Ingegneria del Software

Esercitazione 2

Valutazione Parametri

Illustrare e motivare il valore delle variabili i, counter e counter 2 ad ogni riga del main

```
class Counter {

  int counter = 0;

  public void increment(){
      counter++;
  }

  public void incrementAndSet(int i){
      i++;
      counter=i;
  }

  public void incrementAndSet(Counter c){
       c.counter++;
       counter = c.counter
  }
}
```

```
public static void main(String args[]) {
    Counter counter = new Counter();
    counter.increment();
    int i = 3;
    counter.incrementAndSet(i);
    Counter counter2 = new Counter();
    counter2.incrementAndSet(i);
    counter.incrementAndSet(counter2);
    Counter counter3 = counter;
  }
```

Valutazione Parametri

Tipi primitivi: passati per valore
Oggetti: passati per referenza

Risposta:

```
public static void main(String args[]) {
    Counter counter = new Counter();          counter = 0
    counter.increment();                      counter = 1
    int i = 3;                                i = 3
    counter.incrementAndSet(i);               i = 3, counter = 4
    Counter counter2 = new Counter();         i = 3, counter = 4, counter2 = 0
    counter2.incrementAndSet(i);              i = 3, counter = 4, counter2 = 4
    counter.incrementAndSet(counter2);        i = 3, counter = 5, counter2 = 5
}
```

Strings

Cosa stampa questo programma?

```
public class StringDemo {
 public static void main(String[] args){
  String s1 = “Guess who”;
  String s2 = s1;
  String s3 = “ABC”;
  s1 = s1 + “ is back”;
  s1 = s3;
  System.out.println(s1);
  System.out.println(s2);
 }
}
```

Strings are Immutable

Risposta:

> Guess who is back?

> Guess who

- `s2` punta a `s1`
- `s1` concatenato con un'altra stringa genera un nuovo oggetto
- `s2` continua a puntare all'oggetto precedente

Persons and Students

Definire le classi Person, Student e Grade

Specifiche:

- Una persona ha un nome, cognome e una data *(java.util.Date)*
- Uno studente è una persona con un id e una lista di voti
- Un voto contiene punteggio e crediti
- Lo studente espone due funzionalità:
  - Calcolo media pesata
  - Controllo se è possibile che si laurei (crediti totali >= 180)

Access Modifier

Completare il codice con gli opportuni modificatori di visibilità in modo tale che l'accesso alle variabili e ai metodi sia il più ristretto possibile ma che non crei errori di compilazione

Memento: Access Modifier

`public`	visibile da qualsiasi parte del programma
`private`	visibile solo dall'interno della classe stessa
`protected`	visibile solo dalle classi dello stesso package e dalle sottoclassi
`default`	visibile dallo stesso package e dalle sottoclassi se sono nello stesso pacchetto.

```
package a;
... class First {
  ... int x;
  ... int y;
  ... void h() { y = -1; }
}
... class Second extends First {
  ... void f(int x) { this.x = x; h(); }
}

package b;
imports a.*;
class Third {
  public static void main(String[] s) {
    Second z = new Second();
    z.f(3);
}

class Fourth extends First {   void g() { h(); }  }
```

```
package a;
public class First {
  int x;  // default
  private int y;
  protected void h() { y = -1; }
}
public class Second extends First {
  public void f(int x) { this.x = x; h(); }
}

package b;
imports a.*;
class Third {
  public static void main(String[] s) {
    Second z = new Second();
    z.f(3);
}
class Fourth extends First {   void g(void) { h(); }  }
```

Access Modifier (2)

Questo codice è corretto?
Che output produce una chiamata al metodo m1 di C2?

```
package A;
public class C1 {
    public void m1() { System.out.print(“ 1”); }
    protected void m2() { System.out.print(“ 2”); }
    private void m3() { System.out.print(“ 3”); }
}

package B;
import A.*;
public class C2 extends C1 {
    public void m1() { System.out.print("4"); m2(); m3(); }
    protected void m2() { System.out.print(“ 5”); }
    private void m3() { System.out.print(“6"); }
}
```

Access Modifier (2)

Risposta:

È corretta.

Una chiamata a m1 di C2 stampa “456"

C2 non vede la definizione di *m3* data da *C1*, perché questa è *private*. Pertanto la definizione di *m3* in *C2* è, banalmente, la definizione di un nuovo metodo. Inoltre *C2* ridefinisce i metodi *m1* e *m2* ereditati da *C1*.

Access Modifier (2)

E questa definizione di C2 è corretta? Perché?

```
public class C2 extends C1 {
    public void m1() {
      System.out.print("4");
      m2();
      m3();
   }
    protected void m2() {
      System.out.print(“ 5”);
    }
}
```

Access Modifier (2)

Risposta:

È scorretta.

Il metodo m3 è definito private nella classe C1 e pertanto non è visibile in C2. Si ottiene quindi un errore a compile-time.

Access Modifier (2)

E questa definizione di C2 è corretta? Perché?

```
public class C2 extends C1 {
    public void m1() {
      System.out.print("4");
      m2();
      m3();
   }
    private void m3() {
      System.out.print(“6”);
    }
}
```

Access Modifier (2)

Risposta:

È corretta.

Il metodo m3 è definito nella classe C2 e il metodo m2 è definito protected in C1 quindi visibile in C2. Una chiamata ad m1 stampa “4 2 6”

Static vs Dynamic Types

Sia dato il seguente frammento di codice.

Indicare gli errori a compile-time.

Eliminare le istruzioni che generano errore a compile-time, e dire se il codice genera errori a runtime.

Eliminare anche le istruzioni che generano errore a runtime, e dire cosa produce in output il programma.

Static vs Dynamic Types

```
package C;

public class C3 {
  public static void main(String[] s) {
    C1 c1;    C2 c2;  Object o;
    c1 = new C1();  /*1*/
    c1.m1();        /*2*/
    c2 = new C2();  /*3 */
    c2.m2();        /*4 */
    c1 = c2;        /*5 */
    c1.m1();        /*6 */
    c2 = new C1();  /*7 */
    o = new C1();   /*8 */
    c2 = (C2) o;    /*9 */
    o = new C2();   /*10 */
   c1 = (C1) o;    /*11 */
    c1.m1();        /*12 */
  }
}
```

```
package A;
public class C1 {
    public void m1() { // 1 }
    protected void m2() { // 2 }
    private void m3() { // 3 }
}

package B;
import A.*;
public class C2 extends C1 {
    public void m1() { // 4,5,6 }
    protected void m2() { // 5 }
    private void m3() { // 6 }
}
```

Static vs Dynamic Types

Risposta:

- **1**, **2**, **3** sono **corrette**. *Il costruttore di default non è definito nella classe, ma dal momento che nessun altro costruttore è definito può comunque essere usato. Il metodo m1 è public e quindi può essere usato da chi importa il package, quindi 3 è corretta produce in output "1", essendo C1 il tipo dinamico di c1.*
- **4** è **scorretta**. *Il metodo m2 è protected e C3 non è nello stesso package di C2 e non è neanche una sottoclasse di C2.*
- **5** e **6** sono **corrette**. *c2 conteneva un oggetto valido, e genera in output "456" essendo C2 il tipo dinamico.*
- **7** è **scorretta**. *Si assegna a c2 un oggetto il cui tipo dinamico è un sovra-tipo.*
- **8** e **10** sono **corrette.** C1 e C2 sono sottotipi di Object.
- **9** è **corretta** ma **genera un errore runtime.** *Il casting non può avere successo perché la variabile o, a runtime, riferisce un oggetto il cui tipo dinamico è C1, che è un sovra-tipo di C2, il tipo che viene indicato nell'operatore di casting.*
- **11** e **12** sono **corrette**. *L'ultima riga produce in output "456".*

Java Default Constructors

Cosa stampa questo programma?

E cosa stampa se viene eliminata la definizione del costruttore nella classe Padre?

```
class Padre {
    Padre() { System.out.println(“Padre!"); }
}

class Figlio extends Padre {
    Figlio() { System.out.println(“Figlio!"); }
}

class Example {
    public static void main(String[] args){
      Figlio p = new Figlio();
    }
}
```

Java Default Constructors

Risposta:

> Padre!
> Figlio!

Se si togliesse il costruttore del Padre stamperebbe solo "Figlio!"

Static vs Dynamic Types (2)

Quali sono le istruzioni scorrette nel metodo main?

Una volta eliminate tali istruzioni, cosa stampa il programma?

Qual è il tipo statico e dinamico di ciascuna delle tre variabili al termine dell'esecuzione del main?

Static vs Dynamic Types (2)

```
class Person {
  void greet() { System.out.println(“Arrivederci“);}
}
class EasyPerson extends Person {
  void greet() { System.out.println(“Ciao“); }
}
class FormalPerson extends Person {
  void greet() { System.out.println(“Saluti“); }
}
class VeryFormalPerson extends FormalPerson {
  void greet() { System.out.println(“Distinti saluti“); }
}
```

Static vs Dynamic Types (2)

```
class Example {
 public static void main(String[] args) {
   Person p = new Person();
   EasyPerson ep = new EasyPerson();
   FormalPerson fp = new FormalPerson();
   VeryFormalPerson vfp =new VeryFormalPerson();
   p.greet();                          //1
   ep = p;                             //2
   p = ep;                             //3
   p.greet();                          //4
   ep = fp;                            //5
   ep.greet();                         //6
   fp.greet();                         //7
   p = new FormalPerson();             //8
   p.greet();                          //9
   fp = p;                             //10
   vfp = (VeryFormalPerson) fp;        //11
   vfp.greet();                        //12
 }
}
```

Static vs Dynamic Types (2)

Risposta:

2, 5 e 10 sono **scorrette:** assegnamento di un sovra-tipo a un sotto-tipo o di un tipo non compatibile.
11 crea un **errore a runtime**: casting di un sovra-tipo verso una sottoclasse.

```
class Example {
 public static void main(String[] args) {
   Person p = new Person();
   EasyPerson ep = new EasyPerson();
   FormalPerson fp = new FormalPerson();
   VeryFormalPerson fpp=new VeryFormalPerson();
   p.greet();                          //1
   p = ep;                             //3
   p.greet();                          //4
   ep.greet();                         //6
   fp.greet();                         //7
   p = new FormalPerson();             //8
   p.greet();                          //9
   vfp = (VeryFormalPerson) fp;        //11
   vfp.greet();                        //12
 }
}
```

Static vs Dynamic Types (2)

Risposta:

> Arrivederci
> Ciao
> Ciao
> Saluti
> Saluti

A questo punto l'esecuzione dell'istruzione 11 solleva un'eccezione, dal momento che il tipo dinamico di fp non e' VeryFormalPerson, e il programma termina.

Hierachical Polygons

Definire una gerarchia di poligoni e sfruttare il poliformismo

***Specifiche**:*

- *Polygon* è una classe astratta che definisce il metodo astratto *getPerimeter()*
- *Polygon* implementa una funzione *printPerimeters()* che stampi il perimetro di un array di poligoni
- Implementare le sotto-classi di *Polygon Square, Rectangle* e *Triangle,* ognuna con la propria implementazione di *getPerimeter()*

Cron

Si progetti un package che offra un "demone temporale" simile a cron di Unix

***Specifiche**:*

- L'utente del package deve poter creare un demone, registrare presso di lui una serie di coppie <*orario, azione da compiere*>
- Il demone temporale, una volta avviato, deve eseguire le azioni registrate all'orario prestabilito.
- Si supponga che non si possano registrare più di 10 azioni, che ogni azione debba venir eseguita una volta soltanto e che una volta eseguite tutte le azioni cron termini la sua esecuzione.
- Si può interpretare l'orario di esecuzione come "orario indicativo": viene garantito che l'azione viene eseguita *dopo* l'orario specificato

Comparable Interface

Implementare la classe SortAlgorithms

***Specifiche**:*

- Un solo metodo statico *sort* che ordina un array di oggetti che implementano l'interfaccia *java.lang.Comparable*
- Person implementa *Comparable* controllando l'ordine del cognome e poi, in caso di omonimia, il nome. Student aggiunge a questo comportamento in caso di omonimia sia sul nome che sul cognome il controllo sull'id.

Esercizio: cosa stampa?

```
class Father {
  int x;
  public Father(int x) {
    this.x = x;
  }
  public int m(Father f) {
    return (f.x - this.x);
  }
}
class Son extends Father {
  int y;
  public Son(int x, int y) {
     super(x); this.y = y;
  }
  public int m(Father f) {
    return 100;
  }
  public int m(Son s) {
    return super.m(s) + (s.y - this.y);
  }
}
```

```
public class MainClass {
  public static void main(String args[]) {
    Father f1, f2;    Son s1, s2;
    f1 = new Father(3);
    f2 = new Son(3,10);
    System.out.println(f1.m(f2));          /* 1 */
    System.out.println(f2.m(f1));          /* 2 */
    s1 = new Son(4,21);
    System.out.println(s1.m(f1) + s1.m(f2)); /*3*
    System.out.println(f1.m(s1) + f2.m(s1)); /*4*
    s2 = new Son(5,22);
    System.out.println(s1.m(s2));          /* 5 */
  }
}
```

- Risposta: La classe Son definisce un metodo m(Father), che effettua un overriding del metodo m(Father) in Father, e un overloading di m, con signature m(Son).
- Istruzione 1:
  - Parte statica (overloading): f1 ha tipo statico Father -> il metodo viene cercato nella classe father. f2 ha tipo statico Father -> viene cercato un metodo la cui signature è compatibile con m(Father). Il metodo viene trovato, è proprio Father.m(Father), e occorre cercare tra i metodi che ne effettuano un overriding.
  - Parte dinamica (overriding): f1 ha tipo dinamico Father -> viene scelto il metodo Father.m(Father). Stampato f2.x - f1.x, ossia 0.
- Istruzione 2:
  - Parte statica (overloading): ancora, f1 e f2 hanno tipo statico Father. Quindi viene sempre scelto Father.m(Father) (o uno che ne fa overriding).
  - Parte dinamica (overriding): stavolta f2 ha tipo dinamico Son, e quindi viene scelto il metodo Son.m(Father), che effettua overriding. Viene stampato 100.

- System.out.println(s1.m(f1) + s1.m(f2)); /*3*/
- Istruzione 3:
  - Parte statica: le due chiamate hanno come tipo statico Son.m(Father). Quindi viene scelto questo metodo, o un metodo che ne fa override...
  - Parte dinamica: ...ma nessun metodo fa override di Son.m(Father), quindi per entrambe le chiamate viene eseguito questo. Notare che, nonostante f2 abbia tipo dinamico Son, s.m(f2) NON esegue Son.m(Son)!!! Viene stampato 200.
- System.out.println(f1.m(s1) + f2.m(s1)); /*4*/
- Istruzione 4:
  - Parte statica: le due chiamate hanno come tipo statico Father.m(Son). Non esiste un metodo con questa signature, ma Father.m(Father) è compatibile. Viene scelto quindi Father.m(Father), o un metodo che ne fa override.
  - Parte dinamica: dal momento che f1 e f2 hanno diversi tipi dinamici, la prima chiamata usa il metodo Father.m(Father), la seconda usa il metodo overridden Son.m(Father).

    Il risultato è 1 + 100 = 101.

- Istruzione 5:
  - Statico è Son.m(Son), e non ci sono metodi che ne fanno overriding. All'interno, viene effettuata una chiamata di super.m(s), con s parametro il cui tipo statico è Son; super significa "della superclasse statica" - quindi di Father. Staticamente, questo significa cercare Father.m(Son), che non esiste: Però esiste Father.m(Father), che è compatibile. A runtime viene invocato questo. Quindi, super.m(s) restituisce 1, e l'istruzione 5 stampa 2 a schermo.

Collections and Generics

```
List<Grade> grades = new ArrayList<Grade>()
```

- Questo è un esempio dell'uso dei *generics*, classi il cui tipo è parametrico rispetto ad altri tipi.
- List è una interfaccia, per creare un oggetto ci serve una classe concreta. Nell'esempio ArrayList

Stack

Implementare la classe Stack con i Generics