


PROGRAMMAZIONE 2

3. Primi passi in Java

PR2 2018-2019 1



Oggetti e classi

- *Oggetto*: insieme strutturato di *variabili di istanza* (stato) e *metodi* (operazioni)
- *Classe*: *modello (template)* per la creazione di oggetti

OBJECT
STATO (NASCOSTO)
METODI (PUBBLICI)

PR2 2018-2019 2

Oggetti e classi

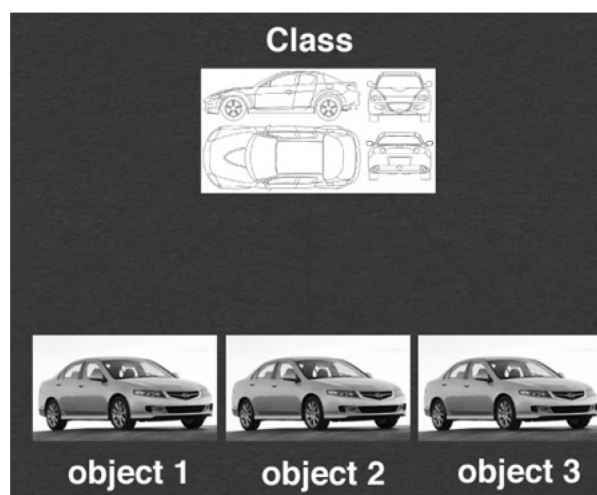


- La definizione di una classe specifica
 - Tipo e valori iniziali dello stato locale degli oggetti (le variabili di istanza)
 - Insieme delle operazioni che possono essere eseguite (metodi)
 - Costruttori (uno o più): codice che deve essere eseguito al momento della creazione di un oggetto
- Ogni oggetto è una istanza di una classe e può (opzionalmente) implementare una interfaccia

PR2 2018-2019


3

Oggetti e classi



PR2 2018-2019

4




- Oggetti hanno una interfaccia ben definita
 - Stato e metodi pubblici
- Implementazione non e' visibile all'esterno dell'oggetto
 - Information hiding

obj1

f1:	
f2:	
m(p1,p2) {...}	
m1() {...}	
m2(p) {...}	
m(p1,p2) {...}	
m1() {...}	
m2(p) {...}	
h1(p,q) {...}	
h2(r) {...}	
h3() {...}	

PR2 2018-2019 5



Il paradigma a oggetti

- Sistema software = insieme di oggetti cooperanti
- Oggetti sono caratterizzati da uno stato e da un insieme di funzionalità
- Oggetti cooperano scambiandosi dei messaggi

OBJ1

A1:	
A1:	
M1(){ ...}	
M2(p, r){ ...}	

OBJ2.M("abc", 7)

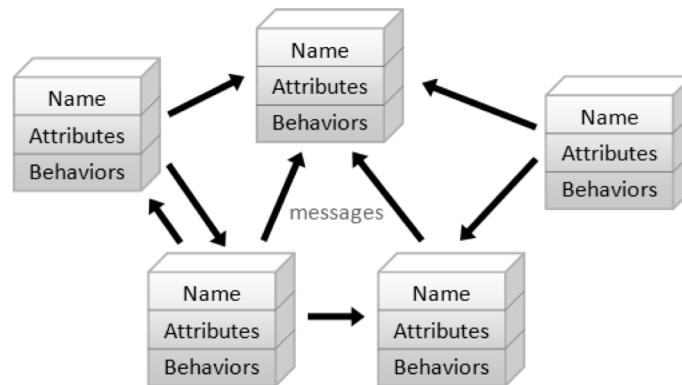
→

OBJ2

A:	
B:	
M(p, q){ ...}	

PR2 2018-2019 6

Oggetti e classi



An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

PR2 2018-2019

7

Oggetti: caratteristiche



- Oggetti sono caratterizzati
 - Stato
 - Identità (nome che individua l'oggetto)
 - Ciclo di vita (creati, riferiti, disattivati)
 - Locazione (di memoria)
 - Comportamenti
- **Rispetto al paradigma imperativo**
 - **Differente struttura dei programmi**
 - **Differente modello di esecuzione**

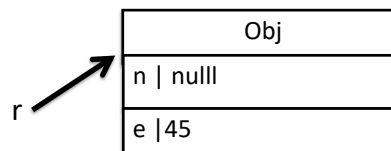
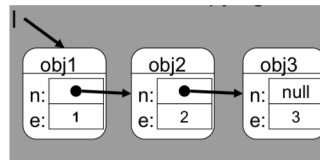
PR2 2018-2019

8

Identità di oggetti



I nomi **l** e **r**
Costituiscono l'identità
dell'oggetto



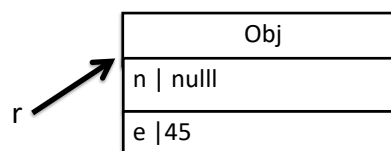
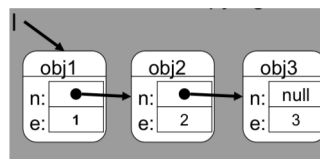
PR2 2018-2019

9

Identità di oggetti: operazioni



```
r = l.rest();
r.set(4711);
int i = l.next.get();
```



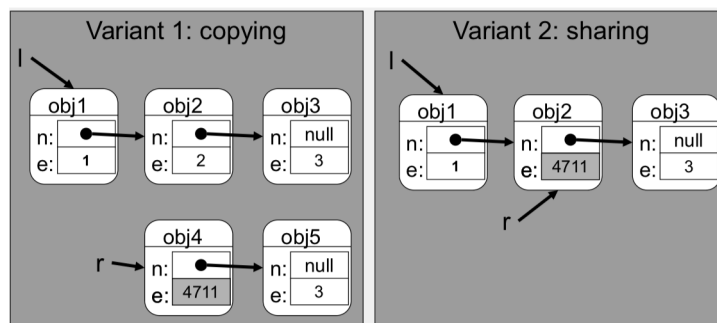
PR2 2018-2019

10

Oggetti: identità



```
r = l.rest( ); r.set( 4711 ); int i = l.next.get();
```



PR2 2018-2019

11

Un primo esempio



```
public class Counter { // nome della classe

    private int cnt; // lo stato locale

    // metodo costruttore
    public Counter ( ) { cnt = 0; }
    // metodo
    public int inc ( ) { cnt++; return cnt; }
    // metodo
    public int dec ( ) { cnt--; return cnt; }
}
```

**DICHIARAZIONE
DI CLASSE**

public = visibile fuori
dell'oggetto

private = visibile solo
all'interno dell'oggetto

PR2 2018-2019

12

Esecuzione di Java



un programma Java è mandato in esecuzione invocando un metodo speciale di una opportuna classe chiamato main

```
public class First {
    public static void main(String[ ] args) {
        Counter c = new Counter( );
        System.out.println(c.inc( ));
        System.out.println(c.dec( ));
    }
}
```

PR2 2018-2019

13

Compilare ed eseguire



```
prompt$ javac Counter.java
```

Viene creato il bytecode Counter.class

```
prompt$ javac First.java
```

Viene creato il bytecode First.class

```
prompt$ java First
```

```
1
```

```
0
```

```
prompt$
```

PR2 2018-2019

14

Cosa è il Java bytecode?



- È il linguaggio della Java Virtual Machine
- Load & store (e.g. aload_0, istore)
- Arithmetic & logic (e.g. ladd, fcmpl)
- Object creation & manipulation (new, putfield)
- Operand stack management (e.g. swap, dup2)
- Control transfer (e.g. ifeq, goto)
- Method invocation & return (e.g. invokespecial, areturn)
- Visualizzabile con javap !!
 - E anche editabile: ad esempio l'appena rilasciato [BCEL](#)

Creare oggetti



Dichiarare una variabile di tipo Counter

Invocare il costruttore per creare l'oggetto di tipo Counter

```
Counter c;  
c = new Counter( )
```

Soluzione alternativa: fare tutto in un passo!!

```
Counter c = new Counter( );
```


Costruttori con parametri



```
public class Counter { // nome della classe

    private int cnt; // lo stato locale

    // metodo costruttore
    public Counter (int v0) { cnt = v0; }
    // metodo
    public int inc ( ) { cnt++; return cnt; }
    // metodo
    public int dec ( ) { cnt--; return cnt; }
}
```

DICHIARAZIONE DI CLASSE

public = visibile fuori
dell'oggetto

private = visibile solo
all'interno dell'oggetto

PR2 2018-2019

17

Costruttori con parametri



```
public class First {
    public static void main(String[ ] args) {
        Counter c = new Counter(25);
        System.out.println(c.inc( ));
        System.out.println(c.dec( ));
    }
}
```

PR2 2018-2019

18

Strutture mutabili



Ogni variabile di oggetto in Java puo' denotare una entità mutabile

```
Counter C1;
Counter C2;
C1 = new Counter(5);
C2 = new Counter(10);
C1.inc( );
C2.dec()

// quale è il valore dello stato locale?
```

PR2 2018-2019

19

Il valore NULL



Il valore **null** è generico e può essere assegnato a qualunque variabile di tipo riferimento

Restituisce un oggetto di tipo Counter
o **null** se non lo trova

```
Counter c = cercaContatore( );
if (C == null) System.out.println("contatore non trovato");
```

Attenzione: come in C
= **singolo**: assegnamento
== **doppio**: test di uguaglianza

PR2 2018-2019

20

Nello heap...



- Gli oggetti Java sono memorizzati nello heap
- Nello heap vengono allocate
 - variabili di istanza, quando si crea un oggetto
 - variabili statiche (o di classe), quando è caricata una classe
- Le variabili allocate nello heap sono inizializzate dal sistema
 - con **0** (zero), per le variabili di tipi numerici
 - con **false**, per le variabili di tipo **boolean**
 - con **null**, per le variabili di tipo riferimento
- Le variabili dichiarate localmente in metodi/costruttori non vengono inizializzate per default: bisogna assegnare loro un valore prima di leggerle

PR2 2018-2019

21

Stato locale



- Modificatori: meccanismo per controllare l'accesso allo stato locale dell'oggetto
 - Public: visibile/accessibile da ogni parte del programma
 - Private: visibile/accessibile solo all'interno della classe
- Design Pattern (suggerimento grossolano)
 - Tutte le variabili di istanza: private
 - Costruttori e metodi: public

PR2 2018-2019

22

Riassunto...



- Il “frammento imperativo” di Java richiama da vicino la sintassi del C
 - **int x = 3;** // dichiara x e lo inizializza al valore 3
 - **int y;** // dichiara y e gli viene assegnato il valore di default 0
 - **y = x+3;** // assegna a y il valore di x incrementato di 3
 - // dichiara un oggetto C di tipo Counter e lo inizializza con
 - // il costruttore
 - **Counter c = new Counter();**
 - **Counter d;** // dichiara d e il suo valore di default è **null**
 - **d = c;** // assegna a d l’oggetto denotato da c => **Aliasing!**

PR2 2018-2019

23

Alcuni comandi...



- Condizionali
 - **if (cond) stmt1;**
 - **if (cond) { stmt1; stmt2; }**
 - **if (cond) { stmt1; stmt2; } else { stmt3; stmt4; }**
- Iterativi
 - **while (exp) { stmt1; stmt2; }**
 - **do { stmt1; stmt2; } until (exp);**
 - **for (init; term; inc) { stmt1; stmt2; }**

PR2 2018-2019

24

Demo



```

class WhileDemo {
    public static void main(String[ ] args) {
        int count = 1;
        while (count < 11) {
            System.out.println("Count is: " + count);
            count++;
        }
    }
}

class ForDemo {
    public static void main(String[ ] args) {
        for(int i = 1; i < 11; i++)
            System.out.println("Count is: " + i);
    }
}

```

PR2 2018-2019

25

Demo 2



```

class BreakDemo {
    public static void main(String[ ] args) {
        int[ ] arrayOfInts = { 32, 87, 3, 589, 12, 1076, 2000, 8, 622, 127 };
        int i, searchfor = 12;
        boolean foundIt = false;

        for (i = 0; i < arrayOfInts.length; i++) {
            if (arrayOfInts[i] == searchfor) {
                foundIt = true;
                break;
            }
        }

        if (foundIt) System.out.println("Found " + searchfor + " at index " + i);
        else System.out.println(searchfor + " not found in the array");
    }
}

```

PR2 2018-2019

26

Tipi primitivi



- **int** // *standard integers*
- **byte, short, long** // *other flavors of integers*
- **char, float** // *unicode characters*
- **double** // *floating-point numbers*
- **boolean** // *true and false*
- **String non è un tipo primitivo (ma quasi...)**

Interfacce in Java



Tipi in Java



- Java è un linguaggio fortemente tipato (ogni entità ha un tipo)
- Le classi definiscono il tipo degli oggetti
 - Classe come definizione del contratto di uso degli oggetti che appartengono a quella classe
- Java prevede un ulteriore meccanismo per associare il tipo agli oggetti: le interfacce

Java Interface



- Una interfaccia definisce il tipo degli oggetti in modo dichiarativo: non viene presentato il dettaglio della implementazione
- Interfaccia = Contratto d'uso dell'oggetto

Esempio



```
public interface Displaceable {
    public int getX ( );
    public int getY ( );
    public void move(int dx, int dy);
}
```

Nome

Dichiarazione dei tipi dei metodi

PR2 2018-2019

31

Una implementazione...



```
public class Point implements Displaceable {
    private int x, y;
    public Point(int x0, int y0) {
        x = x0;
        y = y0;
    }
    public int getX( ) { return x; }
    public int getY( ) { return y; }
    public void move(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }
}
```

Devono essere implementati tutti i metodi dell'interfaccia

32

Un'altra!!



```
class ColorPoint implements Displaceable {
    private Point p;
    private Color c;

    ColorPoint (int x0, int y0, Color c0) {
        p = new Point(x0,y0); c = c0;
    }
    public void move(int dx, int dy) {
        p.move(dx, dy);
    }
    public int getX( ) { return p.getX( ); }
    public int getY( ) { return p.getY( ); }
    public Color getColor( ) { return c; }
}
```

Oggetti che implementano la stessa interfaccia possono avere stato locale differente

Delega all'oggetto point

Numero maggiore di metodi di quelli previsti dal contratto

PR2 2018-2019

33

Tipi e interfacce



- Dichiarare variabili che hanno il tipo di una interfacce

```
Displaceable d;
d = new Point(1, 2);
d.move(-1, 1)
```

- Assegnare una implementazione

```
d = new ColorPoint(1, 2, new Color("red"));
d.move(-1, 1);
```

PR2 2018-2019

34

Sotto-tipi



- La situazione descritta illustra il fenomeno del subtyping (sotto-tipo): un tipo A è un sotto-tipo di B se un oggetto di tipo A in grado di soddisfare tutti gli obblighi che potrebbero essere richiesti dall'interfaccia (o una classe) B
- Intuitivamente, un oggetto di tipo A può fare qualsiasi cosa che può fare (un oggetto di tipo) B
- Maggiori dettagli in seguito (quando ColorPoint potrà essere *sotto-classe* di Point!!)

Interfacce multiple



```
public interface Area {
    public double getArea( );
}
```

```
public class Circle implements Displaceable, Area {
    private Point center;
    private int rad;

    public Circle(int x0, int y0, int r0) {
        rad = r0; center = new Point(x0, y0);
    }

    public double getArea ( ) {
        return Math.PI * rad * rad;
    }

    public int getRadius( ) { return rad; }
    public getX( ) { return center.getX( ); }
    public getY( ) { return center.getY( ); }
    public move(int dx, int dy) { center.move(dx, dy); }
}
```

Esempi d'uso



```
Circle c = new Circle(10,10,5);  
Displaceable d = c;  
Area a = c;  
  
Rectangle r = new Rectangle(10,10,5,20);  
d = r;  
a = r;
```

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



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);
    }
}

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

simile alla definizione di una funzione

PR2 2018-2019 41

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

PR2 2018-2019 42

Metodi statici



- I metodi statici sono “funzioni” definite globalmente
- Variabili di istanza *static* sono variabili globali accessibili 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 esempi sono dati dalle funzioni di conversione: `Integer.toString` e `Boolean.valueOf`

Java at work



Liste



```

/**
 * A ConsCell is an element in a linked list of
 * ints.
 */
public class ConsCell {
    private int head; // the first item in the list
    private ConsCell tail; // rest of the list, or null

    /**
     * Construct a new ConsCell given its head and tail.
     * @param h the int contents of this cell
     * @param t the next ConsCell in the list, or null
     */
    public ConsCell(int h, ConsCell t) {
        head = h;
        tail = t;
    }

```

PR2 2018-2019

47

Liste



```

/**
 * Accessor for the head of this ConsCell.
 * @return the int contents of this cell
 */
public int getHead() {
    return head;
}

/**
 * Accessor for the tail of this ConsCell.
 * @return the next ConsCell in the list, or null
 */
public ConsCell getTail() {
    return tail;
}

```

PR2 2018-2019

48

Liste



```
/**
 * Mutator for the tail of this ConsCell.
 * @param t the new tail for this cell
 */
public void setTail(ConsCell t) {
    tail = t;
}
}
```

OCAML vs JAVA



```
let a = [];          ConsCell a = null;
let b = 2::a;        ConsCell b = new ConsCell(2,a);
let c = 1::b;        ConsCell c = new ConsCell(1,b);
```

IntList



```

**
 * An IntList is a list of ints.
 */
public class IntList {
    private ConsCell start; // list head, or null

    /**
     * Construct a new IntList given its first ConsCell.
     * @param s the first ConsCell in the list, or null
     */
    public IntList(ConsCell s) {
        start = s;
    }

```

PR2 2018-2019

51

IntList



```

/**
 * Cons the given element h onto us and return the
 * resulting IntList.
 * @param h the head int for the new list
 * @return the IntList with head h, and us as tail
 */
public IntList cons (int h) {
    return new IntList(new ConsCell(h,start));
}

```

PR2 2018-2019

52

IntList



```

/**
 * Get our length.
 * @return our int length
 */
public int length() {
    int len = 0;
    ConsCell cell = start;
    while (cell != null) { // while not at end of list
        len++;
        cell = cell.getTail();
    }
    return len;
}
}

```

PR2 2018-2019

53

Usiamo IntList



OCAML:

```

let a = nil;
let b = 2::a;
let c = 1::b;
let x = (length a) + (length b) + (length c);

```

Java:

```

IntList a = new IntList(null);
IntList b = a.cons(2);
IntList c = b.cons(1);
int x = a.length() + b.length() + c.length();

```

PR2 2018-2019

54

Riferimenti



Un riferimento è un valore che identifica un particolare oggetto

Il parametro del costruttore **IntList** è un riferimento all'oggetto.

Cosa memorizziamo nella variabile di istanza **start** è il riferimento all'oggetto non una copia dell'oggetto.

```
public IntList(ConsCell s) {
    start = s;
}
```

Puntatori



- In soldoni: un riferimento e' un puntatore (a la C) all'oggetto.
- Ma
- C (e C++) espongono la natura "indirizzo" (con l'aritmetivca dei puntatori)
- Java non dice come sono implementati i riferimenti: sono valori che identificano univocamente oggetti

Per intenderci



C++	Java
<pre>IntList* p; p = new IntList(0); p->length(); p = q;</pre>	<pre>IntList p; p = new IntList(null); p.length(); p = q;</pre>
<pre>IntList p(0); p.length(); p = q;</pre>	No.

PR2 2018-2019

57

Liste in Java (à la OCaml)



```
interface StringList {
    public boolean 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; }
}
```

PR2 2018-2019

Operare su liste



```
StringList x = new Cons("Cowa", 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 numberOfStrings (StringList pl) {
  if (pl.isNil( )) { return 0; }
  else { return 1 + numberOfStrings (pl.tl( )); }
}
```

- ...o iterazione!

```
public static int numberOfStrings (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("Cowa Bunga", new Nil( ));
    StringList y = map(???, x);
    assertEquals(y.hd( ), "COWA BUNGA");
}
```

PR2 2018-2019

61

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("Cowa Bunga", new Nil( ));
    StringList y = map(new UpperCaseFun( ), x);
    assertEquals(y.hd( ), "COWA BUNGA");
}
```

62



Le stringhe in Java

PR2 2018-2019 63



Java String

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

PR2 2018-2019 64

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 s = new String("test"); String t = new String("test")`
 - `s.equals("test")` --> true
 - `s == "test"` --> false
 - `s == t` --> false

PR2 2018-2019

65

Un quesito interessante...



```
String s1 = "Java";
```

```
String s2 = "Java";
```

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

```
s1==s2 // true... perché?
```

[approfondire gli ["String literals"](#)]

PR2 2018-2019

66

Un quesito più standard...



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

```
str1.equals(str2) // true: stesso  
contenuto
```

```
str1==str2 // false: oggetti differenti
```

PR2 2018-2019

67


String: oggetti immutabili!



```
public class Test {  
    public static void main(String[ ] args) {  
        String s1 = "Cow";  
        if (s1.equals(s1.concat("Mucca")))  
            System.out.println("True!");  
        else  
            System.out.println("False!");  
    }  
}
```


PR2 2018-2019

68



Java Assert

PR2 2018-2019 69



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**

PR2 2018-2019 70

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**

PR2 2018-2019

71

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;
}

```

PR2 2018-2019

72

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`