

Ingegneria del Software

Esercitazione 2

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)

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.

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 `c2` ridefinisce i metodi `m1` e `m2` ereditati da `c1`. Inoltre questa è *private*. Pertanto la definizione di `m3` in `c2` è, banalmente, la definizione di un nuovo metodo.

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”

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

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”.*

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 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"); }  
}
```

```
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.

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.

Hierarchical 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()*

Runtime Type Checking

Cosa stampa questo programma?

```
class Father { }  
class Son extends Father { }  
class Test {  
    public static void main(String[] s) {  
        Father f = new Son();  
        Father f2 = new Father();  
        if (f instanceof Father)  
            System.out.println("True");  
        else  
            System.out.println("False");  
        if (f.getClass() == f2.getClass())  
            System.out.println("True");  
        else  
            System.out.println("False");  
    }  
}
```

Runtime Type Checking

Risposta:

> True

> False

instanceof restituisce true se c'è compatibilità di assegnamento.

getClass() ritorna una reference alla (unica) istanza di tipo *Class* della classe dell'oggetto su cui è chiamato

Esercizio: cosa stampa

```
class Bar implements Cloneable { int z; }
```

```
class Foo implements Cloneable {  
    int x; String y; Bar w;  
    public String toString() {  
        String s="x: " + x;  
        s += ", y: " + y;  
        s += ", w.z: " + w.z;  
        return s;  
    }  
    Foo cloneMe() {  
        try {return (Foo) this.clone();}  
        catch (Exception e) { return null; }  
    }  
}
```

```
class Main {  
    public static void main(String args[]) {  
        Foo a,b;  
        a = new Foo();  
        a.x = 5;  
        a.y = "abc";  
        a.w = new Bar();  
        a.w.z = 11;  
        b = a.cloneMe();  
        b.x = 11;  
        b.y = "def";  
        b.w.z = 42;  
        System.out.println(a);  
    }  
}
```

- Risposta: stampa x: 5, y: abc, w.z: 42. Clone effettua una "shallow copy": clona il contenuto di tutti i campi dell'oggetto, e se un campo è un riferimento copia il riferimento (non clona l'oggetto al quale si riferisce).

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

String Literals

*Illustrare l'effetto delle istruzioni in rosso sullo Heap.
Che uguaglianza c'è tra le quattro String?*

```
class Example1 {  
    public static void main(String args[]) {  
        String s1 = "abc";  
        String s2 = "abc";  
        String s3 = new String("abc");  
        String s4 = s3;  
    }  
}
```

String Literals

Risposta:

s1 e s2 puntano allo stesso oggetto stringa nello heap

s3 punta ad un oggetto diverso nello heap, al quale punta anche s4

s1 == s2

s3 == s4

s1.equals(s2)

s3.equals(s4)

s1.equals(s3)

...

Valutazione Parametri

Illustrare e motivare il valore delle variabili i, counter e counter2 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);  
}
```


Valutazione Parametri

Tipi primitivi: passati per valore

Oggetti: passati per referenza

Risposta:

```
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 = 0

counter = 1

i = 3

i = 3, counter = 4

i = 3, counter = 4, counter2 = 0

i = 3, counter = 4, counter2 = 4

i = 3, counter = 5, counter2 = 5

Linguaggio Procedurale vs OOP

Consideriamo una ipotetica implementazione in C, o in un qualsiasi linguaggio procedurale, di un tipo di dato qualsiasi, ad esempio la seguente per i punti nello spazio. Quale differenza sostanziale esiste con un linguaggio ad oggetti come Java?

```
/* un nuovo tipo per la struttura dati */  
typedef struct { float x,y; } *planepoint;
```

```
/* le operazioni che manipolano le variabili di tipo planepoint */  
void init(planepoint *this, float x, float y) { this->x =x; this->y = y; }
```

```
void translate(planepoint *this, float deltax, float deltay) ...
```

```
float distance(planepoint *this, planepoint *another) ...
```

Linguaggio Procedurale vs OOP

In un linguaggio procedurale le operazioni agiscono su un tipo hanno almeno un parametro di tale tipo. In Java le dichiarazioni delle funzioni vengano inserite nella dichiarazione del tipo e per tanto si chiamano metodi.

/ dichiarazione del tipo E delle operazioni che manipolano le sue variabili */*

```
class PlanePoint {
```

/ i dati di un oggetto di tipo PlanePoint */*

```
float x,y;
```

/ le operazioni che lo manipolano */*

```
PlanePoint(float x, float y) { this.x = x; this.y = y; }
```

```
void translate(float deltax, float deltax) ...
```

```
float distance(PlanePoint another) ...
```

```
}
```

Linguaggio Procedurale vs OOP

In altre parole:

- Il parametro `this` viene automaticamente passato da Java alle tre operazioni, e quindi non va dichiarato (non c'è bisogno!)
- L'operazione che inizializza un nuovo oggetto di un tipo, il costruttore, viene automaticamente invocata da Java non appena viene creato l'oggetto mantenendolo sempre in uno stato consistente.