Esercizi di Programmazione ad Oggetti, Vacanze Natalizie 2017/2018

C++

Esercizio 1

Si consideri il seguente modello concernente una app Gallo© che offre delle funzionalità di galleria di foto/video.

(A) Definire la seguente gerarchia di classi.

1. Definire una classe base polimorfa astratta `GalloFile` i cui oggetti rappresentano un file grafico memorizzabile dalla app Gallo©. Ogni `GalloFile` è caratterizzato dalla dimensione in MB. La classe include un metodo virtuale puro di "clonazione" `GalloFile* clone()`, che prevede il contratto standard della clonazione polimorfa di oggetti, ed un metodo virtuale puro `bool highQuality()` con il seguente contratto puro: `f->highQuality()` ritorna true se il file grafico `*f` è considerato di alta qualità, altrimenti ritorna false.

2. Definire una classe concreta `Foto` derivata da `GalloFile` i cui oggetti rappresentano un file immagine scattato da una fotocamera. Ogni oggetto `Foto` è caratterizzato dalla sensibilità ISO della fotocamera (un intero positivo) e dall'essere stata scattata con il flash oppure senza flash. La classe `Foto` implementa i metodi virtuali puri nel seguente modo: `f.clone()` ritorna un puntatore ad un oggetto `Foto` che è una copia di `f`; inoltre, `f.highQuality()` ritorna true quando la sensibilità ISO di `f` è almeno 500.

3. Definire una classe concreta `Video` derivata da `GalloFile` i cui oggetti rappresentano un file video girato da una fotocamera. Ogni oggetto `Video` è caratterizzato dalla durata in secondi e dall'essere in formato FullHD o superiore oppure no. La classe `Video` implementa i metodi virtuali puri nel seguente modo: `v.clone()` ritorna un puntatore ad un oggetto `Video` che è una copia di `v`; inoltre, `v.highQuality()` ritorna true quando `v` è in formato FullHD o superiore.

(B) Definire una classe `Gallo` i cui oggetti rappresentano una installazione dell'app Gallo©. Un oggetto `g` di `Gallo` è quindi caratterizzato dall'insieme di tutti i file grafici memorizzati da `g`. La classe `Gallo` rende disponibili i seguenti metodi:

1. Un metodo `vector<GalloFile*> selectHQ()` con il seguente comportamento: una invocazione `g.selectHQ()` ritorna il vector dei puntatori ai `GalloFile` memorizzati in `g` che: (i) sono considerati di alta qualità e (ii) se sono una `Foto` allora devono essere stati scattati con il flash.

2. Un metodo `void removeNonFoto(double)` con il seguente comportamento: una invocazione `g.removeNonFoto(size)` elimina tutti i file grafici memorizzati da `g` che: (i) non sono una `Foto` e (ii) hanno una dimensione maggiore di `size`; nel caso non venga eliminato alcun file grafico da `g` allora deve essere sollevata l'eccezione `std::logic_error("NoRemove")`, ricordando che `std::logic_error` è un tipo di eccezioni della libreria standard.

3. Un metodo `const GalloFile* insert(const GalloFile* pf)` con il seguente comportamento: se l'oggetto `*pf` non è un `Video` oppure se `*pf` è un `Video` di durata minore ad un minuto allora l'invocazione `g.insert(pf)` inserisce `*pf` tra i file grafici memorizzati da `g` e quindi ritorna un puntatore all'oggetto inserito; se invece `*pf` non viene inserito tra i file grafici memorizzati da `g` allora ritorna il puntatore nullo.

Esercizio 2

Si consideri il seguente modello di realtà concernente l'app InForma per archiviare allenamenti sportivi.

(A) Definire la seguente gerarchia di classi.

1. Definire una classe base polimorfa astratta `Workout` i cui oggetti rappresentano un allenamento (workout) archiviabile in InForma. Ogni `Workout` è caratterizzato dalla durata temporale espressa in minuti. La classe è astratta in quanto prevede i seguenti **metodi virtuali puri**:

- un metodo di "clonazione": `Workout* clone()`.
- un metodo `int calorie()` con il seguente contratto puro: `w->calorie()` ritorna il numero di calorie consumate durante l'allenamento `*w`.

2. Definire una classe concreta `Corsa` derivata da `Workout` i cui oggetti rappresentano un allenamento di corsa. Ogni oggetto `Corsa` è caratterizzato dalla distanza percorsa espressa in Km. La classe `Corsa` implementa i metodi virtuali puri di `Workout` come segue:
   - implementazione della clonazione standard per la classe `Corsa`.
   - per ogni puntatore `p` a `Corsa`, `p->calorie()` ritorna il numero di calorie dato dalla formula $500K^2/D$, dove K è la distanza percorsa in Km nell'allenamento `*p` e D è la durata in minuti dell'allenamento `*p`.
3. Definire una classe astratta `Nuoto` derivata da `Workout` i cui oggetti rappresentano un generico allenamento di nuoto che non specifica lo stile di nuoto. Ogni oggetto `Nuoto` è caratterizzato dal numero di vasche nuotate.
4. Definire una classe concreta `StileLibero` derivata da `Nuoto` i cui oggetti rappresentano un allenamento di nuoto a stile libero. La classe `StileLibero` implementa i metodi virtuali puri di `Nuoto` come segue:
   - implementazione della clonazione standard per la classe `StileLibero`.
   - per ogni puntatore `p` a `StileLibero`, `p->calorie()` ritorna il seguente numero di calorie: se D è la durata in minuti dell'allenamento `*p` e V è il numero di vasche nuotate nell'allenamento `*p` allora quando $D < 10$ le calorie sono $35V$, mentre se $D \geq 10$ le calorie sono $40V$.
5. Definire una classe concreta `Dorso` derivata da `Nuoto` i cui oggetti rappresentano un allenamento di nuoto a stile dorso. La classe `Dorso` implementa i metodi virtuali puri di `Nuoto` come segue:
   - implementazione della clonazione standard per la classe `Dorso`.
   - per ogni puntatore `p` a `Dorso`, `p->calorie()` ritorna il seguente numero di calorie: se D è la durata in minuti dell'allenamento `*p` e V è il numero di vasche nuotate nell'allenamento `*p` allora quando $D < 15$ le calorie sono $30V$, mentre se $D \geq 15$ le calorie sono $35V$.
6. Definire una classe concreta `Rana` derivata da `Nuoto` i cui oggetti rappresentano un allenamento di nuoto a stile rana. La classe `Rana` implementa i metodi virtuali puri di `Nuoto` come segue:
   - implementazione della clonazione standard per la classe `Rana`.
   - per ogni puntatore `p` a `Rana`, `p->calorie()` ritorna $25V$ calorie dove V è il numero di vasche nuotate nell'allenamento `*p`.

(B) Definire una classe `InForma` i cui oggetti rappresentano una installazione dell'app. Un oggetto di `InForma` è quindi caratterizzato da un contenitore di elementi di tipo `const Workout*` che contiene tutti gli allenamenti archiviati dall'app. La classe `InForma` rende disponibili i seguenti metodi:

1. Un metodo `vector<Nuoto*> vasche(int)` con il seguente comportamento: una invocazione `app.vasche(v)` ritorna un STL vector di puntatori a **copie** di tutti e soli gli allenamenti a nuoto memorizzati in `app` con un numero di vasche percorse $>$ `v`.
2. Un metodo `vector<Workout*> calorie(int)` con il seguente comportamento: una invocazione `app.calorie(x)` ritorna un vector contenente dei puntatori a **copie** di tutti e soli gli allenamenti memorizzati in `app` che : (i) hanno comportato un consumo di calorie $>$ `x`; e (ii) non sono allenamenti di nuoto a rana.
3. Un metodo `void removeNuoto()` con il seguente comportamento: una invocazione `app.removeNuoto()` rimuove dagli allenamenti archiviati in `app` **tutti** gli allenamenti a nuoto che abbiano il massimo numero di calorie tra tutti gli allenamenti a nuoto; se `app` non ha archiviato alcun allenamento a nuoto allora viene sollevata l'eccezione "`NoRemove`" di tipo `std::string`.

Esercizio 3

```
class A {
protected:
  virtual void h() {cout<<" A::h ";}
public:
  virtual void g() const {cout <<" A::g ";}
  virtual void f() {cout <<" A::f "; g(); h();}
  void m() {cout <<" A::m "; g(); h();}
  virtual void k() {cout <<" A::k "; h(); m(); }
  virtual A* n() {cout <<" A::n "; return this;}
};

class B: public A {
protected:
  virtual void h() {cout <<" B::h ";}
public:
  virtual void g() {cout <<" B::g ";}
  void m() {cout <<" B::m "; g(); h();}
  void k() {cout <<" B::k "; g(); h();}
  B* n() {cout <<" B::n "; return this;}
};

class C: public B {
protected:
  virtual void h() const {cout <<" C::h ";}
public:
  virtual void g() {cout <<" C::g ";}
  void m() {cout <<" C::m "; g(); k();}
  void k() const {cout <<" C::k "; h();}
};

A* p2 = new B(); A* p3 = new C(); B* p4 = new B(); B* p5 = new C(); const A* p6 = new C();
```

Le precedenti definizioni compilano correttamente. Per ognuna delle seguenti istruzioni scrivere nell'apposito spazio:

- **NON COMPILA** se la compilazione dell'istruzione provoca un errore;
- **ERRORE RUN-TIME** se l'istruzione compila correttamente ma la sua esecuzione provoca un errore a run-time;
- se l'istruzione compila correttamente e non provoca errori a run-time allora si scriva la stampa che l'esecuzione produce in output su `cout`; se non provoca alcuna stampa allora si scriva **NESSUNA STAMPA**.

```
p2->f(); .....

p2->m(); .....

p3->k(); .....

p3->f(); .....

p4->m(); .....

p4->k(); .....

p4->g(); .....

p5->g(); .....

p6->k(); .....

p6->g(); .....

(p3->n())->m(); .....

(p3->n())->g(); .....

(p3->n())->n()->g(); .....

(p5->n())->g(); .....

(p5->n())->m(); .....

(dynamic_cast<B*>(p2))->m(); .....

(static_cast<C*>(p3))->k(); .....

( static_cast<B*>(p3->n()) )->g(); .....
```

Esercizio 4

Si considerino le seguenti dichiarazioni di classi di qualche libreria grafica, dove gli oggetti delle classi `Container`, `Component`, `Button` e `MenuItem` sono chiamati, rispettivamente, contenitori, componenti, pulsanti ed entrate di menu.

```
class Component;

class Container {
public:
  virtual ~Container();
  vector<Component*> getComponents() const;
};

class Component: public Container {};

class Button: public Component {
public:
  vector<Container*> getContainers() const;
};

class MenuItem: public Button {
public:
  void setEnabled(bool b = true);
};

class NoButton {};
```

Assumiamo i seguenti fatti.

1. Il comportamento del metodo `getComponents()` della classe `Container` è il seguente: `c.getComponents()` ritorna un vector di puntatori a tutte le componenti inserite nel contenitore `c`; se `c` non ha alcuna componente allora ritorna un vector vuoto.
2. Il comportamento del metodo `getContainers()` della classe `Button` è il seguente: `b.getContainers()` ritorna un vector di puntatori a tutti i contenitori che contengono il pulsante `b`; se `b` non appartiene ad alcun contenitore allora ritorna un vector vuoto.
3. Il comportamento del metodo `setEnabled()` della classe `MenuItem` è il seguente: `mi.setEnabled(b)` abilita (con `b==true`) o disabilita (con `b==false`) l'entrata di menu `mi`.

Definire una funzione `Button** Fun(const Container&)` con il seguente comportamento: in ogni invocazione `Fun(c)`

1. Se `c` contiene almeno una componente `Button` allora

   ritorna un puntatore alla prima cella di un array dinamico di puntatori a pulsanti contenente tutti e soli i puntatori ai pulsanti che sono componenti del contenitore `c`; inoltre tutte le componenti del contenitore `c` che sono una entrata di menu e sono contenute in almeno 2 contenitori devono essere disabilitate disabilitate.

2. Se invece `c` non contiene nessuna componente `Button` allora ritorna il puntatore nullo.

Esercizio 5

Definire un template di funzione `Fun(T1*, T2&)` che ritorna un booleano con il seguente comportamento. Consideriamo una istanziazione implicita `Fun(p,r)` dove supponiamo che i parametri di tipo `T1` e `T2` siano istanziati a tipi polimorfi (cioè che contengono almeno un metodo virtuale). Allora `Fun(p,r)` ritorna `true` se e soltanto se valgono le seguenti condizioni:

1. i parametri di tipo `T1` e `T2` sono istanziati allo stesso tipo;
2. siano `D1*` il tipo dinamico di `p` e `D2&` il tipo dinamico di `r`. Allora (i) `D1` e `D2` sono lo stesso tipo e (ii) questo tipo è un sottotipo proprio della classe `ios` della gerarchia di classi di I/O (si ricordi che `ios` è la classe base astratta della gerarchia).

Ad esempio, il seguente `main()` deve compilare e provocare le stampe indicate:

```
#include<iostream>
#include<fstream>
#include<typeinfo>
using namespace std;
```

```
class C { public: virtual ~C() {} };

main() {
  ifstream f("pippo"); fstream g("pluto"), h("zagor"); iostream* p = &h;
  C c1,c2;
  cout << Fun(&cout,cin) << endl; // stampa: 0
  cout << Fun(&cout,cerr) << endl; // stampa: 1
  cout << Fun(p,h) << endl; // stampa: 0
  cout << Fun(&f,*p) << endl; // stampa: 0
  cout << Fun(&g,h) << endl; // stampa: 1
  cout << Fun(&c1,c2) << endl; // stampa: 0
}
```

Esercizio 6

Si considerino i seguenti fatti concernenti la libreria di I/O standard.

- Si ricorda che `ios` è la classe base di tutta la gerarchia di classi della libreria di I/O, che la classe `istream` è derivata direttamente e virtualmente da `ios` e che la classe `ifstream` è derivata direttamente da `istream`.
- La classe base `ios` ha il distruttore virtuale. La classe `ios` rende disponibile un metodo costante e non virtuale `bool fail()` con il seguente comportamento: una invocazione `s.fail()` ritorna `true` se e solo se lo stream `s` è in uno stato di fallimento (cioè, il failbit di `s` vale 1).
- La classe `istream` rende disponibile un metodo non costante e non virtuale `long tellg()` con il seguente comportamento: una invocazione `s.tellg()`:
  1. se `s` è in uno stato di fallimento allora ritorna -1;
  2. altrimenti, cioè se `s` non è in uno stato di fallimento, ritorna la posizione della cella corrente di input di `s`.
- La classe `ifstream` rende disponibile un metodo non costante e non virtuale `bool is_open()` con il seguente comportamento: una invocazione `s.is_open()` ritorna `true` se e solo se il file associato allo stream `s` è aperto.

Definire una funzione `long Fun(const ios&)` con il seguente comportamento: una invocazione `Fun(s)`:

(1) se `s` è in uno stato di fallimento lancia una eccezione di tipo `Fallimento`; si chiede anche di definire tale classe `Fallimento`;

(2) se `s` non è in uno stato di fallimento allora:

  (a) se `s` non è un `ifstream` ritorna -2;

  (b) se `s` è un `ifstream` ed il file associato non è aperto ritorna -1;

  (c) se `s` è un `ifstream` ed il file associato è aperto ritorna la posizione della cella corrente di input di `s`.

Esercizio 7

Definire un template di classe `Coda<T>` i cui oggetti rappresentano una struttura dati coda per elementi di uno stesso tipo `T`, ossia la coda implementa l'usuale politica FIFO (First In First Out) di inserimento/estrazione degli elementi: gli elementi vengono estratti nello stesso ordine in cui sono stati inseriti. Il template `Coda<T>` deve soddisfare i seguenti vincoli:

1. `Coda<T>` non può usare i contenitori STL come campi dati (inclusi puntatori e riferimenti a contenitori STL).
2. Il parametro di tipo del template `Coda<T>` ha come valore di default `int`.
3. Gestione della memoria senza condivisione.
4. Deve essere disponibile un costruttore di default che costruisce la coda vuota.
5. Deve essere disponibile un costruttore `Coda(int k, const T& t)` che costruisce una coda contenente `k` copie dell'elemento `t`.
6. Deve essere disponibile un metodo `void insert(const T&)` con il seguente comportamento: `c.insert(t)` inserisce l'elemento `t` in coda a `c` in tempo costante.

7. Deve essere disponibile un metodo `T removeNext()` con il seguente comportamento: se la coda `c` non è vuota, `c.removeNext()` rimuove l'elemento in testa alla coda `c` in tempo costante e lo ritorna; se invece `c` è vuota allora solleva una eccezione di tipo `Empty`, una classe di eccezioni di cui è richiesta la definizione.

8. Deve essere disponibile un metodo `T* getNext()` con il seguente comportamento: se la coda `c` non è vuota, `c.getNext()` ritorna un puntatore all'elemento in testa a `c` in tempo costante; se invece `c` è vuota ritorna il puntatore nullo.

9. Overloading dell'operatore di uguaglianza.

10. Overloading dell'operatore di somma che agisca come concatenazione: `c + d` è la coda che si ottiene aggiungendo `d` in coda a `c`.

Esercizio 8

```
class A {
  public:
    virtual ~A() {}
};

class B: public A {};

class C: public A {};

class D: public C {};

template<class T>
void Fun(T* pt){
  bool b=0;
  try{ throw T(*pt); }
  catch(B){cout << "B "; b=1;}
  catch(C){cout << "C "; b=1;}
  catch(D){cout << "D "; b=1;}
  catch(A){cout << "A "; b=1;}
  if(!b) cout << "NO ";
}

A a; B b; C c; D d;
A *pa1 = &b, *pa2 = &c, *pa3 = &d;
B *pb1=dynamic_cast<B*>(pa1);  B *pb2=dynamic_cast<B*>(pa3);
```

Le precedenti definizioni compilano senza provocare errori (con gli opportuni `#include` e `using`). Per ognuna delle seguenti istruzioni di invocazione della funzione `Fun` scrivere nell'apposito spazio:

- **NON COMPILA** se la compilazione dell'istruzione provoca un errore;
- **ERRORE RUN-TIME** se l'istruzione compila correttamente ma la sua esecuzione provoca un errore a run-time;
- se l'istruzione compila correttamente e non provoca errori a run-time allora si scriva la stampa che l'esecuzione produce in output su `cout`; se non provoca alcuna stampa allora si scriva **NESSUNA STAMPA**.

`Fun(&a);`	
`Fun(&c);`	
`Fun(&d);`	
`Fun(pa1);`	
`Fun(pa2);`	
`Fun(pa3);`	
`Fun(pb1);`	
`Fun(pb2);`	
`Fun<A>(pb1);`	
`Fun<A>(pb2);`	
`Fun<A>(pa3);`	
`Fun<B>(&d);`	
`Fun<B>(pb1);`	
`Fun<C>(pa3);`	
`Fun<C>(&d);`	
`Fun<D>(pa3);`	

JAVA

Esercizio 9

Si consideri la seguente gerarchia di tipi.

```
interface X { void f(); }

class B {
    public void g() { System.out.print("B.g() "); }
}

class C extends B implements X {
    public void f() { System.out.print("C.f() "); }
    public void f(Object ref) { System.out.print("C.f(Object) "); }
}

abstract class A extends B implements X {
    public void f() { System.out.print("A.f() "); }
    public void g() { System.out.print("A.g() "); }
    public abstract B f(B ref);
}

class D extends A {
    public static B st = new B();
    public void f() { System.out.print("D.f() "); }
    public B f(B ref) { if (ref instanceof A) return (D)ref; return st; }
}
```

Si supponga che ognuno dei seguenti frammenti sia il codice di un metodo `main()` che può accedere alla precedente gerarchia. Si scriva nell'apposito spazio contiguo:

- **NON COMPILA** quando il `main()` non compila;
- **ECCEZIONE** quando il `main()` compila ma la sua esecuzione provoca una eccezione;
- la stampa che produce in output nel caso in cui il `main()` compili correttamente ed esegua senza sollevare eccezioni; se non provoca alcuna stampa si scriva **NESSUNA STAMPA**.

```
B b = new A();
X x = (A)b;
x.f();              ..........................................................................

A a = new D();
B b = a;
b.g();
a.g();
a.f(b);             ..........................................................................

B b  = new D();
A a1 = (A)b;
A a2 = (D)b;
a1.f();
a2.f();             ..........................................................................

D d = new D();
B b1 = d;
B b2 = d.f(b1);
b2.g();             ..........................................................................

B b1  = new B();
A a  = new D();
B b2 = a.f(b1);
X x = (D)b2;        ..........................................................................

X x = new C();
C c  = (C)x;
B b = new D();
c.f(b);             ..........................................................................

X x = new C();
B b = new B();
x.f(b);             ..........................................................................

A a = new D();
C c = (C)(a.f(a));  ..........................................................................
```

Esercizio 10

Si considerino i seguenti fatti concernenti la libreria API:

- `AbstractButton` è una classe astratta; `JButton` è una sottoclasse concreta di `AbstractButton`; `BasicArrowButton` è una sottoclasse di `JButton` i cui oggetti sono pulsanti con una freccia direzionale in una delle direzioni cardinali
- La classe `AbstractButton` include un metodo concreto `public void setText(String)` tale che: `ab.setText(s)` imposta la label di `ab` alla stringa `s`
- La classe `BasicArrowButton` include un metodo `public void setDirection(int)` tale che: `bab.setDirection(x)` imposta la direzione cardinale di `bab` di invocazione, dove `x` è una direzione tra i campi dati statici final `SwingConstants.NORTH`, `SwingConstants.SOUTH`, `SwingConstants.EAST`, `SwingConstants.WEST`

Definire una funzione statica (da inserire in qualsiasi classe) `void Fun(AbstractButton)` con il seguente comportamento: una invocazione `Fun(ab)` imposta la label di `ab` a "`pippo`" quando `ab` è un `JButton` che non è un `BasicArrowButton`, mentre se `ab` è un `BasicArrowButton` imposta `ab` con direzione "North".

Esercizio 11

Si considerino i seguenti fatti concernenti il package `java.util` della libreria API:

- `Collection` è un'interfaccia i cui oggetti rappresentano una collezione di oggetti di qualsiasi tipo detti elementi. `Collection` include la dichiarazione di metodo `public int size()` con il seguente contratto: "ritorna il numero di elementi della collezione d'invocazione".

- `List` è una sottointerfaccia di `Collection` i cui oggetti rappresentano una "lista", ovvero una collezione ordinata. `List` include la dichiarazione di metodo `public Object get(int i)` (quindi non incluso in `Collection`) con il seguente contratto: "ritorna l'elemento in posizione `i` nella lista d'invocazione" (come al solito gli indici partono da 0).
- `Vector` è l'implementazione di `List` già nota.
- `LinkedList` è una implementazione di `List` i cui oggetti rappresentano una lista realizzata tramite l'usuale definizione ricorsiva. `LinkedList` include inoltre un metodo `public Object removeFirst()` che rimuove il primo elemento della lista di invocazione e lo ritorna.

Definire una classe `Controllata` di eccezioni controllate (cioè da controllare obbligatoriamente), una classe `NonControllata` di eccezioni non controllate e una funzione statica (da inserire in qualsiasi classe) `Object Fun(Collection)` con il seguente comportamento:

una invocazione `Fun(c)`

1. lancia una eccezione `Controllata` quando `c` è il reference nullo;
2. ritorna il secondo elemento di `c` quando `c` è un `Vector` che contiene almeno 3 elementi;
3. rimuove e ritorna il primo elemento di `c` quando `c` è una `LinkedList` non vuota;
4. lancia una eccezione `NonControllata` in tutti gli altri casi.

Esercizio 12

Si consideri la seguente gerarchia di tipi.

```
interface X { public void f(); }

interface Y { public int g(); }

class F implements X {
    public void f() { System.out.print("F.f() "); }
}
class C implements X {
    public void f() { System.out.print("C.f() "); }
}
class D extends C implements Y {
    public int g() { System.out.print("D.g() "); return 1; }
}
class E extends D {
    public int g() { System.out.print("E.g() "); return 2; }
}
```

Si supponga che ognuno dei seguenti frammenti sia il codice di un metodo `main()` di qualche classe che può accedere alla precedente gerarchia. Si scriva nell'apposito spazio contiguo:

- **NON COMPILA** quando il `main()` non compila;
- **ECCEZIONE** quando il il `main()` compila correttamente ma la sua esecuzione provoca una eccezione;
- la stampa che produce in output nel caso invece il `main()` compili correttamente ed esegua senza lanciare eccezioni; se non provoca alcuna stampa si scriva **NESSUNA STAMPA**.

```
X x = new D();
C c = x;
x.f();              .....................................................................

X x  = new F();
Y y  = x;
y.g();              .....................................................................

X x  = new C();
Y y  = (C)x;
y.g();              .....................................................................
```

```
Y y  = new E();
D d  = (D)y;
d.f(); y.g();      ..........................................................................

Y y  = new D();
X x  = (E)y;
x.f(); y.g();      ..........................................................................

C c  = new E();
Y y  = (D)c;
((E)c).f();
y.f();             ..........................................................................

C c  = new D();
X x  = (D)c;
Y y =  (D)x;
y.g(); x.f();      ..........................................................................

Y y = new D();
X x  = (D)y;
((E)x).g();        ..........................................................................
```

Esercizio 13

Si considerino le seguenti classi la cui compilazione non provoca errori.

```
class A {
  public void print(String s) { System.out.print(s + " "); }
  public void m1() { print("A.m1"); m2(); }
  public void m2() { print("A.m2"); }
}
class B extends A {
  public void m2() { print("B.m2"); }
  public void m3() { print("B.m3"); }
}
class C extends A {
  public void m1() { print("C.m1"); }
  public void m2() { print("C.m2"); m1(); }
}
class D extends C {
  public void m1() { super.m1(); print("D.m1"); }
  public void m3() { print("D.m3"); }
}
```

Si considerino inoltre le seguenti dichiarazioni (esterne alla precedente gerarchia), la cui compilazione non provoca errori.

```
A ref1 = new B(); A ref2 = new D();  B ref3 = new B();
C ref4 = new C(); C ref5 = new D();  Object ref6 = new C();
```

Per ognuna delle seguenti istruzioni si scriva nell'apposito spazio a fianco:

- **NON COMPILA** se la compilazione dell'istruzione provoca un errore;
- **ECCEZIONE** se l'istruzione compila correttamente ma la sua esecuzione provoca una eccezione;
- la stampa che produce in output nel caso in cui l'istruzione compili correttamente ed esegua senza provocare eccezioni; se non provoca nessuna stampa si scriva **NESSUNA STAMPA**.

`ref1.m1();`	
`ref2.m1();`	
`ref4.m1();`	
`ref5.m1();`	
`ref6.m1();`	
`ref1.m2();`	
`ref2.m2();`	
`ref3.m2();`	
`ref4.m2();`	
`ref6.m2();`	
`ref3.m3();`	
`ref5.m3();`	
`((B)ref1).m3();`	
`((D)ref4).m3();`	
`((D)ref5).m3();`	
`((B)ref2).m3();`	
`((C)ref2).m2();`	
`((D)ref6).m2();`	

Esercizio 14

Definire una classe `Orario` i cui oggetti rappresentano un orario della giornata, ad esempio "00:00:00", "10:03:45", "15:20:00", "23:59:59", etc. La classe `Orario` deve soddisfare le seguenti specifiche:

- fornisce un costruttore senza argomenti `Orario()`, un costruttore a due argomenti `Orario(int h,int m)` che inizializza ad `h` le ore e a `m` i minuti, ed un costruttore `Orario(int h, int m, int s)` che costruisce l'oggetto "`h:m:s`"
- fornisce dei metodi `int Ore()`, `int Minuti()`, `int Secondi()` che ritornano le ore, i minuti ed i secondi dell'orario di invocazione
- fornisce un metodo `boolean minore(Orario r)` che ritorna `true` se e soltanto se l'oggetto di invocazione rappresenta un orario antecedente o uguale all'orario rappresentato dall'argomento `r`
- fornisce un metodo `boolean sera()` che ritorna `true` se e soltanto se l'oggetto di invocazione rappresenta un orario tra le "19:00:00" e le "23:59:59"
- ridefinisce il metodo `boolean equals(Object r)` di `Object` in modo che ritorni `true` se e soltanto se l'oggetto di invocazione e l'argomento `r` rappresentano lo stesso orario

Definire inoltre una sottoclasse `DataOra` derivata da `Orario` i cui oggetti rappresentano una data[1] con orario, ad esempio "11/7/2003, 10:00:00", "31/12/2003, 23:59:59", etc. La classe `DataOra` deve soddisfare le seguenti specifiche:

- fornisce un costruttore senza argomenti `DataOra()` che costruisce Capodanno del 2004 (cioè "1/1/2004, 00:00:00"), ed un costruttore a tre argomenti `DataOra(int g, int m, int a)` che costruisce l'oggetto "`g`/`m`/`a`, 12:00:00"
- fornisce un metodo `boolean agosto1215()` che ritorna `true` se e soltanto se l'oggetto di invocazione rappresenta un qualsiasi giorno del mese di agosto nell'orario compreso tra le "12:00:00" e le "15:00:00" (inclusi)

[1] Non si consideri il problema degli anni bisestili.

- ridefinisce il metodo `boolean sera()` in modo che ritorni `true` se e soltanto se l'orario dell'oggetto di invocazione è compreso tra le "19:00:00" e le "23:59:59" (cioè soddisfa il vincolo del metodo `sera` della superclasse `Orario`) ed inoltre il mese dell'oggetto di invocazione è Luglio o Agosto
- fornisce un metodo `boolean minore(DataOra r)` che ritorna `true` se e soltanto se l'oggetto di invocazione rappresenta una data antecedente o uguale alla data rappresentata dall'argomento `r`. Quindi tale condizione non considera gli orari degli oggetti
- fornisce un metodo statico `void stampa(Orario r)` che stampa su `System.out` le informazioni dell'argomento `r`: quindi se il tipo dinamico di `r` è `DataOrario` stampa la data seguita dall'orario, altrimenti stampa solamente l'orario

Infine, si definisca una classe `Ecc` i cui oggetti sono delle eccezioni che rappresentano una situazione di inconsistenza in un orario o in una data. La classe deve poter discriminare tra le due diverse tipologie di inconsistenza. Ad esempio, "25:23:55", "-12:30:12" e "10:25:87" sono orari inconsistenti mentre "2/14/2003", "6/-5/2002", "45/3/2000" sono date inconsistenti. La classe `Eccezione` fornisce un costruttore `Ecc(String s)` tale che `Ecc("orario")` costruisce una eccezione che rileva una inconsistenza di orario, mentre `Ecc("data")` costruisce una eccezione che rileva una inconsistenza di data. Inoltre, `Ecc` fornisce un metodo `String getReason()` che ritorna la stringa `"orario"` se l'eccezione di invocazione rappresenta una inconsistenza di orario mentre ritorna `"data"` per l'inconsistenza di data. Quindi i costruttori delle classi `Orario` e `DataOrario` dovranno sollevare un'opportuna eccezione della classe `Ecc` a seconda della tipologia di incosistenza rilevata.

Esercizio 15

(i) Definire una classe `Chiamata` che rappresenta le informazioni relative ad una chiamata effettuata da un telefono cellulare. Ogni oggetto della classe `Chiamata` contiene le informazioni relative al numero chiamato, alla durata della chiamata e al costo della chiamata. La classe deve fornire i seguenti metodi:

- un costruttore per costruire una chiamata ad un dato numero e di una data durata espressa in secondi;
- un metodo `void printInfo()` che stampa su `System.out` tutte le informazioni sulla chiamata.

(ii) Il costo di una chiamata dipende dai seguenti parametri: durata della chiamata; numero chiamato; tipologia di abbonamento ricaricabile, dove sono possibili due tipologie: `Flat` e `EvenOdd`.

Si chiede di definire una gerarchia di tipi che realizzi le diverse tipologie di abbonamento ricaricabile. La gerarchia ha come radice una classe astratta `AbbonamentoR`, che prevede un campo dati `creditoResiduo` che rappresenta il credito residuo dell'abbonamento in eurocent (sono ammessi i decimali) e i seguenti metodi:

- un metodo astratto `int costo(Chiamata c)` che restituisce il costo della chiamata in eurocent, utilizzando le informazioni del parametro `c`;
- un metodo `double disponibile()` che restituisce il credito residuo dell'abbonamento;
- un metodo `double addebita(Chiamata c)`: addebita il costo della chiamata `c` aggiornando il credito residuo; se il costo della chiamata è superiore al credito residuo dell'abbonamento allora viene sollevata una eccezione di tipo `CreditoEsaurito` (tale classe eccezione deve essere opportunamente definita); in questo caso il credito deve venire aggiornato a `0`. Il metodo ritorna il costo dell'addebito della chiamata `c`.

Le due tipologie di abbonamento `Flat` e `EvenOdd` corrispondono ciascuna ad una sottoclasse concreta di `AbbonamentoR` che implementa il metodo `costo()` a seconda della regola di tariffazione associata all'abbonamento e aggiunge le eventuali informazioni necessarie per l'implementazione corretta di tale metodo `costo()`. Le regole di tariffazione sono le seguenti:

- per la classe `Flat`: tutte le chiamate costano 10c/minuto;
- per la classe `EvenOdd`: le chiamate ai numeri pari costano 15c/minuto, le chiamate ai numeri dispari costano 5c/minuto.

Esercizio 16

Si consideri la seguente interfaccia `Iteratore` i cui oggetti rappresentano degli iteratori su oggetti di qualche classe collezione di oggetti.

```
interface Iteratore {
  boolean end();
  Object current();
  void next();
}
```

I metodi di `Iteratore` prevedono i seguenti contratti:

- `it.end();` ritorna `true` se e solo se `it` è l'iteratore "past-the-end".
- `it.current();` ritorna l'oggetto "puntato" dall'iteratore `it`.
- `it.next();` sposta l'iteratore `it` all'elemento successivo.

Si consideri inoltre la seguente definizione parziale della classe `Sequenza` i cui oggetti rappresentano una sequenza di oggetti implementata come un array di `Object`.

```
class Sequenza {
  private Object[] a; // array di Object
  private int next = 0; // indice dell'array per il prossimo inserimento
  public Sequenza(int dim) { a = new Object[dim]; }
  public void add(Object x) { if(next < a.length) {a[next] = x; next++;} }
  ...
}
```

A. Implementare l'interfaccia `Iteratore` come classe interna a `Sequenza`. Oltre questa classe interna **non** è permesso aggiungere ulteriori membri (campi dati, metodi o classi) alla classe `Sequenza`.

B. Definire una classe `SequenzaNulla` di eccezioni non controllate.

C. Definire un metodo statico `int numero(Sequenza)` (da inserire in qualche classe esterna a `Sequenza`) che soddisfa la seguente specifica: ogni invocazione `numero(s)` deve avere il seguente comportamento:

- se `s` è un reference nullo allora lancia una eccezione di tipo `SequenzaNulla`;
- se `s` è un reference non nullo allora ritorna il numero di riferimenti non nulli a stringhe (cioè di tipo `String`) contenuti nella sequenza `s`.

Ad esempio, il seguente frammento di codice deve compilare correttamente ed eseguire senza eccezioni provocando la stampa riportata.

```
Sequenza s = new Sequenza(5); String str = new String("pippo");
s.add(new Integer(0)); s.add(str); s.add(new Integer(0)); s.add(str);
System.out.println(Esercizio.numero(s)); // stampa: 2
```