

settembre 2023

```
public class P {
    String m(Number... o) { return "P.m(Number...)"; }
    String m(Object... o) { return "P.m(Object...)"; }
}

public class H extends P {
    String m(String s) { return super.m(s) + " H.m(String)"; }
    String m(Double d1, Double d2) { return super.m(d1, d2) + " H.m(Double,Double)"; }
}

public class Test {
    public static void main(String[] args) {
        P p = new P();
        H h = new H();
        P p2 = h;
        System.out.println(...);
    }
}
```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato, assumendo che tutte le classi siano dichiarate nello stesso package.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

- (a) `p.m("42")`
- (b) `p2.m("42")`
- (c) `h.m("42")`
- (d) `p.m(42, 0)`
- (e) `p2.m(42, 0)`
- (f) `h.m(42, 0)`

luglio 2019

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

(a) Il tipo statico di `p` è `P`, il literal `42` ha tipo statico `int`.

- primo tentativo (solo sottotipo): poiché `int` \leq `Number` e `int` \leq `Number[]` (al primo e al secondo tentativo `Number...` viene trattato come `Number[]`), non esistono metodi di `P` accessibili e applicabili per sottotipo;

- secondo tentativo (boxing/unboxing e sottotipo): dopo aver applicato una conversione boxing da `int` a `Integer`, poiché `Integer` \leq `Number` e `Integer` \leq `Number[]`, solo il metodo `m(Number)` di `P` è applicabile per boxing e reference widening.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Number)` in `P` e viene stampata la stringa "`P.m(Number)`".

(b) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, ma poiché il metodo con segnatura `m(Number)` non è ridefinito in `H`, viene eseguito lo stesso metodo del punto precedente e, quindi, viene stampata la stringa "`P.m(Number)`".

(c) Il tipo statico di `h` è `H` e l'argomento ha tipo statico `int` come ai punti precedenti.

- primo tentativo (solo sottotipo): nessun metodo accessibile definito in `H` o ereditato da `P` è applicabile per sottotipo (`int` \leq `Number`, `int` \leq `Number[]`, `int` \leq `Integer`);
- secondo tentativo (boxing/unboxing e sottotipo): dopo aver applicato una conversione boxing da `int` a `Integer`, poiché `Integer` \leq `Number`, `Integer` \leq `Integer` \leq `Number[]`, solo i metodi `m(Number)` e `m(Integer)` sono applicabili per boxing e reference widening (quest'ultimo richiesto solo nel caso `m(Number)`) e poiché `Integer` \leq `Number`, l'overloading viene risolto con la segnatura più specifica `m(Integer)`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo di `H` con segnatura `m(Integer)`; poiché il parametro `i` ha tipo statico `Integer` e `super` si riferisce alla classe `P`, la chiamata `super.m(i)` si comporta analogamente al caso illustrato al punto (a); viene quindi stampata la stringa "`P.m(Number) H.m(Integer)`".

(d) Il literal `42` ha tipo statico `int` e il tipo statico di `p` è `P`.

- primo tentativo (solo sottotipo): poiché nessun metodo ha due parametri (al primo e al secondo tentativo `m(Number... o)` viene trattato come metodo con un solo parametro), non esistono metodi di `P` accessibili e applicabili per sottotipo;
- secondo tentativo (boxing/unboxing e sottotipo): continuano a non esistere metodi di `P` accessibili e applicabili, anche dopo una conversione boxing da `int` a `Integer`, poiché nessun metodo ha due parametri;
- terzo tentativo (metodi con numero variabile di parametri): `m(Number... o)` viene trattato come metodo con numero variabile di parametri di tipo `Number`; dopo una conversione boxing da `int` a `Integer`, poiché `Integer` \leq `Number`, il metodo `m(Number... o)` è l'unico applicabile per boxing e reference widening.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Number...)` in `P` e viene stampata la stringa "`P.m(Number...)`".

(e) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, ma poiché il metodo con segnatura `m(Number...)` non è ridefinito in `H`, viene eseguito lo stesso metodo del punto precedente e, quindi, viene stampata la stringa "`P.m(Number...)`".

(f) Il literal `42` ha tipo statico `int` e il tipo statico di `h` è `H`.

- primo tentativo (solo sottotipo): nessun metodo accessibile definito in `H` o ereditato da `P` è applicabile per sottotipo dato che l'unico metodo con due parametri (al primo e al secondo tentativo `m(Number... o)` viene trattato come metodo con un solo parametro) ha segnatura `m(Integer, Integer)` e `int` $\not\leq$ `Integer`;
- secondo tentativo (boxing/unboxing e sottotipo): dopo una conversione boxing da `int` a `Integer`, `m(Integer, Integer)` è il solo metodo applicabile per boxing e sottotipo.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo di `H` con segnatura `m(Integer, Integer)`; poiché i parametri `i1` e `i2` hanno tipo statico `Integer` e `super` si riferisce alla classe `P`, la chiamata `super.m(i1, i2)` si comporta analogamente al caso illustrato al punto (d); viene quindi stampata la stringa "`P.m(Number...) H.m(Integer, Integer)`".

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, tutti i metodi sono accessibili.

(a) Il tipo statico di `p` è `P`, il literal "`42`" ha tipo statico `String`.

- prima fase (solo sottotipo): poiché `String` $\not\leq$ `Number[]` e `String` $\not\leq$ `Object[]` (nella prima e seconda fase `Number...` e `Object...` vengono considerati come i tipi `array Number[]` e `Object[]`), non esistono metodi di `P` applicabili per sottotipo;
- seconda fase (boxing/unboxing e sottotipo): nessuna conversione boxing/unboxing è applicabile al tipo `String`, quindi anche in questo caso non esistono metodi applicabili;
- terza fase (arità variabile, boxing/unboxing e sottotipo): è corretto considerare che i due metodi abbiano un solo parametro di tipo `Number` e `Object`; poiché `String` $\not\leq$ `Number`, ma `String` \leq `Object`, solo il metodo `m(Object...)` di `P` è applicabile per boxing e reference widening.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Object...)` in `P` e viene stampata la stringa "`P.m(Object...)`".

(b) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, ma poiché il metodo con segnatura `m(Object...)` non è ridefinito in `H`, viene eseguito lo stesso metodo del punto precedente e, quindi, viene stampata la stringa "`P.m(Object...)`".

(c) Il tipo statico di `h` è `H` mentre l'argomento ha sempre tipo statico `String`.

- prima fase (solo sottotipo): il metodo con due parametri non è applicabile a un solo argomento; inoltre, poiché `String` $\not\leq$ `Number[]`, `String` $\not\leq$ `Object[]` (nella prima e seconda fase `Number...` e `Object...` vengono considerati come i tipi `array Number[]` e `Object[]`) e `String` \leq `String`, l'unico metodo di `H` applicabile per sottotipo ha segnatura `m(String)`;

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo di `H` con segnatura `m(String)`; poiché il parametro `s` ha tipo statico `String` e `super` si riferisce alla classe `P`, la chiamata `super.m(s)` si comporta come al punto (a); viene quindi stampata la stringa "`P.m(Object...) H.m(String)`".

(d) Il literal `42.0` ha tipo statico `double` e il tipo statico di `p` è `P`.

- prima fase (solo sottotipo): poiché `double` $\not\leq$ `Number[]` e `double` $\not\leq$ `Object[]` (nella prima e seconda fase `Number...` e `Object...` vengono considerati come i tipi `array Number[]` e `Object[]`), non esistono metodi di `P` applicabili per sottotipo;

- seconda fase (boxing/unboxing e sottotipo): il tipo `double` è convertibile a `Double` per boxing, ma poiché `Double` $\not\leq$ `Number[]` e `Double` $\not\leq$ `Object[]`, non esistono metodi di `P` applicabili;
- terza fase (arità variabile, boxing/unboxing e sottotipo): è corretto considerare che i due metodi abbiano un solo parametro di tipo `Number` e `Object`; poiché `Double` \leq `Number` e `Double` \leq `Object`, entrambi i metodi sono applicabili, ma dato che `Number` \leq `Object`, il metodo con segnatura `m(Number...)` è più specifico.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Number...)` in `P` e viene stampata la stringa "`P.m(Number...)`".

(e) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, ma poiché il metodo con segnatura `m(Number...)` non è ridefinito in `H`, viene eseguito lo stesso metodo del punto precedente e, quindi, viene stampata la stringa "`P.m(Number...)`".

(f) Il literal `42.0` ha tipo statico `double` e il tipo statico di `h` è `H`. I due metodi in `H` non sono comunque applicabili, quindi la risoluzione della chiamata procede come nei due punti precedenti; infatti, il metodo con segnatura `m(String)` non è applicabile perché non è possibile convertire un argomento da `double` a `String`, mentre quello con segnatura `m(Double, Double)` non è applicabile a un solo argomento perché ha due parametri.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi come al punto precedente viene eseguito il metodo con segnatura `m(Number...)` in `P` e viene stampata la stringa "`P.m(Number...)`".

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
    String m(Number... o) { return "P.m(Number...)"; }
    String m(Number o) { return "P.m(Number)"; }
}

public class H extends P {
    String m(Integer i) { return super.m(i) + " H.m(Integer)"; }
    String m(Integer i1, Integer i2) { return super.m(i1, i2) + " H.m(Integer, Integer)"; }
}

public class Test {
    public static void main(String[] args) {
        P p = new P();
        H h = new H();
        P p2 = h;
        System.out.println(...);
    }
}
```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato, assumendo che tutte le classi siano dichiarate nello stesso package.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

(a) `p.m(42)`

(b) `p2.m(42)`

(c) `h.m(42)`

(d) `p.m(42, 42)`

(e) `p2.m(42, 42)`

(f) `h.m(42, 42)`

giugno 2019

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
    String m(Number... o) {
        return "P.m(Number...)";
    }
    String m(Number o) {
        return "P.m(Number)";
    }
}
public class H extends P {
    String m(Short s) {
        return super.m(s) + " H.m(Short)";
    }
    String m(Float f) {
        return super.m(f) + " H.m(Float)";
    }
}
public class Test {
    public static void main(String[] args) {
        P p = new P();
        H h = new H();
        P p2 = h;
        System.out.println(...);
    }
}
```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato, assumendo che tutte le classi siano dichiarate nello stesso package.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

- (a) `p.m((short) 42)`
- (b) `p2.m((short) 42)`
- (c) `h.m((short) 42)`
- (d) `p.m(42.0f)`
- (e) `p2.m(42.0f)`
- (f) `h.m(42.0f)`

giugno 2019

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

(a) Il literal 42 ha tipo statico `int` e il tipo statico di `p` è `P`.

- primo tentativo (solo sottotipo): poiché `int` $\not\leq$ `Object` e `int` $\not\leq$ `Object[]` (al primo e al secondo tentativo `Object...` viene trattato come `Object[]`), non esistono metodi di `P` accessibili e applicabili per sottotipo;
- secondo tentativo (boxing/unboxing e sottotipo): applicando una conversione boxing da `int` a `Integer` e poiché `Integer` \leq `Object`, `Integer` \leq `Object[]`, solo il metodo `m(Object)` di `P` è applicabile per boxing e reference widening.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Object)` in `P` e viene stampata la stringa "`P.m(Object)`".

(b) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, ma poiché il metodo con segnatura `m(Object)` non è ridefinito in `H`, viene eseguito lo stesso metodo del punto precedente `e`, quindi, viene stampata la stringa "`P.m(Object)`".

(c) Il literal 42 ha tipo statico `int` e il tipo statico di `h` è `H`.

- primo tentativo (solo sottotipo): nessun metodo accessibile definito in `H` o ereditato da `P` è applicabile per sottotipo (`int` $\not\leq$ `Object`, `int` $\not\leq$ `Object[]`, `int` $\not\leq$ `Integer`, `int` $\not\leq$ `Double`);
- secondo tentativo (boxing/unboxing e sottotipo): applicando una conversione boxing da `int` a `Integer` e poiché `Integer` \leq `Object`, `Integer` \leq `Object[]`, `Integer` $\not\leq$ `Object`, solo i metodi `m(Object)` e `m(Integer)` sono applicabili per boxing e reference widening (quest'ultimo richiesto solo nel caso `m(Object)`) e poiché `Integer` \leq `Object`, l'overloading viene risolto con la segnatura più specifica `m(Integer)`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo di `H` con segnatura `m(Integer)`; poiché il parametro `i` ha tipo statico `Integer` e `super` si riferisce alla classe `P`, semantica statica e dinamica della chiamata `super.m(i)` coincidono con il caso illustrato al punto (a); viene quindi stampata la stringa "`P.m(Object) H.m(Integer)`".

(d) Il literal 42.0 ha tipo statico `double` e il tipo statico di `p` è `P`.

- primo tentativo (solo sottotipo): poiché `double` $\not\leq$ `Object` e `double` $\not\leq$ `Object[]` (al primo e al secondo tentativo `Object...` viene trattato come `Object[]`), non esistono metodi di `P` accessibili e applicabili per sottotipo;
- secondo tentativo (boxing/unboxing e sottotipo): applicando una conversione boxing da `double` a `Double` e poiché `Double` \leq `Object`, `Double` \leq `Object[]`, solo il metodo `m(Object)` di `P` è applicabile per boxing e reference widening.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Object)` in `P` e viene stampata la stringa "`P.m(Object)`".

(e) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, ma poiché il metodo con segnatura `m(Object)` non è ridefinito in `H`, viene eseguito lo stesso metodo del punto precedente `e`, quindi, viene stampata la stringa "`P.m(Object)`".

(f) Il literal 42.0 ha tipo statico `double` e il tipo statico di `h` è `H`.

- primo tentativo (solo sottotipo): nessun metodo accessibile definito in `H` o ereditato da `P` è applicabile per sottotipo (`double` $\not\leq$ `Object`, `double` $\not\leq$ `Object[]`, `double` $\not\leq$ `Integer`, `double` $\not\leq$ `Double`);
- secondo tentativo (boxing/unboxing e sottotipo): applicando una conversione boxing da `double` a `Double` e poiché `Double` \leq `Object`, `Double` \leq `Object[]`, `Double` $\not\leq$ `Integer`, solo i metodi `m(Object)` e `m(Double)` sono applicabili per boxing e reference widening (quest'ultimo richiesto solo nel caso `m(Object)`) e poiché `Double` \leq `Object`, l'overloading viene risolto con la segnatura più specifica `m(Double)`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo di `H` con segnatura `m(Double)`; poiché il parametro `d` ha tipo statico `Double` e `super` si riferisce alla classe `P`, semantica statica e dinamica della chiamata `super.m(d)` coincidono con il caso illustrato al punto (d); viene quindi stampata la stringa "`P.m(Object) H.m(Double)`".

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

(a) Il tipo statico di `p` è `P`, il literal 42 ha tipo statico `int`, il cast a `short` è staticamente corretto (narrowing primitive conversion) e l'argomento (`short`) 42 ha tipo statico `short`.

- primo tentativo (solo sottotipo): poiché `short` $\not\leq$ `Number` e `short` $\not\leq$ `Number[]` (al primo e al secondo tentativo `Number...` viene trattato come `Number[]`), non esistono metodi di `P` accessibili e applicabili per sottotipo;
- secondo tentativo (boxing/unboxing e sottotipo): dopo aver applicato una conversione boxing da `short` a `Short`, poiché `Short` \leq `Number` e `Short` $\not\leq$ `Number[]`, solo il metodo `m(Number)` di `P` è applicabile per boxing e reference widening.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Number)` in `P` e viene stampata la stringa "`P.m(Number)`".

(b) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, ma poiché il metodo con segnatura `m(Number)` non è ridefinito in `H`, viene eseguito lo stesso metodo del punto precedente `e`, quindi, viene stampata la stringa "`P.m(Number)`".

(c) Il tipo statico di `h` è `H` e l'argomento ha tipo statico `short` come ai punti precedenti.

- primo tentativo (solo sottotipo): nessun metodo accessibile definito in `H` o ereditato da `P` è applicabile per sottotipo (`short` $\not\leq$ `Number`, `short` $\not\leq$ `Number[]`, `short` $\not\leq$ `Short`, `short` $\not\leq$ `Float`);
- secondo tentativo (boxing/unboxing e sottotipo): dopo aver applicato una conversione boxing da `short` a `Short`, poiché `Short` \leq `Number`, `Short` \leq `Short` e `Short` $\not\leq$ `Number[]`, `Short` $\not\leq$ `Float`, solo i metodi `m(Number)` e `m(Short)` sono applicabili per boxing e reference widening (quest'ultimo richiesto solo nel caso `m(Number)`) e poiché `Short` \leq `Number`, l'overloading viene risolto con la segnatura più specifica `m(Short)`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo di `H` con segnatura `m(Short)`; poiché il parametro `s` ha tipo statico `Short` e `super` si riferisce alla classe `P`, la chiamata `super.m(s)` si comporta analogamente al caso illustrato al punto (a); viene quindi stampata la stringa "`P.m(Number) H.m(Short)`".

(d) Il literal 42.0 ha tipo statico `float` e il tipo statico di `p` è `P`.

- primo tentativo (solo sottotipo): poiché `float` $\not\leq$ `Object` e `float` $\not\leq$ `Number[]`, non esistono metodi di `P` accessibili e applicabili per sottotipo;
- secondo tentativo (boxing/unboxing e sottotipo): dopo aver applicato una conversione boxing da `float` a `Float`, poiché `Float` \leq `Number`, `Float` \leq `Float` e `Float` $\not\leq$ `Number[]`, solo il metodo `m(Number)` di `P` è applicabile per boxing e reference widening.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Number)` in `P` e viene stampata la stringa "`P.m(Number)`".

(e) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, ma poiché il metodo con segnatura `m(Number)` non è ridefinito in `H`, viene eseguito lo stesso metodo del punto precedente `e`, quindi, viene stampata la stringa "`P.m(Number)`".

(f) Il literal 42.0 ha tipo statico `float` e il tipo statico di `h` è `H`.

- primo tentativo (solo sottotipo): nessun metodo accessibile definito in `H` o ereditato da `P` è applicabile per sottotipo (`float` $\not\leq$ `Number`, `float` $\not\leq$ `Number[]`, `float` $\not\leq$ `Short`, `float` $\not\leq$ `Float`);
- secondo tentativo (boxing/unboxing e sottotipo): dopo aver applicato una conversione boxing da `float` a `Float`, poiché `Float` \leq `Number`, `Float` \leq `Float` e `Float` $\not\leq$ `Number[]`, solo i metodi `m(Number)` e `m(Float)` sono applicabili per boxing e reference widening (quest'ultimo richiesto solo nel caso `m(Number)`) e poiché `Float` \leq `Number`, l'overloading viene risolto con la segnatura più specifica `m(Float)`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo di `H` con segnatura `m(Float)`; poiché il parametro `f` ha tipo statico `Float` e `super` si riferisce alla classe `P`, la chiamata `super.m(f)` si comporta analogamente al caso illustrato al punto (d); viene quindi stampata la stringa "`P.m(Number) H.m(Float)`".

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
    String m(Object... o) {
        return "P.m(Object...)";
    }
    String m(Object o) {
        return "P.m(Object)";
    }
}
public class H extends P {
    String m(Integer i) {
        return super.m(i) + " H.m(Integer)";
    }
    String m(Double d) {
        return super.m(d) + " H.m(Double)";
    }
}
public class Test {
    public static void main(String[] args) {
        P p = new P();
        H h = new H();
        P p2 = h;
        System.out.println(...);
    }
}
```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato, assumendo che tutte le classi siano dichiarate nello stesso package.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

(a) `p.m(42)`

(b) `p2.m(42)`

(c) `h.m(42)`

(d) `p.m(42.0)`

(e) `p2.m(42.0)`

(f) `h.m(42.0)`

febbraio 2019

3. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {  
    String m(Object o) { return "P.m(Object)"; }  
    String m(Number n) { return "P.m(Number)"; }  
}  
public class H extends P {  
    String m(int i) { return super.m(i) + " H.m(int)"; }  
}  
public class Test {  
    public static void main(String[] args) {  
        P p = new P();  
        H h = new H();  
        P p2 = h;  
        System.out.println(...);  
    }  
}
```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato, assumendo che tutte le classi siano dichiarate nello stesso package.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

- (a) `p.m(42)`
- (b) `p2.m(42)`
- (c) `h.m(42)`
- (d) `p.m(42.0)`
- (e) `p2.m(42.0)`
- (f) `h.m(42.0)`

gennaio 2019

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

(a) Il literal "42" ha tipo statico `String` e il tipo statico di `p` è `P`, quindi solo il metodo di `P` con segnatura `m(String)` è accessibile e applicabile per sottotipo, dato che `String` $\not\leq$ `Number`. A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(String)` in `P`. Viene stampata la stringa "`P.m(String)`".

(b) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo con segnatura `m(String)` ridefinito in `H`; poiché il parametro `s` ha tipo statico `String`, la chiamata `super.m(s)` viene risolta come al punto precedente e viene invocato il metodo della classe `P` con segnatura `m(String)`; viene stampata la stringa "`P.m(String) H.m(String)`".

(c) Il literal "42" ha tipo statico `String` e il tipo statico di `h` è `H`, l'unico metodo accessibile e applicabile per sottotipo per gli stessi motivi dei punti precedenti.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi il comportamento è lo stesso descritto al punto precedente; viene stampata la stringa "`P.m(String) H.m(String)`".

(d) Il literal 42 ha tipo statico `int` e il tipo statico di `p` è `P`, quindi non esistono metodi accessibili e applicabili per sottotipo, dato che `int` $\not\leq$ `Number`, `String`; dato che `int` può essere implicitamente convertito a `Integer` per boxing e che `Integer` \leq `Number`, `Integer` $\not\leq$ `String`; l'unico metodo applicabile ha segnatura `m(Number)`.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Number)` in `P`. Viene stampata la stringa "`P.m(Number)`".

(e) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo con segnatura `m(Number)` ridefinito in `H`; l'invocazione `super.m(n)` viene risolta con il metodo con segnatura `m(Number)` poiché il parametro `n` ha tipo statico `Number`, `Number` \leq `Number` e `Number` $\not\leq$ `String`; viene stampata la stringa "`P.m(Number) H.m(Number)`".

(f) Il literal 42 ha tipo statico `int` e il tipo statico di `h` è `H`; dato che i metodi accessibili di `H` hanno le stesse segnature di quelli di `P`, la chiamata viene risolta come al punto precedente.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi il comportamento è lo stesso del punto precedente; viene stampata la stringa "`P.m(Number) H.m(Number)`".

Settembre 2018

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {  
    String m(long l) { return "P.m(long)"; }  
    String m(int i) { return "P.m(int)"; }  
}  
public class H extends P {  
    String m(long l) { return super.m(l) + " H.m(long)"; }  
    String m(Integer i) { return super.m(i) + " H.m(Integer)"; }  
}  
public class Test {  
    public static void main(String[] args) {  
        P p = new P();  
        H h = new H();  
        P p2 = h;  
        System.out.println(...);  
    }  
}
```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato, assumendo che tutte le classi siano dichiarate nello stesso package.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

- (a) `p.m(42)`
- (b) `p2.m(42)`
- (c) `h.m(42)`
- (d) `p.m(42L)`
- (e) `p2.m(42L)`
- (f) `h.m(42L)`

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

(a) Il literal 42 ha tipo statico `int` e il tipo statico di `p` è `P`; poiché `int` $\not\leq$ `Object` e `int` $\not\leq$ `Number`, non esistono metodi di `P` accessibili e applicabili per sottotipo; tuttavia, applicando una conversione di tipo boxing da `int` a `Integer` e osservando che `Integer` \leq `Object` e `Integer` \leq `Number`, entrambi i metodi di `P` sono applicabili per boxing e reference widening. Dato che `Number` \leq `Object`, la chiamata viene risolta con il metodo che ha la segnatura più specifica `m(Number)`.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Number)` in `P` e viene stampata la stringa "`P.m(Number)`".

(b) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, ma poiché il metodo con segnatura `m(Number)` non è ridefinito in `H`, viene eseguito lo stesso metodo del punto precedente e, quindi, viene stampata la stringa "`P.m(Number)`".

(c) Il literal 42 ha tipo statico `int` e il tipo statico di `h` è `H`, l'unico metodo accessibile e applicabile per sottotipo ha segnatura `m(int)`, viste le considerazioni sui metodi ereditati da `P` riportate al punto (a).

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo di `H` con segnatura `m(int)`; poiché il parametro `i` ha tipo statico `int` e `super` si riferisce alla classe `P`, la risoluzione e il comportamento della chiamata `super.m(i)` coincidono con il caso illustrato al punto (a); viene quindi stampata la stringa "`P.m(Number) H.m(int)`".

(d) Il literal 42.0 ha tipo statico `double` e il tipo statico di `p` è `P`; poiché `double` $\not\leq$ `Object` e `double` $\not\leq$ `Number`, non esistono metodi di `P` accessibili e applicabili per sottotipo; tuttavia, applicando una conversione di tipo boxing da `double` a `Double` e osservando che `Double` \leq `Object` e `Double` \leq `Number`, entrambi i metodi di `P` sono applicabili per boxing e reference widening. Dato che `Number` \leq `Object`, la chiamata viene risolta con il metodo che ha la segnatura più specifica `m(Number)`.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Number)` in `P` e viene stampata la stringa "`P.m(Number)`".

(e) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, ma poiché il metodo con segnatura `m(Number)` non è ridefinito in `H`, viene eseguito lo stesso metodo del punto precedente e, quindi, viene stampata la stringa "`P.m(Number)`".

(f) Il literal 42.0 ha tipo statico `double` e il tipo statico di `h` è `H`; per le considerazioni riportate al punto (a) poiché `double` $\not\leq$ `int`, né il metodo definito in `H`, né quelli ereditati da `P` sono applicabili per sottotipo; tuttavia, applicando una conversione di tipo boxing da `double` a `Double` e osservando che `Double` \leq `Object` e `Double` \leq `Number`, ma `Double` $\not\leq$ `int`, i soli metodi ereditati da `P` sono applicabili per boxing e reference widening. Dato che `Number` \leq `Object`, la chiamata viene risolta con il metodo che ha la segnatura più specifica `m(Number)`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, ma poiché il metodo con segnatura `m(Number)` non è ridefinito in `H`, viene eseguito lo stesso metodo del punto precedente e, quindi, viene stampata la stringa "`P.m(Number)`".

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {  
    String m(Number n) { return "P.m(Number)"; }  
    String m(String s) { return "P.m(String)"; }  
}  
public class H extends P {  
    String m(Number n) { return super.m(n) + " H.m(Number)"; }  
    String m(String s) { return super.m(s) + " H.m(String)"; }  
}  
public class Test {  
    public static void main(String[] args) {  
        P p = new P();  
        H h = new H();  
        P p2 = h;  
        System.out.println(...);  
    }  
}
```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato, assumendo che tutte le classi siano dichiarate nello stesso package.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

- (a) `p.m("42")`
- (b) `p2.m("42")`
- (c) `h.m("42")`
- (d) `p.m(42)`
- (e) `p2.m(42)`
- (f) `h.m(42)`

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

(a) Il literal 42 ha tipo statico `int` e il tipo statico di `p` è `P`, quindi entrambi i metodi di `P` sono accessibili e applicabili per sottotipo, ma `m(int)` è più specifico poiché `int` \leq `long`. A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(int)` in `P`. Viene stampata la stringa "`P.m(int)`".

(b) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo con segnatura `m(int)` ereditato da `H` e viene stampata la stringa "`P.m(int)`".

(c) Il literal 42 ha tipo statico `int` e il tipo statico di `h` è `H`, i metodi applicabili per sotto-tipo sono gli stessi dei due punti precedenti visto che `int` $\not\leq$ `Integer`, quindi viene selezionato il metodo `m(int)`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi il comportamento è lo stesso del punto precedente; viene stampata la stringa "`P.m(int)`".

(d) Il literal 42L ha tipo statico `long` e il tipo statico di `p` è `P`, quindi `m(long)` è l'unico metodo accessibile e applicabile per sottotipo (dato che `long` $\not\leq$ `int`).

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(long)` in `P`.

Viene stampata la stringa "`P.m(long)`".

(e) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo con segnatura `m(long)` ridefinito in `H`; l'invocazione `super.m(l)` viene risolta come al punto precedente poiché `l` ha tipo statico `long`; viene stampata la stringa "`P.m(long) H.m(long)`".

(f) Il literal 42L ha tipo statico `long` e il tipo statico di `h` è `H`, il metodo applicabile per sotto-tipo ha segnatura `m(long)` come per i due punti precedenti visto che `long` $\not\leq$ `Integer`, `int`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi il comportamento è lo stesso del punto precedente; viene stampata la stringa "`P.m(long) H.m(long)`".

Luglio 2018

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
    String m(long l) { return "P.m(long)"; }
    String m(int i) { return "P.m(int)"; }
}
public class H extends P {
    String m(long l) { return super.m(l) + " H.m(long)"; }
    String m(Integer i) { return super.m(i) + " H.m(Integer)"; }
}
public class Test {
    public static void main(String[] args) {
        P p = new P();
        H h = new H();
        P p2 = h;
        System.out.println(...);
    }
}
```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato, assumendo che tutte le classi siano dichiarate nello stesso package.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

- (a) `p.m(42)`
- (b) `p2.m(42)`
- (c) `h.m(42)`
- (d) `p.m(42L)`
- (e) `p2.m(42L)`
- (f) `h.m(42L)`

giugno 2018

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

(a) Il literal `42` ha tipo statico `int` e il tipo statico di `p` è `P`, quindi nessun metodo di `P` accessibile è applicabile per sottotipo poiché `int` \leq `Long, Integer`, mentre l'unico metodo accessibile è applicabile per boxing conversion è quello con segnatura `m(Integer)` poiché `Integer` \leq `Long`. A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Integer)` in `P`.

Viene stampata la stringa "`P.m(Integer)`".

(b) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo con segnatura `m(Integer)` ereditato da `H` e viene stampata la stringa "`P.m(Integer)`".

(c) Il literal `42` ha tipo statico `int` e il tipo statico di `h` è `H`, quindi solo il metodo accessibile di `H` con segnatura `m(int)` è applicabile per sottotipo poiché `int` \leq `Long, Integer`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo con segnatura `m(int)` di `H`; l'invocazione `super.m(i)` viene risolta come al punto precedente poiché `i` ha tipo statico `int`; viene stampata la stringa "`P.m(Integer) H.m(int)`".

(d) Il literal `42L` ha tipo statico `long` e il tipo statico di `p` è `P`, quindi nessun metodo di `P` accessibile è applicabile per sottotipo poiché `long` \leq `Long, Integer`, mentre l'unico metodo accessibile è applicabile per boxing conversion è quello con segnatura `m(Long)` poiché `long` \leq `Integer`. A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Long)` in `P`.

Viene stampata la stringa "`P.m(Long)`".

(e) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo con segnatura `m(Long)` ridefinito in `H`; l'invocazione `super.m(l)` viene risolta come al punto precedente poiché `l` ha tipo statico `long`; viene stampata la stringa "`P.m(Long) H.m(long)`".

(f) Il literal `42L` ha tipo statico `long` e il tipo statico di `h` è `H`, quindi nessun metodo di `H` accessibile è applicabile per sottotipo poiché `long` \leq `int, Long, Integer`, mentre l'unico metodo applicabile per boxing conversion è quello con segnatura `m(Long)` poiché `long` \leq `Integer`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo con segnatura `m(Long)` di `H` come nel punto precedente; viene quindi stampata la stringa "`P.m(Long) H.m(Long)`".

giugno 2018

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
    String m(Double d) {
        return "P.m(Double)";
    }
    String m(Float f) {
        return "P.m(Float)";
    }
}
public class H extends P {
    String m(Double d) {
        return super.m(d) + " H.m(Double)";
    }
    String m(float f) {
        return super.m(f) + " H.m(float)";
    }
}
public class Test {
    public static void main(String[] args) {
        P p = new P();
        H h = new H();
        P p2 = h;
        System.out.println(...);
    }
}
```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato, assumendo che tutte le classi siano dichiarate nello stesso package.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

- (a) `p.m(42.0)`
- (b) `p2.m(42.0)`
- (c) `h.m(42.0)`
- (d) `p.m(42f)`
- (e) `p2.m(42f)`
- (f) `h.m(42f)`

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

(a) Il literal `42.0` ha tipo statico `double` e il tipo statico di `p` è `P`, quindi nessun metodo di `P` accessibile è applicabile per sottotipo poiché `double` \leq `Float, Double`, mentre l'unico metodo accessibile è applicabile per boxing conversion è quello con segnatura `m(Double)` poiché `Double` \leq `float`. A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Double)` in `P`.

Viene stampata la stringa "`P.m(Double)`".

(b) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo con segnatura `m(double)` ereditato da `H` e viene stampata la stringa "`P.m(double)`".

(c) Il literal `42.0` ha tipo statico `double` e il tipo statico di `h` è `H`, i metodi applicabili per sotto-tipo sono gli stessi dei due punti precedenti visto che `double` \leq `float`, quindi viene selezionato il metodo `m(double)`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi il comportamento è lo stesso del punto precedente; viene stampata la stringa "`P.m(double)`".

(d) Il literal `42L` ha tipo statico `long` e il tipo statico di `p` è `P`, quindi `m(long)` è l'unico metodo accessibile e applicabile per sotto-tipo (dato che `long` \leq `double`).

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(long)` in `P`.

Viene stampata la stringa "`P.m(long)`".

(e) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo con segnatura `m(long)` ridefinito in `H`; l'invocazione `super.m(1)` viene risolta come al punto precedente poiché `1` ha tipo statico `long`; viene stampata la stringa "`P.m(long) H.m(long)`".

(f) Il literal `42L` ha tipo statico `long` e il tipo statico di `h` è `H`, il metodo applicabile per sotto-tipo è "`m(long)`" come per i due punti precedenti visto che `long` \leq `double`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi il comportamento è lo stesso del punto precedente; viene stampata la stringa "`P.m(long) H.m(long)`".

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
    String m(Long l) {
        return "P.m(Long)";
    }
    String m(Integer i) {
        return "P.m(Integer)";
    }
}
public class H extends P {
    String m(Long l) {
        return super.m(l) + " H.m(Long)";
    }
    String m(int i) {
        return super.m(i) + " H.m(int)";
    }
}
public class Test {
    public static void main(String[] args) {
        P p = new P();
        H h = new H();
        P p2 = h;
        System.out.println(...);
    }
}
```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato, assumendo che tutte le classi siano dichiarate nello stesso package.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

- (a) `p.m(42)`
- (b) `p2.m(42)`
- (c) `h.m(42)`
- (d) `p.m(42L)`
- (e) `p2.m(42L)`
- (f) `h.m(42L)`

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

(a) Il literal `42.0` ha tipo statico `double` e il tipo statico di `p` è `P`, quindi nessun metodo di `P` accessibile è applicabile per sottotipo poiché `double` \leq `float, Double`, mentre l'unico metodo accessibile è applicabile per boxing conversion è quello con segnatura `m(Double)` poiché `Double` \leq `float`. A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Double)` in `P`.

Viene stampata la stringa "`P.m(Double)`".

(b) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo con segnatura `m(double)` ridefinito in `H`; l'invocazione `super.m(d)` viene risolta come al punto precedente poiché `d` ha tipo statico `double` e l'unico metodo accessibile è applicabile per sottotipo è quello con segnatura `m(Double)`; viene stampata la stringa "`P.m(Double) H.m(double)`".

(c) Il literal `42.0` ha tipo statico `double` e il tipo statico di `h` è `H`, quindi nessun metodo di `H` accessibile è applicabile per sottotipo poiché `double` \leq `float, Double`, mentre l'unico metodo applicabile per boxing conversion è quello con segnatura `m(float)` poiché `float` \leq `double`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo con segnatura `m(double)` di `H` come nel punto precedente; viene quindi stampata la stringa "`P.m(Double) H.m(double)`".

(d) Il literal `42f` ha tipo statico `float` e il tipo statico di `p` è `P`, quindi nessun metodo di `P` accessibile è applicabile per sottotipo poiché `float` \leq `double, Double`, mentre l'unico metodo applicabile per boxing conversion è quello con segnatura `m(Float)` poiché `float` \leq `double`. A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Float)` in `P`.

Viene stampata la stringa "`P.m(Float)`".

(e) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo con segnatura `m(float)` ereditato da `H` e viene stampata la stringa "`P.m(float)`".

(f) Il literal `42f` ha tipo statico `float` e il tipo statico di `h` è `H`, il metodo applicabile per sottotipo è "`m(float)`" è applicabile per sottotipo poiché `float` \leq `double`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi il comportamento è lo stesso del punto precedente; viene stampata la stringa "`P.m(float) H.m(float)`".

febbraio 2018

3. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {  
    String m(double d) {  
        return "P.m(double)";  
    }  
  
    String m(float f) {  
        return "P.m(float)";  
    }  
}  
  
public class H extends P {  
    String m(double d) {  
        return super.m(d) + " H.m(double)";  
    }  
  
    String m(int i) {  
        return super.m(i) + " H.m(int)";  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        P p = new P();  
        H h = new H();  
        P p2 = h;  
        System.out.println(...);  
    }  
}
```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato, assumendo che tutte le classi siano dichiarate nello stesso package.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

- (a) `p.m(42)`
- (b) `p2.m(42)`
- (c) `h.m(42)`
- (d) `p.m(42.0)`
- (e) `p2.m(42.0)`
- (f) `h.m(42.0)`

gennaio 2018

4. Assumere che le seguenti dichiarazioni di classi Java siano contenute nello stesso package:

```
public class P {  
    String m(Number n) {  
        return "P.m(Number)";  
    }  
  
    String m(double d) {  
        return "P.m(double)";  
    }  
}  
  
public class H extends P {  
    String m(Double d) {  
        return super.m(d) + " H.m(Double)";  
    }  
  
    String m(Integer i) {  
        return super.m(i) + " H.m(Integer)";  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        P p = new P();  
        H h = new H();  
        P p2 = h;  
        System.out.println(...);  
    }  
}
```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

- (a) `p.m(new Integer(42))`
- (b) `p2.m(new Integer(42))`
- (c) `h.m(new Integer(42))`
- (d) `p.m(new Double(42))`
- (e) `p2.m(new Double(42))`
- (f) `h.m(new Double(42))`

Settembre 2017

4. Assumere che le seguenti dichiarazioni di classi Java siano contenute nello stesso package:

```
public class P {  
    String m(Double d) {  
        return "P.m(Double)";  
    }  
  
    String m(Long l) {  
        return "P.m(Long)";  
    }  
}  
  
public class H extends P {  
    String m(double d) {  
        return super.m(d) + " H.m(double)";  
    }  
  
    String m(long l) {  
        return super.m(l) + " H.m(long)";  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        P p = new P();  
        H h = new H();  
        P p2 = h;  
        System.out.println(...);  
    }  
}
```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

- (a) Il literal `42` ha tipo statico `int` e il tipo statico di `p` è `P`, quindi entrambi i metodi di `P` sono accessibili e applicabili per sottotipo poiché `int` \leq `float`, `double`, ma il metodo con segnatura `m(float)` è più specifico del metodo con segnatura `m(double)` poiché `float` \leq `double`. A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(float)` in `P`. Viene stampata la stringa "`P.m(float)`".
- (b) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che i tipi statici sono gli stessi. A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo con segnatura `m(float)` ereditato da `H`. Viene stampata la stringa "`P.m(float)`".
- (c) L'espressione `42.0` ha tipo statico `int`, mentre il tipo statico di `h` è `H`; tutti e tre i metodi di segnatura `m(double)`, `m(int)` e `m(Number)` sono accessibili e applicabili per sottotipo, ma quello con segnatura `m(int)` è il più specifico poiché `int` \leq `float` \leq `double`. A runtime `h` contiene un'istanza di `H`, quindi viene eseguito il metodo di segnatura `m(int)` in `H`; l'invocazione `super.m(i)` viene risolta come al punto (a) poiché `i` ha tipo statico `int`, quindi viene stampata la stringa "`P.m(float) H.m(int)`".
- (d) Il literal `42.0` ha tipo statico `double` e il tipo statico di `p` è `P`, quindi il solo metodo accessibile in `P` è applicabile per sottotipo ha segnatura `m(double)` poiché `double` \leq `float`. A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(double)` in `P`. Viene stampata la stringa "`P.m(double)`".
- (e) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che `p2` ha lo stesso tipo statico di `p`. A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo con segnatura `m(double)` ridefinito in `H`; l'invocazione `super.m(d)` viene risolta come al punto precedente poiché `d` ha tipo statico `double`, quindi viene stampata la stringa "`P.m(double) H.m(double)`".
- (f) Il literal `42.0` ha tipo statico `double` e il tipo statico di `h` è `H`, quindi solo il metodo di `H` con segnatura `m(double)` è accessibile e applicabile per sottotipo infatti `double` $\not\leq$ `int`, `float`. A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi il comportamento è lo stesso di quello al punto precedente e viene stampata la stringa "`P.m(double) H.m(double)`".

Soluzione: assumendo che le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

- (a) Il literal `42` ha tipo statico `int`, mentre `p` ha tipo statico `P`; la classe `Integer` ha un costruttore pubblico con parametro di tipo `int` che è l'unico applicabile per sottotipo, quindi l'argomento dell'invocazione ha tipo statico `Integer`. Poiché `Integer` \leq `Number`, `Integer` \leq `double` l'unico metodo di `P` accessibile e applicabile per sottotipo ha segnatura `m(Number)`. A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Number)` di `P`. Viene stampata la stringa "`P.m(Number)`".
- (b) Per analogia con il caso precedente i tipi statici coinvolti sono identici, quindi anche in questo caso viene selezionato il metodo con segnatura `m(Number)`. A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi la ricerca del metodo inizia dalla classe `H`; non essendo ridefinito in `H`, viene eseguito come nel caso precedente il metodo con segnatura `m(Number)` di `P`. Viene stampata la stringa "`P.m(Number)`".
- (c) Come nei casi precedenti, l'argomento dell'invocazione ha tipo statico `Integer`, mentre `h` ha tipo statico `H`. Poiché `Integer` \leq `Number`, `Integer` \leq `Double`, `Integer` \leq `double`, esistono due metodi di `H` accessibili e applicabili, aventi segnatura `m(Integer)` e `m(Number)`; viene selezionato il più specifico `m(Integer)`, dato che `Integer` \leq `Number`. A runtime `h` contiene un'istanza di `H`, quindi viene eseguito il metodo `m(Integer)` di `H`. La chiamata `super.m(i)` viene risolta come ai punti precedenti visto che il tipo statico dell'oggetto receiver è `P` e il tipo statico dell'argomento `i` è `Integer`, quindi viene invocato il metodo `m(Number)` di `P`. Viene stampata la stringa "`P.m(Number) H.m(Integer)`".
- (d) Il literal `42` ha tipo statico `int`, mentre `p` ha tipo statico `P`; la classe `Double` ha un costruttore pubblico con parametro di tipo `double` che è l'unico applicabile per sottotipo, quindi l'argomento dell'invocazione ha tipo statico `Double`. Poiché `Double` \leq `Number`, `Double` \leq `double` l'unico metodo di `P` accessibile e applicabile per sottotipo ha segnatura `m(Number)`. A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Number)` di `P`. Viene stampata la stringa "`P.m(Number)`".
- (e) Per analogia con il caso precedente i tipi statici coinvolti sono identici, quindi anche in questo caso viene selezionato il metodo con segnatura `m(Number)`. A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi la ricerca del metodo inizia dalla classe `H`; non essendo ridefinito in `H`, viene eseguito come nel caso precedente il metodo con segnatura `m(Number)` di `P`. Viene stampata la stringa "`P.m(Number)`".
- (f) Come nei due casi precedenti, l'argomento dell'invocazione ha tipo statico `Double`, mentre `h` ha tipo statico `H`. Poiché `Double` \leq `Number`, `Double` \leq `Double`, `Double` \leq `double`, esistono due metodi di `H` accessibili e applicabili, aventi segnatura `m(Double)` e `m(Number)`; viene selezionato il più specifico `m(Double)`, dato che `Double` \leq `Number`. A runtime `h` contiene un'istanza di `H`, quindi viene eseguito il metodo `m(Double)` di `H`. La chiamata `super.m(d)` viene risolta come ai punti precedenti visto che il tipo statico dell'oggetto receiver è `P` e il tipo statico dell'argomento `d` è `Double`, quindi viene invocato il metodo `m(Number)` di `P`. Viene stampata la stringa "`P.m(Number) H.m(Double)`".

Soluzione: assumendo che le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

- (a) Il literal `42` ha tipo statico `int`, mentre `p` ha tipo statico `P`; non esistono metodi di `P` accessibili e applicabili né per sottotipo, né per boxing, dato che `int` $\not\leq$ `Long`, `int` $\not\leq$ `Double` `Integer` \leq `Long` e `Integer` \leq `Double`, quindi viene emesso un errore di compilazione.
- (b) Il literal `42` ha tipo statico `int`, mentre `h` ha tipo statico `H`; esistono solo due metodi di `H`, `m(long)` e `m(double)`, entrambi accessibili e applicabili per sottotipo (`int` \leq `long`, `int` \leq `double`, `int` \leq `Long` e `int` \leq `Double`); viene selezionato il più specifico `m(long)` dato che `long` \leq `double`. A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo con segnatura `m(long)` di `H`. L'invocazione `super.m(l)` viene risolta con il metodo `m(Long)` di `P` per boxing dato che `l` ha tipo statico `long` e `long` \leq `Long`, `long` \leq `double`, `long` \leq `Long`. Viene stampata la stringa "`P.m(Long) H.m(long)`".
- (c) Il literal `42L` ha tipo statico `long`, mentre `p` ha tipo statico `P`; essendo i tipi statici gli stessi di `super.m(l)` del caso precedente, l'invocazione viene risolta con il metodo `m(Long)`.
- (d) Il literal `42L` ha tipo statico `long`, mentre `h` ha tipo statico `H`; esistono solo due metodi di `H`, `m(long)` e `m(double)`, entrambi accessibili e applicabili per sottotipo (`long` \leq `long`, `long` \leq `double`, `long` \leq `Long` e `long` \leq `Double`); viene selezionato il più specifico `m(long)` come per il caso (b). A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo con segnatura `m(long)` di `H`. Analogamente al caso (b), viene stampata la stringa "`P.m(Long) H.m(long)`".
- (e) Il literal `42L` ha tipo statico `long`, mentre `p` ha tipo statico `P`; non esistono metodi di `P` accessibili e applicabili per sottotipo (`long` $\not\leq$ `Long` e `long` $\not\leq$ `Double`), mentre `m(Double)` è l'unico accessibile e applicabile per boxing (`long` $\not\leq$ `Long` e `long` $\not\leq$ `Double`). A runtime `p` contiene un'istanza di `P`, quindi viene eseguito il metodo `m(Long)` di `P` e viene stampata la stringa "`P.m(Long)`".
- (f) L'invocazione viene risolta con il metodo `m(Double)` come per il caso precedente, dato che i tipi statici sono gli stessi. A runtime `p2` contiene un'istanza di `H`, ma poiché il metodo `m(Double)` non viene ridefinito in `H`, viene eseguito il metodo ereditato da `P` e viene stampata la stringa "`P.m(Double)`" come nel caso precedente.

luglio 2017

4. Considerare le seguenti dichiarazioni di classi Java contenute nello stesso package:

```
public class P {
    String m(Long i) {
        return "P.m(Long)";
    }
    String m(long i) {
        return "P.m(long)";
    }
}
public class H extends P {
    String m(Integer i) {
        return super.m(i) + " H.m(Integer)";
    }
    String m(int i) {
        return super.m(i) + " H.m(int)";
    }
}
public class Test {
    public static void main(String[] args) {
        P p = new P();
        H h = new H();
        P p2 = h;
        System.out.println(...);
    }
}
```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

- (a) `p.m(42L)`
- (b) `p2.m(42L)`
- (c) `h.m(42L)`
- (d) `p.m(Integer.valueOf(42))`
- (e) `p2.m(Integer.valueOf(42))`
- (f) `h.m(Integer.valueOf(42))`

giugno 2017

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
    String m(Object o) {
        return "P.m(Object)";
    }
    String m(Number n) {
        return "P.m(Number)";
    }
    String m(Object... os) {
        return "P.m(Object...)";
    }
}
public class H extends P {
    String m(Number n) {
        return super.m(n) + " H.m(Number)";
    }
    String m(double d) {
        return super.m(d) + " H.m(double)";
    }
}
public class Test {
    public static void main(String[] args) {
        P p = new P();
        H h = new H();
        P p2 = h;
        System.out.println(...);
    }
}
```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato, assumendo che tutte le classi siano dichiarate nello stesso package.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

- (a) `p.m(Double.valueOf(42.0))`
- (b) `p2.m(Double.valueOf(42.0))`
- (c) `h.m(Double.valueOf(42.0))`
- (d) `p.m(42.0)`
- (e) `p2.m(42.0)`
- (f) `h.m(42.0)`

Soluzione: assumendo che le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

- (a) Il literal `42L` ha tipo statico `long`, mentre `p` ha tipo statico `P`; l'unico metodo di `P` accessibile e applicabile per sottotipo ha segnatura `m(Long)` poiché `long` \leq `Long`.
A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(long)` in `P`. Viene stampata la stringa `"P.m(long)"`.
- (b) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che `p2` ha lo stesso tipo statico di `p`.
A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo con segnatura `m(long)` ereditato da `P` e viene stampata la stringa `"P.m(long)"`.
- (c) Il literal `42L` ha tipo statico `long`, mentre `h` ha tipo statico `H`; l'unico metodo di `H` accessibile e applicabile per sottotipo è quello ereditato da `P` con segnatura `m(long)`, poiché `long` \leq `Long`, `long` \leq `Integer` e `long` \leq `int`.
Visto che `h` contiene un'istanza di `H`, il comportamento a runtime del metodo è lo stesso del punto precedente e quindi viene stampata la stringa `"P.m(long)"`.
- (d) Il literal `42` ha tipo statico `int` e l'unico metodo statico `valueOf` di `Integer` accessibile e applicabile per sottotipo ha segnatura `valueOf(int)` e restituisce un valore di tipo `Integer` (le altre due versioni di `valueOf` hanno segnatura `valueOf(String)` e `valueOf(String, int)` e quindi non sono applicabili per sottotipo), perciò `Integer.valueOf(42)` ha tipo statico `Integer`.
Il tipo statico di `p` è `P` e non esistono metodi di `P` accessibili e applicabili per sottotipo poiché `Integer` $\not\leq$ `Long` e `Integer` $\not\leq$ `long`; tramite unboxing l'argomento può essere convertito al tipo `int` e dato che `int` \leq `long`, ma `int` $\not\leq$ `Long`, il metodo con segnatura `m(long)` è l'unico applicabile per unboxing e widening primitive conversion.
A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(long)` in `P`. Viene stampata la stringa `"P.m(long)"`.
- (e) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che `p2` ha lo stesso tipo statico di `p`.
A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi il comportamento è lo stesso di quello al punto (b). Viene stampata la stringa `"P.m(long)"`.
- (f) Come a due punti precedenti, il literal `Integer.valueOf(42)` ha tipo statico `Integer`, mentre il tipo statico di `h` è `H`. Poiché `Integer` $\not\leq$ `Long`, `Integer` $\not\leq `long` e `Integer` $\not\leq$ `int`, l'unico metodo della classe `H` accessibile e applicabile per sottotipo ha segnatura `m(Integer)`.
A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo con segnatura `m(Integer)` in `H`. L'argomento `i` ha tipo statico `Integer`, quindi come al punto (d) la chiamata `super.m(i)` viene staticamente risolta con il metodo con segnatura `m(long)` e viene eseguito il metodo di `P` (diretta superclasse di `H`) con tale segnatura.
Viene stampata la stringa `"P.m(long) H.m(Integer)"`.$

Soluzione: assumendo che le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

- (a) Il literal `42.0` ha tipo statico `double` e l'unico metodo statico `valueOf` della classe `Double` accessibile e applicabile restituisce un valore di tipo `Double`. Il tipo statico di `p` è `P`, quindi esistono due metodi accessibili in `P` applicabili per sottotipo, ma il metodo con segnatura `m(Number)` è più specifico del metodo con segnatura `m(Object)` poiché `Number` \leq `Object`.
A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Number)` in `P`.
Viene stampata la stringa `"P.m(Number)"`.
- (b) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che `p2` ha lo stesso tipo statico di `p`.
A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo con segnatura `m(Number)` ridefinito in `H`. L'argomento `n` ha tipo statico `Number`, quindi per ragioni analoghe a quelle del punto precedente, per l'invocazione `super.m(n)` il metodo accessibile, applicabile e più specifico in `P` ha segnatura `m(Number)`.
Viene stampata la stringa `"P.m(Number) H.m(Number)"`.
- (c) Il tipo statico di `h` è `H` e come nei casi precedenti, l'argomento del metodo `m` ha tipo statico `Double`. Rispetto ai casi precedenti, i metodi applicabili e accessibili sono gli stessi, quindi l'overloading viene risolto come al punto precedente.
Visto che `h` contiene un'istanza di `H`, il comportamento a runtime del metodo è lo stesso del punto precedente e quindi viene stampata la stringa `"P.m(Number) H.m(Number)"`.
- (d) Il literal `42.0` ha tipo statico `double` e il tipo statico di `p` è `P`, non esistono metodi in `P` accessibili e applicabili per sottotipo, mentre i due metodi con segnatura `m(Number)` e `m(Object)` sono accessibili e applicabili per boxing e widening reference conversion. Il primo è più specifico poiché `Number` \leq `Object`.
A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Number)` in `P`.
Viene stampata la stringa `"P.m(Number)"`.
- (e) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che `p2` ha lo stesso tipo statico di `p`.
A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi il comportamento è lo stesso di quello al punto (b).
Viene stampata la stringa `"P.m(Number) H.m(Number)"`.
- (f) Il literal `42.0` ha tipo statico `double`, il tipo statico di `h` è `H` e l'unico metodo della classe `H` accessibile e applicabile per sottotipo è il metodo con segnatura `m(double)`.
A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo con segnatura `m(double)` in `H`. L'argomento `d` ha tipo statico `double`, quindi per ragioni analoghe a quelle del punto precedente, per l'invocazione `super.m(d)` il metodo accessibile, applicabile e più specifico in `P` ha segnatura `m(Number)`.
Viene stampata la stringa `"P.m(Number) H.m(double)"`.

febbraio 2017

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
    String m(long l) {
        return "P.m(long)";
    }
    String m(double d) {
        return "P.m(double)";
    }
    String m(Object... os) {
        return "P.m(Object...)";
    }
}

public class H extends P {
    String m(long l) {
        return super.m(double) l + " H.m(long)";
    }
    String m(Double d) {
        return super.m(d, d + 1) + " H.m(Double)";
    }
}

public class Test {
    public static void main(String[] args) {
        P p = new P();
        H h = new H();
        P p2 = h;
        System.out.println(...);
    }
}
```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato, assumendo che tutte le classi siano dichiarate nello stesso package.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

- (a) `p.m(42L)`
- (b) `p2.m(42L)`
- (c) `h.m(42L)`
- (d) `p.m(42.0)`
- (e) `p2.m(42.0)`
- (f) `h.m(Double.valueOf(42.0))`

Settembre 2016

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
    String m(Number n) {
        return "P.m(Number)";
    }
    String m(Double d) {
        return "P.m(Double)";
    }
}

public class H extends P {
    String m(Integer i) {
        return super.m(i) + " H.m(Integer)";
    }
    String m(int i) {
        return super.m(i) + " H.m(int)";
    }
    String m(Number n) {
        return super.m(n) + " H.m(Number)";
    }
}

public class Test {
    public static void main(String[] args) {
        P p = new P();
        H h = new H();
        P p2 = h;
        System.out.println(...);
    }
}
```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato, assumendo che tutte le classi siano dichiarate nello stesso package.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

- (a) `p.m(42)`
- (b) `p2.m(42)`
- (c) `h.m(42)`
- (d) `p.m(4.2)`
- (e) `p2.m(4.2)`
- (f) `h.m(4,0)`

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

- (a) Il literal `42L` ha tipo statico `long` e il tipo statico di `p` è `P`, quindi esistono due metodi accessibili in `P` applicabili per sottotipo, ma il metodo con segnatura `m(long)` è più specifico del metodo con segnatura `m(double)` poiché `long ≤ double`.
A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(long)` in `P`.
Viene stampata la stringa "`P.m(long)`".
- (b) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che `p2` ha lo stesso tipo statico di `p`.
A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo con segnatura `m(long)` ridefinito in `H`. L'espressione `(double) l` è staticamente corretta poiché i cast tra tipi numerici sono sempre ammessi; dato che il tipo statico dell'argomento è `double`, per l'invocazione `super.m(double) l` esiste in `P` un solo metodo accessibile e applicabile per sottotipo, quello con segnatura `m(double)`.
Viene stampata la stringa "`P.m(double)` `H.m(long)`".
- (c) L'espressione `42L` ha tipo statico `long`, mentre il tipo statico di `h` è `H`; poiché il metodo di `H` con segnatura `m(Double d)` non è applicabile per sottotipo, l'invocazione viene risolta come al punto precedente. Visto che `h` contiene un'istanza di `H`, il comportamento a runtime del metodo è lo stesso del punto precedente e quindi viene stampata la stringa "`P.m(double)` `H.m(long)`".
- (d) Il literal `42.0` ha tipo statico `double` e il tipo statico di `p` è `P`, quindi il solo metodo accessibile in `P` è applicabile per sottotipo ha segnatura `m(double)`.
A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(double)` in `P`.
Viene stampata la stringa "`P.m(double)`".
- (e) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che `p2` ha lo stesso tipo statico di `p`.
A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo di `P` con segnatura `m(double)`, visto che la sottoclasse `H` non ridefinisce il metodo, ma lo eredita da `P`.
Viene stampata la stringa "`P.m(double)`".
- (f) Il literal `42.0` ha tipo statico `double` e l'unico metodo della classe `Double` accessibile e applicabile per sottotipo è il metodo statico con segnatura `valueOf(double)` e tipo di ritorno `Double`. Il tipo statico di `h` è `H` e l'unico metodo accessibile e applicabile per sottotipo è quello con segnatura `m(Double)`. Gli argomenti dell'invocazione `super.m(d, d + 1)` hanno, rispettivamente, tipo `Double` e `double`; nella classe `P` non esistono metodi accessibili e applicabili solo per sottotipo o sottotipo e boxing/unboxing, mentre il metodo con segnatura `m(Object...)` è applicabile per arità variabile e per boxing del secondo argomento e conseguente widening reference conversion.
Viene stampata la stringa "`P.m(Object...)` `H.m(Double)`".

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, tutti i metodi di `P` e `H` sono accessibili da `Test`; si hanno i seguenti casi:

- (a) Il literal `42` ha tipo statico `int` e il tipo statico di `p` è `P`; nessun metodo di `P` è applicabile per sottotipo, ma il metodo con segnatura `m(Number)` è l'unico applicabile per boxing e widening reference conversion poiché `Integer ≤ Number` (mentre `Integer ≤ Double`).
A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Number)` in `P`. Viene stampata la stringa "`P.m(Number)`".
- (b) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che `p2` ha lo stesso tipo statico di `p`.
A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo con segnatura `m(Number)` ridefinito in `H`; per la chiamata `super.m(n)`, poiché `n` ha tipo statico `Number`, l'unico metodo della classe `P` applicabile per sottotipo ha segnatura `m(Number)`, quindi viene eseguito lo stesso metodo del punto precedente e viene stampata la stringa "`P.m(Number)` `H.m(Number)`".
- (c) Il literal `42` ha tipo statico `int` e il tipo statico di `h` è `H`; l'unico metodo di `H` applicabile per sottotipo ha segnatura `m(int)`.
A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo dichiarato in `H` con segnatura `m(int)`; per la chiamata `super.m(i)`, poiché `i` ha tipo statico `int`, l'invocazione viene risolta come al punto (a), quindi viene stampata la stringa "`P.m(Number)` `H.m(int)`".
- (d) Il literal `4.2` ha tipo statico `double` e il tipo statico di `p` è `P`; nessun metodo di `P` è applicabile per sottotipo, mentre i metodi con segnatura `m(Number)` e `m(Double)` sono entrambi applicabili per boxing e widening reference conversion; poiché `Double ≤ Number`, il metodo più specifico è quello con segnatura `m(Double)`.
A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Double)` in `P` e viene stampata la stringa "`P.m(Double)`".
- (e) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che `p2` ha lo stesso tipo statico di `p`.
A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, ma poiché il metodo con segnatura `m(Double)` non viene ridefinito in `H`, viene eseguito lo stesso metodo del punto precedente e viene stampata la stringa "`P.m(Double)`".
- (f) I literal `4` e `0` hanno tipo statico `int` e il tipo statico di `h` è `H`; dato che `h` non ha metodi con due parametri, né con arità variabile, l'invocazione non è staticamente corretta.

Giugno 2016

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
    String m(int i) {
        return "P.m(int)";
    }
    String m(double d) {
        return "P.m(double)";
    }
}

public class H extends P {
    String m(Integer i) {
        return super.m(i) + " H.m(Integer)";
    }
    String m(Double d) {
        return super.m(d) + " H.m(Double)";
    }
    String m(Integer... ia) {
        return super.m(ia[0]) + " H.m(Integer...)";
    }
}

public class Test {
    public static void main(String[] args) {
        P p = new P();
        H h = new H();
        P p2 = h;
        System.out.println(...);
    }
}
```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato, assumendo che tutte le classi siano dichiarate nello stesso package.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

(a) `p.m(42)`

(b) `p2.m(42)`

(c) `h.m(42)`

(d) `p.m(4.2)`

(e) `p2.m(4.2)`

(f) `h.m(42, 0)`

Giugno 2016

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
    String m(Integer i) { return "P.m(Integer)"; }
    String m(Long l) { return "P.m(Long)"; }
}

public class H extends P {
    String m(Integer i) { return super.m(i) + " H.m(Integer)"; }
    String m(Long l) { return super.m(l) + " H.m(Long)"; }
    String m(int... ia) { return super.m(ia[0]) + " H.m(int[])"; }
}

public class Test {
    public static void main(String[] args) {
        P p = new P();
        H h = new H();
        P p2 = h;
        System.out.println(...);
    }
}
```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato, assumendo che tutte le classi siano dichiarate nello stesso package.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

(a) `p.m(new Long(42))`

(b) `p2.m(new Long(42))`

(c) `h.m(new Long(42))`

(d) `p.m(new Integer(42))`

(e) `p2.m(new Integer(42))`

(f) `h.m(42, 0)`

Febbraio 2016

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

(a) Il literal `42` ha tipo statico `int`; il tipo statico di `p` è `P` quindi non esistono metodi accessibili in `P` applicabili per sottotipo, ma il metodo con segnatura `m(Integer)` è accessibile e applicabile per boxing, mentre l'altro no, dato che `Integer` $\not\leq$ `Long`.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Integer)` in `P`. Viene stampata la stringa `"P.m(Integer)"`.

(b) L'espressione è staticamente corretta per esattamente lo stesso motivo del punto precedente, visto che `p2` ha lo stesso tipo statico di `p`.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo con segnatura `m(Integer)` ridefinito in `H`. Poiché `i` ha tipo statico `Integer`, l'overloading per l'invocazione `super.m(i)` viene risolta con il metodo con segnatura `m(Integer)` che è l'unico accessibile e applicabile per sottotipo.

Viene stampata la stringa `"P.m(Integer) H.m(Integer)"`.

(c) L'espressione `new Long(42)` ha tipo statico `Long` poiché `42` ha tipo `int` che è sotto-tipo del tipo `long` del parametro dell'unico costruttore applicabile per sottotipo di `Long` (l'altro ha tipo `String`); il tipo statico di `p` è `P`. L'unico metodo della classe `P` applicabile per sottotipo ha segnatura `m(Long)`.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Long)` in `P`. Viene stampata la stringa `"P.m(Long)"`.

(d) L'espressione `42L` ha tipo statico `long` e il tipo statico di `h` è `H`; nessun metodo di `H` è applicabile per sottotipo, mentre esistono due metodi accessibili e applicabili per boxing (e successivo reference widening in uno dei due casi) con segnatura `m(Long)` e `m(Object)`; poiché `Long` \leq `Object`, il metodo con segnatura `m(Long)` è il più specifico.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo in `H` con segnatura `m(Long)`. Viene stampata la stringa `"P.m(Long)"`.

(e) Il literal `42` ha tipo statico `int`, il cast è corretto staticamente (la conversione in questo caso è senza perdita di informazione), l'argomento `(byte)42` ha tipo statico `byte`; il tipo statico di `h` è `H` e l'unico metodo accessibile e applicabile per sottotipo ha segnatura `m(int)`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo in `H` con segnatura `m(int)`. Poiché `i` ha tipo statico `int`, l'overloading per l'invocazione `super.m(i)` viene risolta come al punto (a) e, quindi, viene chiamato il metodo della classe `P` con segnatura `m(Integer)`.

Viene stampata la stringa `"P.m(Integer) H.m(int)"`.

(f) Il literal `42.0` ha tipo statico `double` e il tipo statico di `h` è `H`; nessun metodo di `H` (inclusi quelli ereditati da `P`) è applicabile per sottotipo, mentre l'unico metodo accessibile e applicabile per boxing e reference widening ha segnatura `m(Object)`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo in `H` con segnatura `m(Object)`. In seguito alla conversione per boxing, l'argomento del metodo ha tipo dinamico `Double` che non è sottotipo di `Integer`, quindi il cast (`Integer`) solleva l'eccezione `ClassCastException`.

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

- Il literal `42` ha tipo statico `int` e il tipo statico di `p` è `P`; entrambi i metodi di `P` sono accessibili e applicabili per sottotipo, ma il metodo con segnatura `m(int)` è più specifico poiché `int` è sottotipo di `double`. A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(int)` in `P`. Viene stampata la stringa `"P.m(int)"`.
- L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che `p2` ha lo stesso tipo statico di `p`. A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, ma poiché il metodo con segnatura `m(int)` non viene ridefinito in `H`, viene eseguito lo stesso metodo del punto precedente e, quindi, viene stampata la stringa `"P.m(int)"`.
- Il literal `42` ha tipo statico `int` e il tipo statico di `h` è `H`; poiché nessuno dei metodi dichiarati in `H` è applicabile per sottotipo, gli unici due metodi accessibili e applicabili per sottotipo sono quelli ereditati da `P`, quindi la risoluzione dell'overloading e il comportamento a runtime sono gli stessi del punto precedente e viene stampata la stringa `"P.m(int)"`.
- Il literal `4.2` ha tipo statico `double` e il tipo statico di `p` è `P`; l'unico metodo di `P` accessibile e applicabile per sottotipo, ha segnatura `m(double)`. A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(double)` in `P`. Viene stampata la stringa `"P.m(double)"`.
- L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che `p2` ha lo stesso tipo statico di `p`. A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, ma poiché il metodo con segnatura `m(double)` non viene ridefinito in `H`, viene eseguito lo stesso metodo del punto precedente e, quindi, viene stampata la stringa `"P.m(double)"`.
- Il literal `42` e `0` hanno tipo statico `int` e il tipo statico di `h` è `H`; dato che gli argomenti sono due, nessun metodo è applicabile per sottotipo o per conversione boxing/unboxing e l'unico metodo accessibile e applicabile è quello di arità variabile e segnatura `m(Integer...)`. Il tipo dinamico dell'oggetto in `h` è `H` e nel corpo del metodo di segnatura `m(Integer...)` l'espressione `ia[0]` ha tipo statico `Integer`; per l'invocazione `super.m(ia[0])` non esistono metodi in `P` accessibili e applicabili per sottotipo, mentre entrambi i metodi di `P` sono applicabili per unboxing e sottotipo, ma il metodo con segnatura `m(int)` è più specifico poiché `int` è sottotipo di `double`. Viene stampata la stringa `"P.m(int) H.m(Integer...)"`.

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

- Il literal `42` ha tipo statico `int`, gli unici due costruttori pubblici della classe `Long` hanno segnatura `Long(long)` e `Long(String)` e il primo è applicabile per sottotipo, quindi l'espressione `new Long(42)` ha tipo statico `Long`. Il tipo statico di `p` è `P` quindi non esistono metodi accessibili in `P` applicabili per sottotipo, ma il metodo con segnatura `m(long)` è accessibile e applicabile per unboxing. A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(long)` in `P`. Viene stampata la stringa `"P.m(long)"`.
- L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che `p2` ha lo stesso tipo statico di `p`. A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo con segnatura `m(long)` in `P`, visto che la sottoclasse `H` non ridefinisce il metodo, ma lo eredita da `P`. Viene stampata la stringa `"P.m(long)"`.
- L'espressione `new Long(42)` ha tipo statico `Long` come già spiegato ai punti precedenti, mentre il tipo statico di `h` è `H`; tra i metodi definiti in `H` e quelli ereditati da `P` solo `m(Long)` è accessibile e applicabile per sottotipo. A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo con segnatura `m(Long)` in `H`. L'invocazione `super.m(1)` è staticamente corretta e l'overloading viene risolto come ai punti precedenti e, quindi, viene invocato il metodo `m(Long)` in `P`. Viene stampata la stringa `"P.m(long) H.m(Long)"`.
- Il literal `42` e `0` hanno tipo statico `int` e il tipo statico di `h` è `H`; nessun metodo di `H` (inclusi quelli ereditati da `P`) è applicabile per sottotipo o per boxing/unboxing, mentre l'unico metodo ad arità variabile `m(int...)` (definito nella classe `H`) è accessibile e applicabile. A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo in `H` con segnatura `m(Integer...)`. L'invocazione `super.m(1)` è staticamente corretta, il tipo statico di `ia[0]` è `int` e l'unico metodo accessibile e applicabile ha segnatura `m(Long)`. Viene stampata la stringa `"P.m(Integer) H.m(Integer...)"`.
- L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che `p2` ha lo stesso tipo statico di `p`. A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo con segnatura `m(Integer)` ridefinito in `H`. L'invocazione `super.m(1)` è staticamente corretta e l'overloading viene risolto allo stesso modo, quindi, viene invocato il metodo `m(Integer)` in `P`. Viene stampata la stringa `"P.m(Integer) H.m(Integer)"`.
- Il literal `42` e `0` hanno tipo statico `int` e il tipo statico di `h` è `H`; nessun metodo di `H` (inclusi quelli ereditati da `P`) è applicabile per sottotipo o per conversione boxing/unboxing, mentre l'unico metodo ad arità variabile `m(int...)` (definito nella classe `H`) è accessibile e applicabile. A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo in `H` con segnatura `m(int...)`. L'invocazione `super.m(1)` è staticamente corretta, il tipo statico di `ia[0]` è `int` e l'unico metodo accessibile e applicabile ha segnatura `m(Long)`. Viene stampata la stringa `"P.m(long) H.m(int[])"`.

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
    String m(Integer i) { return "P.m(Integer)"; }
    String m(Long l) { return "P.m(Long)"; }
}

public class H extends P {
    String m(int i) { return super.m(i) + " H.m(int)"; }
    String m(Integer i) { return super.m(i) + " H.m(Integer)"; }
    String m(Object o) { return super.m((Integer) o) + " H.m(Object)"; }
}

public class Test {
    public static void main(String[] args) {
        P p = new P();
        H h = new H();
        P p2 = h;
        System.out.println(...);
    }
}
```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato, assumendo che tutte le classi siano dichiarate nello stesso package.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

(a) `p.m(42)`

(b) `p2.m(42)`

(c) `p.m(new Long(42))`

(d) `h.m(42L)`

(e) `h.m((byte)42)`

(f) `h.m(42, 0)`

Gennaio 2016

4. Considerare le seguenti dichiarazioni di classi Java:

```

public class P {
    String m(int i) { return "P.m(int)"; }
    String m(long l) { return "P.m(long)"; }
}

public class H extends P {
    String m(int i) { return super.m(i) + " H.m(int)"; }
    String m(Integer i) { return super.m(i) + " H.m(Integer)"; }
    String m(Integer... l) { return (l.length > 0 ? super.m(l[0]) : "") + " H.m(Integer...)"; }
}

public class Test {
    public static void main(String[] args) {
        P p = new P();
        H h = new H();
        P p2 = h;
        System.out.println(...);
    }
}

```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato, assumendo che tutte le classi siano dichiarate nello stesso package.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

- (a) `p.m(42)`
- (b) `p2.m(42)`
- (c) `p.m(new Long(42))`
- (d) `h.m(new Long(42))`
- (e) `p2.m(42, 42)`
- (f) `h.m(42, 42)`

Settembre 2015

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

(a) Il literal '`a'` ha tipo statico `char`; il tipo statico di `p` è `P` ed esiste un solo metodo di `P` accessibile e applicabile per sotto-tipo, `String m(char c)`.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo `String m(char c)` in `P`.

Viene stampata la stringa "`P.m(char)`".

(b) L'espressione è staticamente corretta per esattamente lo stesso motivo del punto precedente, visto che `p2` ha lo stesso tipo statico di `p`.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo `String m(char c)` in `H`. Poiché `c` ha tipo statico `char`, per l'invocazione `super.m(c)` esiste un solo metodo in `P` accessibile e applicabile per sotto-tipo, `String m(char c)`.

Viene stampata la stringa "`P.m(char) H.m(char)`".

(c) Il literal '`a'` ha tipo statico `char` e per l'invocazione `Character.valueOf('a')` esiste un solo metodo statico nella classe `Character` accessibile e applicabile per sotto-tipo, che restituisce un valore di tipo `Character`. Il tipo statico di `i` è `H` ed esiste un solo metodo di `H` accessibile e applicabile per sotto-tipo, `String m(Character c)`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo `String m(Character c)` in `H`.

Poiché `c` ha tipo statico `Character`, per l'invocazione `super.m(c)` non esistono metodi in `P` accessibili e applicabili per sotto-tipo, mentre `String m(char c)` è l'unico accessibile e applicabile per unboxing.

Viene stampata la stringa "`P.m(char) H.m(Character)`".

(d) Il literal "`a`" ha tipo statico `String`; il tipo statico di `p` è `P` ed esiste un solo metodo di `P` accessibile e applicabile per sotto-tipo, `String m(String s)`.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo `String m(String s)` in `P`.

Viene stampata la stringa "`P.m(String)`".

(e) L'espressione `new char[]{'4', '2'}` ha tipo statico `char[]`, il tipo statico di `p2` è `P` e non esiste alcun metodo di `P` accessibile e applicabile, quindi viene segnalato un errore durante la compilazione.

(f) L'espressione `new Character[]{'4', '2'}` ha tipo statico `Character[]`, il tipo statico di `h` è `H` e l'unico metodo di `H` che è accessibile e applicabile è `String m(Character... cs)`, poiché `Character...` corrisponde a `Character[]`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo `String m(Character... cs)` in `H`. La variabile `sb` viene inizializzata con un'istanza di `StringBuilder` corrispondente alla stringa vuota. In seguito, gli elementi dell'array vengono inseriti nell'istanza di `StringBuilder` nell'ordine e separati da uno spazio bianco, dopo di che viene concatenata in fondo la stringa "`H.m(Character...)`".

Viene stampata la stringa "`4 2 H.m(Character...)`".

Luglio 2015

4. Considerare le seguenti dichiarazioni di classi Java:

```

public class P {
    String m(byte b) { return "P.m(byte)"; }
    String m(int i) { return "P.m(int)"; }
    String m(Number... n) { return "P.m(Number...)"; }
}

public class H extends P {
    String m(float f) { return super.m(f) + " H.m(float)"; }
    String m(short s) { return super.m(s) + " H.m(short)"; }
    String m(Double... ds) { return super.m(ds) + " H.m(Double...)"; }
}

public class Test {
    public static void main(String[] args) {
        P p = new P();
        H h = new H();
        P p2 = h;
        System.out.println(...);
    }
}

```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato, assumendo che tutte le classi siano dichiarate nello stesso package.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

- (a) `p.m(42)`
- (b) `p2.m((byte) 42)`
- (c) `h.m((float) 42)`
- (d) `p.m(Short.valueOf((byte) 42))`
- (e) `p2.m(new Double[] { 4.2 })`
- (f) `h.m(new double[]{4.2})`

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

(a) Il literal `42` ha tipo statico `int`; il tipo statico di `p` è `P` ed entrambi i metodi di `P` sono accessibili e applicabili per sotto-tipo, ma quello con segnatura `m(int)` è più specifico.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(int)` in `P`. Viene stampata la stringa "`P.m(int)`".

(b) L'espressione è staticamente corretta per esattamente lo stesso motivo del punto precedente, visto che `p2` ha lo stesso tipo statico di `p`.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo con segnatura `m(int)` ridefinito in `H`. Poiché `i` ha tipo statico `int`, l'overloading per l'invocazione `super.m(i)` viene risolta come al punto precedente e, quindi, viene chiamato il metodo della classe `P` con segnatura `m(int)`.

Viene stampata la stringa "`P.m(int) H.m(int)`".

(c) L'espressione `new Long(42)` ha tipo statico `Long` poiché `42` ha tipo `int` che è sotto-tipo del tipo `long` del parametro dell'unico costruttore applicabile per sotto-tipo di `Long` (l'altro ha tipo `String`); il tipo statico di `p` è `P`. Nessun metodo della classe `P` è applicabile per sotto-tipo, mentre per unboxing l'unico metodo accessibile e applicabile è quello con segnatura `m(long)`.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(long)` in `P`. Viene stampata la stringa "`P.m(long)`".

(d) L'espressione `new Long(42)` ha tipo statico `Long` per gli stessi motivi del punto precedente e il tipo statico di `h` è `H`; nessun metodo in `H` è applicabile per sotto-tipo o per unboxing, quindi l'overloading viene risolto come al punto precedente.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo in `P` con segnatura "`P.m(long)`". Viene stampata la stringa "`P.m(long)`".

(e) Il literal `42` ha tipo statico `int`; il tipo statico di `p2` è `P` e nessun metodo accessibile di `P` è applicabile visto che entrambi i metodi hanno arità costante 1, quindi verrà segnalato un errore di tipo.

(f) Il literal `42` ha tipo statico `int` e il tipo statico di `h` è `H`; nessun metodo di `H` è applicabile per sotto-tipo o per boxing, dato che in entrambi i casi tutti i metodi considerati hanno arità costante 1. Il metodo in `H` con arità variabile e segnatura `m(Integer...)` è applicabile visto che `int` può essere convertito a `Integer` tramite boxing.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo in `H` con segnatura `m(Integer...)`. Poiché in questo caso vengono passati due argomenti `l.length` si valuta in `true` e viene eseguita l'invocazione `super.m(l[0])`; l'argomento `l[0]` ha tipo statico `Integer`, nessun metodo accessibile in `P` è applicabile per sotto-tipo, mentre entrambi i metodi sono applicabili per unboxing, ma quello con segnatura `m(int)` è più specifico.

Viene stampata la stringa "`P.m(int) H.m(Integer...)`".

```

public class P {
    String m(char c) { return "P.m(char)"; }
    String m(String s) { return "P.m(String)"; }
}

public class H extends P {
    String m(char c) { return super.m(c) + " H.m(char)"; }
    String m(Character c) { return super.m(c) + " H.m(Character)"; }
    String m(Character... cs) {
        StringBuilder sb = new StringBuilder();
        for (Character c : cs)
            sb.append(c).append(" ");
        return sb.append("H.m(Character...)").toString();
    }
}

public class Test {
    public static void main(String[] args) {
        P p = new P();
        H h = new H();
        P p2 = h;
        System.out.println(...);
    }
}

```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato, assumendo che tutte le classi siano dichiarate nello stesso package.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

- (a) `p.m('a')`
- (b) `p2.m('a')`
- (c) `h.m(Character.valueOf('a'))`
- (d) `p.m("a")`
- (e) `p2.m(new char[] { '4', '2' })`
- (f) `h.m(new Character[] { '4', '2' })`

Soluzione:

(a) Il literal `42` ha tipo statico `int`; il tipo statico di `p` è `P` ed esiste un solo metodo di `P` accessibile e applicabile per sotto-tipo, `String m(int i)`.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo `String m(int i)` in `P`. Viene stampata la stringa "`P.m(int)`".

(b) Il literal `42` ha tipo statico `int`, il cast a `byte` risulta staticamente corretto e il tipo statico dell'argomento è `byte`; il tipo statico di `p2` è `P` ed esistono due metodi di `P` accessibili e applicabili per sotto-tipo: `String m(byte b)` e `String m(int i)`. Viene selezionato il primo metodo, poiché è il più specifico.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi il metodo `String m(byte b)` viene cercato a partire da `H`, ma poiché il metodo non viene ridefinito viene eseguito quello di `P`. La conversione di tipo dovuta al cast non comporta in questo caso perdita di informazione, dato che la costante `42` è correttamente rappresentabile con un valore di tipo `byte`.

Viene stampata la stringa "`P.m(byte)`".

(c) Il literal `42` ha tipo statico `double`, il cast a `float` risulta staticamente corretto e il tipo statico dell'argomento è `float`; il tipo statico di `h` è `H` ed esiste un solo metodo di `H` accessibile e applicabile per sotto-tipo, `String m(float f)`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo `String m(float f)` in `H`. La conversione di tipo dovuta al cast non comporta in questo caso perdita di informazione, dato che la costante `42` è correttamente rappresentabile con un valore di tipo `float`.

Poiché il parametro `f` ha tipo statico `float`, per la chiamata `super.m(f)` non esiste alcun metodo in `P` di arità costante accessibile e applicabile per sotto-tipo o per conversione di tipo boxing/unboxing, mentre il metodo di arità variabile è accessibile e applicabile per boxing da `float` a `Float` e reference widening da `Float` a `Number`.

Viene stampata la stringa "`P.m(Number...) H.m(float)`".

(d) L'argomento di `Short.valueOf` ha tipo statico `byte` come al punto (b) ed esiste un solo metodo statico nella classe `Short` accessibile e applicabile per sotto-tipo, che restituisce un valore di tipo `Short`. Il tipo statico di `p` è `P` ed esiste un solo metodo di `P` accessibile e applicabile per sotto-tipo, `String m(int i)`.

Viene stampata la stringa "`P.m(int)`".

(e) Il literal `42` ha tipo statico `double`, l'argomento `new Double[]{4.2}` ha tipo statico `Double[]` (`4.2` viene implicitamente convertito tramite boxing) e il tipo statico di `p2` è `P`; esiste un solo metodo di `P` accessibile e applicabile per sotto-tipo, `String m(Number... n)` (durante il controllo di applicabilità per sotto-tipo il parametro `n` viene considerato di tipo `Number[]` e `Double[]` ≤ `Number[]` poiché `Double` ≤ `Number`).

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi il metodo `String m(byte b)` viene cercato a partire da `H`, ma poiché il metodo non viene ridefinito viene eseguito quello di `P`.

Viene stampata la stringa "`P.m(Number...)`".

(f) Il literal `42` ha tipo statico `double`, l'argomento `new double[]{4.2}` ha tipo statico `double[]` e il tipo statico di `h` è `H`; poiché `double[]` ≤ `Number[]` non esistono metodi accessibili e applicabili, quindi la chiamata non è staticamente corretta.

giugno 2015

4. Considerare le seguenti dichiarazioni di classi Java:

```

public class P {
    String m(double d) { return "P.m(double)"; }
    String m(int i) { return "P.m(int)"; }
    String m(Double d) { return "P.m(Double)"; }
}
public class H extends P {
    String m(double d) { return super.m(d) + " H.m(double)"; }
    String m(Object... os) { return "H.m(Object...)"; }
}
public class Test {
    public static void main(String[] args) {
        P p = new P();
        H h = new H();
        P p2 = h;
        System.out.println(...);
    }
}

```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

- (a) `p.m(42.)`
- (b) `p2.m(42.)`
- (c) `h.m(42.)`
- (d) `p.m(Integer.valueOf(42))`
- (e) `p2.m(new double[]{4.2})`
- (f) `h.m(new double[]{4.2})`

febbraio 2015

4. Considerare le seguenti dichiarazioni di classi Java:

```

public class P {
    String m(double d) {
        return "P.m(double)";
    }
    String m(Object o) {
        return "P.m(Object)";
    }
}
public class H extends P {
    String m(double d) {
        return super.m(d) + " H.m(double)";
    }
    String m(Double d) {
        return super.m(d) + " H.m(Double)";
    }
    String m(double... ds) {
        return "H.m(double...)";
    }
}
public class Test {
    public static void main(String[] args) {
        P p = new P();
        H h = new H();
        P p2 = h;
        System.out.println(...);
    }
}

```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

- (a) `p.m(42.42)`
- (b) `p2.m(42.42)`
- (c) `p.m(Double.valueOf(42))`
- (d) `p2.m(Float.valueOf(42))`
- (e) `h.m(Double.valueOf(42))`
- (f) `h.m(4,2)`

gennaio 2015

4. Considerare le seguenti dichiarazioni di classi Java:

```

public class P {
    String m(int i) {
        return "P.m(int)";
    }
    String m(Object o) {
        return "P.m(Object)";
    }
}
public class H extends P {
    String m(int i) {
        return super.m(i) + " H.m(int)";
    }
    String m(Integer i) {
        return super.m(i) + " H.m(Integer)";
    }
    String m(int... is) {
        return "H.m(int...)";
    }
}
public class Test {
    public static void main(String[] args) {
        P p = new P();
        H h = new H();
        P p2 = h;
        System.out.println(...);
    }
}

```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

- (a) Il literal `42.` ha tipo statico `double`; il tipo statico di `p` è `P` ed esiste un solo metodo di `P` accessibile e applicabile per sotto-tipo, `String m(double d)`.
A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo `String m(double d)` in `P`. Viene stampata la stringa "`P.m(double)`".
- (b) L'espressione è staticamente corretta per esattamente lo stesso motivo del punto precedente, visto che `p2` ha lo stesso tipo statico di `p`.
A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo `String m(double d)` in `H`. Per gli stessi motivi del punto precedente, viene stampata la stringa "`P.m(double) H.m(double)`".
- (c) Il literal `42.` ha tipo statico `double`; il tipo statico di `h` è `H` ed esiste un solo metodo di `H` accessibile e applicabile per sotto-tipo, `String m(double d)`.
A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo `String m(double d)` in `H`. Per gli stessi motivi del punto precedente, viene stampata la stringa "`P.m(double) H.m(double)`".
- (d) Il literal `42` ha tipo statico `int` e per l'invocazione `Integer.valueOf(42)` esiste un solo metodo statico nella classe `Integer` accessibile e applicabile per sotto-tipo, che restituisce un valore di tipo `Integer`. Il tipo statico di `p` è `P`, non esistono metodi in `P` accessibili e applicabili per sotto-tipo, mentre se l'argomento viene convertito in un valore di tipo `int` tramite unboxing i due metodi di `P` `String m(double d)` e `String m(int i)` sono entrambi accessibili e applicabili, ma il secondo è più specifico.
Viene stampata la stringa "`P.m(int)`".
- (e) L'espressione `new double[]{4.2}` ha tipo statico `double[]`, il tipo statico di `p2` è `P` e non esiste alcun metodo di `P` accessibile e applicabile, quindi viene segnalato un errore durante la compilazione.
- (f) L'espressione `new double[]{4.2}` ha tipo statico `double[]`, il tipo statico di `h` è `H` e l'unico metodo di `H` che è accessibile e applicabile è `String m(Object... os)`, poiché nessun metodo è applicabile per sotto-tipo o per boxing/unboxing, mentre `Object...` corrisponde a `Object` e `double[] ≤ Object`.
A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo `String m(Object... os)` in `H`. La conversione da `int` a `double` è senza perdita di informazione.
Viene stampata la stringa "`H.m(Object...)`".

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

- (a) Il literal `42.42` ha tipo statico `double`; il tipo statico di `p` è `P` ed esiste un solo metodo di `P` accessibile e applicabile per sotto-tipo, `String m(double d)`.
A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo `String m(double d)` in `P`. Viene stampata la stringa "`P.m(double)`".
- (b) L'espressione è staticamente corretta per esattamente lo stesso motivo del punto precedente, visto che `p2` ha lo stesso tipo statico di `p`.
A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo `String m(double d)` in `H`. Poiché `d` ha tipo statico `double`, per l'invocazione `super.m(d)` esiste un solo metodo in `P` accessibile e applicabile per sotto-tipo, `String m(double d)`.
Viene stampata la stringa "`P.m(double) H.m(double)`".
- (c) Il literal `42` ha tipo statico `int` e per l'invocazione `Double.valueOf(42)` esiste un solo metodo statico nella classe `Double` accessibile e applicabile per sotto-tipo, che restituisce un valore di tipo `Double`. Il tipo statico di `p` è `P` ed esiste un solo metodo di `P` accessibile e applicabile per sotto-tipo, `String m(Object i)`.
A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo `String m(Object o)` in `P`. La conversione di `42` da `int` a `double` non comporta perdita di informazione.
Viene stampata la stringa "`P.m(Object)`".
- (d) Il literal `42` ha tipo statico `int` e per l'invocazione `Float.valueOf(42)` esiste un solo metodo statico nella classe `Float` accessibile e applicabile per sotto-tipo, che restituisce un valore di tipo `Float`. Il tipo statico di `p2` è `P` ed esiste un solo metodo di `P` accessibile e applicabile per sotto-tipo, `String m(Object i)`.
A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo `String m(Object o)` in `H`. La conversione di `42` da `int` a `double` non comporta perdita di informazione.
Viene stampata la stringa "`P.m(Object)`".
- (e) Come al punto (c), l'espressione `Double.valueOf(42)` ha tipo statico `Double`. Il tipo statico di `h` è `H` ed esiste un solo metodo di `H` accessibile e applicabile per sotto-tipo, `String m(Double d)`.
A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo `String m(Double d)` in `H`. Poiché `d` ha tipo statico `Double`, per l'invocazione `super.m(d)` esiste un solo metodo in `P` accessibile e applicabile per sotto-tipo, `String m(Object o)`.
Viene stampata la stringa "`P.m(Object) H.m(Double)`".
- (f) La variabile `h` ha tipo statico `H`, mentre il literal `4` e `2` hanno entrambi tipo `int`. e poiché in `P` non esiste alcun metodo con due argomenti, l'unico metodo accessibile e applicabile è `String m(double... ds)`, visto che `int` è sotto-tipo di `double`.
A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo `String m(double... ds)` in `H`. La conversione da `int` a `double` è senza perdita di informazione.
Viene stampata la stringa "`H.m(double...)`".

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

- (a) Il literal `42` ha tipo statico `int` il cast è staticamente corretto (narrowing primitive conversion) e l'espressione `(short) 42` ha tipo `short`; in accordo con la sua dichiarazione, il tipo statico di `p` è `P` ed esiste un solo metodo di `P` accessibile e applicabile per sotto-tipo, `String m(int i)`.
A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo `String m(int i)` in `P`. Durante la conversione da `int` a `short` non c'è perdita di informazione visto che `42` è rappresentabile in complemento a 2 su 16 bit e, ovviamente, non c'è perdita di informazione nella conversione inversa da `short` a `int` dovuta al passaggio del parametro.
Viene stampata la stringa "`P.m(int)`".
- (b) L'espressione è staticamente corretta per esattamente lo stesso motivo del punto precedente, visto che `p2` ha lo stesso tipo statico di `p`.
A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo `String m(int i)` in `H`. Per le conversioni a cui è soggetto il literal `42` valgono le stesse considerazioni del punto precedente. Poiché `i` ha tipo statico `int`, per l'invocazione `super.m(i)` esiste un solo metodo in `P` accessibile e applicabile per sotto-tipo, `String m(int i)`.
Viene stampata la stringa "`P.m(int) H.m(int)`".
- (c) Il literal `42` ha tipo statico `int` e per l'invocazione `Integer.valueOf(42)` esiste un solo metodo statico nella classe `Integer` accessibile e applicabile, che restituisce un valore di tipo `Integer`. Il tipo statico di `p` è `P` ed esiste un solo metodo di `P` accessibile e applicabile per sotto-tipo, `String m(Object i)`.
A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo `String m(Object o)` in `P`. Viene stampata la stringa "`P.m(Object)`".
- (d) L'espressione è staticamente corretta per esattamente lo stesso motivo del punto precedente, visto che `p2` ha lo stesso tipo statico di `p`.
A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, ma poiché `H` non definisce alcun metodo "`P.m(Object)`", viene eseguito il metodo ereditato dalla superclasse `P`.
Viene stampata la stringa "`P.m(Object)`".
- (e) Per gli stessi motivi dei punti (c) e (d), l'espressione `Integer.valueOf(42)` ha tipo statico `Integer`, mentre `h` ha tipo statico `H` ed esiste un solo metodo di `H` accessibile e applicabile per sotto-tipo, `String m(Integer i)`.
A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo `String m(Integer i)` in `H`. Poiché `i` ha tipo statico `Integer`, per l'invocazione `super.m(i)` esiste un solo metodo in `P` accessibile e applicabile per sotto-tipo, `String m(Integer i)`.
Viene stampata la stringa "`P.m(Integer) H.m(Integer)`".
- (f) La variabile `p2` ha tipo statico `P` e poiché in `P` non esiste alcun metodo accessibile e applicabile a due argomenti, l'espressione non è staticamente corretta.