Esercizio 1

Considerate la seguente gerarchia di classi:

```
interface M { M m(); }
interface N { void n(); }

class A implements M {
  public M m() { return this; }
}

class B extends A {
  public void k() { }
}

class C extends A implements N {
  public void n() {}
  public void p() {}
}
```

Quale è il risultato della compilazione e della (eventuale, nel caso la compilazione non dia errori) esecuzione dei seguenti frammenti?

1. `N x = new C(); M y = (B)x;`

2. `M x = new B(); B y = x.m();`

3. `M x = new B(); ((B)x.m()).k();`

Soluzione

1. `N x = new C();` – `x` ha tipo statico `N`, tipo dinamico `C`, sottotipo di `N`. Assegnamento valido.
   `M y = (B)x;` assegnazione valida: `B` è sottotipo di `M`.
   `(B)x` cast invalido: `x` ha tipo dinamico `C` che non è sottotipo di `B`.
   Risultato: eccezione a runtime.

2. `M x = new B();` – `x` ha tipo statico `M`, tipo dinamico `B`, sottotipo di `M`. Assegnamento valido.
   `x.m()` chiama `A.m()`. il risulatato ha tipo statico `M`, tipo dinamico `B` (viene restituito lo stsso oggetto riferito da `x`.
   `B y = x.m();` assegnamento invalido: `x.m()` ha tipo statico `M` che non è sottotipo di `B`.
   Risultato: errore di compilazione.

3. `M x = new B();` – `x` ha tipo statico `M`, tipo dinamico `B`, sottotipo di `M`. Assegnamento valido.
   `x.m()` chiama `A.m()`. il risulatato ha tipo statico `M`, tipo dinamico `B` (viene restituito lo stsso oggetto riferito da `x`.
   `(B)x.m()` cast valido: il tipo dinamico `B` è sottotipo di `B`.
   `((B)x.m()).k();` operazione valida: `k()` è un metodo di `B`.

Esercizio 2

Considerate le seguenti classi che descrivono alcuni prodotti in vendita in un negozio online.

```
class Book
{
  private int ISBN;
  private double price;
  ...
  public double getPrice(){ return price; }
  public Preview showPreview() {...}
}

class Toy
{
  private int age;
  private double price;
  ...
  public double getPrice(){ return price; }
}
```

Definite un nuovo tipo `ItemForSale` che descriva in modo uniforme i diversi prodotti in vendita. Definite una classe `Catalog` che contiene una collezione di item ed è in grado di restituire diversi `Iterator<ItemForSale>` che iterino a) su tutti i prodotti; b) solo sui libri; c) solo sui giocattoli. La classe dovrà anche avere un metodo `public double total()` che restituisce il prezzo totale degli item in catalogo.

Soluzione

```
abstract class ItemForSale
{
  private double price;

  protected ItemForSale(double p) { price=p; }

  public double getPrice() { return price; }
}

class Book extends ItemForSale
{
  private int ISBN;
  private Preview preview;

  public Book(int ISBN, double price, Preview preview)
  {
    super(price);
    this.ISBN=ISBN;
    this.preview=preview;
  }

  public Preview showPreview() { return preview; }
}

class Toy extends ItemForSale
{
  private int age;

  public Toy(int age, double price)
  {
    super(price);
    this.age=age;
  }
}
```

```
class Catalog
{
  private ArrayList<ItemForSale> GeneralCatalog = new ArrayList<ItemForSale>();
  private ArrayList<ItemForSale> ToyCatalog = new ArrayList<ItemForSale>();
  private ArrayList<ItemForSale> BookCatalog = new ArrayList<ItemForSale>();

  public void add(ItemForSale item)
  {
    GeneralCatalog.add(item);
    if (item instanceof Toy) ToyCatalog.add(item);
    if (item instanceof Book) BookCatalog.add(item);
  }

  public boolean remove(ItemForSale item)
  {
    if (item instanceof Book) BookCatalog.remove(item);
    if (item instanceof Toy) ToyCatalog.remove(item);
    return GeneralCatalog.remove(item);
  }

  Iterator<ItemForSale> iterator() { return GeneralCatalog.iterator(); }
  Iterator<ItemForSale> ToyIterator() { return ToyCatalog.iterator(); }
  Iterator<ItemForSale> BookIterator() { return BookCatalog.iterator(); }
}
```

Esercizio 3

State scrivendo il codice per un videogioco che deve gestire delle entità astratte da un opportuno tipo (classe o interfaccia) `Entity`. Ogni tipo di entità può avere varie proprietà:

- `Visible` – l'entità è *visibile* e implementa un metodo `draw()`;
- `Solid` – l'entità è *solida* e può verificare la collisione con altre entità attraverso il metodo `collide(Entity entities[])`;
- `Movable` – l'entità è *mobile* e implementa un metodo `update()` che ne altera la posizione in funzione della fisica del gioco.

Scrivete il codice Java per implementare le seguenti entità:

- `Player` – sia *solida*, che *mobile*, che *visibile*;
- `Cloud` – sia *visibile* che *mobile*;
- `Building` – sia *solida* che *visibile*;
- `Trap` – che è solo *solida*.

con implementazioni di default per i metodi `draw`, `collide` e `update`. Motivate le scelte nella strutturazione delle classi ed interfacce utilizzate.
Scrivete inoltre una classe `EntityArray` che gestisca un array di `Entity` a dimensioni fisse specificate in costruzione e che abbia un metodo `makeSolidIterator()` che restituisca un iteratore di tutte le entità *solide* nell'array.

Soluzione

Valutare pro e contro della soluzione proposta (robustezza, scalabilità, localizzazione di eventuali modifiche, duplicazione di codice e/o logica), nonché la capacità di rispondere ai requisiti (incluso fornire comportamenti di default).