida desde la versión 1.5. Generics te ayuda a escribir código para un tipo (digamos T) que es aplicable para todos los tipos, en lugar de escribir clases separadas para cada tipo. Antes de los genéricos, la clase base Object era usada como una alternativa. Sin embargo, usar la clase Object para almacenar elementos en un contenedor resultaba en errores de type-safe, esto es, usando un recipiente tal permite errores de tipo que no se pueden identificar por el compilador de java y resultaban en excepciones en el runtime. Si no entiendes esta descripción de tipos seguros, no te preocupes porque lo vamos a explicar con detalle. Entender los tipos seguros es importante. De hecho, la principal motivación por agregar los genéricos al lenguaje es porque los contenedores originales no son de tipo seguro.

Usar tipo Object y tipo Safety

Antes de la introducción de los genéricos en java, las estructuras de datos eran implementadas mediante el almacenamiento y proceso de elementos usando la clase base Object. Bueno, discutamos un ejemplo para entender por qué estos contenedores no son de tipo seguro. Considera el programa en el siguiente código que usa la clase Vector. ¿Puedes predecir la salida de este programa?

// OldContainerTest.java

import java.util.Vector;

// Este programa te demostrará la falta de tipo seguro en contenedores basados en Object

class OldContainerTest {

public static void main(String[] args) {

Vector floatValues = new Vector();

floatValues.add(10.0f);

floatValues.add(100.0);

for(int i = 0; i < floatValues.size(); i++) {

System.out.println(floatValues.get(i));

}

}

}

// imprimirá

10.0

100.0

Ahora vamos a realizar un cambio en el programa: en lugar de imprimir directamente los valores regresados por el método get de la clase “Vector”, primero almacenarás en una variable temporal de tipo Float y después imprimirás el valor temporal usando println()

Float temp = (Float) floatValues.get(i);

System.out.println(temp)

// ahora esta es la salida

10.0

Exception in thread “main” java.lang.ClassCastException: java.lang.Double cannot be cast to java.lang.Float at OldContainerTest.main(OldConteinerTest.java:9)

Esto pasa porque el contenedor Vector es capaz de almacenar cualquier objeto. Cuando intentas colocar un tipo primitivo, este es encajado automáticamente en la clase envoltorio y esta se coloca en el contenedor (autoboxing) . Para 10.0 el tipo correspondiente es float por el sufijo “f”, entonces el envoltorio de tipo Float es creado y almacenado en el vector. Para 100.0 el primitivo es Double ya que no especifica ningún sufijo. Entonces, el objeto creado es de tipo Double y este es colocado en el Vector.

Un vector es implementado en términos del tipo Object, Entonces puedes colocar un objeto de cualquier tipo en el vector. No puedes forzar a que solo elementos de un solo tipo puedan ser almacenados en el vector en tiempo de compilación. De hecho, cuando compilas el programa, el compilador lanza una advertencia referente al uso de tipos raw. Bueno, discutiremos esto más adelante.

Un Vector que usa la clase base Object para almacenar elementos te permite agregar elementos de cualquier tipo, mientras la lógica de la aplicación requiere que solo Strings puedan agregarse. Con Generics, puedes especificar al compilador el tipo de elemento que necesitas agregar al contenedor. Generics asegurará que cualquier intento de agregar elementos de otro tipo que el especificado, será cachado en tiempo de compilación. En otras palabras, el uso de estructuras de datos implementadas usando Generics es revisado su tipo seguro en tiempo de compilación, por lo tanto estas estructuras son de tipo seguro.

Usar la clase Object vs Generics

En la sección anterior, nos enfocamos en la discusión de los tipos seguros al usar estructuras de datos que eran implementadas usando la clase base Object. En esta sección nos enfocaremos en definir nuestras propias estructuras de datos. Con un ejemplo de una clase contenedora usando Object , entenderás por qué tales contenedores no son de tipo seguro. Después de esto, revisaremos el concepto de Generics y cómo se usan para definir contenedores de tipo seguro.

Contenedor usando Object

Asume que quieres imprimir el valor del objeto dentro de unas corcheas. Por ejemplo, para imprimir un objeto Integer con el valor 10, en lugar de imprimirlo “10” imprimirlo “[10]”. La definición por default del método toString en Integer, solo te brinda una representación en cadena de caracteres del valor del objeto; tú quieres mejorar el método toString() para que imprima las corcheas. ¿Cómo puedes hacer esto? Puedes escribir la clase BoxPrinter que sobre-escriba el método toString() . Puedes usar la clase Object para almacenar el valor, ya que esta es la clase base para todos los tipos:

// BoxPrinter.java

class BoxPrinter {

private Object val;

public BoxPrinter(Object arg) { val = arg; }

public String toString() {

return “[” + val + “]”;

}

}

// BoxPrinterTest.java

class BoxPrinterTest1 {

public static void main(String[] args) {

BoxPrinter value1 = new BoxPrinter(new Integer(10));

System.out.println(value1);

BoxPrinter value2 = new BoxPrinter(“Hello world”);

System.out.println(value2);

}

}

// el programa trabaja bien e imprime:

[10]

[Hello world]

Puedes pasar cualquier tipo de objeto, y est e código imprime entre corchetes correctamente. Este es el mismo mecanismo que usan los contenedores originales como Vector. Entonces, ¿Cuál es el problema? Bueno, aquí esta un escenario: asume que quieres agregar un método llamado getValue() que te retorne el valor almacenado el el objeto BoxPrinter. ¿Cómo retornarás el tipo original que almacenaste cuando creaste el objeto BoxPrinter? Aquí es donde el problema empieza:

class BoxPrinter {

private Object val;

public BoxPrinter(Object arg) {

val = arg;

}

public String toString() {

return “[” + val + “]”;

}

public Object getValue() {

return val;

}

}

class BoxPrinterTest2 {

public static void main(String… args) {

BoxPrinter value1 = new BoxPrinter(new Integer(10));

System.out.println(value1);

Integer intValue1 = (Integer) value1.getValue();

BoxPrinter value2 = new BoxPrinter(“Hello world”);

System.out.println(value2);

Integer intValue2 = (Integer) value2.getValue();

}

}

// la salida será

[10]

[Hello world]

Exception in thread “main” java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer at BoxPrinterTest2.main(Main.java:22)

in the line

Integer intValue2 = (Integer) value2.getValue();

Por error ejecutaste un downcast de Object a Integer en lugar de Object a String. Aquí no hay forma de que el compilador detecte el error. Que tú recuerdes como el valor es de tipo Object y el único camino para checar el tipo correcto antes del downcast es checando el tipo usando “instanceof”. En general, sin embargo, cuando no sabes el tipo de objeto que está almacenado en el contenedor, no podrás ejecutar un “instanceof” para hacer el downcast. Esta es la rason del por qué los contenedores antiguos de java no son de tipo seguro.

Contenedor usando Generics

class BoxPrinter<T> {

private T val;

public BoxPrinter(T arg) {

val = arg;

}

public String toString() {

return “[” + val + “]”;

}

}

class BoxPrinterTest3 {

public static void main(String… args) {

BoxPrinter<Integer> value1 = new BoxPrinter<>(new Integer(10));

System.out.println(value1);

BoxPrinter<String> value2 = new BoxPrinter<>(“Hello world”);

System.out.println(value2);

}

}

// imprime

[10]

[Hello world]

En este código hay varias cosas que debes notar.

1. Observa la declaración de la clase BoxPrinter

class BoxPrinter<T>

Has dado a la clase BoxPrinter un marcador de tipo <T>. Puedes usar este tipo para indicar el tipo actual que será provisto a la clase.

1. Dentro de la clase utilizas el parámetro T en la declaración de los campos

private T val;

Con esto declaras “val” como genérico

1. Usas T como parámetro del constructor

public BoxPrinter(T arg)

Con esto declaras el argumento del constructor como genérico y con esto aseguras que el valor que se pase al contenedor pueda ser checado en tiempo de compilación.

Ahora vamos a agregar el método getValue() como tipo seguro

class BoxPrinter<T> {

private T val;

public BoxPrinter(T arg) {

val = arg;

}

public String toString() {

return “[” + val + “]”;

}

public T getValue() {

return val;

}

}

class BoxPrinterTest4 {

public static void main(String… args) {

BoxPrinter<Integer> value1 = new BoxPrinter<>(new Integer(10));

System.out.println(value1);

Integer intValue1 = value1.getValue();

BoxPrinter<String> value2 = new BoxPrinter<>(“Hello world”);

System.out.println(value2);

// error a propósito y se obtiene un error de compilación

Integer intValue2 = value2.getValue();

}

}

// error de compilación

Esto es mejor: En lugar de obtener un error en tiempo de ejecución ahora tienes un error de compilación. Ahora puedes corregir este error y el programa correrá correctamente.

Crear clases Genéricas

Vamos a crear la clase genérica “Pair” que puede contener objetos de dos tipos diferentes, T1 y T2. No te preocupes demasiado de qué tan útil puede ser para resolver un problema del mundo real. Solo intenta entender como escribir genéricos.

class Pair <T1, T2> {

T1 object1;

T2 object2;

Pair(T1 one, T2 two) { object1 = one; object2 = two; }

public T1 getFirst() { return object1;}

public T2 getSecond { return object2;}

}

class PairTest {

public static void main(String[] args) {

Pair<Integer, String> worldCup = new Pair<>(2010, “South Africa”);

System.out.println(“World cup ” + worldCup.getFirst() +

“ in ” + worldCup.getSecond();

}

}

// imprimirá

World cup 2010 in South Africa

Los parámetros T serán reemplazados internamente por el tipo provisto en el argumento a la clase, dentro de los símbolos <>.

Interoperabilidad entre los tipos RAW y los Genéricos

Un tipo genérico puede usarse sin especificar su tipo asociado; en este caso, el tipo es referido como RAW (cuando no se especifica el parámetro T en la construcción del objeto). Por ejemplo, List<T> puede usarse con un tipo asociado List<String>; sin embargo, puede usarse sin especificar el tipo (List). En el último caso, la List es referida como tipo RAW.

Cuando usas un tipo RAW, pierdes la ventaja de tipo seguro que otorgan los genéricos. Por ejemplo, el tipo Vector en el siguiente código, es un tipo RAW. En tiempo de compilación, el compilador genera una advertencia. Los tipos RAW dejan pasar el chequeo de tipo en tiempo de compilación; sin embargo, podrían lanzar excepciones en tiempo de ejecución. Por lo tanto no es recomendado usar tipos RAW en código nuevo.

Ok, ahora entiendes que no debes usar tipos RAW. Pero, puedes preguntarte ¿Por qué el compilador no prohíbe este tipo de declaraciones? La respuesta está en la compatibilidad anterior. Los genéricos de Java fueron introducidos hasta la versión 5. Java soporta RAW con el fin de hacer compatible el código genérico con el código original. Vamos a analizar por qué no usarlos:

class RawTest {

public static void main(String[] args) {

List list = new LinkedList();

list.add(“first”);

list.add(“second”);

List<String> strList = list; //#1

for(Iterator<String> itemItr = strList.iterator(); itemItr.hasNext(); ) {

System.out.println(“Item : ” + itemItr.next());

}

List<String> strList2 = new LinkedList<>();

strList2.add(“first”);

strList2.add(“Second”);

List list2 = strList2; // #2

for(Iterator<String> itemItr = list2.iterator(); itemItr.hasNext(); ) {

System.out.println(“Item : ” + itemItr.next());

}

}

}

El programa anterior compila y se ejecuta sin problemas.

Ahora hagamos un para de cambios al código

class RawTest {

public static void main(String[] args) {

List list = new LinkedList(); // raw

list.add(“first”);

list.add(“second”);

List<String> strList = list;

strList.add(10);// genera un error de compilación

for(Iterator<String> itemItr = strList.iterator(); itemItr.hasNext(); )

System.out.println(“Item : ” + itemItr.next());

List<String> strList2 = new LinkedList<>(); // generic

strList2.add(“First”);

strList2.add(“Second”);

List list2 = strList2;

list2.add(10); // compila pero genera una excepción en tiempo de ejecución

for(Iterator<String> itemItr = list2.iterator(); itemItr.hasNext() ; )

System.out.println(“Item : ” + ítemItr.next());

}

}

En el código anterior intentas agregar un ítem Integer en un List<String>, entonces obtienes un error de compilación. En el segunrdo error, list2 es declarado como raw einicializado con un tipo genérico List<String>. Después de la inicialización agregas un Integer en la Lista tipo raw. Esto esta permitido ya que list2 es de tipo raw. Sin embargo, resultará en un error en tiempo de ejecución “ClassCastException”.

¿Qué puedes aprender de estos ejemplos? La lección es evitar mezclar tipos raw y tipos genéricos en tus programas, ya que puede resultar en un comportamiento erróneo en tiempo de ejecución. Si necesitas usar los dos tipos en un programa, asegúrate que agregas un único tipo de ítems a los contenedores y recuperalos usando el mismo tipo.

Métodos Genéricos

Similarmente a las clases genéricas, puedes crear métodos genéricos. Esto es, métodos que toman un parámetro de tipo genérico. Los métodos genéricos son útiles para escribir métodos que son aplicables en un amplio rango de tipos mientras la funcionalidad es la misma. Por ejemplo, hay numerosos métodos genéricos en la clase java.util.Collections.

Vamos a implementar un método simple llamado fill(). Dado un contenedor, el método fill() llena todos los elementos del contenedor con el valor “val”. Es siguiente código contine la implementación del método fill() en la clase “Utilities”:

// Utilities.java

class Utilities {

public static <T> void fill(List<T> list, T val) {

for(int i = 0; i < list.size(); i++)

list.set(i,val);

}

}

// UtilitiesTest.java

class UtilitiesTest {

public static void main(String[] args) {

List<Integer> intList = new ArrayList<>();

intList.add(10);

intList.add(20);

System.out.printn(“The original list is: ” + intList);

Utilities.fill(intList, 100);

System.out.println(The list after fill() method call is: + intList);

}

}

// imprime

The original list is: [10, 20]

The list after fill() method cal lis: [100, 100]

Vamos a revisar paso a paso este código.

Primero: creas un método llamado fill() en la clase “Utilities”. Declaras un parámetro de tipo genérico <T> en este método. Después de los cualificadores “public static”, colocas <T> y después el tipo de retorno, el nombre del método y sus parámetros. Esta declaración es diferente a las clases genéricas ya que tú provees el tipo genérico <T> después del nombre de la clase.

Segundo: en el cuerpo del método escribes código como un método normal. El loop for recorre desde 0 hasta el tamaño del List, y toma la posición del índice en el contenedor como el primer argumento y el valor actual se establece como el segundo argumento.

Tercero: en el método main() en la clase “UtilitiesTest” está como llamas al método genérico fill(). Nota que tú no proporcionaste el tipo genérico de manera explícita. Ya que IntList es de tipo Integer y 100 puede ser encajado en un Integer, el compilador infiere el tipo T en el método fill().

NOTA: es un error común importar la clase List de java.awt en lugar de java.util. Tu programa no compilará o producirá advertencias si no usas el import correcto. Recuerda que java.util.List es genérico y java.awt.List no.

Hemos discutido los métodos genéricos estáticos. ¿Cómo serían los métodos genéricos no-estáticos? En realidad no hay diferencia . Oviamente necesitas crear una instancia de la clase para poder utilizar un método no-estático.

Genéricos y subtyping

Recuerda que para llenar el diamante de los genéricos debes usar el mismo tipo.

Puedes asignar un objeto de tipo derivado a su referencia de tipo base; esto significa cuando dices suptyping. Sin embargo, para los genéricos, el tipo del parámetor debe coincidir exactamente, de otra manera obtendrás un error de compilación. En otras palabras, suptyping no funciona en parámetros genéricos. Sí, esta es una regla difícil de recordar, pero discutamos en más detalle por qué.

// imagina que este código compila

List<Number> intList = new ArrayList<Integer>();

intList.add(new Integer(10)); // okay

intList.add(new Float(10.0f)); // oops

Como ves en este ejemplo, el uso de subtyping en parámetros genéricos no es permitido porque no es seguro; pero aun así se trata de una limitación inconveniente. Afortunadamente, Java soporta el parámetro de tipo wildcard (comodín) en donde puedes usar subtipos. Vamos a explorar esa capacidad ahora.

Parámetros comodines

// no compila

List<Number> intList = new ArrayList<Integer>();

// compila

List<?> intList = new ArrayList<Integer>(); // usa el comodín ?

¿Qué significa comodín? Justo como un comodín, lo usas para sustituir por cualquier carta en un juego de cartas, puedes usar el comodín para indicar que puede coincidir con cualquier tipo. Con List<?> quieres decir que es una lista de cualquier tipo. En otras palabras, es una lista de tipo desconocido.

Pero espera un minuto… cuando querías indicar “cualquier tipo” usas la clase Object. ¿Qué pasa si usas el mismo código anterior pero usando la clase Object?

List<Object> intList = new ArrayList<Integer>(); // no compila

En otras palabras, estás intentando hacer subtyping para los tipos genéricos y estos siguen sin funcionar. Como puedes ver, List<Object> no es lo mismo que List<?>. De hecho List<?> es un supertipo de cualquier tipo de List, lo que significa que puedes pasar List<Integer>, List<String> e incluso List<Object> donde se espera una List<?>.

Vamos a usar el comodín en un ejemplo y observa si funciona.

class WildCardUse {

static void printList(List<?> list) {

for (Object l: list)

System.out.println(“[” + l + “]”);

}

public static void main(String[] args) {

List<Integer> list = new ArrayList<>();

list.add(10);

list.add(100);

printList(list);

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

strList.add(“10”);

strList.add(“100”);

printList(strList);

}

}

// imprimirá

[10]

[100]

[10]

[100]

Bueno, funciona y la lista usando el comodín se le pueden pasar listas de Integer como de String como de cualquier otra clase.

Limitaciones de los comodines

Vamos a analizar el siguiente trozo de código:

List<?> wildCardList = new ArrayList<Integer>();

wildCardList.add(new Integer(10));

System.out.println(wildCardList);

// se obtiene el siguiente error de compilación

WildCardUse.java:7: cannot find symbol

symbol : method add(java.lang.Integer)

location : interface java.util.List<capture#145 of ? extends java.lang.Number>

wildCardList.add(new Integer(10));

¿Por qué? Estás absolutamente seguro que el método add() existe en la interface List. Entonces ¿Por qué el compilador no encuentra el método?

El problema requere la explicación de algunos detalles. Cuando usas el tipo comodín <?> le dices al compilador que ignoras el tipo de la información, entonces <?> permanece como de tipo desconocido. Siempre que intentes pasar argumentos a un tipo genérico, el compilador intenta inferir el tipo de los argumentos pasados, así como el tipo de los genéricos y esto justifica el tipo seguro. Ahora, estás intentando usar el método add() para insertar un elemento a la lista. Ya que wildCardList no conoce qué tipo de objetos almacena, esto incluye un riesgo al añadir elementos a la lista. Podrías terminar la adición de una cadena (“hello” por ejemplo) en lugar de un valor entero. Para evitar este problema, el compilador no te permite llamar métodos que modifiquen el objeto, Ya que el método add() modifica el objeto obtienes un error de compilación que se ve un poco confuso.

NOTA: en general, cuando usas un parámetro comodín no puedes llamar métodos que modifiquen el objeto. Si lo intentas, el compilador te dará mensajes confusos de errores. Sin embargo, puedes llamar a métodos que accedan al objeto.

Comodines acotados

Vamos a recapitular aspectos importantes de los comodines para entender por qué necesitas comodines acotados. Tú tienes errores de compilación cuando intentas crear genéricos que difieran en el tipo de sus parámetros:

// compiler error

List<Number> numList = new ArrayList<Integer>();

// ahora cambias por comodines para evitar este error de compilación

List<?> numList = new ArrayList<Integer>();

// asume que quieres que se pueda almacenar solo una lista de números en numList.

// sin embargo, podrías terminar almacenando una lista de cualquier tipo, como sigue:

List<?> numList = new ArrayList<Integer>();

numList = new ArrayList<String>(); //compila ya que <?> es el supertipo de todos

// Compila sin errores. ¿Cómo puedes restringir numList para referirse solo a la clase Number y sus clases derivadas como Integer, Double, etc.? Necesitas usar comodines acotados como sigue:

List<? extends Number> numList = new ArrayList<Integer>();

// si intentas

numList = new ArrayList<String>();

// obtendrás un error de compilación

// si lo intentas con este código compilará a la perfección

List<? extends Number> numList = new ArrayList<Integer>();

numList = new ArrayList<Double>(); // compila ya que Double es derivado de Number

En el último ejemplo compila bien. ¿Qué pasa aquí? en List<? extends Number>, el comodín es acotado con “extends Number”. Esto significa que cualquier tipo que quieras sustituir para el comodín (?) debe satisfacer la condición “extends Number”, por lo que todas las clases que hereden de Number podrán ser sustituidas. Entonces la compilación será exitosa para todas estas sustituciones. Cualquier intento por sustituír un tipo que no herede de Number generará un error de compilación (por ejemplo String).

Puedes usar comodines acotados también en los argumentos de los métodos, tipos de retorno, etc. Aquí mostraremos un simple método que usa comodines acotados.

public static Double sum(List<? extends Number> numList) {

Double result = 0.0;

for(Number num: numList) {

result += num.doubleValue();

}

return result;

}

Aquí hay una explicación paso por paso de este método:

1. El método sum() se entiende que toma una lista de Numbers, y retorna la suma de los elementos de la lista
2. Ya que List es limitado por la clase Number, declaras List<? extends Number>
3. Ya que no conoces el tipo exacto de los elementos en la lista, tú quieres usar Double como tipo de retorno para la suma de los elementos. Ya que los tipos primitivos como int y double son implícitamente encajados/desencajados cuando los usas en colecciones, declaras el tipo de retorno como Double, ya que es más conveniente que usar datos primitivos como double
4. En el cuerpo del método, ya que la suma de los elementos se convertirá en Double, declaras la variable “result” como Double y la inicializas a cero
5. En el for-each usas a Number como tipo para el loop. Ya que el comodín es acotado por la clase Number, tu conoces que el elemento que sea, será un elemento de Number
6. Obtienes el valor double del tipo Number usando el método doubleValue()
7. Retorna la suma de los elementos ya que haya terminado

El siguiente código contiene la implementación del método sum()

class BoundedWildCardUse {

public static Double sum(List<? extends Number> numList> {

Double result = 0.0;

for(Number num: numList) {

retult += num.doubleValue();

}

return result;

}

public static void main(String[] args) {

List<Integer> intList = new ArrayList<>();

List<Double> doubleList = new ArrayList<>();

for(int i = 0; i < 5; i++) {

intList.add(i);

doubleList.add((double)(i \* i));

}

System.out.println(sum(intList));

System.out.println(sum(doubleList));

}

}

Vamos a revisar el código anterior, paso por paso.

1. Creas dos ArrayList, una de tipo Integer y la otra de tipo Double
2. En el for insertas cinco elementos a cada una de las colecciones
3. Imprimes los contenidos de las colecciones así como su suma derivada del método sum() que funciona bien para ambas

Similarmente a usar la palabra “extends” con comodines <? extends Number>, puedes usar también la palabra “super” <? super Integer>. La expresión <? super X> significa que puedes usar cualquier supertipo (clase o interface) incluyendo el tipo X. Por ejemplo, el siguiente código compila bien. Puedes observar que <? super Integer> no solo permite los supertipos de Integer; esta expresión permite también a Integer.

NOTA: En el comodín delimitado <? extends X> X puede ser una interface o una clase (se usa “extends” tanto para clases como para interfaces). L a sustitución válida para “?” no es solo el conjunto de subclases de X, sino también X en sí misma. Esta regla también se aplica para las expresiones <? super X>.

Los comodines en la clase “Collections”

El framework collections usa extensamente. Para entender algunas características de los comodines acotados y como son usados en la práctiaca, aquí hay algunos ejemlos contenidos en la clase “Collections”.

El primer ejemplo es en el método nCopies():

static <T> List<T> nCopies(int num, T obj);

// este método retorna una lista de solo lectura de el número (num) de elementos con el

// valor obj

// El siguiente ejemplo es el método reverse()

static void reverse(List<?> list);

// Este método invierte el orden de los elementos de la lista que se pasa como argumento. Puedes pasar elementos de cualquier tipo; la clase Collections usa el comodín para esta lista

// Aquí otro ejemplo

static <T> void fill(List<? super T> list, T obj)

// Este método llena todos los elementos de la lista con el valor obj. Aquí se usa <? super T>. ¿Por qué? Aquí hay un ejemplo:

List<Object> objList = new ArrayList<Object>();

objList.add(new Object());

objList.add(new Object());

Collections.fill(objList, “Hello”);

System.out.println(objList);

el código anterior imprime

[Hello, Hello]

En el código anterior creas un List<Object> que apubta a un ArrayList<Object>. Creas dos objetos de prueba y los insertas en objList. Luego llenas a objList con la cadena “Hello” y funciona. Como puedes ver, para el método fill() puedes pasar el tipo base de List como primer argumento.

Ahora veamos un ejemplo más duro

static <T> void copy(List<? super T> dest, List<? extends T> src)

El método copy() copia todos los elementos de src List a dest List. Aquí un ejemplo del porqué dest es <? super T> y src es <? extends T>:

List<? extends Number> intList = Collections.nCopies(5, new Integer(10));

List<Object> objList = new ArrayList<Object>();

for(int i = 0; i < 5; i++) {

objList.add(new Object());

}

Collections.copy(objList, intList);

System.out.println(objList);

imprime

[10, 10, 10, 10, 10]

Vamos a ver paso a paso.

1. Primero creas una List de tipo List<? extends Number>. Inicializas la lista con cinco instancias de Integer con el valor 10. Esta es la List para el parámetro src del cual se copiarán los elementos a la otra lista, con el método copy()
2. Creas ahora una List<Object> y inicializas com ArrayList<Object>. Inicializas esta lista objList con cinco instancias de prueba, de la clase Object. Vas a usar objList como objetivo para el método copy(). ¿Por qué no usar el método nCopies() como se usó para intList? Porque la List que regresa el método nCopies() es una lista de solo lectura, por lo que si la utilizas como objetivo para el método copy() obtienes una excepción: UnsopportedOperationException
3. Usas objList como el objetivo destino y intList como el origen para el método copy(). Ahora, el tipo <T> inferido es Number. Como puedes ver, el parámetro src es de tipo <? extends T>; cuando sustituyes Integer como en <Integer extends Number>, la compilación es exitosa. Similarmente el tipo de dest es <? super T>; cuando substituyes como <Object super Number> la compilación es exitosa también.
4. Imprimes los Integer copiados en la objList

Nota: cuando usas (o quieres usar) <T>, <?>, <? extends T> o <? super T> con un tipo específico, substituye T con el tipo actual y visualiza como el tipo reemplazado se ve, Esta es la manera más fácil de entender el correcto uso de genéricos y comodines en las clases de collection.

Puntos a recordar

* Es posible definir o delcarar métodos genéricos en una interface o clase, incluso aunque estas no sean genéricas.
* Una clase genérica usada sin el argumento de tipo <T> se conoce como de tipo raw. Por supuesto, los tipos raw no son de tipo seguro. Java soporta tipos raw para las implementaciones antiguas a Java 5. El compilador generará una advertencia cuando uses tipos raw en tu código. Puedes usar la anotación @SuppressWarnings({“unchecked”}) para suprimir la advertencia asociada con los tipos raw.
* List<?> es el supertipo de cualquier tipo de List, significa que puedes pasar un List<Object>, o List<String> o cualquier otro tipo de cualquier clase.
* La implementación de genéricos es estática por naturaleza, lo que significa que el compilador de java interpreta los genéricos especificados en el código fuente y reemplasa el código genérico con tipos concretos. Esto es conocido como type erasure. Después de la compilación, el código se verá similar al que hubiera escrito el programador con tipos concretos. Esencialmente, el uso de genéricos ofrece dos ventajas: primero, intruduce una abstracción que permite escribir implementaciones genéricas; segundo, te permite escribir implementación genérica con conversión de tipo seguro.
* Hay muchas limitaciones en los genéricos debido al proceso de borrado (erasure). Los más importantes son:
  + No puedes instanciar un tipo genérico usando el operador new()
    - T mem = new T(); // mala utilización – error de compilación
  + No puedes instanciar un array de tipo genérico. Por ejemplo, asume que mem es un campo, la siguiente sentencia resulta en un error de compilación
    - T[] mem = new T[100];
  + Puedes declarar campos no estáticos de tipo T, pero no puedes declarar campos estáticos de tipo T:
    - Class X<T> { T instanceMem; static T statMem; // statMem genera error de compilación
  + No es posible tener excepciones genéricas; como un resultado, el código siguiente no compila:
    - class GenericException<T> extends Throwable{}
  + No puedes instanciar un tipo genérico con tipos primitivos
  + El significado de “extends” y “super” cambia en el contexto de los genéricos. Por ejemplo, cuando escribes <? extends X>, te refieres a todos los tipos que extienden a X y X en sí mismo

El Framework Collections (colecciones)

En la primera parte de el capítulo, discutimos el uso de genéricos en detalle. El principal uso de los genéricos es para ser capaz de escribir estructuras de datos y algoritmos reusables y de tipo seguro. La librería de Java tiene el Framework Collections, que hace uso extensivo de los genéricos y provee un juego de contenedores y algoritmos.

En esta sección, nos centraremos en como usar el Framework Collections. Para usar las colecciónes correctamente, debes de entender ciertos conceptos. Para usar colecciones como HashSet, debes de sobre-escribir los métodos hashCode() y equals() de manera correcta. Para comparar objetos y almacenarlos en colecciones, debes de aprender como usar Comparator, Comparable, etc. Ya que este Framework usa los genéricos de manera extensa, tendrás que revisar los tópicos relacionados a genéricos en el contexto del framework para obtener mejor entendimiento de los genéricos.

¿Por qué clases reusables?

Asume que quieres escribir un único programa que obtenga el número de extensión de un colega en tu compañía. Para esto, necesitas tener una lista de los nombres de tus colegas y sus números de extención. En este simple directorio raramente agregas o borras entradas, pero frecuentemente las consultas, por lo que la búsqueda debe ser muy rápida.

Para implementar este simple directorio, debes de implementar una clase que mapee (maps) el nombre (un String) y la extención (un entero o String). Debes de implementar métodos para agregar o borrar registros, buscar extenciones dando el nombre, etc. Adicionalmente, puedes implementar métodos que devuelvan todos los nombres de tus colegas, todas las extenciones, etc. Implementar una clase con todas estas características y implementarla correctamente así como checar la características, toma mucho tiempo.

Afortunadamente, Java tiene estructuras de datos como esta implementadas en el paquete java.util. Tú solo reutilizas la clase java.util.HashMap para implementar tu directorio simple. Puedes ver ejemplos de implementaciones como esta después, pero es importante que conozcas que tu puedes hacer uso de las estructuras de datos que ya están disponibles en las librerías de Java en lugar de implementarlas por ti mismo.

Componentes básicos del Framework Collections

El Framework Collections de Java tiene tres componentes principales:

1. Clases abstractas e interfaces: este framework tiene muchas clases abstractas e interfaces que proveen funcionalidad general. Aprendiendo de ellos, conocerás la funcionalidad ofrecida en términos de métodos públicos
2. Clases concretas: Estas son las instancias actuales de contenedores que uesaras en los programas
3. Algoritmos: la clase java.util.Collections implementa funcionalidad requerida de manera común por las colecciones, como ordenar (sorting), buscar (searching), etc. Estos métodos son genéricos: puedes usarlos con diferentes contenedores.

Nota: nota que colecciones (collection(s)) es un término genérico, mientras que Colection y Collections son aplicaciones específicas del paquete java.util. Collections es una clase utilitaria que contiene solo métodos estáticos. El término general collections ser refiere a contenedores como map, stack, queue, etc. Usaremos este término containers cuando nos refiramos a estas colecciones, para evitar confusión.

Clases abstractas e interfaces

La jerarquía de tipos en la librería de java.util consiste en numerosas clases abstractas e interfaces que proveen funcionalidad genérica. Las siguientes figuras y tablas muestran unas cuantos tipos importantes en esta jerarquía. Cubrimeros algunos de esos tipos en más detalle en la siguiente sección.

Clases abstractas e interfaces importantes en el Framework Collections

|  |  |
| --- | --- |
| Clase abstracta/Interface | Descripción corta |
| Iterable | La clase que implemente esta interface puede ser usada para iterar en un bucle for:each |
| Collection | Interface base común para las clases en la jerarquía de colecciones. Cuando tu quieras escribir métodos que sean verdaderamente generales, puedes pasar la interface Collection. Por ejemplo, el método max() en java.util.Collections toma una colección y retorna un objeto |
| List | Interface base para los contenedores que almacenan una sequiencia de elementos. Puedes a los elementos usando un índice, y recuperar el mismo elemento después (ya que mantienen su orden de inserción). Puedes almacenar elementos duplicados en un List |
| Set, SortedSet, NabigableSet, Queue, Deque | Interfaces para contenedores que no permiten almacenar elementos duplicados. SortedSet mantiene los elementos en un orden. NavigableSet permite buscar el set de las opciones más cercanas. Queue es la interface base para los contenedores que almacenan una secuencia de elementos para procesar. Por ejemplo, las clases que implementan Queue pueden ser LIFO o FIFO. En una Deque puedes insertar o remover elementos de ambos lados de la estructura. |
| Map, SortedMap, NavigableMap | Clases base para los contenedores que mapean llaves a valores. En un SortedMap, las llaves están en un orden. Un NavigableMap te permite buscar y retornar la ocurrencia más cercana al criterio de búsqueda. Nota que la jerarquía Map no extiende a la interface Collection. |
| Iterator, ListIterator | Puedes recorrero sobre el contenedor en dirección hacia adelante si la clase implemnta la interface Iterator. Puedes recorrer en ambos sentidos de la colección si la clase implementa la interface ListIterator |

Interfaces importantes en el paquete java.util y su relación de herencia

Estos son pocos tipos base, pero no te dejes confundir por ellos . Verás clases concretas específicas que usan algunos de estos tipos base. Cubriremos solo la interface Collection y pasaremos a cubrir clases concretas para revisar cada parte de esta jerarquía de colecciones.

La interface Collection

La interface Colleciton provee métodos como add() y remove() que son comunes para todos los contenedores. La siguiente tabla lista los métodos más importantes en esta interface. Revísalos antes de que los uses.

|  |  |
| --- | --- |
| Método | Descripción corta |
| boolean add(Element elem) | Agrega un elemento dentro del contenedor subyacente |
| void clear() | Remueve todos los elementos del contenedor |
| boolean isEmpty() | Checa si el contenedor tiene elementos o no |
| Iterator<Element> iterator() | Retorna un objeto Iterator<Element> para iterar sobre el contenedor |
| boolean remove(Object obj) | Remueve el elemento si el objeto del argumento está presente en el contenedor |
| int size() | Regresa el número de elementos del contenedor |
| Object[] toArray() | Regresa un array que contiene todos los elementos del contenedor |

Métodos como add() y remove() pueden fallar dependiendo del contenedor subyacente. Por ejemplo, si el contenedor es de solo lectura, no estarás permitido para agregar o remover elementos. Aparte de estos métodos, hay muchos métodos en la interfaceque aplican a múltiples elementos en el contenedor y se muestran algunos en la siguiente tabla.

|  |  |
| --- | --- |
| Método | Descripción corta |
| boolean addAll(Collection<? extends Element> coll) | Agrega todos los elementos de coll en el contenedor subyacente |
| boolean containsAll(Collection<?> coll) | Checa si todos los elementos de coll están presentes en el contenedor subyacente |
| boolean removeAll(Collection<?> coll) | Remueve todos los elementos del contenedor que estén presentes en coll |
| boolean retainAll(Collection<?> coll) | Retiene los elementos del contenedor, solo si están presentes en coll; los otros elementos son removidos |

Clases concretas

Numerosas interfaces y clases abstractas in la jerarquía de Collection proveen los métodos comunes que especifican clases concretas que las implementan o extienden. Las clases concretas proveen la funcionalidad actual, y necesitas aprender solo un puñado de ellas para estar preparado para el examen. La siguiente tabla muestra estas clases que debes conocer:

|  |  |
| --- | --- |
| Clase | Descripción corta |
| ArrayList | Internamente implementada como un array de tamaño variable. Esta es una de las clases más usadas. Rápida para buscar pero lenta para insertar o borrar elementos. Permite duplicados. |
| LinkedList | Internamente implementada como una lista enlazada doble. Rápida para insertar o borrar elementos, pero lenta para buscar elementos. Adicionalmente, LinkedList puede ser usada cuando necesitas una pila (LIFO) o cola (FIFO). Permite duplicados. |
| HashSet | Internamente implementada como una hash-table. Usada para almacenar un grupo de elementos, no permite elementos duplicados. Rápida para búsqueda y recuperación de elementos. No mantiene ningún órden. |
| TreeSet | Internamente implementa un árbol red-black. Como HashSet, TreeSet no permite duplicados. Sin embargo, a diferencia de HashSet, este almacena los elementos en un órden. Usa la estructura de datos de árbol para decidir donde almacenar o buscar elementos, y la posición es decidida por el orden de clasificación |
| HashMap | Internamente implementado como una hash-table. Almacena pares llave-valor. Usa hashing para encontrar un lugar para buscar o almacenar un par. Buscar o insertar es muy rápido. No almacena los elementos en ningún orden |
| TreeMap | Internamente implementado como un árbol-red-black. A diferencia de HashMap, TreeMap almacena los elementos en un órden de clasificación. Utiliza la estructura de árbol para decidir donde almacenar o buscar las llaves, y la posición es decidida por el orden de clasificación |
| PriorityQueue | Internamente implementada usando la estructura heap. Una PriorityQueue es para recuperar elementos basado en la prioridad. Independientemente del orden de inserciónm cuando remueves los elementos, el elemento con la prioridad más alta será recuperado primero |

NOTA: Hay muchas clases antiguas en java.util (ahora conocidas como legacy) que han sido sustituidas por nuevas clases. Algunas de ellas son (con nuevos tipos entre paréntesis): Enumeration(Iterator), Vector(ArrayList). Dictionary(Map) y HashTable(HashMap). En suma, Stack y Properties son clases legacy que no tienen reemplazo directo.

La interface Iterator

Vamos a discutir la interface Iterator, ya que utilizaremos Iterator para otras clases concretas. La interface Iterator es una simple interface con solo tres métodos: hasNext(), next() y remove().

* boolean hasNext(): checka si el Iterator tiene más elementos para recorrer
* E next(): mueve el iterador al siguiente elemento y lo retorna.
* void remove(): remueve el último elemento visitado del contenedor. El método next() debe de llamarse, antes de llamar a remove(), de otra manera obtendrás un IllegalStateException

Clases List (Listas)

Las listas son usadas para almacenar una secuencia de elementos. Puedes insertar un elemento al contenedor en una posición específica usando un índice, y recuperar el mismo elemento después, ya que mantiene el orden en el que se insertan los elementos. Puedes almacenar elementos duplicados. Hay dos clases concretas que necestas conocer: ArrayList y LinkedList.

ArrayList

La clase ArrayList implementa un array redimensionable. Cuando creas un array nativo (digamos new String[10]), el tamaño del array es conocido y fijo al momento de la creación. Sin embargo, ArrayList es un array dinámico, este puede crecer su tamaño, como sea requerido. Internamente, un ArrayList asigna un bloque de memoria y lo aumenta si es requerido más espacio. Entonces, acceder a los elementos del array es muy rápido en un ArrayList. Sin embargo, cuando agregas o remueves elementos, internamente, el resto de los elementos son copiados; por lo tanto agregar o borrar elementos es una operación costosa.

Aquí hay un ejemplo simple para conocer los elementos un un ArrayList y uso de un bucle for-each para recorrer la colección:

ArrayList<String> languageList = new ArrayList<>();

languageList.add(“C”);

languageList.add(“C++”);

languageList.add(“Java”);

for(String language: languageList) {

System.out.println(language);

}

imprimirá

C

C++

Java

Este bucle for-each es equivalente al siguiente código, que usa explícitamente al Iterator:

for(Iterator<String> languageIter = languageList.iterator(); languageIter.hasNext(); ) {

String language = languageIter.next();

System.out.println(language);

}

Este segmento imprimirá la misma salida del bucle anterior. Aquí revisaremos paso por paso, la descripción de como este for-each funciona:

1. Usas el método iterator para obtener el iterador del contenedor. Ya que languageList es una ArrayList de tipo <String>, necesitas crear un Iterator con String. Nombrado languageIter.
2. Antes de etrar en el loop, checas si hay elementos a visitar en el ArrayList. Llamas al método hasNext para checar esto. Si este retorna true, quiere decir que hay más elementos, todavía; si retorna falso, la iteración termino y puedes salir del loop
3. Ya que entras al cuerpo del loop, la primer cosa que haces is llamar al método next() y mover el iterator. El método next() retorna el valor iterado. Puedes capturar este valor en la variable language.
4. Imprimes por consola el valor de la variable language, y el loop continúa

Este idioma de iteración, la manera en que llamas a los métodos iterator(), hasNext() y next() es importante aprenderlos; usaremos el bucle for-each en este sentido en muchos de los ejemplos.

Nota que tu creas un ArrayList<String> y un Iterator<String> en lugar de solo usar ArrayList o Iterator (provees la información de tipo a estas clases). Las clases Collection son clases genéricas; por lo tanto necesitas especificar los parámetros de tipo para usarlas. Aquí tu estás almacenando/iterando una lista de Strings, ya que usas <String>.

Puedes remover elementos mientras recorres el contenedor usando iteradores. Vamos a crear un objeto de ArrayList<Integer> con diez elementos. Iterarás sobre estos elementos y removerás todos ellos (en lugar de usar el método removeAll()):

import java.util.\*

class TestIterator {

public static void main(String… args) {

ArrayList<Integer> nums = new ArrayList<>();

for(int i = 1; i < 10; i++) {

nums.add(i);

}

System.out.println(“Original list: ” + nums);

Iterator<Integer numsIter = nums.iterator();

while(numsIter.hasNext()) {

numsIter.remove();

}

System.out.println(“List after removing all elements: ” + nums);

}

}

Este código imprimirá:

Original list [1, 2, 3, 4, 5, 6, 7, 8, 9]

Exception in thread “main” java.lang.IllegalStateException

at java.util.AbstractList$Itr.remove(AbstractList.java:356)

at TestIterator.main(Main.java:12)

Oops!! ¿Qué pasó? El problema es que no llamaste nunca al método next() antes de llamar a remove(). Checar hasNext() in la condición del while, mover al siguiente elemento usando next() y llamar a remove() es la correcta forma para remover un elemento. Si tú no sigues esto correctamente, estarás en un problema. Similarmente, si tu llamas remove() dos veces sin intercalar el método next() obtendrás la misma excepción.

Vamos a arreglar el programa llamando al método next() antes de remove(). Solo se muestra la parte importante del código.

Iterator<Integer> numsIter = nums.iterator();

while(nums.Iter.hasNext()) {

numsIter.next();

numsIter.remove();

}

System.out.println(“List after removing all elements: ” + nums);

imprime

List after removing all elements []

NOTA: recuerda que el método next() debe ser llamado antes de llamar al método remove() en un Iterator; de otra manera, obtienes una excepción IllegalStateException. Similarmente, llamar al método remove() en sentencias subsecuentes sin llamar a next() entre estas sentencias resulta en esta misma excepción. Básicamente, cualquier modificación al contenedor subsecuente cuando un iterador está recorriendo a través del contenedor resulta en una excepción.

Interface ListIterator

Necesitas entender a ListIterator primero antes de ver la clase LinkedList. Esta interface extiende a la interface Iterator, entonces hereda los métodos hasNext(), next() y remove(). Adicionalmente, ListIterator tiene muchos otros métodos. Usando estos métodos puedes recorrer en reversa, obtener la posición previa o la siguiente y ajustar o agregar nuevos elementos al contenedor.

Esta tabla contiene los métodos de esta interface

|  |  |
| --- | --- |
| Método | Descripción corta |
| boolean hasPrevious() | Checa si el iterador tiene más elementos que recorrer en reversa |
| Element previous() | Mueve el iterador en reversa al siguiente elemento devolviéndolo |
| int nextIndex() | Retorna el índice del siguiente elemento hacia adelante |
| int previousIndex() | Retorna el índice del siguiente elemento en reversa |
| void set(Element) | Ajusta el último elemento visitado y reemplaza al elemento |
| void add(Element) | Agrega el elemento en la lista en la posición actual de la iteración |

La clase LinkedList

La clase LinkedList internamente usa una lista-enlazada-doble. Por lo tanto, insertar y eliminar elementos es muy rápido en esta clase. Sin embargo, acceder a un elemento implica atravesar los nodos uno por uno, lo que es muy lento. Cuando tú quieres agregar o remover elementos frecuentemente en una lista de elementos, es mejor usar una LinkedList. Verás un ejemplo de LinkedList junto con el uso de la interface ListIterator.

Un palindroma es una palabra o frase que se lee igual tanto hacia adelante como hacia atrás. Una cadena palindroma “abcba” se lee igual en ambas direcciones. Dada una cadena, ¿cómo puedes determinar si es un palindroma o no?

Bueno, puedes determinar cuando una cadena es un palindroma o no almacenando la cadena en un objeto String y usando el método charAt() en un loop for para comparar los caracteres (uno del principio y otro del final). Para mostrarte como usar ListIterator y LinkedList que solucionan este problema, observa el código siguiente:

import java.util.\*;

class ListIteratorTest {

public static void main(String[] args) {

String palStr = “abcba”;

List<Character> palindroma = new LinkedList<>();

for(char ch: palStr.toCharArray())

palindrome.add(ch);

System.out.println(“Input String is: ” + palStr);

ListIterator<Character> iterator = palindrome.listIterator();

ListIterator<Character> revIterator = palindrome.listIterator(palindrome.size());

boolean result = true;

while(revIterator.hasPrevious() && iterator.hasNext()) {

if(iterator.next() != revIterator.previous()) {

result = false;

break;

}

}

if(result)

System.out.println(“is palindrome”);

else

System.out.println(“is not a palindrome”);

}

}

En este programa, usas una LinkedList de caracteres para almacenar cada carácter de la cadena de entrada. ¿Por qué LinkedList<Character> en lugar de LinkedList<char>? Las clases conenedor almacenan referencias a objetos, por lo que no puedes usar tipos primitivos con ninguna de las clases collection. Ya que tienes clases envoltorio que auto-encajan a su respectivo tipo primitivo, esto no es un gran problema.

También nota como asignas una LinkedList<Character> a una List<Character>. Ya que la clase LinkedList implementa la interface List, puedes hacer esta asignación. Usando interfaces de Collection, en lugar de clases concretas para almacenar referencias, es una buena práctica de programación, ya que si quieres cambiar de una LinkedList a una ArrayList en el futuro, el cambio puede ser fácil usando la interface List.

El método toCharArray() en el String retorna un array de char[] y lo usas en el for-each para recorrer cada carácter en el array. Mientras el loop se ejecuta, pones cada carácter en la LinkedList. Ahora ¿Cómo vas a recorrer la LinkedList en ambas direcciones al mismo tiempo?

Puedes usar los métodos hasNext() y next() para recorrer hacia adelante la lista; y los métodos hasPrevious() y previous() en el ListIterator para moverte en dirección reversa. ¿Cómo puedes obtener el ListIterator de la LinkedList? Usando el método listIterator() y listIterator(index). En el primer caso, obtienes el Iterator referenciado al principio del contenedor; en el otro caso, pasas el tamaño del String como un argumento al método listIterator para obtener el iterador en reversa. Ya que obtuviste ambos iteradores es muy sencillo ir comparando ambos de hacia adelante y hacia atrás respectivamente.

La interface Set

Set, como hemos estudiado en nuestras clases de matemáticas en la Universidad, contiene elementos no duplicados. A diferencia de List, un Set no recuerda dónde inseraste los elementos, ni el orden de la inserción.

Hay dos clases concretas importantes que implementan la interface Set: HashSet y TreeSet. Un HashSet is para rápida insersión y recuperación de elementos; esta no mantiene ninguna clasificación de los elementos que mantiene. Un TreeSet almacena los elementos en un orden clasificado e implementa la interface SortedSet.

La clase HashSet

Dando una sentencia, ¿Cómo puedes remover palabras repetidas en esta sentencia? Set no permite duplicados, y HashSet puede ser usado para una inserción rápida y búsquedas. Entonces puedes usar HashSet para resolver este problema. Observa el siguiente código.

import java.util.\*;

class RemoveDuplicates {

public static void main(String… args) {

String tongueTwister = “I feel, a feel, a funny feel, a funny feel I feel ” +

if you feel the feel I feel, I feel the feel you feel”;

Set<String> words = new HashSet<>();

// parte la sentencia en palabras y intenta colocarlas en el set

for(String word: tongueTwister.split(“\\W+”))

words.add(word);

System.out.println(“The tongueTwister is: ” + tongueTwister);

System.out.println(“The words used are: ” + words);

}

}

En este ejemplo, el trabalenguas tiene solo dos separadores de palabras, la coma y el espacio en blanco. Puedes separar la cadena usando estos separadores usando el método split(). El método split() toma expresiones regulares como argumento (las expresiones regulares serán revisadas en el capítulo 7). La expresión regular [\\W](file:///\\W)+ significa que se dividirán en los límites de las palabras. Entonces, el String se separa en palabras, ignorando la puntuación como comas.

Intentas insertar cada palabra en el set. Si la palabra ya existe, el método add() falla y retorna falso (aunque en este programa no se está guardando este resultado). Cuando ya insetaste todos los elementos en el HashSet, puedes imprimirlos uno a uno y encontrarás que el trabalenguas de 25 palabras, solo usa 7 palabras.

La clase TreeSet

Dada una sentencia, ¿Cómo puedes ordenar las letras usadas en esta sentencia en orden alfabético? Un TreeSet pone los valores en un orden clasificado, entonces puedes usar un TreeSet para resolver este problema.

class TreeSetTest {

public static void main(String… args) {

String pangram = “the quick Brown fox jumps over the lazy dog”;

Set<Character> aToZee = new TreeSet<>();

for(char gram: pangram.toCharArray());

aToZee.add(gram);

System.out.println(“The pangram is: ” + pangram);

System.out.println(“Sorted pangram characters are: ” + aToZee);

}

}

Un pangram es una sentencia que usa todas las letras del alfabeto, por lo menos una vez. Quieres almacenar los caracteres de un pangram en el set. Ya que necesitas usar referencia de tipos para los contenedores, debes de crear un TreeSet de Character.

Ahora, ¿Cómo obtienes los caracteres del String? Llamando al método toCharArray() retorna un arreglo de caracteres char[]. Entonces, usas este método para recorrer sobre la cadena y obtener todos los caracteres. Como agregas los caracteres en el TreeSet, los caracteres son almácenados de manera ordenada.

Nota que en la salida hay una coma que comienza. El pangram tiene varios espacios en blanco, uno de ellos es almacenado en el set y por lo tanto es impreso.

La interface Map

Un Map almacena pares llave-valor (key-value). La interface Map no extiende a la interface Collection. Sin embargo, hay métodos en la interface Map que puedes usar para obtener clases en la Collection para trabajar con este problema. También, los nombres de los métodos en Map son bastante similares a los métodos en Collection, por lo que es fácil entender y usar Map. Hay dos clases concretas importantes de Map que vamos a cubrir: HashMap y TreeMap.

Un HashMap usa una estructura de datos tabla de hash de manera interna. En un HashMap, buscar o encontrar elementos es rápido. Sin embargo, HashMap nunca recuerda el órden en el que se insertaron los elementos y no mantiene los elementos en ningún orden determinado.

Un TreeMap usa una estructura de datos red-blak tree de manera interna. A diferencia de HashMap, TreeMap mantiene los elementos en un orden determinado (por ejemplo sus llaves). Pero la búsqueda y la inserción de elementos es un poco más lenta que en un HashMap.

La clase HashMap

Asume que estás implementando un simple checador de ortografía. Dada una String de entrada, el checador de ortografía checa las palabras que son usualmente mal escritas; si estas se emparejan, imprime la forma correcta. Entonces, el checador de ortografía debe mantener una lista de estas palabras. ¿Cómo puedes implementar esto?

Dada una llave, puedes buscar un valor usando un Map. Ahora, qué map usar HashMap o TreeMap. No es necesario mantener las palabras en un orden específico, y la búsqueda por las palabras debe ser rápida. Entonces, usa HashMap para resolver este problema.

Este programa simple te muestra como implementar un checador de ortografía.

public class SpellChecker {

public static void main(String[] args) {

Map<String, String> mWords = new HashMap<>();

mWords.put(“calender”, “calendar”);

mWords.put(“tomatos”, “tomatoes”);

mWords.put(“existance”, “existence”);

mWords.put(“aquaintance”, “acquaintance”);

String sentence = “Buy a calender for the year 2013”;

System.out.println(“La sentencia dada es: ” + sentence);

for(String word: sentence.split(“\\W+”)) {

if(mWords.containsKey(word)) {

System.out.print(“La palabra correcta para ” + word);

System.out.println(“ es: ” + mWords.get(word));

}

}

}

}

Primero necesitas crear una tabla de palabras incorrectas y sus correcciones. Ya que ambas, llave y valor son Strings, puedes crear un HashMap<String, String>. Insertas palabras mal escritas como llave y su palabra corregida como valor. Usas el método put() en lugar de add() para insertar el par llave-valor en el Map.

Usas una simple aproximación de separar las palabras en la sentencia con el método split(). Por cada palabra, checas si la palabra coincide exactamente con una palabra mal escrita, buscando en las llaves del Map; si sí, imprime el valor que coincide con la llave. Usas el método containsKey() para checar si la llave existe en el Map; este método retorna un valor booleano. Usas el método get() para retornar el valor del Map dando la llave respectiva como argumento.

Ahora vamos a ver las llaves en el HashMap mWords. Puedes obtener todas las llaves en el HashMap usando el método keySet(). Ya que tienes un HashMap<String, String>, el Set que retorna es de tipo Set<String>.

Set<String> keys = mWord.keySet();

System.out.println(“Misspelt words in spellchecker are: ” + keys);

Sobre-escribir el método hashCode()

Sobre-escribir los métodos equals() y hashCode correctamente es importante para usar las clases con contenedores, particularmente HashMap y HashSet. En el siguiente código se muestra una simple clase Circle para que puedas entender qué puede salir mal:

class Circle {

private int xPos, yPos, radius;

public Circle(int x, int y, int r) {

xPos = x;

yPos = y;

radius = r;

}

public boolean equals(Object arg) {

if(arg == null) return false;

if(this == arg) return true;

if(arg instanceof Circle) {

Circle that = (Circle) arg;

if((this.xPos == that.xPos) && (this.yPos == that.yPos) &&

(this.radius == that.radius )) {

return true;

}

}

return false;

}

}

class TestCircle {

public static void main(String[] args) {

Set<Circle> circleList = new HashSet<>();

circleList.add(new Circle(10, 30, 40));

System.out.println(circleList.contains(new Circle(10, 30, 40)));

}

}

Imprimirá false. La clase Circle sobre-escribe el método equals(), pero no sobre-escribe el método hashCode. Cuando usas objetos Circle en contenedores estándar, esto se vuelve un problema. Para un vistaso rápido, los contenedores comparan el hashcode de los objetos. Si el método hashCode() no es sobre-escrito entonces el contenedor no encontrará el objeto, incluso si se pasa un objeto con los mismos valores. Por lo que tendrás que sobre-escribir el método hashCode().

NOTA: Los métodos hashCode() y equals() necesitan ser consistentes para una clase. Para propósitos prácticos, asegura que sigues esta única regla: el método hashCode() debe retornar el mismo valor hash si dos objetos retornan true con el método equals().

Cuando implementas el método hashCode(), puedes usar los valores de los miembros de instancia de la clase para crear un valor hash. Aquí hay una simple implementación del método hashCode para la clase Circle:

public int hashCode() {

// usa los operadores de manipulación de bits como ^ para generar lo mejor posible

// códigos hash únicos

// aquí usamos los números mágicos 7, 11 y 53, pero puedes usar cualquier número

// preferentemente primos

return (7 \* xPos) ^ (11 \* yPos) ^ (53 \* radius);

}

Ahora si vuelves a correr el método main() imprimirá true. En esta implementación del método hashCode, multiplicas los valores por un número primo así como una operación de bit a bit. Puedes escribir código más complejo para el método si quieres tener mejor función de hash, pero esta implementación es suficiente para propósitos prácticos.

Puedes usar operadores de bit a bit para valores enteros. ¿Qué pasa con otros tipos como float o valores de referencia? Para darte un ejemplo, aquí esta la implementación de java.awt.Point2D, que tiene valores de punto flotante para “x” y “y”. Los métodos getX() y getY() retornan estos valores respectivamente:

public int hashCode() {

long bits = java.lang.Double.doubleToLongBits(getX());

bits ^= java.lang.Double.doubleToLongBits(getY()) \* 31;

return (((int) bits) ^ ((int) bits >> 32)));

}

Este método usael método doubleToLongBits, que toma un valor double y retorna un valor long. Para los valores de punto flotante x,y tienes los valores long en bits y usas la manipulación de bits para obtener el hashCode().

Ahora cómo implementar el método hashCode si la clase tiene referencias a objetos en sus miembros. Por ejemplo, cosidera el uso de una instancia (center) de la clase Point como miembro de la clase Circle:

public int hashCode() {

// en este caso se llama al método hashCode de la referencia

return center.hashCode() ^ radius;

}

NOTA: Si estás usando un objeto en un contenedor como HashSet o HashMap, asegúrate de sobre-escribir el método hashCode() y equals() correctamente.

La interface NavigableMap

La interface NavigableMap extiende la interface SortedMap. En la jerarquía de Collections, la clase ThreeMap es la más usada que implementa esta interface. Como su nombre lo indica, con NavigableMap, puedes navegar el Map de manera fácil. Puedes obtener el valor más cercano, que coincida con una llave, todos los valores menores de la llave dada, todos los valores más grandes de la llave dada, etc. Vamos a ver un ejemplo: Lennon, McCartney, Harrison y Starr están tomando un examen en línea. En este examen, el score máximo es 100, con un passing score de 40. Si tú quieres encontrar detalles como quién pasó el examen y ordenar los scores de los exámenes, NavigableMap es muy conveniente para esta tarea. Observa el siguiente código:

import java.util.\*;

public class NabigableMapTest {

NavigableMap<Integer, String> examScores = new TreeMap<>();

examScores.put(90, “Sophia”);

examScores.put(20, “Isabella”);

examScores.put(10, “Ema”);

examScores.put(50, “Olivea”);

System.out.println(“Los datos en el map son: ” + examScores);

System.out.pritnln(“En orden descendiente son: ” + examScores.descendingMap());

System.out.println(“Pasó el examen: ” + examScores.tailMap(40));

System.out.println(“La marca más baja fue: ” + examScores.firstEntry());

}

En este programa tienes un NavigableMap<Integer, String> dodne actualmente almacenas los resultados de los exámenes. Por default, el TreeMap almacena los datos en orden ascendente. Si quieres los datos en orden descendiente, es fácil, solo usas el método descendingMap() (o descendingKeySet() si solo te interesan las llaves).

Dado el passing score de 40, pues querer obtener el mapa con los datos de quién falló el examen. Para esto, puedes usar el método headMap() con el valor de llave 40 (ya que los datos están en orden ascendiente, tú quieres la cabeza “head” desde la posición dada). Similarmente es el método tailMap, con el que obtienes la cola desde un un valor de llave dado.

Si quieres los valores inmediatos bajo el passing score, puedes usar los métodos higerEntry() y lowerEntry(), respectivamente. Los métodos firstEntry() y lastEntry() muestran las entradas más bajas y más altas respectivamente.

La interface Queue (cola)

Una cola sigue el mecanismo FIFO: el primero que entra será el primero que sale. Para obtener el comportamiento de una cola puedes crear un objeto de tipo LinkedList y referirte a el a través de la interface Queue. Cuando llamas los métodos de la referencia Queue, el objeto se comporta como una cola. El siguiente código muestra un ejemplo y esto será más claro: Lennon, McCartney, Harrison y Starr están tomando un examen en línea. Vamos a ver cómo puedes recordar la secuencia en cómo cada uno ingresó para realizar el examen.

import java.util.\*;

class QueueTest {

public static void main(String… args) {

Queue<String> loginSequence = new LinkedList<>();

loginSequence.add(“Harrison”);

loginSequence.add(“McCartney”);

loginSequence.add(“Starr”);

loginSequence.add(“Lennon”);

System.out.println(“The login sequence is: ” + loginSequence);

while(¡loginSequence.isEmpty())

System.out.println(“Removing: ” + loginSequence.remove());

}

}

En este ejemplo, tú creaste una Queue<String> que apunta a una LinkedList<String>. Cuando agregas cuatro elementos (nombrados en la secuencia que ingresaron al examen) a la cola. Despues de que los elementos fueron agregados, imprimes el Queue completo (implícitamente llama al método toString). Después llamas al método remove() para remover uno a uno los elementos de la cola. Obtienes la misma secuencia de salida en el órden que ingresaste los elementos.

La interface Deque

Una Deque (cola doble) es una estructura de datos que te permite insertar y remover elementos de ambos lados de la cola. Esta interface fue introducida en la versión 6 de Java en el paquete java.util.collection. La interface Deque extiende la funcionalidad de la interface Queue, que revisamos anteriormente. Por lo tanto, todos los métodos proveídos por la interface Queue están también disponibles en la interface Deque. Vamos a examinar los métodos comúnmente usados de la interface Deque, en la siguiente tabla.

|  |  |
| --- | --- |
| Método | Descripción corta |
| void addFirst(Element) | Agrega un elemento al frente del Deque |
| void addLast(Element) | Agrega un elemento al final del Deque |
| Element removeFirst() | Remueve un elemento del frente del Deque y lo retorna |
| Element removeLast() | Remueve un elemento del final del Deque y lo retorna |
| Element getFirst() | Retorna el primer elemento del Deque, pero no lo remueve |
| Element getLast() | Retorna el último elemento del Deque, pero no lo remueve |

Todos los métodos de la tabla anterior alcanzan excepciones apropiadas si fallan. Este es otro set de métodos, listados en la siguiente tabla que logran la misma funcionalidad. Sin embargo, estos no lanzan una exepción si fallan; en su lugar, retornan un valor especial. Por ejemplo, el método getFirst() retorna el primer elemento de la Deque pero no lo remueve. Si la Deque está vacía, lanza una exepción NoSuchElementException. Al mismo tiempo, el método peekFirst() tiene la misma funcionalidad. Sin embargo, este retorna null si la Deque está vacía. Cuando la Deque es creada con una capacidad predefinida, los métodos listados en la siguiente tabla son preferidos sobre los métodos de la tabla anterior.

|  |  |
| --- | --- |
| Método | Descripción corta |
| boolean offerFirst(Element) | Agrega el elemento al frente de la Deque si no viola la restricción de capacidad |
| boolean offerLast(Element) | Agrega el elemento al final de la Deque si no viola la restricción de capacidad |
| Element pollFirst() | Remueve un elemento del frente de la Deque y lo retorna, retorna null si la Deque está vacía |
| Element pollLast() | Remueve un elemento del final de la Deque y lo retorna, retorna null si la Deque está vacía |
| Element peekFirst() | Retorna el primer elemento de la Deque pero no lo remueve, retorna null si la Deque está vacía |
| Element peekLast() | Retorna el último elemento de la Deque pero no lo remueve, retorna null si la Deque está vacía |

NOTA: Ahora observas en estos últimos métodos que retornan null si fallan. ¿Qué pasa si almacenas un valor null como elemento en el Deque? Bueno, no es recomendable que almacenes un null como argumento, ya que hay métodos en la interface Deque que retornan null, y puede ser difícil distinguir cuando el método es exitoso o falla.

Hay tres implementaciones concretas de la interface Deque: LinquedList, ArrayDeque y LinkedBlockingDeque. Vamos a usar ArrayDeque para entender las características de la interface Deque.

Es evidente de la lista de métodos soportados por Deque que es posible realizar el comportamiento estándar de una Queue, un Stack y Deque. Vamos a implementar una cola especial (digamos, para pagar una factura de servicios) cuando se puede agregar un cliente solo al final de la cola y puede remover del frente de la cola cuando el cliente paga o del final de la cola cuando el cliente se desespera y se va. El siguiente código muestra cómo hacer esto.

import java.util.\*;

class SplQueue {

private Deque<String> splQ = new ArrayDeque<>();

void addInQueue(String cliente) {

splQ.addLast(customer);

}

void removeFront() {

splQremoveFirst();

}

void removeBack() {

splQ.removeLast()

}

void printQueue() {

System.out.println(“Special queue contains: ” + splQ);

}

}

class SplQueueTest {

public static void main(String[] args) {

SplQueue splQ = new SplQueue();

splQ.addInQueue(“Harrison”);

splQ.addInQueue(“McCartney”);

splQ.addInQueue(“Starr”);

splQ.addInQueue(“Lennon”);

splQ.printQueue();

splQ.removeFront();

splQ.removeBack();

splQ.printQueue();

}

}

Primero defines la clase SplQueue que define un contenedor splQ de tipo ArrayDeque con cuatro operaciones básicas addInQueue(), removeBack(), removeFront() y el método printQueue(). Segundo, utilizas los métodos de la interface Deque para implementar cada uno de estos métodos.

Intefaces Comparable y Comparator

Como sus nombres lo sugieren, estas interfaces son usadas para comparar objetos similares (por ejemplo, para ejecutar búsquedas u ordenamientos de acuerdo a un criterio). Asume que tienes un contenedor que contiene una lista de objetos Person. Ahora ¿Cómo puedes comparar dos objetos Person? Hay muchas atributos comparables, como el número de seguro social, el nombre, licencia de conducir, etc. Dos objetos pueden compararse bun por el número de seguro social tanto como por el nombre; esto depende del contexto. Por lo tanto, el criterio para comparar objetos Person no puede ser predefinido; un desarrollador tiene que definir este criterio. Java define esta s interfaces para lograr esto.

La clase Comparable tiene solo un método compareTo() , que está declarado como sigue:

int compareTo(Element that)

Ya que tú estás implementando este método en una clase, tu tienes la referencia this disponible. Puedes comparar el objeto actual con el elemento que se pasa como parámetro y retornar un valor int. ¿Qué es el valor entero? Bueno, estas son las reglas para retornar el valor entero:

return 1 if current object > passed object

return 0 if current object == passed object

return -1 if currend object < passed object

Ahora una pregunta importante: ¿Qué hace que un objeto sea mayor, menor o igual a otro? Esto lo tienes que decidir tú. Pero el significado de la comparación debe ser natural; en otras palabras, la comparación debe significar un órden natural. Por ejemplo, viste cómo los Integers son comparados con cada otro, basado en el órden numérico, lo cual es el órden natural para números enteros. Similarmente, comparaste Strings usando la comparación lexicográfica, que es el órden natural de los Strings. Para clases definidas por el usuario, necesitas encontrar el órden natural en el que puedes comparar los objetos. Por ejemplo, para la clase Student, studentId puede ser el órden natural de comparar objetos Student.

Vamos a ver este ejemplo en el siguiente código:

import java.util.\*;

class Student implements Comparable<Student> {

String id;

String name;

Double cgpa;

public Student(String id, String name, double cgpa) {

this.id = id;

this.name = name;

this.cgpa = cgpa;

}

public String toString() {

return “ \n ” + id + “ \t ” + name + “ \t ” + cgpa;

}

public int compareTo(Student that) {

return this.id.compareTo(that.id);

}

}

class ComparableTest {

public static void main(String[] args) {

Student[] students = {

new Student(“cs011”, “Lennon”, 3.1),

new Student(“cs021”, “McCartney”, 3.4),

new Student(“cs012”, “Harrison”, 2.7),

new Student(“cs022”, “Starr”, 3.1)

};

System.out.println(“Before sorting by student ID”);

System.out.pritnln(“Student-ID \t Name \t CGPA (for 4.0)”);

System.out.println(“Arrays.toString(students)”);

Arrays.sort(students);

System.out.println(“After sorting by student ID”);

System.out.pritnln(“Student-ID \t Name \t CGPA (for 4.0)”);

System.out.println(“Arrays.toString(students)”);

}

}

Implementas la interface Comparable<Student>. Cuando llamas al método sort(), el llama al método compareTo() para comparar el objeto Student por su id. Ya que los id’s son únicos, esta es una forma natural de comparar y ordenar, por lo que funciona bien.

Ahora, puedes necesitar organizar estudiantes basado en su puntos promedio cgpa que tienen. También puedes necisitar comparar los estudiantes por su nombre. Si necesitas implementar dos o más alternativas para comparar dos objetos similares, entonces tienes que implementar la clase Comparator. El siguiente código es una implementación de esta clase utilizando la clase Student vista anteriormente.

import java.util.\*;

class CGPAComparator implements Comparator<Student> {

public int compare(Student s1, Student s2) {

return(s1.cgpa.compareTo(s2.cgpa));

}

}

class ComparatorTest {

public static void main(String… args) {

Student[] students = {

new Student(“cs011”, “Lennon ”, 3.1),

new Student(“cs021”, “McCartney”, 3.4),

new Student(“cs012”, “Harrison”, 2.7),

new Student(“cs022”, Starr, 3.7)

};

System.out.println(“Antes de ordenar por cgpa: ”);

System.out.println(“Student-id \t Name \t CGPA (for 4.0)”);

System.out.println(Arrays.toString(students));

Arrays.sort(students, new CGPAComparator());

System.out.println(“Después de ordenar por cgpa: ”);

System.out.println(“Student-id \t Name \t CGPA (for 4.0)”);

System.out.println(Arrays.toString(students));

}

}

Sí, el programa imprime los datos de los estudiantes ordenados por cgpa. No hiciste ningún cambio a la clase Student; sigue implementando la interface Comparable<String> y define su propio método compareTo(), pero no usas el método compareTo() en el programa. Creas una clase separada llamada CGPAComparator y implementas la interface Comparator<Student>. Defines el método compare() que toma dos estudiantes como argumentos.

Diferencias entre implementar las interfaces Comparable y Comparator

|  |  |
| --- | --- |
| Comparable | Comparator |
| Usada cuando los objetos necesitan ser comparados en su orden natural | Usado cuando los objetos necesitan ser comparados en un órden definido por el usuario (otro que el natural) |
| No puedes crear una clase separada para implementar esta interface | Necesitas crear una clase separada para implementar esta interface |
| Para el tipo de clase dada, tienes solo esa clase (sola) para implementar la interface | Puedes tener muchas clases separadas implementando esta interface, y cada clase puede definir qué camino usar para comparar los objetos |
| El método en la interface está declarado como int compareTo(ClassType type) | El método en la interface está declarado como int compare(ClassType tipe1, ClassType type2) |

Muchas clases tienen un orden natural para comparar objetos, entonces implementa la interface Comparable en estos casos. Si quieres compara los objetos en otro orden que el natural o no tiene un orden natural, entonces crea clases separadas implementando la interface Comparator.

Algoritmos (Clase Collections)

Has visto dos componentes importantes del Framework Collections: clases abstractas/interfaces y las implementaciones en sus clases concretas. El Framework Collections también tiene una clase utilitaria llamada Collections. Esta provee algoritmos que son útiles para manipular las estructuras de datos provistas por este Framework. Veremos métodos importantes como sort(), binarySearch(), reverse(), shuffle(), etc., en esta pequeña sección. Observa la siguiente tabla.

Algoritmos importantes (Métodos estáticos en la clase Collections)

|  |  |
| --- | --- |
| Método | Descripción corta |
| int binarySearch(List<? extends Comparable<? super T>> list, T key) | Busca la llave en la List. Si la encuentra, retorna un valor mayor o igual a cero; si no la encuentra retorna un valor negativo. Hay una versión sobrecargada que toma una clase Comparator para comparar los elementos. |
| void copy(List<? super T> list, List<? extends T> src) | Copia todos los elementos de src a dest |
| void fill(List<? super T> list, T obj) | Llena la lista entera con el valor obj |
| T max(Collection<? extends T> coll) | Retorna el elemento máximo en una lista. Tiene un método sobrecargado que recibe un Coparator |
| T min(Collection<? extends T> coll) | Retorna el elemento mínimo de coll |
| boolean replaceAll(List<T> list, T oldVal, T newVal) | Reemplaza todas las ocurrencias de oldVal por newVal en list |
| void reverse(List<?> list) | invierte el órden de todos los elementos de list |
| void rotate(List<?> list, int distance) | Rota la lista dada por el valor distancia |
| void shuffle(List<?> list) | Revuelve los elementos de list |
| void sort(List<T> list) | Ordena los elementos por su orden natural. Existe una versión sobrecargada que recibe un Comparator como argumento |
| void swap(List<?> list, int i, int j) | Intercambia los elementos de la lista en las posiciones i y j |

Asume que estás creando un playlist de tus canciones favoritas de Michael Jackson. Hay muchas cosas que puedes hacer con la playlist: puedes ordenar, revolver, buscar, invertir o repetir las canciones. Vamos a hacer todo eso en el programa PlayList. Observa el siguiente ejemplo.

import java.util.\*;

class PlayList {

public static void main(String[] args) {

List<String> playList = new LinkedList<>();

playList.add(“Rock with you”);

playList.add(“Billie Jean”);

playList.add(“Man in the mirror”);

playList.add(“Black or White”);

System.out.println(“La lista original”);

System.out.println(playList);

System.out.println(“\nLa lista en reversa”);

Collections.reverse(playList);

System.out.println(playList);

System.out.println(“\nShuffle la lista”);

Collections.shuffle(playList);

System.out.println(playList);

System.out.println(“\nOrdenar por nombre”);

Collections.sort(playList);

System.out.println(playList);

System.out.println(“Está Black or White en la lista?”);

String song = “Black or White”;

int index = Collections.binarySearch(playList, song);

if(index >= 0)

System.out.printf(“Sí, es la canción numero %d\n”, (index + 1));

else

System.out.printf(“No está esa canción”);

System.out.println(“\nVamos a adelantar dos canciones”);

Collections.rotate(playList, 2);

System.out.println(palyList);

}

}

Solo te tomó unos minutos observar la salida del programa y observar qué pasa. En el método binarySearch(), existen dos argumentos, la List y la llave que se está buscando. Si la llave es encontrada, el entero retornado será igual o mayor que 0 y este valor es el índice de la llave en la lista. Si es negativo significa que la llave no se encontró.

La clase Arrays

Similar a la clase Collections, Arrays también es una clase utilitaria. Los métodos en la clase Collections son muy similares a los métodos en Arrays. La clase Collections es para clases contenedoras y Arrays para arrays nativos.

Métodos en la clase Arrays

El siguiente código es un ejemplo trivial para mostrar por qué la clase Arrays es muy útil trabajando con arrays nativos.

public class PrintArray {

public static void main(String… args) {

int[] intArray = {1,2,3,4,5};

}

System.out.println(intArray);

}

// imprime

[I@2je34a6

¿Qué estuvo mal?: El método println() implícitamente llama al método toString(). Los arrays nativos no son clases, sin embargo ellas heredan de la clase Object, pero no es posible sobre-escribir sus métodos. Entonces ¿Cómo puedes imprimir el contenido de un array?

Afortunadamente exste la clase Arrays que tiene métodos como toString. El siguiente código es la manera correcta de imprimir los contenidos de un array nativo.

import java.util.\*

class PrintArray {

public static void main(String… args) {

int[] intArray = {1,2,3,4,5};

System.out.println(Arrays.toString(intArray));

}

}

Ahora veremos en la siguiente tabla, los métodos más importantes de la clase Arrays; se parecen mucho a los métodos de la clase Collections.

|  |  |
| --- | --- |
| Método | Descripción |
| List<T> asList(T… a) | Crea una lista fija del array dado |
| int binarySearch(Object[] objArray,Object key) | Busca a key en objArray, si lo encuentra, retorna el índice en el array, si no, devuelve un valor negativo. Está disponible para tipos primitivos, aunque hay una versión sobrecargada que toma un Comparator como argumento |
| boolean equals(Object[] arr1, Object[] arr2) | Checa si el contenido de los array son iguales. |
| void fill(Object[] objArray, Object val) | Llena el objArray con el valor val |
| void sort(Object[] objArray) | ordena los elementos de objArray basado en el órden natural usando compareTo(). Hay una versión que recibe un Comparator como argumento |
| String toString(Object[] a) | Retorna la representación String de a |

Todos los métodos de esta lista están disponibles para tipos primitivos.

El método Arrays.sort(Object[] objArray) llama al método compareTo() para comparar los elementos. Entonces los elementos del array deben de implementar la interface Comparable. El método sort está sobrecargado para recibir tipos primitivos. Para los elementos, el ordenamiento será en orden ascendente numérico. Veamos el siguiente código que muestra el método sort() con int y con String.

class CollectionsTest {

public static void main(String… args) {

String[] strArr = {“21”, “1”, “111”, “12”, “123”};

Arrays.sort(strArr);

System.out.println(Arrays.toString(strArr));

int[] intArr = {21,1,111,12,123};

Arrays.sort(intArr);

System.out.println(Arrays.toString(intArr));

}

}

El contenido de los arrays se ven similares pero la salida del programa es diferente. ¿Por qué? Esto muestra la diferencia en el ordenamiento para los Strings y tipos primitivos. El método compareTo() en la clase String, hace una comparación lexicográfica. El contenido del String es comparado carácter por carácter. Esto tiene sentido, por ejeplo, si tu tienes un String “john”, “johannes”, “johann”, “johnny”, etc. Sin embargo, para números necesitas comparar valores. Por esta razón, después de “1” sigue “111” y después “12”…

NOTA: Ten cuidado cómo los valores son comparados cuando usas el método sort(). Por ejemplo, la comparación numérica está bien para enteros, mientras la comparación lexicográfica está bien para los Strings.

En el código anterior intentaste ordenar. Ahora intentarás buscar valores en el array. Veamos el siguiente código.

import java.util.Arrays;

public class BinarySearchTest {

public static void main(String… args) {

String[] strArr = {“21”, “22”, “11”, “12”, “13”};

System.out.println(“Buscar el dato 22”);

int index = Arrays.binarySearch(strArr, “22”);

System.out.println(“El índice del 22 es ” + index);

}

}

El método anterior imprime -6 ¿Qué está mal? el método binarySearch() toma dos argumentos: el primero es el array en el que se va a buscar el segundo argumento. Si es exitoso, retorna el índice de elemento buscado en el array. Si la llave no es encontrada, retorna un valor negativo. Ahora, el método espera que sea llamado en un array ordenado de antemano. Aquí olvidaste llamar al método sort para realizar la búsqueda, por lo tanto el método falla. Vamos a ver el código corregido.

import java.util.\*;

class BynarySearchTest {

public static void main(String… args) {

String[] strArr = {“21”, “22”, “11”, “12”, “13”};

System.out.println(“El array es: ” + Arrays.toString(strArr));

Arrays.sort(strArr);

System.out.println(“Array después de ordenado: ” + Arrays.toString(strArr) );

int index = Arrays.binarySearch(strArr, “22”);

System.out.println(“El valor del índice: ” + index);

}

}

CUIDADO: Asegúrate de llamar al método binarySearch() en un array ordenado de antemano.

Array como List

Asume que tienes temperaturas registradas en tu ciudad por una semana. ¿Cómo puedes escribir un programa que imprima la temperatura máxima y la mínima?

Para almacenar temperaturas puedes usar un array. Sin embargo, la clase Arrays no tiene los métodos max() y min(). Un camino para evitar escribir tu propio método es convertir el array en un List usando el método asList, y usar los métodos max() y min() de la clase Collections. Recuerda que el array puede ser un tipo de referencia. NO puedes usar el método asList en un array de tipo primitivo.

import java.util.\*;

public class ArrayAsList {

public static void main(String… args) {

Double[] temperaturas = {31.1, 30.0,32.5, 34.9, 33.7, 27.8};

List<Double> tempList = Arrays.asList(temperaturas);

System.out.println(“La temp. máxima fue: ” + Collections.max(tempList));

System.out.println(“la mínima: ” + Collections.min(tempLis));

}

}

Sí, funcionó. Pero tu hiciste un error lógico simple. En este programa tienes las temperaturas dadas de seis días y no siete. Puedes agregar más valores al array, pero tendrías que intentar agregar el dato a la List<Double> que ya tienes. Este es el código.

List<Double> tempList = Arrays.asList(teperaturas);

tempList.add(32.3);

Se obtiene el siguiente error

Exception in thread “main” java.lang.UnsupportedOperationException

at java.util.AbstractList.add(AbstractList.java:131)

at java.util.AbstractList.add(AbstractList.java:91)

at ArrayAsList.main(ArrayAsList.java:13)

No puedes agregar elementos a la lista retornada por el método asList(). La solución es crear una nueva List, por ejemplo ArrayList por ti mismo y agregar los elementos a esta lista. Observa el código

import java.util.\*;

class ArrayAsList {

public static void main(String… args) {

Double[] temp = {31.1,30.0,32.5,34.9,33.7,27.8};

System.out.println(“El array original es: ” + Arrays.toString(temp));

List<Double> tempList = new ArrayList<>(Arrays.asList(temp));

tempList.add(32.3);

System.out.println(“La lista nueva es: ” + tempList);

}

}

Ahora ya funciona!!!!

Puntos a recordar

* La diferencia entre una ArrayList y ArrayDeque es que puedes agregar elementos donde sea en el array list usando un índice; sin embargo, puedes agregar un elemento solo al principio o al final de un array deque. Esto hace que la inserción en un ArrayDeque sea más eficiente que en ArrayList; sin embargo, la navegación en ArrayDeque se vuelve más complicada que en ArrayList
* Esta es otra cosa que necesitas recordar acerca del uso de la List retornada por el método Arrays.asList(). Aunque no puedes agregar elementos a esta lista, puedes modificar la List, por lo tanto las modificaciones que hagas en la List son reflejadas en el array original. Por ejemplo, di modificas la temperatura del primer día en la List, el array original es modificado como se ve en el siguiente código:

import java.util.\*;

class ArrayAsList3 {

public static void main(String[] args) {

Double[] temp = {31.1,30.0,32.5,34.9,33.7,27.8};

System.out.println(“El array original es: ” + Arrays.toString(temp));

List<Double> tempList = Arrays.asList(temp);

tempList.set(0, 35.2);

System.out.println(“El array modificado es: ” + Arrays.toString(temp));

}

}

La clase Arrays provee solo funcionalidad limitada y a menudo utilizaras los métodos en la clase Collections. Para eso utiliza el método Arrays.asList()

Recuerda que no puedes agregar elementos a la List retornada por el método Arrays.asList(), pero puedes hacer cambios en la misma y se verán reflejados en el array original.

Pregunta 2 de generics: Predice la salida del siguiente código:

import java.io.\*;

class LastError<T> {

private T lastError;

void setError(T t) {

lastError = t;

System.out.println(“LastError: setError”);

}

}

class StrLastError<S extends CharSequence> extends LastError<String> {

public StrLastError(S s) {

}

void setError(S s) {

System.out.println(“StrLastError: setError”);

}

}

class Test {

public static void main(String… args) {

StrLastError<String> err = new StrLastError<>(“Error”);

err.setError(“Last error”);

}

}

Se obtiene un error de compilación ya que parece que el método setError en la clase StrLastError está sobre-escribiendo al método setError en la clase LastError. Sin embargo, no es el caso. En tiempo de compilación, el tipo conocido para “S” no está disponible. Por lo tanto, el compilador graba las signaturas de estos dos métodos como setError<String> para LastError y setError(S extends CharSequence) en StrLastError, tratando a ambos métodos como sobrecargados y no como sobre-escritos. En este caso, cuando se llama en el main() a setError(), el compilador encuentra que los dos métodos coinciden, resultando en un error de llamada ambígua a método.

Resumen

Generics

* Generics asegurará que cualquier intento de agregar elementos de otros tipos especificados en el parámetro de tipo, se cogerá en tiempo de compilación.
* Java 7 introdujo la sintaxis de diamante <> donde los parámetros de tipo pueden omitirse después del operador new, en la creación de un objeto. El compilador infiere el tipo de acuerdo al tipo en la declaración
* Generics no son covariantes. Esto es, el subtipado no funciona en generics; no puedes asignar un tipo genérico derivado a un parámetro de tipo base
* El <?> especifica un tipo desconocido en generics y se conoce como comodín. Por ejemplo, List<?> es una lista de objetos desconocidos
* Los comodines pueden limitarse. Por ejemplo <? extends Runnable> especifica que ? puede coincidir con cualquier tipo Runnable o cualquier clase derivada. Nota que extends es inclusivo, por lo tanto puedes reemplazar a Runnable en ?. Sin embargo, en <? super Runnable>, ? podrá coincidir solo con los supertipos de Runnable y Runnable por sí mismo no coincidirá: es exclusivo
* Puedes usar la palabra clave extends para clases e interfaces cuando especificas tipos limitados en genéricos. Para especificar multiples tipos base, puedes usar el amperson “&”. Por ejemplo List<? extends X & Y>, ? puede conicidir con cualquiera de los supertipos X o Y.

Framework Collections

* Evita usar tipos raw con tipos genéricos. En otros casos, asegúrate de los tipos seguros manualmente
* Los términos Colleciton, Collections, y collection son diferentes
  + Collection: java.util.Colleciton<E> es la interface raíz en la herarquía de colecciones
  + Collecctions: java.util.Collections es una clase utilitaria que contiene solo métodos estáticos.
  + collection(s): se refiere a los contenedores como map, stack, queue, etc.
* Las clases contenedoras almacenan referencias a los objetos, por lo tanto no puedes usar tipos primitivos en ninguna de estas clases
* Los métodos hashCode() y equals() necesitan ser consistentes para una clase. Para propósitos prácticos, asegúrate que sigues esta regla: el método hashCode() debe retornar el mismo valor hash para dos objetos, si el método equals() retorna true para los dos
* Si estás usando un objeto en los contenedores como HashSet o HashMap, asegúrate que sobre-escribiste los métodos del punto anterior correctamente
* La interface Map no extiende a la interface Collection
* No es recomendable que almacenes un valor null como argumento, ya que hay métodos en la interface Deque que retornan null y puede ser difícil distinguir entre el éxito o la falla de la llamada al método
* Implementa la interface Comparable para tus clases cuando un órden natural es posible. Si tú quieres comparar los objetos con otro órden distinto al natural crea clases separadas que implementen la interface Comparator.