Esonero del corso di Programmazione a Oggetti

Roma, 9 novembre 2006

Soluzioni

Domanda 1 L'istruzione `p.bacia(sec)` stamperà
☑ `giammai!`
☐ `se mi piace!`
☐ `subito!`

Domanda 2 L'istruzione `p.bacia(g)` stamperà
☐ `giammai!`
☐ `se mi piace!`
☑ `subito!`

Domanda 3 L'istruzione `p.bacia(ing)` stamperà
☐ `giammai!`
☑ `se mi piace!`[1]
☐ Niente, perché non ci sono versioni del metodo `bacia` con parametro di tipo `StudenteIngegneria`

Domanda 4 L'istruzione `p.bacia((Studente) sec)` stamperà
☐ `giammai!`
☑ `se mi piace!`[2]
☐ `subito!`

[1] osservate che viene scelto il metodo che *approssima meglio* la chiamata (*best match*): sarà quello che ha come parametro il più piccolo supertipo. Non sarebbe viceversa safe chiamare un metodo che ha un parametro formale dichiarato sottotipo del tipo del parametro attuale.

[2] osservate che la scelta di un metodo overloaded avviene sulla base del tipo statico del parametro. Osservate anche che l'upcast viene sempre considerato safe e quindi non viene controllato a tempo di esecuzione se `sec` sia effettivamente uno `Studente`. Se ne deduce che una `Pupa` considera l'ipotesi di baciare un `Secchione`, solo se questo maschera la sua vera natura!

Domanda 5 L'istruzione `((Studente) p).bacia(sec)` causerà
☑ Un errore in fase di compilazione, perché `Studente` non ha relazioni di sottotipaggio con `Pupa`[3]
☐ Un errore in fase di esecuzione perché il metodo `bacia()` non è definito nella classe `Studente`
☐ Stamperà `se mi piace!`

Domanda 6 L'istruzione `p.bacia((Goliarda) s_sun)` causerà:
☐ Un errore in fase di compilazione, perché `Goliarda` non ha relazioni di sottotipaggio con `Studente`
☑ Un errore in fase di esecuzione perché `s_sun` non punta ad un oggetto di tipo `Goliarda`[4]
☐ Stamperà `subito!`

Domanda 7 L'istruzione `p.bacia((Goliarda) sec)` causerà:
☑ Un errore in fase di compilazione, perché `Goliarda` non ha relazioni di sottotipaggio con `Secchione`[5]
☐ Un errore in fase di esecuzione perché `sec` non punta ad un oggetto istanziato dalla classe `Goliarda`
☐ Stamperà `subito!`

Domanda 8 L'istruzione `ps.bacia(sec)` stamperà
☑ `giammai!`[6]
☐ `fossi matta`
☐ `si puo' fare`

Domanda 9 L'istruzione `ps.bacia(s_sec)` stamperà
☐ `giammai!`
☑ `fossi matta`[7]
☐ `si puo' fare`

[3] i cast si possono fare solo tra tipi che stanno nella stessa gerarchia, e questo viene verificato staticamente. Insomma, un `Secchione` potrà fingersi uno `Studente`, ma non riuscira' a farsi passare per un `Goliarda` o per una `Pupa`.

[4] viene sollevata un'eccezione `ClassCastException`, perché il compilatore genera codice per verificare *dinamicamente* la correttenza del downcast. Posso fare il downcast a patto che il tipo dinamico dell'oggetto sia un sottotipo del tipo nel cast.

[5] come per la domanda 5

[6] osservate che il metodo `void bacia(Studente s)` della classe `PupaSecchiona` non fa overriding del metodo `bacia(Secchione s)` della classe `Pupa`: tale metodo viene quindi ereditato dalla classe `PupaSecchiona`.

[7] riflettete qui sul fatto che l'overloading viene risolto *staticamente* sulla base del tipo dei parametri... se preferite ciascun metodo viene identificato da nome e tipi dei parametri, poi dinamicamente sulla base di chi è l'oggetto ricevente si sceglierà la versione più specializzata.

Domanda 10 L'istruzione `ps.bacia(ing)` stamperà
☐ `se mi piace!`
☑ `fossi matta`[8]
☐ `si puo' fare`
Domanda 11 L'istruzione `ps.bacia(sun)` stamperà
☐ `se mi piace!`
☐ `fossi matta`
☑ `si puo' fare`[9]

Domanda 12 L'istruzione `ps.bacia(s_g)` stamperà
☑ `non sei serio`[10]
☐ `fossi matta`
☐ `subito!`

Domanda 13 L'istruzione `ps.bacia(g)` stamperà
☐ `non sei serio`
☐ `fossi matta`
☑ `subito!`[11]

Domanda 14 L'istruzione `ps.bacia((Goliarda) sun)` causerà:
☑ Un errore in fase di esecuzione perché `sun` non punta ad un oggetto istanziato dalla classe `Goliarda`[12]
☐ Un errore in fase di compilazione, perché `Goliarda` non ha relazioni di sottotipaggio con `StudenteUniversitario`
☐ Stamperà `subito!`

Domanda 15 Dire quali delle seguenti affermazioni è falsa:
☐ `Secchione` è sottotipo di `Studente`[13]
☐ `Secchione` è sottotipo di `StudenteIngegneria`
☑ `Goliarda` è sottotipo di `Secchione`

[8] Osservate qui che se `B<:A` e `b` oggetto di tipo dinamico `B`, allora `b instanceof A` dà come risultato `true`. Questo comportamento è corretto in quanto si mantiene il codice flessibile ad adattarsi a future nuove sottoclassi e coerente col tipico uso di `instaceof`: la verifica del tipo dinamico si fa per assicurarsi che le invocazioni su quell'oggetto di metodi appartenenti all'interfaccia `A` non diano errori di `message not understood`: ma tali metodi saranno presenti anche in tutti i sottotipi di `A`.

[9] infatti `StudenteUniversitario` (tipo dinamico di `sun` non è sottotipo ne di `StudenteIngegneria`, ne di di `Goliarda`.

[10] come domanda 9.

[11] come domanda 8.

[12] come per domanda 4.

[13] per chiusura transitiva della relazione di sottotipaggio.

Domanda 16 L'istruzione `s_g.canta()` stamperà:
☐ `Dottore, dottore...`
☑ Niente, perché darà un errore a tempo di compilazione[14]
☐ Niente, perché darà un errore a tempo di esecuzione
☐ Niente, ma il programma compila ed esegue correttamente

Domanda 17 L'istruzione `((Goliarda)s_g).canta()` stamperà:
☑ `Dottore, dottore...`[15]
☐ Niente, perché darà un errore a tempo di compilazione
☐ Niente, perché darà un errore a tempo di esecuzione
☐ Niente, ma il programma compila ed esegue correttamente

Domanda 18 L'istruzione `g.canta()` stamperà:
☑ `Dottore, dottore...`
☐ Niente, perché darà un errore a tempo di compilazione
☐ Niente, perché darà un errore a tempo di esecuzione
☐ Niente, ma il programma compila ed esegue correttamente

Domanda 19 L'istruzione `if (sec istanceof Goliarda) ((Goliarda) sec).canta()` stamperà:
☐ `Dottore, dottore...`
☑ Niente, perché darà un errore a tempo di compilazione[16]
☐ Niente, perché darà un errore a tempo di esecuzione
☐ Niente, ma il programma compila ed esegue correttamente

Domanda 20 L'istruzione `if (s_sec istanceof Goliarda) ((Goliarda) s_sec).canta();` stamperà:
☐ `Dottore, dottore...`
☐ Niente, perché darà un errore a tempo di compilazione
☐ Niente, perché darà un errore a tempo di esecuzione
☑ Niente, ma il programma compila ed esegue correttamente[17]

[14] infatti staticamente `s_g` è un semplice `Studente`, tipo che non ha il metodo `canta()` nella sua interfaccia.

[15] osservate perché il downcast in questo caso sia legale staticamente e dinamicamente.

[16] come per la domanda 5. In realtà anche l'`instanceof` dà errore se il tipo statico della variabile non è nella stessa gerarchia: del resto tale controllo non potrebbe, per come è strutturato il sistema dei tipi di Java, dare mai esito positivo (i tipi dinamici sono *sempre* sottotipi dei tipi statici

[17] si tratta di un caso di buon uso combinato di `instanceof` e downcast: il downcast è correttamente protetto da una verifica dinamica del tipo dell'oggetto

Domanda 21 L'istruzione `if (g.laureato) P.print("congratulazioni");` stamperà:
☐ `congratulazioni`
☑ Niente, perché darà un errore a tempo di compilazione[18]
☐ Niente, perché darà un errore a tempo di esecuzione
☐ Niente, ma il programma compila ed esegue correttamente

Domanda 22 L'istruzione `p.canta();` stamperà:
☐ `Dottore, dottore...`
☑ `Boys, boys, boys...`
☐ Niente, perché darà un errore a tempo di esecuzione
☐ Niente, perché darà un errore a tempo di compilazione

Domanda 23 L'istruzione `p.canta(g);` stamperà:
☑ `Dottore, dottore...`
☐ `Boys, boys, boys...`
☐ Niente, perché darà un errore a tempo di esecuzione
☐ Niente, perché darà un errore a tempo di compilazione

Domanda 24 L'istruzione `p.canta(ps);` stamperà:
☐ `Dottore, dottore...`
☐ `Boys, boys, boys...`
☐ Niente, perché darà un errore a tempo di esecuzione
☑ Niente, perché darà un errore a tempo di compilazione[19]

Domanda 25 L'istruzione `p.canta(sun);` stamperà:
☐ `Dottore, dottore...`
☐ `Boys, boys, boys...`
☑ Niente, perché darà un errore a tempo di esecuzione[20]
☐ Niente, perché darà un errore a tempo di compilazione

[18] `laureato` infatti è una variabile di istanza dichiarata `protected` e quindi non accessibile fuori dallo scope della classe `Studente` e relative sottoclassi

[19] infatti non esiste nessun metodo `void canta(Pupa)`.

[20] ancora una volta è il downcast fatto senza precauzioni che genera l'errore a tempo di esecuzione, mentre la chiamata è staticamente ben tipata, perchè `StudenteUniversitario <: Studente`.

Domanda 26 L'istruzione `g.sostieneEsame(26);` stamperà:
☐ `rifiuto!`
☑ `rifiuto!`
`Sono un genio!  Beviamoci su!`[21]
☐ `Mi hanno dato 26`

Domanda 27 L'istruzione `s_g.sostieneEsame(23);` stamperà:
☐ `rifiuto!`
☐ `rifiuto!`
`Sono un genio!  Beviamoci su!`
☑ `Mi hanno dato 23`
`Sono un genio!  Beviamoci su!`
☐ Niente, perché darà un errore a tempo di compilazione

Domanda 28 L'istruzione `s_sec.sostieneEsame(28);` stamperà:
☑ `rifiuto!`
☐ `Mi hanno dato 28`
☐ Niente, perché darà un errore a tempo di compilazione

[21] il dynamic method lookup farà si che anche se il metodo `sostieneEsame` nella classe `Goliarda` è ereditato da `Studente`, verrà messo in esecuzione il metodo `rifiuta()` di `Goliarda`, che inverte il risultato ottenuto dal metodo `rifiuta()` di `Studente`. Stesso dicasi per le prossime 3 domande.