Esercizio 8

Scrivere una classe classe `Modular` che rappresenti un elemento dell'anello modulare $\mathbb{Z}/n\mathbb{Z}$ (somma e prodotti modulo n). La classe deve avere:

- una funzione statica `void init(int n)` che setti il modulo per tutte le istanze e da chiamare almeno una volta prima di tutte le altre operazioni;
- un costruttore che prende un `int` come input;
- due metodi statici `plus` e `times` che implementano addizione e moltiplicazione tra due `Modular` passati come argomento;
- due metodi membro `add` e `multiply` che aggiungono e moltiplicano l'oggetto per il `Modular` argomento;
- la funzione `equals` per verificare che due `Modular` rappresentino lo stesso numero.

Soluzione

```
public class Modular
{
  private static int modulus;
  public static void init(int n) { modulus=n; }
  public static int getModulus() { return modulus; }

  private int value;

  public Modular(int n) { value=n%modulus; }

  public Modular add(Modular x)
  {
    value += x.value;
    value %= modulus;
    return this;
  }

  public Modular multiply(Modular x)
  {
    value *= x.value;
    value %= modulus;
    return this;
  }

  public static Modular plus(Modular x, Modular y)
  {
    Modular result = new Modular(x.value);
    return result.add(y);
  }

  public static Modular times(Modular x, Modular y)
  {
    Modular result = new Modular(x.value);
    return result.multiply(y);
  }

  public boolean equals(Modular x) { return value%modulus == x.value%modulus; }
  public boolean equals(Object o)
  {
    if (o instanceof Modular) return equals((Modular)o);
    else return false;
  }
}
```

Esercizio 9

Scrivere una classe `PostOffice` che rappresenti un ufficio postale. Questa deve implementare il pattern *singleton* (una classe con una istanza unica riferita da un membro `public static final` della classe). La classe `PostOffice` deve mantenere una associazione tra *ricevente* e la sua *mailbox*, dove il ricevente è rappresentato da un `id` intero e la mailbox è una lista di coppie id-mittente, stringa. Deve inoltre implementare i seguenti metodi:

- `send( int senderId, int receiverId, String text)`: invia il messaggio `(senderId, text)` alla mailbox di `receiverId`;
- `receive(int receiverId)`: riceve la mailbox di `receiverId`.

Scrivere anche una classe `Agent` che rappresenti degli agenti che si mandino messaggi tra di loro. Questi devono essere caratterizzati da un Id univoco assegnato all'atto della costruzione dell'agente ed aver i metodi `send` e `receive` per mandare un messaggio ad un altro agente e ricevere i propri messaggi.

Soluzione

```
public class PostOffice
{
  public class Message
  {
    public int senderId;
    public String text;
    public Message(int sId, String txt) { senderId=sId; text=txt; }
  }

  // messages è una mappa che associa l'id del ricevente ad una lista di messaggi destinati a lui
  private Map< Integer, ArrayList<Message> > messages = new HashMap< Integer, ArrayList<Message> >();

  // costrutore privato: non possono essere costruite altre istanze
  private PostOffice() {}

  // istanza unica dell'ggetto PostOffice
  public static final PostOffice instance = new PostOffice();

  // send mette il messaggio nella mailbox di receiverId
  void send( int senderId, int receiverId, String text)
  {
    ArrayList<Message> mailbox = messages.get(receiverId);
    if (mailbox==null)
      {
      mailbox = new ArrayList<Message>();
      messages.put(receiverId,mailbox);
      }
    mailbox.add( new Message(senderId,text) );
  }

  // receive restitisce la mailbox (lista dei messaggi) del receiverId
  ArrayList<Message> receive(int receiverId) { return messages.get(receiverId); }
}

public class Agent
{
  private static int nextId=0;
  private int Id;
  public Agent() { Id = nextId++; }

  int getId() { return Id; }

  void send(int receiverId, String text) { PostOffice.instance.send(Id, receiverId, text); }
    ArrayList<PostOffice.Message> receive() {  return PostOffice.instance.receive( Id); }
 }
```

Esercizio 10

Sia data la seguente interfaccia che astrae il comportamento di una funzione di una variabile

```
interface Function
{
  // calcola il valore della funzione in x, i.e.,  f(x)
  double value(double x);
}
```

Implementate le seguenti classi che implementano `Function` e che costituiscono elementi per la costituzione di funzioni complesse:

- `Constant`: questa classe mantinene un `double` e rappresenta una funzione costante: `value` su questa calsse deve restituire la costante immagazzinata;
- `Variable`: questa classe corrisponde alla x: `value` restituisce il suo input;
- `Plus`: questa classe mantiene due `Function` e ne rappresenta la somma;
- `Times`: questa classe mantiene due `Function` e ne rappresenta il prodotto.

Inoltre, scrivete del codice (classe, metodo, quello che ritenete più opportuno) che data una `Function` restituisca un `Function` che ne rappresenti la derivata.
Nota: È possiblie aggiungere metodi all'interfaccia `Function`.

Soluzione

In questa soluzione viene usato il pattern *Visitor* anche se era possibile semplicemente aggiungere un metodo `Function differenate()` all'interfaccia `Function`.

```
interface Function
{
  double value(double x);

  void accept(FunctionVisitor v);
}

class Constant implements Function
{
  private double val;
  public Constant(double c) { val=c; }

  public double value(double x) { return val; }
  public void accept(FunctionVisitor v) { v.visit(this); }
}

class Variable implements Function
{
  public double value(double x) { return x; }
  public void accept(FunctionVisitor v) { v.visit(this); }
}

class Plus implements Function
{
  private Function left;
  private Function right;
  public Plus(Function l, Function r) { left=l; right=r; }

  public double value(double x) { return left.value(x)+right.value(x); }
  public void accept(FunctionVisitor v) { v.visit(this); }

  public Function getLeft() { return left;}
  public Function getRight() { return right;}
}
```

```
class Times implements Function
{
  private Function left;
  private Function right;
  public Times(Function l, Function r) { left=l; right=r; }

  public double value(double x) { return left.value(x)*right.value(x); }
  public void accept(FunctionVisitor v) { v.visit(this); }

  public Function getLeft() { return left;}
  public Function getRight() { return right;}
}

interface FunctionVisitor
{
  void visit(Constant f);
  void visit(Variable f);
  void visit(Plus f);
  void visit(Times f);
}

class Differentiate implements FunctionVisitor
{
  private Function function=null;

  public Function getFunction() { return function; }

  public void visit(Constant f) { function = new Constant(0.0); }

  public void visit(Variable f) { function = new Constant(1.0); }

  public void visit(Plus f)
  {
    // derivata della sotto-espressione sinistra
    Differentiate lDiff = new Differentiate();
    f.getLeft().accept(lDiff);

    // derivata della sotto-espressione destra
    Differentiate rDiff = new Differentiate();
    f.getRight().accept(rDiff);

    // derivata della somma: (f+g)' = f' + g'
    function = new Plus(lDiff.getFunction(),rDiff.getFunction());
  }

  public void visit(Times f)
  {
    // derivata della sotto-espressione sinistra
    Differentiate lDiff = new Differentiate();
    f.getLeft().accept(lDiff);

    // derivata della sotto-espressione destra
    Differentiate rDiff = new Differentiate();
    f.getRight().accept(rDiff);

    // derivata del prodotto: (f*g)' = f'*g + f*g'
    function = new Plus(
  new Times(lDiff.getFunction(),f.getRight()),
  new Times(f.getLeft(),rDiff.getFunction()) );
  }
```

```
}
```

Con questa infrastruttura, il calcolo della dervata di una funzione avverrà nel seguente modo:

```
public static Function derivative(Fuction f)
{
  Differentiate df=new Differentiate();
  f.accept(df);
  return df.getFunction();
}
```

In alternativa, rinunciando alla possibilità di agire in altro modo sulle `Function`, possiamo fare a meno del meccanismo di double dispatch e aggiungere il metodo `differentiate()` all'interfaccia `Function`

```
interface Function
{
  double value(double x);

  Function differentiate();
}

class Constant implements Function
{
  private double val;
  public Constant(double c) { val=c; }

  public double value(double x) { return val; }
  public Function differentiate() { return new Constant(0.0); }
}

class Variable implements Function
{
  public double value(double x) { return x; }
  public Function differentiate() { return new Constant(1.0); }
}

class Plus implements Function
{
  private Function left;
  private Function right;
  public Plus(Function l, Function r) { left=l; right=r; }

  public double value(double x) { return left.value(x)+right.value(x); }
  public Function differentiate() { return new Plus(left.differentiate(), right.differentiate()); }
}

class Times implements Function
{
  private Function left;
  private Function right;
  public Times(Function l, Function r) { left=l; right=r; }

  public double value(double x) { return left.value(x)*right.value(x); }
  public Function differentiate()
  {
    return  new Plus(
        new Times(left.differentiate(), right),
        new Times( left, right.differentiate())
        );
  }
}
```