Paso de parámetros por Valor y por Referencia

Parámetros por referencia en Java

En los lenguajes de programación suelen existir dos formas de pasar los parámetros a los métodos. Parámetros por valor (dónde se realiza una copia de las variables) o parámetros por referencia (dónde se pasa una referencia a la variable original).

Paso de parámetros por valor

Lo primero que tenemos que ver es que para los datos primitivos en Java se realiza claramente una copia.

```
1.  public void metodo(int p) {
2.    p=3;
3.  }
4.
5.  public static void main ....{
6.     int p1=2;
7.     metodo(p1);
8.     System.out.println(p1); //p1 = 2 porque el valor no ha cambiado
9.  }
```

Paso de parámetros "por referencia": referencia de objetos

Pero ahora pasemos a manejar un objeto como parámetro, **incluídos los arrays**. Lo que sucede al manejar los objetos en Java es que las variables pasan una referencia al objeto:

Creamos una clase básica llamada MiClase:

```
1.  public class MiClase {
2.   public int valor;
3.  }
```

Y ahora un método que modifica ese valor:

```
1.  public static void metodo_referencia(MiClase m) {
2.    m.valor =3;
3.  }
```

Veamos cómo se pasa por valor, aunque parece que hay una referencia:

```
1.  public static void main ...{
2.     MiClase m1 = new MiClase();
3.     m1.valor = 2;
4.     System.out.println(m1.valor); // Devuelve 2
5.     metodo_referencia(m1);
6.     System.out.println(m1.valor); // Devuelve 3 porque el valor sí ha cambiado
7.  }
```

Hemos instanciado con un valor de 2 el atributo de la clase y el método lo cambia a 3. Como la variable m en el método sigue manteniendo la referencia al objeto original, se produce un cambio en dicho objeto.

Pero para verlo mejor veamos otro caso. Ahora lo que vamos a hacer es crear un nuevo objeto y cambiar la referencia de la variable pasada por copia.

```
1.  public static void metodo_referencia2(MiClase m1) {
2.    MiClase m2 = new MiClase();
3.    m1 = m2;
4.    m1.valor = 3;
5.  }
```

Lo que sucede es que ahora m1, que mantenía una referencia al objeto inicial, pasa a tener una referencia a un nuevo objeto. Por lo tanto los cambios que hagamos en m1 no afectan ya al objeto inicial.

Si volvemos a realizar la misma secuencia de salida vemos que no hay cambios en el objeto inicial.

```
1.  public static void main...{
2.     MiClase m2 = new MiClase();
3.     m2.valor = 2;
4.     System.out.println(m2.valor); // Devuelve 2
5.     metodo_referencia2(m2);
6.     System.out.println(m2.valor); // Devuelve 2
7.  }
```

Paso de argumentos tipo wrappers:

```
1  public static void main(String[] args) {
2  Integer obj1 = new Integer(1);
3  Integer obj2 = new Integer(2);
4  System.out.print("Values of obj1 & obj2 before wrapper modification: ");
5  System.out.println("obj1 = " + obj1.intValue() + " ; obj2 = " + obj2.intValue());
```

```
6   modifyWrappers(obj1, obj2);
7   System.out.print("Values of obj1 & obj2 after wrapper modification: ");
8   System.out.println("obj1 = " + obj1.intValue() + " ; obj2 = " + obj2.intValue());
9   }
10 private static void modifyWrappers(Integer x, Integer y)
11 {
12 x = new Integer(5);
13 y = new Integer(10);
14 }
15
16
17
18
19
```

Salida:

```
1 Values of obj1 & obj2 before wrapper modification: obj1 = 1 ; obj2 = 2
2 Values of obj1 & obj2 after wrapper modification: obj1 = 1 ; obj2 = 2
```

Descripción de la salida:

Los tipos wrappers aunque son objetos, ocurre como los Strings, son pasados por valor y no por referencia. Cualquier cambio en la referencia dentro del método modifica el valor de las referencias y no los valores originales.

Interfaces en Java

En la programación orientada a objetos, a veces es útil definir qué debe hacer una clase, pero no cómo lo hará. Ya has visto un ejemplo de esto: el método abstract. Un método abstracto define la estructura de un método pero no proporciona ninguna implementación. Una subclase debe proporcionar su propia implementación de cada método abstracto definido por su superclase. Por lo tanto, **un método abstracto especifica la interfaz para el método pero no la implementación**.

Una Interface es como una clase abstracta pura, donde todo lo que contiene es abstracto, pero sin necesidad de utilizar la cláusula "abstract".

Si bien las clases y métodos abstractos son útiles, es posible llevar este concepto un paso más allá. En Java, puede separar por completo la interfaz de una clase de su implementación utilizando la palabra clave interface.

Table de Contenido

1. Qué es una interface en Java

Una interfaz (interface) es sintácticamente similar a una clase abstracta, en la que puede especificar uno o más métodos que no tienen cuerpo (*{}*). Esos métodos deben ser implementados por una clase para que se definan sus acciones.

Por lo tanto, **una interfaz especifica qué se debe hacer, pero no cómo hacerlo**. Una vez que se define una interfaz, cualquier clase puede implementarla. Además, una clase puede implementar más de una interfaz.

Para implementar una interfaz, la clase debe implementar todos los métodos descritos por la interfaz. Cada clase es libre de determinar los detalles de su propia implementación. Dos clases pueden implementar la misma interfaz de diferentes maneras, pero **cada clase tiene que implementar el mismo conjunto de métodos**.

2. Interface en el nuevo JDK

Antes de continuar, se necesita hacer un punto importante. JDK 8 agregó una función a **interface** que hizo un cambio significativo en sus capacidades. Antes de JDK 8, una interfaz no podía definir ninguna implementación de ningún tipo. Por lo tanto, antes de JDK 8, una interfaz podría definir solo el qué, pero no el cómo, como se acaba de describir. **JDK 8** cambió esto.

Hoy, es posible agregar una implementación predeterminada a un método de interfaz. Además, ahora se admiten los métodos **de interfaz estática** y, a partir de JDK 9, una interfaz también puede incluir métodos **privados**. Por lo tanto, ahora es posible que la interfaz especifique algún comportamiento.

Sin embargo, tales métodos constituyen lo que son, en esencia, características de uso especial, y la intención original detrás de la interfaz aún permanece. Por lo tanto, como regla general, con frecuencia creará y utilizará interfaces en las que no se utilizarán estas nuevas funciones. Por esta razón, comenzaremos discutiendo la interfaz en su forma tradicional. Las nuevas funciones de la interfaz se describen más adelante.

Aquí hay una forma general simplificada de una interfaz tradicional:

```
1. acceso interface nombre {
2.         tipo var1 = valor;
3.         tipo var2 = valor;
4.
5.         // ...
6.         tipo-retorno metodo-nombre1(lista-parametros);
7.         tipo-retorno metodo-nombre2(lista-parametros);
8.         tipo-retorno metodo-nombreN(lista-parametros);
9. }
```

En una interfaz, los métodos son implícitamente públicos.

Las variables declaradas en una interfaz no son variables de instancia. En cambio, son implícitamente *public*, *final*, y *static*, y deben inicializarse. Por lo tanto, son esencialmente constantes.

Aquí hay un ejemplo de una definición de interfaz. Especifica la interfaz a una clase que genera una serie de números.

```
1. public interface Series {
2.     int getSiguiente(); //Retorna el siguiente número de la serie
3.     void reiniciar(); //Reinicia
4.     void setComenzar(int x); //Establece un valor inicial
5. }
```

Esta interfaz se declara pública para que pueda ser implementada por código en cualquier paquete.

3. Implementación de interfaces

Una vez que se ha definido una interfaz, una o más clases pueden implementar esa interfaz. Para implementar una interfaz, es necesario incluir la cláusula implements en una definición de clase y luego crear los métodos requeridos por la interfaz. La forma general de una clase que incluye la cláusula de **implements** es:

```
1. class nombreclase implements interface {
2.     // cuerpo-clase
3. }
```

Para implementar más de una interfaz, las interfaces se separan con una coma.

Los métodos que implementan una interfaz deben declararse públicos.

Aquí hay un ejemplo que implementa la interfaz de *Series* mostrada anteriormente. Crea una clase llamada *DeDos*, que genera una serie de números, cada uno mayor que el anterior.

```
1. class DeDos implements Series {
```

```
2.      int iniciar;
3.      int valor;
4.
5.      DeDos(){
6.          iniciar=0;
7.          valor=0;
8.      }
9.
10.
11.     public int getSiguiente() {
12.        valor+=2;
13.        return valor;
14.     }
15.
16.     public void reiniciar() {
17.         valor=iniciar;
18.     }
19.
20.     public void setComenzar(int x) {
21.         iniciar=x;
22.         valor=x;
23.     }
24. }
```

Observe que los métodos *getSiguiente()*, *reiniciar()* y *setComenzar()* se declaran utilizando el especificador de acceso público (public). Esto es necesario.

Es posible y habitual para las clases que implementan interfaces, definir miembros y métodos adicionales propios. Por ejemplo, la siguiente versión de *DeDos* agrega el método *getAnterior()*, que devuelve el valor anterior:

```
1.  class DeDos implements Series {
2.      int iniciar;
3.      int valor;
4.      int anterior;
5.
6.      DeDos(){
7.          iniciar=0;
8.          valor=0;
9.      }
10.
11.
12.     public int getSiguiente() {
13.        anterior=valor;
14.        valor+=2;
15.       return valor;
16.    }
17.
18.    public void reiniciar() {
19.       valor=iniciar;
20.      anterior=valor-2;
21.    }
22.
23.    public void setComenzar(int x) {
24.       iniciar=x;
25.       valor=x;
```

```
26.       anterior=x-2;
27.     }
28.
29.     //Añadiendo un método que no está definido en Series
30.     int getAnterior(){
31.         return anterior;
32.      }
33. }
```

Observe que la adición de *getAnterior()* requirió un cambio en las implementaciones de los métodos definidos por *Series*. Sin embargo, dado que la interfaz con esos métodos permanece igual, el cambio es continuo y no rompe el código preexistente. Esta es una de las ventajas de las interfaces.

Un punto más: si una clase incluye una interfaz pero no implementa completamente los métodos definidos por esa interfaz, **esa clase debe declararse como abstracta** (abstract). No se pueden crear objetos de dicha clase, pero se puede usar como una **superclase abstracta**, lo que permite que las subclases proporcionen la implementación completa.

4. Uso de referencias a interface

Es posible que se sorprenda al descubrir que puede declarar una variable de referencia de un tipo de interfaz. En otras palabras, puede crear una **variable de referencia de interfaz**.

Dicha variable puede referirse a cualquier objeto que implemente su interfaz. Cuando llama a un método en un objeto a través de una referencia de interfaz, es la versión del método implementado por el objeto que se ejecuta. Este proceso es similar al uso de una referencia de superclase para acceder a un objeto de subclase.

El siguiente ejemplo ilustra este proceso. Utiliza la misma variable de referencia de interfaz para llamar a métodos en objetos de *DeDos* y *DeTres*.

```
1.  //Demostración de referencia de interface
2.  class DeDos implements Series {
3.      int iniciar;
4.      int valor;
5.
6.      DeDos(){
7.  iniciar=0;
```

```
8.  valor=0;
9.      }
10.
11.
12.     public int getSiguiente() {
13. valor+=2;
14. return valor;
15.     }
16.
17.     public void reiniciar() {
18. valor=iniciar;
19.     }
20.
21.     public void setComenzar(int x) {
22.         iniciar=x;
23.         valor=x;
24.     }
25. }
```

```
1.  public class DeTres implements Series{
2.  int iniciar;
3.  int valor;
4.
5.  DeTres(){
6.  iniciar=0;
7.  valor=0;
8.  }
9.
10.
11. public int getSiguiente() {
12. valor+=3;
13. return valor;
14. }
15.
16. public void reiniciar() {
17. valor=iniciar;
18. }
19.
20. public void setComenzar(int x) {
21. iniciar=x;
22. valor=x;
23. }
24. }
```

```
1.  class SeriesDemo {
2.  public static void main(String[] args) {
3.  DeDos dosOb=new DeDos();
4.  DeTres tresOb=new DeTres();
5.  Series ob;
6.
7.  for (int i=0;i<5;i++) {
8.  ob = dosOb;
9.  System.out.println("Siguiente valor DeDos es: " + ob.getSiguiente());
10. ob = tresOb;
11. System.out.println("Siguiente valor DeTres es: " + ob.getSiguiente());
12. }
13. }
14. }
```

Salida:

```
Siguiente valor DeDos es: 2
Siguiente valor DeTres es: 3
```

```
Siguiente valor DeDos es: 4
Siguiente valor DeTres es: 6
[...]
```

En *main()*, **ob** se declara como una referencia a una interfaz de *Series*. Esto significa que se puede usar para almacenar referencias a cualquier objeto que implemente *Series*. En este caso, se utiliza para referirse a *dosOb* y *tresOb*, que son objetos de tipo *DeDos* y *DeTres*, respectivamente, que implementan *Series*.

Una variable de referencia de interfaz solo tiene conocimiento de los métodos declarados por su declaración de interfaz. Por lo tanto, ***ob*** no se podría usar para acceder a otras variables o métodos que puedan ser compatibles con el objeto.

5. Variables en interfaces

Como se mencionó, las variables se pueden declarar en una interfaz, pero son implícitamente públicas, estáticas y finales (*public*, *static*, y *final*). A primera vista, podría pensar que habría un uso muy limitado para tales variables, pero ocurre lo contrario.

Los programas grandes normalmente hacen uso de varios valores constantes que describen cosas como el tamaño de la matriz, diversos límites, valores especiales y similares. Dado que un programa grande generalmente se mantiene en una cantidad de archivos fuente separados, debe haber una forma conveniente de hacer que estas constantes estén disponibles para cada archivo. En Java, las variables de interfaz ofrecen una solución.

Para definir un conjunto de constantes compartidas, cree una interfaz que contenga solo estas constantes, sin ningún método. Cada archivo que necesita acceso a las constantes simplemente “implementa” la interfaz. Esto trae las constantes a la vista. Aquí hay un ejemplo:

```
1. //Una interfaz que contiene constantes
2. interface Constante {
3.
4. //Definiendo 3 constantes
5. int MIN=0;
6. int MAX=10;
7. String MSJERROR="LIMITE ERROR";
8. }
```

```
class ConstanteD implements Constante{
public static void main(String[] args) {
int numeros[]=new int[MAX];

for (int i=MIN; i<11; i++){
if (i>=MAX) System.out.println(MSJERROR);
else {
numeros[i]=i;
System.out.println(numeros[i]+ " ");
}
}
}
}
```

La técnica de usar una interfaz para definir constantes compartidas es controvertida. Veremos más ejemplos más adelante.

6. Las interfaces pueden ser extendidas

Una interfaz puede heredar otra mediante el uso de la palabra clave **extends**. La sintaxis es la misma que para heredar clases. Cuando una clase implementa una interfaz que hereda otra interfaz, debe proporcionar implementaciones para todos los métodos requeridos por la cadena de herencia de la interfaz. Lo siguiente es un ejemplo:

```
//Una interface puede extender de otra
interface A{
void metodo1();
void metodo2();
}
```

```
//B ahora incluye metodo1() y metodo2() - y añade metodo3()
interface B extends A{
void metodo3();
}
```

```
//Esta clase debe implementar los métodos de A y B
class MiClase implements B{

public void metodo1() {
System.out.println("Implementación de metodo1().");
}

public void metodo2() {
System.out.println("Implementación de metodo2().");
}

public void metodo3() {
System.out.println("Implementación de metodo3().");
```

```
14. }
15. }
```

```
1.  public class Extender {
2.  public static void main(String[] args) {
3.  MiClase mc=new MiClase();
4.
5.  mc.metodo1();
6.  mc.metodo2();
7.  mc.metodo3();
8.  }
9.  }
```

Salida:

```
Implementación de metodo1().
Implementación de metodo2().
Implementación de metodo3().
```

Como experimento, puede intentar eliminar la implementación de *metodo1()* en *MiClase*. Esto causará un error en tiempo de compilación. Como se dijo anteriormente, cualquier clase que implemente una interfaz debe implementar todos los métodos requeridos por esa interfaz, **incluidos los heredados de otras interfaces**.

Fechas y horas (API Java)

Java Date Examples

Few examples to work with `Date` APIs.

Example 1.1 – Convert Date to String.

```
SimpleDateFormat sdf = new SimpleDateFormat("dd/M/yyyy");
String date = sdf.format(new Date());
System.out.println(date); //15/10/2013
```

Example 1.2 – Convert String to Date.

```
SimpleDateFormat sdf = new SimpleDateFormat("dd-M-yyyy hh:mm:ss");
String dateInString = "31-08-1982 10:20:56";
Date date = sdf.parse(dateInString);
System.out.println(date); //Tue Aug 31 10:20:56 SGT 1982
```

P.S Refer to this – SimpleDataFormat JavaDoc for detail date and time patterns.

Example 1.3 – Get current date time

```
SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");
Date date = new Date();
System.out.println(dateFormat.format(date)); //2013/10/15 16:16:39
```

Example 1.4 – Convert Calendar to Date

```
Calendar calendar = Calendar.getInstance();
        Date date = calendar.getTime();
```

2. Java Calendar Examples

Few examples to work with `Calendar` APIs.

Example 2.1 – Get current date time

```
SimpleDateFormat sdf = new SimpleDateFormat("yyyy MMM dd HH:mm:ss");
Calendar calendar = new GregorianCalendar(2013,0,31);
System.out.println(sdf.format(calendar.getTime()));
```

Output

```
2013 Jan 31 00:00:00
```

Example 2.2 – Simple Calendar example

```
SimpleDateFormat sdf = new SimpleDateFormat("yyyy MMM dd HH:mm:ss");
Calendar calendar = new GregorianCalendar(2013,1,28,13,24,56);

int year       = calendar.get(Calendar.YEAR);
int month      = calendar.get(Calendar.MONTH); // Jan = 0, dec = 11
int dayOfMonth = calendar.get(Calendar.DAY_OF_MONTH);
int dayOfWeek  = calendar.get(Calendar.DAY_OF_WEEK);
int weekOfYear = calendar.get(Calendar.WEEK_OF_YEAR);
int weekOfMonth= calendar.get(Calendar.WEEK_OF_MONTH);
```

```
int hour       = calendar.get(Calendar.HOUR);        // 12 hour clock
int hourOfDay  = calendar.get(Calendar.HOUR_OF_DAY); // 24 hour clock
int minute     = calendar.get(Calendar.MINUTE);
int second     = calendar.get(Calendar.SECOND);
int millisecond= calendar.get(Calendar.MILLISECOND);

System.out.println(sdf.format(calendar.getTime()));

System.out.println("year \t\t: " + year);
System.out.println("month \t\t: " + month);
System.out.println("dayOfMonth \t: " + dayOfMonth);
System.out.println("dayOfWeek \t: " + dayOfWeek);
System.out.println("weekOfYear \t: " + weekOfYear);
System.out.println("weekOfMonth \t: " + weekOfMonth);

System.out.println("hour \t\t: " + hour);
System.out.println("hourOfDay \t: " + hourOfDay);
System.out.println("minute \t\t: " + minute);
System.out.println("second \t\t: " + second);
System.out.println("millisecond \t: " + millisecond);
```

Output

```
2013 Feb 28 13:24:56
year             : 2013
month            : 1
dayOfMonth       : 28
dayOfWeek        : 5
weekOfYear       : 9
weekOfMonth     : 5
hour             : 1
hourOfDay        : 13
minute           : 24
second           : 56
millisecond     : 0
```

Example 2.3 – Set a date manually.

```
SimpleDateFormat sdf = new SimpleDateFormat("yyyy MMM dd HH:mm:ss");

Calendar calendar = new GregorianCalendar(2013,1,28,13,24,56);
System.out.println("#1. " + sdf.format(calendar.getTime()));

//update a date
calendar.set(Calendar.YEAR, 2014);
calendar.set(Calendar.MONTH, 11);
calendar.set(Calendar.MINUTE, 33);

System.out.println("#2. " + sdf.format(calendar.getTime()));
```

Output

```
#1. 2013 Feb 28 13:24:56
#2. 2014 Dec 28 13:33:56
```

Example 2.4– Add or subtract from a date.

```
SimpleDateFormat sdf = new SimpleDateFormat("yyyy MMM dd");

Calendar calendar = new GregorianCalendar(2013,10,28);
System.out.println("Date : " + sdf.format(calendar.getTime()));

//add one month
```

```
calendar.add(Calendar.MONTH, 1);
System.out.println("Date : " + sdf.format(calendar.getTime()));

//subtract 10 days
calendar.add(Calendar.DAY_OF_MONTH, -10);
System.out.println("Date : " + sdf.format(calendar.getTime()));
```

Output

```
Date : 2013 Nov 28
Date : 2013 Dec 28
Date : 2013 Dec 18
```

Example 2.5– Convert Date to Calendar.

```
SimpleDateFormat sdf = new SimpleDateFormat("dd-M-yyyy hh:mm:ss");
String dateInString = "22-01-2015 10:20:56";
Date date = sdf.parse(dateInString);

        Calendar calendar = Calendar.getInstance();
calendar.setTime(date);
```