CSC 207 **Assignment 3: Fields** **Fall 2009**

Due: Tuesday 17 February 2009

Topics: Parameterized types, interfaces, inheritance, generics, collections

Collaboration: This homework assignment is to be completed individually.

Submission: Follow the instructions for submitting programs via P-Web and handing in a printed copy. Be sure to generate test cases and a statement justifying their sufficiency.

In mathematics, a field is a type of algebraic structure that supports the common operations of addition, subtraction, multiplication, and division. In this assignment, you will create a generic field interface that supports these operations, an abstract implementation, and two fully concrete implementations: rational and complex numbers.

Field Interface

Define a typed interface `Field<AnyType>` that supports the `add`, `subtract`, `multiply`, and `divide` operations. These should all return *new* objects of the appropriate type (rather than being mutators).

In addition, add two methods to the interface for the `additiveIdentity` and the `multiplicativeIdentity`. By definition,

```
member.add( additiveIdentity() ).equals( member )
```

should always be `true`, and

```
member.multiply( multiplicativeIdentity() ).equals( member )
```

shoud also be true.

By virtue of the fact that there must be an additive identity and that subtraction is defined, every field can also support a `negate()` operation. Thus, the following statement should also be `true` for any field

```
member.add ( member.negate() ).equals (additiveIdentity ())
```

AbstractField Class

Define a class `AbstractField<AnyType>` that implements the field interface. Given all the methods above, implement as many of the required methods as possible. (Note that these implementations may rely on abstract methods that have not themselves been implemented.)

Rational Class

Create a (concrete) subclass of `AbstractField` called `Rational`. What should the parameterized type be for the extends clause? If you're confused, you might refer to the `Comparable<T>` interface and also see the declaration of the `Integer` class.

Implement all the necessary methods. You may either adapt your `Rational` class from the previous lab using `BigIntegers`, or you may use `long` types internally. You may also choose any representational invariants you wish.

Complex Class

Create another (concrete) subclass of `AbstractField` called `Complex`. The parameterized type will be analagous to `Rational`.

Complex numbers consist of a real and imaginary part, and are commonly written as $a+bi$ where $a, b \in \mathbb{R}$ are real numbers (approximated in Java as `double`) and $i \equiv \sqrt{-1}$ is the imaginary root. Their operations may be defined as follows:

Addition $(a + bi) + (c + di) = (a + b) + (c + d)\,i$

Subtraction $(a + bi) - (c + di) = (a - b) + (c - d)\,i$

Multiplication $(a + bi) * (c + di) = (ac - bd) + (bc + ad)\,i$

Division $\frac{(a+bi)}{(c+di)} = \left(\frac{ac+bd}{c^2+d^2}\right) + \left(\frac{bc-ad}{c^2+d^2}\right) i$

Implement all the necessary methods.

Field Utilities

Add two static methods to your `AbstractField` class: one (called `sum`) that will compute the `sum` of all the elements in an `ArrayList` of any objects that implement the `Field` interface, and another (called `product`) that will compute the `product` of the elements in such an `ArrayList`. You should use Java's enhanced `for` loop.

In creating your function, it may also help to be able to answer the question: what is the product of an empty `ArrayList`?

Note that you will need to use type bounds. That is, every item in the `ArrayList` must be the same *kind* of `Field` object, otherwise you probably would not be able to add them together.

Putting It Together

Create a separate driver class with methods (called by a `main`) that tests the functionality of all of your classes and methods. Be sure to add `toString` methods to your concrete classes that give reasonable textual representations for the field elements.