AA 2015-2016

PROGRAMMAZIONE 2
4.Ancora su Java

Una prima analisi

- Alcune caratteristiche di Java richiedono una conoscenza dettagliata delle librerie
- Sistemi di supporto forniscono molti strumenti per programmare con Java (Eclipse è un esempio significativo)
- La nostra risposta: affrontiamo il problema in termini di *problem solving*

Espressioni vs. Comandi

- Java ha sia espressioni che comandi!
  - Le espressioni restituiscono valori
  - I comandi operano via side effect

Metodi statici

```
public class Max {
      public static int max (int x, int y) {
            if (x > y) { return x; }
            else { return y; }
      }

      public static int max3 (int x, int y, int z) {
            return max(max(x, y), z) ;
      }
}
```

Lsimile alla definizione di una funzione

```
public class Main {
      public static void main (String[ ] args) {
            System.out.println(Max.max(3, 4));
             return;
      }
}
```

Metodi statici

- Sono metodi indipendenti dall'oggetto (valgono per tutti gli oggetti della classe)
  - Non possono dipendere dai valori delle variabili di istanza
- Quando devono essere usati?
  - Per la programmazione non OO
  - Per il metodo main
- I metodi non statici sono entità dinamiche!
  - Devono conoscere e lavorare sulle variabili di istanza degli oggetti

Metodi statici

- I metodi statici sono "funzioni" definite globalmente
- Variabili di istanza *static* sono variabili globali accessibile tramite il nome della classe
- Variabili di istanza statiche non possono essere inizializzate nel costruttore della classe (dato che non possono essere associate a oggetti istanza della classe)

Esempio

```
public class C {
      private static int big = 23;
      public static int m(int x) {
            return big + x; //OK
      }
}
```

```
public class C {
      private static int big = 23;
      private int nonStaticField;
      private void setIt(int x) { nonStaticField = x + x; }
      public static int m(int x) {
            setIt(x);  // Errore: un metodo static non può
                       // accedere a metodi non statici e a
                       // variabili di istanza non statiche
      }
}
```

Quindi?

- I metodi statici sono utilizzati per implementare quelle funzionalità che non dipendono dallo stato dell'oggetto
- Esempi significativi nella API Math che fornisce funzioni matematiche come Math.sin
- Altri esempio sono dati dalle funzioni di conversione: Integer.toString e Boolean.valueOf

Java Demo

Liste in Java

```
interface StringList {
        public bolean isNil( );
        public String hd( );
        public StringList tl( );
}
```

```
class Cons implements StringList {
        private String head;
        private StringList tail;
        public Cons (String h, StringList t) {
                head = h; tail = t; }
        public boolean isNil( ) {
                return false; }
        public String hd( ) {
                return head ; }
         public StringList tl( ) {
                return tail; }
}
```

```
class Nil implements StringList {



        public boolean isNil( ) {
                return true; }
        public String hd( ) {
                return null; }
        public StringList tl( ) {
                return null; }
}
```

Operare su liste

```
StringList x = new Cons("Bunga", new Cons("bunga", new Nil( )))
```

- Regole pragmatiche generali
  - Per ogni tipo di dato definire la relativa interfaccia
  - Aggiungere una classe per ogni costruttore

Operare su liste

- Con ricorsione…

```
public static int numberOfSongs (StringList pl) {
      if (pl.isNil( )) { return 0; }
      else { return 1 + numberOfSongs (pl.tl( )); }
}
```

- …o iterazione!

```
public static int numberOfSongs (StringList pl) {
      int count = 0 ;
      StringList curr = pl;
      while (! Curr.isNil( )) {
            count = count + 1;
            curr = curr.tl( );
      }
      return count ;
}
```

Usare variabili di istanza per value-oriented programming

Higher-order?!?


```
public static StringList Map (??? f, StringList pl) {
       if (pl.isNil( )) { return new Nil( ); }
       else { return new Cons(???, Map(f, pl.tl( ))); }
}
public static testMap( ) {
       StringList x = new Cons("bunga bunga", new Nil( )):
       StringList y = map(???, x);
      assertEqual(y.hd( ), "BUNGA BUNGA");
}
```

Con le interfacce!!


```
Interface Fun { public static String apply (String x); }
Class UpperCaseFun implements Fun {
       public static String apply (String x) {
         return x.toUpperCase( );
       }
}
```

```
public static StringList Map (Fun f, StringList pl) {
       if (pl.isNil( )) { return new Nil( ); }
       else { return new Cons(f.apply(pl.hd( )), Map(f, pl.tl( ))); }
}
public static testMap( ) {
       StringList x = new Cons("bunga bunga", new Nil( )):
       StringList y = map(new UpperCaseFun( ), x);
      assertEqual(y.hd( ), "BUNGA BUNGA");
}
```

Le stringhe in Java

Java String

- Le stringhe (sequenze di caratteri) in Java sono una classe predefinita
  - **"3" + " " + "Volte 3"** equivale a "**3 Volte 3**"
  - Il "+" è pure l'operatore di concatenazione di stringhe
- Le stringhe sono oggetti immutabili (*a là* OCaml)

Uguaglianza

- Java ha due operatori per testare l'uguaglianza
  - o1 == o2 restituisce true se le variabili o1 e o2 denotano lo stesso riferimento (pointer equality)
  - o1.equals(o2) restituisce true se le variabili o1 e o2 denotano due oggetti identici (deep equality)
- Esempio
  - String("test").equals("test") --> true
  - new String("test") == "test" --> false
  - new String("test") == new String("test") --> false

Un quesito interessante…


```
String s1 = "java";
String s2 = "java";
```

```
s1.equals(s2)  // true… perché?
s1==s2  // true… perché?
```

Un altro quesito…


```
String str1 = new String("Java");
String str2 = new String("Java");

str1.equals(str2)  // true: stesso contenuto
str1==str2  // false: oggetti differenti
```

String: oggetti immutabili!


```
public class Main {
    public static void main(String[ ] args) {
        String s1 = "Programmare in ";
        if (s1.equals(s1.concat("Java")))
            System.err.println("True!");
        else
            System.err.println("False!");
    }
}
```

Java Assert

Assert

- Java fornisce una primitiva linguistica per "testare" proprietà di un programma
- Il comando assert
  - assert ***booleanExpression***;
  - valuta l'espressione booleana
  - se l'espressione viene valutata true il comando non modifica lo stato, se l'espressione restituisce false viene sollevata l'eccezione AssertionError

Assert

- assert booleanExpression : expression;
- In questo caso, se l'espressione booleana viene valutata false la seconda espressione è utilizzata all'interno del messaggio di AssertionError
- La seconda espressione può avere qualunque tipo con l'eccezione del tipo void

Invarianti

```
if (i % 3 == 0) {
   ...
} else if (i % 3 == 1) {
   ...
} else { // we know (i % 3 == 2)
   ...
}
```

Se la variabile *i* ha valore negativo l'assert solleva un errore

```
if (i % 3 == 0) {
   ...
} else if (i % 3 == 1) {
   ...
} else {
   assert i % 3 == 2 : i;
}
```

Assert e contratti

- Noi utilizzeremo il comando assert per supportare la programmazione “by contract”
  - Precondizioni
  - Postcondizioni
  - Invarianti di rappresentazione

Esempio

```
private void setRefreshRate(int rate) {
      @REQUIRES (rate <= 0 || rate > MAX_REFRESH_RATE)
      :
      :
}
```

```
private void setRefreshInterval(int rate) {
      assert rate > 0 && rate <= 1000 : rate;
      :
      :
}
```

Abilitare assert

- Il comando assert nella normale esecuzione di applicazioni Java non ha alcun effetto
- Le asserzioni devono essere abilitate
  - prompt$ java -enableassertions prog
  - prompt$ java -ea prog