Abstract Data Types

Numeri complessi

Definire un ADT **Complex** in grado di rappresentare un numero complesso.

La classe deve essere immutabile.

Cosa serve

```
/*@ pure @*/ public class Complex{
    //Creators
    //Observers
    //Producers
    //Mutators
}
```

Creators/Constructors

```
public Complex();

public Complex(double real,
               double imaginary);
```

Observers

```
public double re();

public double im();

public double rho();

public double phi();
```

Producers

```
public Complex(Complex c);

public Complex sum(Complex c);

public Complex sub(Complex c);

public Complex mul(Complex c);

public Complex div(Complex c);

public Complex conjugate();
```

Specifiche

Fornire le specifiche JML per
l'ADT Complex

Esprimere lo stato

- Scegliamo un insieme di osservatori puri opportuno per descrivere lo stato dell'ADT nelle specifiche degli altri metodi.
  - Per esempio, re() e im(), ma anche rho() e phi() sono scelte possibili in questo caso.
  - Privilegiare sempre lo stesso insieme di osservatori, a meno che la specifica non sia più semplice con un altro insieme equivalente

Creators - specifiche

```
//@ ensures this.re() == 0 && this.im() == 0;
public Complex();

/*@ ensures !Double.isNaN(re) && !Double.isNaN(im) &&
  @ !Double.isInfinite(re) && !Double.isInfinite(im) &&
  @ this.re() == re && this.im() == im;
  @ signals (IllegalArgumentException e) Double.isNaN(re) ||
  @ Double.isNaN(im) ||
  @ Double.isInfinite(re) || Double.isInfinite(im);
  @*/
public Complex(double re, double im);
```

In questo caso, abbiamo scelto re() e im() per esprimere le postcondizioni dei creators. Attenzione a ricordarsi sempre che double potrebbe avere dei valori non validi per rappresentare un complesso

Observers - specifiche

```
//@ ensures (*Parte reale di this*);
public double re();

//@ ensures (*Parte immaginaria di this*);
public double im();
```

re() e im() non hanno sostanzialmente specifica: i loro specifici valori di ritorno sono determinati dalle postcondizioni degli altri metodi.

Observers - specifiche

```
//@ ensures \result ==
        Math.sqrt(this.re()*this.re() +
                  this.im()*this.im());
public double rho();

/*@ ensures
  @ re() > 0 => \result == Math.atan(im()/re());
  @ re() < 0 && im() >= 0 =>
  @   \result == Math.atan(im()/re()) + Math.PI;
  @ re() < 0 && im() < 0 =>
  @   \result == Math.atan(im()/re()) - Math.PI;
  @ re() = 0 && im() >= 0 => \result == Math.PI/2;
  @ re() = 0 && im() < 0 => \result == - Math.PI/2;
  @*/
public double phi();
```

rho() e phi() vengono definiti in funzione di re() e im()

Producers - specifiche

```
//@ ensures this.re() == c.re() &&
            this.im() == c.im();
public Complex(Complex c);
```

```
/*@ ensures
  @     \result.re() =  this.re() &&
  @     \result.im() = - this.im();
*/
public Complex conjugate();
```

La specifica del produttore conjugate() e' piu' semplice se espressa in funzione di re() e im()

Producers - specifiche

```
//@ requires c != null;
/*@ ensures
  @  (\result.re() == c.re() + this.re()) &&
  @  (\result.im() == c.im() + this.im());
*/
public Complex sum(Complex c);

//@ requires c != null;
/*@ ensures
  @  (\result.re() == c.re() - this.re()) &&
  @  (\result.im() == c.im() - this.im());
*/
public Complex sub(Complex c);
```

La specifica dei produttori sum() e sub() e' piu' semplice se espressa in funzione di re() e im()
Se espressa in funzione di rho() e phi() diventa piu' complessa

Producers - specifiche

```
//@ requires c != null;
/*@ ensures
  @  (\result.rho() == c.rho()*this.rho()) &&
  @  (\result.phi() == c.phi() + this.phi());
*/
public Complex mul(Complex c);

/*@ requires c != null && c.rho() != 0;
  @ ensures
  @  (\result.rho() == this.rho()/c.rho()) &&
  @  (\result.phi() == this.phi() - c.phi());
*/
public Complex div(Complex c);
```

La specifica dei produttori, in questo caso, e' piu' semplice se espressa in funzione di rho() e phi() rispetto a re() e im()

Rappresentazione

Deve soddisfare la specifica appena definita

Abbiamo due possibilità:
(re, im)
(rho, phi)

Rappresentazione e RI

```
private double re, im;
```

```
/*@ private invariant !Double.isNan(re) &&
  @ !Double.isNaN(im) && !Double.isInfinity(re) &&
  @ !Double.isInfinity(im);
  @*/
```

Rappresentazione e RI

```
private double rho, phi;
```

```
/*@ private invariant !Double.isNan(rho) &&
  @ !Double.isNaN(phi) && !Double.isInfinity(rho) &&
  @ !Double.isInfinity(phi) && rho >= 0;
  @*/
```

Altre funzioni utili

Implementazione dell'Abstraction Function (AF)

```
public String toString();
```

equals per ADT immutabili

```
public boolean equals(Object o);
```

toString()

```
public String toString(){
   return re + " + " + im + "j";
}
```

```
public String toString(){
   return rho + " * e^ ( " + phi + "j )";
}
```

equals()

```
@Override
public boolean equals(Object o){
   if(!(o instanceOf Complex))
      return false;
   else
      return equals((Complex) o);
}

public boolean equals(Complex c){
   return re == c.re && im == c.im;
}
```

Matrici

Definire un ADT **Matrix** in grado di rappresentare una matrice di double.

Attenzione: se devo creare una matrice di double senza ulteriori vincoli, allora qualunque possibile valore di double e' lecito (anche NaN e infinito), contrariamente al caso di Complex.

La classe deve essere mutabile; tuttavia, l'unico metodo per cambiarne lo stato e' quello di modificare un singolo elemento della matrice.

Cosa serve

```
public class Matrix{
    //Creators
    //Observers
    //Producers
    //Mutators
}
```

Creators

```
public Matrix(int rows, int cols);

public Matrix(double [][] data);
```

Observers

```
public /*@ pure @*/ double element(int r, int c);

public /*@ pure @*/ double[] column(int c);

public /*@ pure @*/ double[] row(int r);

public /*@ pure @*/ double[][] data();
  //Non esporre la rappresentazione!
public /*@ pure @*/ int rows();

public /*@ pure @*/ int cols();

public /*@ pure @*/ int rank();
```

Producers

```
public /*@ pure @*/ Matrix sum(Matrix m);

public /*@ pure @*/ Matrix sub(Matrix m);

public /*@ pure @*/ Matrix mul(Matrix m);

public /*@ pure @*/ Matrix transpose();
```

Mutators

```
public void setElement(double value,
                       int r, int c);
```

Specifiche

Fornire le specifiche JML per
l'ADT Matrix

Specifiche

Quali pure observers scegliamo
per descrivere lo stato di un
oggetto Matrix?

```
public double element(int r, int c);
public int rows();
public int cols();
```

Creators - specifiche

```
/*@ ensures rows > 0 && cols > 0 &&
            rows() == rows &&
            cols() == cols &&
            (\forall int i; 1<=i<=rows();
              (\forall int j; 1<=j<cols();
                element(i,j) == 0));
    signals (MatrixException e) rows <=  0 || cols <= 0
@*/
public Matrix(int rows, int cols) throws MatrixException;
```

Creators - specifiche

```
/*@ ensures
  data != null && data.length!= 0 &&
  data[0] != null && data[0].length != 0 &&
  (\forall int i; 1<= i <= data.length;
    data[i] != null && data[i].length==data[0].length) &&
  rows() == data.length && cols() == data[0].length &&
  (\forall int i; 1<=i<=rows();
   (\forall int j; 1<=j<=cols();
     element(i,j) == data[i-1][j-1]))

    signals (MatrixException e)
      data == null || data.length == 0 ||
      (\exists int i; 0<=i<=data.length;
       (\exists int j; 0<=i<=data.length;
        data[i].length != data[j].length));
@*/
public Matrix(double[][] data) throws MatrixException;
```

Observers - specifiche

```
/*@ ensures
  @  0<i<=rows() && 0<j<=cols() &&
  @    \result == data[i][j];
  @ signals (MatrixException e)
  @   r<=0 || r>rows() || c<=0 || c>cols();
  @*/
public double /*@ pure */ element(int i, int j) throws
MatrixException;

/*@ ensures
  @  0<c<=rows() &&
  @    \result.length == rows() &&
  @    (\forall int i; 0<=i<\result.length;
  @      \result[i]==element(i+1,c));
  @  signals (MatrixException e) c<=0 || c>cols();
  @*/
public double[] /* @pure */ column(int c) throws
MatrixException;
```

Mutators - specifiche

```
// Prima versione scorretta:
/*@ ensures
  @  0<r && r<=rows() && 0<c && c<=cols() &&
  @  element(r,c) == value &&
  @  (\forall int i; 1<=i<=rows();
  @   (\forall int j; 1<=j<=cols();
  @    (i!=r && j!=c) =>
  @         element(i,j) == \old(element(i,j))))
  @  signals (MatrixException e)
  @   r<=0 || r>rows() || c<=0 || c>cols();
  @*/
public void setElement(double value, int r, int c)
       throws MatrixException;
```

Mutators - specifiche

```
// Prima versione scorretta:
/*@ ensures
  @  0<r && r<=rows() && 0<c && c<=cols() &&
  @  element(r,c) == value &&
  @  (\forall int i; 1<=i<=rows();
  @   (\forall int j; 1<=j<=cols();
  @    (i!=r && j!=c) =>
  @         element(i,j) == \old(element(i,j))))
  @  signals (MatrixException e)
  @   r<=0 || r>rows() || c<=0 || c>cols();
  @*/
public void setElement(double value, int r, int c)
       throws MatrixException;
```

Il metodo non predica nulla sul valore di rows() e cols() dopo l'invocazione del metodo!

Mutators - specifiche

```
// Prima versione scorretta:
/*@ ensures
  @  0<r && r<=rows() && 0<c && c<=cols() &&
  @  element(r,c) == value &&
  @  (\forall int i; 1<=i<=rows();
  @   (\forall int j; 1<=j<=cols();
  @    (i!=r && j!=c) =>
  @         element(i,j) == \old(element(i,j))))
  @  signals (MatrixException e)
  @   r<=0 || r>rows() || c<=0 || c>cols();
  @*/
public void setElement(double value, int r, int c)
       throws MatrixException;
```

In particolare, il metodo potrebbe rimpicciolire la matrice (cosa impedisce, invece, che possa essere piu' grande?)

Mutators - specifiche

```
// Seconda versione corretta:
/*@ ensures
  @  0<r && r<=rows() && 0<c && c<=cols() &&
  @  element(r,c) == value &&
  @  rows() == \old(rows) && cols() == \old(cols) &&
  @  (\forall int i; 1<=i<=rows();
  @   (\forall int j; 1<=j<=cols();
  @    (i!=r && j!=c) =>
  @         element(i,j) == \old(element(i,j))))
  @  signals (MatrixException e)
  @   r<=0 || r>rows() || c<=0 || c>cols();
  @*/
public void setElement(double value, int r, int c)
       throws MatrixException;
```