Passaggio dei parametri e inizializzazione di oggetti

Passaggio dei parametri

In Java il passaggio dei parametri è per valore (all'atto della chiamata di un metodo viene creata una copia del parametro attuale che è assegnata al parametro formale

Passaggio dei parametri.

```
public class PassReferences {
static void f( int i) {
        i++;
        System.out.println(i);
}
public static void main(String[] args) {
        int x=0;
        System.out.println(x);
        f(x);
        System.out.println(x);
}
}
```

Passaggio dei parametri.

```
class Index{
        int i;
}

public class PassReferences {
static void f( Index index) {
        index.i++;
        System.out.println(index.i);
}
public static void main(String[] args) {
    Index x=new Index()
    System.out.println(x.i);
    f(x);
    System.out.println(x.i);
}
}
```

Clonazione di oggetti…

In Java è possibile *clonare* un oggetto ovvero farne una copia locale attraverso il metodo *clone()*

Questo metodo definito *protected* nella classe base *Object* può essere usato all'interno della sottoclasse.

Il metodo *clone()* genera un *Object* a cui deve essere applicato il casting al proprio tipo.

…Clonazione di oggetti…

Anche se clone() è definito PROTECTED nella classe base *Object*, non tutte le classi Java sono clonabili. Se si vuole rendere una classe clonabile bisogna aggiungere del codice specifico.

In particolare è necessario:

- che la classe implementi l'interfaccia *Cloneable* (che non definisce, comunque, alcun metodo);
- creare un proprio metodo *clone()* per la classe rendendolo *public*;

```
// Creazione di copie locali con clone().
import java.util.*;

class MyObject implements Cloneable {
        private int i;
        MyObject(int ii) { i = ii; }
        public Object clone() {
                Object o = null;
                try {
                        o = super.clone();
                } catch(CloneNotSupportedException e) {
                        System.err.println("MyObject can't clone");
                }
                return o;
    }
        public String toString() {return Integer.toString(i); }
}
```

```
public class LocalCopy {
        static MyObject g(MyObject v) {
                v.i++;
                return v;
        }
        static MyObject f(MyObject v) {
                v = (MyObject) v.clone(); // Copia locale
                v.i++;
                return v;
        }
    public static void main(String[] args) {
        MyObject a = new MyObject(11);
        MyObject b = g(a);
        System.out.println("b = " + b);
        MyObject d = f(b);
        System.out.println("d = " + d);
        System.out.println("b = " + b);

    }
}
```

```
b = 12
d = 13
b = 12
```

Clonazione di oggetti (istanze di classi con composizione)

Anche le classi che vengono coinvolte nella composizione devono essere Cloneable

```
import java.util.*;

class MyObject implements Cloneable {
        private int i;
        MyObject(int ii) { i = ii; }
        public Object clone() {
                Object o = null;
                try {
                        o = super.clone();
                } catch(CloneNotSupportedException e) {}
                return o;
    }
class MyObject2 implements Cloneable {
        private MyObject i;
        MyObject(MyObject ii) { i = ii; }
        public Object clone() {
                Object o = null;
                try {
                        o = super.clone();
                } catch(CloneNotSupportedException e) {}
                return o;
    }
}
```

Classi a sola lettura…

Java permette di creare delle classi a sola-lettura (*read-only class*) i cui oggetti sono immutabili.

Infatti si possono definire classi prive di metodi che causano il cambiamento dello stato interno degli oggetti ed evitare così il problema dell'aliasing.

Esempi di classe *read-only* sono le classi "wrapper" della libreria standard di Java per tutti i tipi primitivi.

Esempio

Il valore intero all'interno della classe integer non può essere modificato.

```
// The Integer class cannot be changed.
import java.util.*;
public class ImmutableInteger {
    public static void main(String[] args) {
         Integer v []= new Integer[10];
         for(int i = 0; i < 10; i++)
                  v[i]=new Integer(i));
         // But how do you change the int
         // inside the Integer?
         for(int i = 0; i < 10; i++)
                  v[i]++; // equal to v[i]=new Integer(v[i]+1);
    }
}
```

…Classi a sola lettura…

Inoltre è possibile anche creare delle classi read-only proprie.

Tutti i dati sono *private* e nessuno dei metodi *public* modifica tali dati.

La creazione di una classe immutabile, pur essendo una soluzione elegante, può causare frequenti attivazioni del garbage collector oltre ad avere un costo aggiuntivo dovuto alla creazione di nuovi oggetti. Questo problema è particolarmente sentito per esempio per la classe *String*.

La soluzione consiste nel creare una classe simile ma modificabile e saltare da una all'altra a seconda delle esigenze.

```
// Gli oggetti che non possono essere modificati
// sono immuni agli effetti di alias
public class Immutable1 {
   private int data;
   public Immutable1(int initVal) {data = initVal;}
   public int read() { return data; }
   public boolean nonzero() { return data != 0; }
   public Immutable1 quadruple() {return new Immutable1(data * 4);}
   static void f(Immutable1 i1) {
         Immutable1 quad = i1.quadruple();
         System.out.println("i1 = " + i1.read());
         System.out.println("quad = " + quad.read());
                  }
   public static void main(String[] args) {
         Immutable1 x = new Immutable1(47);
         System.out.println("x = " + x.read());
         f(x);
         x=x.quadruple().quadruple().quadruple().quadruple();
         System.out.println("x = " + x.read());
    }
}
```

…Classi a sola lettura…

La classe **Immutable2** contiene metodi che mantengono gli oggetti immutati, producendo nuovi oggetti ogni qualvolta che è necessaria una modifica di stato.

Viceversa **Mutable** contiene metodi che modificano gli oggetti passati ai metodi **add()** e **multiply()**.

Il metodi statici **modify1( )** e **modify2( )** producono gli stessi risultati. Tuttavia mentre il primo richiede la creazione di quattro oggetti, il secondo richiede la creazione di solo due oggetti.

```
class Mutable {
private int data;
public   Mutable(int initVal) {
     data = initVal;
}
public   Mutable add(int x) {
     data += x;
     return this;
}
public   Mutable multiply(int x) {
     data *= x;
     return this;
}
public   Immutable2 makeImmutable2() {
     return new Immutable2(data);
}
}
```

```
public class Immutable2 {
private int data;
public   Immutable2(int initVal) {data = initVal;}
public int read() { return data; }
public boolean nonzero() { return data != 0; }
public   Immutable2 add(int x) {return new Immutable2(data + x);}
public Immutable2 multiply(int x) {return new Immutable2(data * x);}
public   Mutable makeMutable() {return new Mutable(data);}
public static Immutable2 modify1(Immutable2 y){ // (((y+12)*3+11)*2+4)*7
    Immutable2 val = y.add(12);
    val = val.multiply(3); // crea nuovo oggetto
    val = val.add(11); // crea nuovo oggetto
    val = val.multiply(2); // crea nuovo oggetto
    return val;
}
// Questa procedura restituisce gli stessi risultati
public static Immutable2 modify2(Immutable2 y){
    Mutable m = y.makeMutable();
    m.add(12).multiply(3).add(11).multiply(2);
    return m.makeImmutable2();
}
}
```

```
public class MainClass {
public static void main(String[] args) {
     Immutable2 i2 = new Immutable2(47);
     Immutable2 r1 = modify1(i2);
     Immutable2 r2 = modify2(i2);
     System.out.println("i2 = "+ i2.read());
     System.out.println("r1 = "+ r1.read());
     System.out.println("r2 = "+ r2.read());
    }

}
```

…Classi a sola lettura…

Come già osservato, l'immutabilità di una classe può causare dei problemi di inefficienza.

Esempio: la concatenazione di stringhe

```
String s = "abc" + foo + "def" +
  Integer.toString(47);
```

La soluzione al problema consiste nell'uso di una "compagno di classe" che sia mutevole ☺.

Per **String** il compagno di classe pensato dai progettisti Java si chiama **StringBuffer**. Il compilatore crea automaticamente un oggetto **StringBuffer** per valutare certe espressioni, come quelle degli operatori **+** e **+=** della classe **String**.

Inizializzazione dei membri…

Java garantisce che alcune variabili siano inizializzate coerentemente prima di essere usate.

In particolare, vengono inizializzate tutte le variabili di istanza e di classe.

Se sono di tipo primitivo viene assegnato un valore di default.

Se sono il riferimento a un oggetto viene assegnato valore `null`.

Per variabili locali a un metodo viene segnalato un errore al momento della compilazione.

…Inizializzazione dei membri.

Un membro di tipo dato può essere inizializzato attraverso il valore restituito da un metodo (con o senza argomenti).

L'ordine di inizializzazione è determinato dall'ordine in cui le variabili sono definite nella classe.

…Inizializzazione dei membri…

Esempio

```
class Measurement {
Depth o = new Depth();
boolean b = true;
// . . . }
```

dove *Depth* è una classe precedentemente definita.

- Nel caso di inizializzazione mediante chiamata di un metodo

```
class CInit {
int i = f();
int j = g(i);//...
int f() {return 5;}
int g(int a) {return a+1;}
}
```

…Inizializzazione dei membri.

- Ma non potremo avere mai:

```
class CInit {
int j = g(i);  //i non è stato ancora
int i = f();      //inizializzato
//...
int f() {return 5;}
int g(int a) {retur a+1;}
}
```

Si noti che l'inizializzazione di variabili d'istanza non è consentita in C++

Inizializzazione di array…

Un array in Java è una sequenza di elementi (oggetti o tipi primitivi) dello stesso tipo impacchettati insieme e identificati da un unico nome.

Un array in Java si definisce nel seguente modo:

```
int[] a1   oppure          int a1[]
```

L'inizializzazione per gli array può avvenire in qualsiasi punto del codice ed avviene nel seguente modo:

```
int[] a1 = { 1, 2, 3, 4, 5 }
```

È possibile anche inizializzare gli array in altri modi senza però rispettare la compatibilità con le versioni precedenti del JDK.

…Inizializzazione di array…

Ogni array ha un membro instrinseco, **`length`**, a cui possiamo accedere (ma non modificare) che indica il numero di elementi presenti nell'array.

In Java si contano gli elementi di un array a partire da 0 pertanto l'ultimo elemento di un array avrà sempre posizione ***length-1***

```
public class ArrayInit {
    public static void main(String[] args) {
        Integer[] a = {
                new Integer(1),
                new Integer(2),
                new Integer(3),
        };


        Integer[] b = new Integer[3];
        b[0]=new Integer(1);
        b[1]=new Integer(2);
        b[2]=new Integer(3);
    =
    }
}
```