Tipos básicos de java con sus wrappers

En ocasiones es muy conveniente poder tratar los datos primitivos (int, boolean, etc.) como objetos. Por ejemplo, los contenedores definidos por el API en el package java.util (Arrays dinámicos, listas enlazadas, colecciones, conjuntos, etc.) utilizan como unidad de almacenamiento la clase Object. Dado que Object es la raíz de toda la jerarquía de objetos en Java, estos contenedores pueden almacenar cualquier tipo de objetos. Pero los datos primitivos no son objetos, con lo que quedan en principio excluidos de estas posibilidades.

Para resolver esta situación el API de Java incorpora las clases envoltorio (wrapper class), que no son más que dotar a los datos primitivos con un envoltorio que permita tratarlos como objetos. Por ejemplo podríamos definir una clase envoltorio para los enteros, de forma bastante sencilla.

La API de Java hace innecesario esta tarea al proporcionar un conjunto completo de clases envoltorio para todos los tipos primitivos. Adicionalmente a la funcionalidad básica que se muestra en el ejemplo las clases envoltorio proporcionan métodos de utilidad para la manipulación de datos primitivos (conversiones de / hacia datos primitivos, conversiones a String, etc.)

Las clases envoltorio existentes son:

```
Byte para byte.
Short para short.
Integer para int.
Long para long.
Boolean para boolean
Float para float.
Double para double y
Character para char.
```

Obsérvese que las clases envoltorio tienen siempre la primera letra en mayúsculas.

Las clases envoltura se usan como cualquier otra:

```
Integer i = new Integer(5);
int x = i.intValue();
```

Hay que tener en cuenta que las operaciones aritméticas habituales (suma, resta, multiplicación ...) están definidas solo para los datos primitivos por lo que las clases envoltura no sirven para este fin.

Las variables primitivas tienen mecanismos de reserva y liberación de memoria más eficaces y rápidos que los objetos por lo que deben usarse datos primitivos en lugar de sus correspondientes envolturas siempre que se pueda.

```
public class Entero {
    private int valor;

    Entero(int valor) {
        this.valor = valor;
    }
    int intValue() {
        return valor;
    }
```

Sobrecarga es la capacidad de un lenguaje de programación, que permite nombrar con el mismo identificador diferentes variables u operaciones.

En programación orientada a objetos la sobrecarga se refiere a la posibilidad de tener dos o más funciones con el mismo nombre pero funcionalidad diferente. Es decir, dos o más funciones con el mismo nombre realizan acciones diferentes. El compilador usará una u otra dependiendo de los parámetros usados. A esto se llama también sobrecarga de funciones.

También existe la sobrecarga de operadores que al igual que con la sobrecarga de funciones se le da más de una implementación a un operador.

El mismo método dentro de una clase permite hacer cosas distintas en función de los parámetros.

Java no permite al programador implementar sus propios operadores sobrecargados, pero sí utilizar los predefinidos como el +. C++, por el contrario si permite hacerlo.

Algunos **métodos** en una clase pueden tener el mismo nombre. Estos métodos deben contar con diferentes argumentos. El compilador decide qué método invocar comparando los argumentos. Se generara un error si los métodos tienen definidos los mismos parámetros

```
public class Articulo {
   private float precio;
   public void setPrecio() {
        precio = 0;
   }
   public void setPrecio(float nuevoPrecio) {
        precio = nuevoPrecio;
   }
}
```

Sobrecarga de métodos y de constructores

La firma de un método es la combinación del nombre y los tipos de los parámetros o argumentos.

La sobrecarga de métodos es la creación de varios métodos con el mismo nombre pero con diferente lista de tipos de parámetros. Java utiliza el número y tipo de parámetros para seleccionar cuál definición de método ejecutar.

Java diferencia los métodos sobrecargados con base en el número y tipo de parámetros o argumentos que tiene el método y no por el tipo que devuelve.

Tambien existe la sobrecarga de constructores: Cuando en una clase existen constructores múltiples, se dice que hay sobrecarga de constructores.

Ejemplo

```
/* Métodos sobrecargados */
int calculaSuma(int x, int y, int z){
    ...
}
int calculaSuma(double x, double y, double z){
    ...
}

/* Error: estos métodos no están sobrecargados */
int calculaSuma(int x, int y, int z){
     ...
}
double calculaSuma(int x, int y, int z){
    ...
}
```

Prueba unitaria

En programación, una **prueba unitaria** es una forma de comprobar el correcto funcionamiento de una unidad de código. Por ejemplo en diseño estructurado o en diseño funcional una función o un procedimiento, en diseño orientado a objetos una clase. Esto sirve para asegurar que cada unidad funcione correctamente y eficientemente por separado. Además de verificar que el código hace lo que tiene que hacer, verificamos que sea correcto el nombre, los nombres y tipos de los parámetros, el tipo de lo que se devuelve, que si el estado inicial es válido, entonces el estado final es válido también.

La idea es escribir casos de prueba para cada función no trivial o método en el módulo, de forma que cada caso sea independiente del resto. Luego, con las Pruebas de Integración, se podrá asegurar el correcto funcionamiento del sistema o subsistema en cuestión.

Características

Para que una prueba unitaria tenga la *calidad suficiente* se deben cumplir los siguientes requisitos:

- Automatizable: No debería requerirse una intervención manual. Esto es especialmente útil para integración continua.
- Completas: Deben cubrir la mayor cantidad de código.
- Repetibles o Reutilizables: No se deben crear pruebas que sólo puedan ser ejecutadas una sola vez. También es útil para integración continua.
- Independientes: La ejecución de una prueba no debe afectar a la ejecución de otra.
- Profesionales: Las pruebas deben ser consideradas igual que el código, con la misma profesionalidad, documentación, etc.

Aunque estos requisitos no tienen que ser cumplidos al pie de la letra, se recomienda seguirlos o de lo contrario las pruebas pierden parte de su función.

Ventajas[editar]

El objetivo de las pruebas unitarias es aislar cada parte del programa y mostrar que las partes individuales son correctas. Proporcionan un contrato escrito que el trozo de código debe satisfacer. Estas pruebas aisladas proporcionan cinco ventajas básicas:

- Fomentan el cambio: Las pruebas unitarias facilitan que el programador cambie el código para mejorar su estructura (lo que se ha dado en llamar refactorización), puesto que permiten hacer pruebas sobre los cambios y así asegurarse de que los nuevos cambios no han introducido defectos.
- Simplifica la integración: Puesto que permiten llegar a la fase de integración con un grado alto de seguridad de que el código está funcionando correctamente. De esta manera se facilitan las pruebas de integración.
- Documenta el código: Las propias pruebas son documentación del código, puesto que ahí se puede ver cómo utilizarlo.
- Separación de la interfaz y la implementación: Dado que la única interacción entre los casos de prueba y las unidades bajo prueba son las interfaces de estas últimas, se puede cambiar cualquiera de los dos sin afectar al otro, a veces usando objetos maquetados (mock object - maqueta) que habilitan de forma aislada (unitaria) el comportamiento de objetos complejos.
- Los errores están más acotados y son más fáciles de localizar: Dado que tenemos pruebas unitarias que pueden desenmascararlos.

REGLA DE TRES A

```
1. ublic class ReglaDeTres {
2.
3.
4.         public static void main(String[] args) {
5.                 Scanner sc = new Scanner(System.in);
6.                 int a, b, x, y;
7.
8.                 System.out.println("Teniendo en cuenta que Y = (B * X)
   / A \ny que A, B y X son valores conocidos...");
9.                 System.out.println("Introduzca el valor de A: ");
10.                       a = sc.nextInt();
11.                       System.out.println("Introduzca el valor de B:
   ");
12.                       b = sc.nextInt();
13.                       System.out.println("Introduzca el valor de X:
   ");
14.                       x = sc.nextInt();
15.
16.                       //hallamos la regla de 3
17.                       // Y = (B * X) / A
18.                       y = ( b * x ) / a;
19.                       System.out.println("Valor de Y = " + y);
20.                       sc.close();
21.               }
```

La Palabra Clave this

Cuando se llama a un método, se pasa automáticamente un argumento implícito que es una referencia al objeto invocado (es decir, el objeto sobre el que se llama el método). Esta referencia se llama **this**. Para comprender esto, considere primero un programa que crea una clase llamada *Potencia* que calcula el resultado de un número elevado a una potencia entera:

```
class Potencia {
        double b;
        int e;
        double valor;
       Potencia(double base, int exp){
            b=base;
            e=exp;
            valor=1;
        if (exp==0) return;
            for ( ; exp>0; exp--) valor = valor * base;
        }

         double get_potencia(){
            return valor;
    }
}
class DemoPotencia {
    public static void main(String[] args) {

        Potencia x=new Potencia(4.0,2);
        Potencia y=new Potencia(2.5,1);
        Potencia z=new Potencia(2.7,2);

        System.out.println(x.b+ " elevado a la potencia de "+ x.e+", es igual a: "+x.
get_potencia());
        System.out.println(y.b+ " elevado a la potencia de "+ y.e+", es igual a: "+y.
get_potencia());
        System.out.println(z.b+ " elevado a la potencia de "+ z.e+", es igual a: "+z.
get_potencia());
    }
}
```