Richard Lindner

teaching Java some algebra

a summary

April 6, 2004

MSM3G0 - Project with Richard Kaye

I warrant that the content of this dissertation is the direct result of my own work and that any use made in it of published or unpublished material is fully and correctly referenced.

Signature: ..... Date: .....

Contents

1

Introduction

1.1 Goals

Welcome to the Jalgebra project. [Jalgebra] is a java package that implements some basic algebraic structures. Should you not have this java package now be sure to check the reference section at the end. It will tell you exactly where to get [Jalgebra].

In this accompanying write-up we want to explore how this package was created and what problems occurred at the time. This will help us to understand how much a computer can 'comprehend' mathematics.

1.1.1 Package Contents

We will see how it is possible to get the integers $\mathbb{Z}$ working in Java. From there we go on to construct the fields of prime order as the modulo structure $\mathbb{Z}/p\mathbb{Z}$.

Then we go even further and implement a polynomial ring over an arbitrary ring. It will be proven and implemented that this polynomial ring is euclidean if and only if it is constructed over a field. Then we plug in the fields we have constructed already into the polynomial ring implementation to get euclidean rings.

At this point we try to go for the primepower fields. All we need is an irreducible polynomial over $\mathbb{F}_p$ of degree q and a reliable check that this thing really is irreducible in order to get a working implementation of $\mathbb{F}_{p^q}$. This then will complete the implementation of all finite fields.

There are some interesting design considerations in putting together such a package. These will be discussed in detail within this report. For example

there will be an argument about which representatives we should choose in both the prime fields and primepower fields. It turns out, that we can get a good representation system on the primepower fields that is backwards compatible with the one we will take for the prime fields.

This can be done despite the fact that the elements of the primepower field are internally polynomials. In fact the idea we will use for the representatives, employs a function which is almost but not quite a homomorphism and this will be shown as well. We will also see a good reason, that there isn't a homomorphism between these structures, as it would map the primes directly on irreducible polynomials of all kinds.

Another interesting design point that will concern us is how to calculate the multiplicative inverses in these fields fairly fast. Two methods will be very thoroughly introduced for this, namely Euclids Algorithm and Repeated Squaring. We will show that Euclids method, while being slightly slower in special circumstances, has the upper hand most of the time.

Both these methods can be used in [Jalgebra]. Since they are both included in the package as algorithms on abstract rings, interchanging them takes only one line of code as we will see. Also the two methods will give the same results but work in a very different way, as they use very different mathematical approaches to inverses in finite fields. These approaches as well as their implementation will be explained in detail.

1.1.2 Following a System

We will not only focus on the implementation of finite fields here, but we will keep a broader view and talk in detail about the system in which these mathematical ideas are made understandable to the java compiler. Because by understanding this system more fully we will see that it is possible to add further mathematical ideas and structures to java in a way that would make them 'compatible' with the ones that are already there.

This is started by introducing the Three Way System. This general guideline, which [Jalgebra] follows, will be explained in detail and we will see that it helps to stay organized and compatible with future work. We will also discuss a way to deal with Java's shortcomings when it comes to multiple inheritance.

Later some regulations will be added on top of the Three Way System to make our implementations faster and more efficient. These will just be suggestions to speed up runtime. Each suggestion will come with an explanation on how it will help the programmer.

1.1.3 Prior Knowledge

It is suggested that the reader of this project has a minor background in programming pure java as well as mathematics. Very basic computer science, like what a java-'class' or the 'main'-method is will not be explained. The same goes for mathematics. It should be clear how basic algebraic structures like the integers or abstract rings work. Everything that is more complicated will be explained thoroughly though. So don't be afraid to be left behind and enjoy reading.

1.2 Acknowledgments

These are some acknowledgments for the help of certain people and products. Had it not been for some of these, this project would not have gotten off the ground. Thanks.

- A lot of motivation for *me* and the code written in **jalgebra** came from my tutor, *Dr. Richard Kaye* and his own java math-package **kaye.maths**. Should he decide to put it on the net it will be available at his homepage
  `http://web.mat.bham.ac.uk/R.W.Kaye/`

- A lot of mathematical definitions that are used here came from the great internet collection center of *Wolfram Research* at
  `http://mathworld.wolfram.com/`

- A TEX AddOn Package was used to draw some of the nicer diagrams. It's called XY-pic: Graphs and Diagrams by *Kristoffer H. Rose* and *Ross Moore*, and their homepage is
  `http://www.brics.dk/~krisrose/Xy-pic.html`

- The Java IDE used to actually write jalgebra is the **JCreator LiteEdition** by *Xinox Software*. It is available for free in the 'lite' version, which is already very powerful, at
  `http://www.jcreator.com/`

- The very similar TEX frontend used to write the TEX code of this project is **TeXnicCenter** by *Sven Wiegand*. A lot of people are still writing on this program to make it even better. Try out their latest efforts at
  `http://www.toolscenter.org/`

2

The Three Way System

This chapter is about a concept that we will refer to as the "Three Way System". It is a convention invented by my mentor on how to implement mathematical objects in Java. But let us find out first why it is so important to follow a concept here.

2.1 Java's Shortcomings

Java may not be the ideal language to use in our pursuit to teach a computer mathematical ideas! The major reason for this is Java's incapability to do multiple inheritance of classes or interface in a suitable way for us. Many things we will want to do concerning mathematics stack upon each other in a multiple fashion. Just imagine the structure of a ring which is by definition an additive abelian group and a multiplicative semigroup on the same set. Here we already see three concepts coming together in one definition, an abelian group, a semigroup and a set. In fact what we really do in our heads when thinking about rings is quite akin to multiple inheritance. So how do we cope with a programming language that cannot do this? The answers is discussed in detail in *Section 3.3: Interface Inhertance Problem*.

It has been widely talked about whether or not multiple inheritance of classes should be added to Java. Jim Waldo writes about the benefits of multiple inheritance in his paper [Wal91]. Arnold and Gosling in their write up [AG96] suggest, that the `interface` concept if used correctly not only achives as much as multiple inheritance, but is also much safer to use. The technique we will be using to emulate mutliple inheritance is called *delegation*. This delegation luckily turns out to be better for modelling mathematics then multiple inheritance of classes would have been in any case. Tempero and Biddle wrote a report [TB98], that is worth reading for further information on delegation.

So why do we even bother using Java at all? The shortest answer to this is: It is the most compatible programming language there is. While Java is not the fastest or most convenient language it really makes up for this by running on virtually any system. It really runs on a nonexistent 'virtual machine' that is itself emulated by whichever system you are using. But Java's advantages don't stop there it is also a "free to use for non-commercial purposes"-language so you and I and those others who want to see our progress don't have to pay for it. It is also quite a modern and internet related language and can easily be incorporated into web-pages and in many other places. So Java is after all worth doing all our work in, because in this way everyone may prosper from it.

It is now time to answer our previous question about how to cope with Java's shortcoming. The solution is simply what will be introduced in the next section: the "Three Way System". It is really a way of conveniently using Java's multiple inheritance possibilities when it comes to interfaces. We will try to understand this concept together with its biggest advantages and disadvantages.

2.2 The System in Detail

Since the [Jalgebra]-package was made following the 3-Way System we might as well understand it by looking at the structure of this package. Inside the main directory the convention begins. There must be three subdirectories, hence a three way choice for all the work. They are called impl, intf, and algo. These are abbreviations for implementation, interface, and algorithm.

Let us discuss these three in further detail. The interface directory naturally holds all the interfaces. It turns out that many mathematical rules can be expressed most directly to a compiler via interfaces. So here is where the rule making happens. Defining mathematical rules (basically axioms) in Java terms is quite straightforward most of the time although little mistakes here will have great effects.

The next directory is about implementations. Here most of the coding happens. Once the structure of everything is defined in the interfaces their concrete implementations are found here. For example it is one thing to say a polynomial ring has a multiplication (axiom/interface) but it's quite a harder thing to implement how this multiplication actually works.

Implementations should usually hold exactly those methods they need to implement all the interfaces they wish to claim. Everything else is in the third and final directory the algorithms. These work on abstract interfaces and are

used together with their implementations. To understand this triple mixture more clearly here is an example:

Example 2.2.1. ***The Power Algorithm***
We want to implement an algorithm that simply takes an element to its power. The least structure we need to do this is a multiplicative semigroup. Together with the `Semigroup` interface *we can define a generic static* algorithm *that is given a semigroup and an element inside this semigroup. It will probably use the '*`op()`*' method (guaranteed by the Semigroup interface) as many times as it's necessary to obtain the power.*

To actually see this algorithm working though we will have to go one step further and create an object that implements *our semigroup interface. Let us say we create the integers or just take the ones that are provided. Then we can call this static algorithm, give it our integer's multiplicative monoid and just watch it work.*

If we take a second to think here we should make our algorithm smart enough to realize that it has been given a monoid and can thus take elements to the power of zero as well. We could either use 'instanceof' or make two algorithms and let java figure out what is given to the algorithm. When this is done we should of course implement one for groups that allows negative powers as well (we return to this idea later in section 4.4).

For more convenience we extending the convention which would be finished at this point. We add yet another directory in the package which is util. This name stands for utility and should contain things that are too Java specific in their work to fit in the other three parts. Here we find so far just an exception class, that throws an error for this package and is used all over the place in implementations and algorithms. We could however also implement a class here that would quickly plot a polynomial in a window. This is clearly not a mathematical challenge but it is a very useful utility. We might also write a window based graphical user interface for the other parts of the package here or things of that nature.

2.3 Positive and Negative Aspects

In this section we take a short critical look at the small concept, which was introduced in this chapter. The Three Way System is not very restricitve, it gives the ideas we want to implement just a little bit bottom structure so that we may start out in an ordered fashion. If we wish to add new code to the [Jalgebra]-package, then thanks to this concept's rules, it is often easy to see what has been done in which area and what still needs doing. Those are the

advantages - a basic order yielding easier group-work.

The disadvantages are even easier to list. It is the one disadvantage that comes with everything that makes long-term or group working more comfortable - the individual has to work, and in our case write more. As soon as we create rules to guide us we limit our personal efficiency for the sake of more structure, which in turn gives others an easy entry to understand what we did (it's the same with object oriented programming in general by the way). So the bottom line is if we decide to follow the Three Way System, as [Jalgebra] does we'll be required to write a bit more code.

3

Inside the Interfaces

We shall see that writing interfaces in Java so that they actually do what we want is not an easy matter. There have been some problems with this and at some point the whole [Jalgebra] code has been rewritten because a different approach was discovered. This however turns out to have new drawbacks, which are less relevant. In this chapter we will mainly see what Javas idea behind interfaces is and when the translations between mathematical axioms and interfaces is not smooth sailing anymore.

3.1 Interfaces Explained

Java uses interfaces so that you can pre-define a minimum of structure an object has to have. Moreover the compiler will only let a class implement a given interface if all functions mentioned there have an implementation in the class. The little clash that we will see is that Java purposely allows an interface to inherit multiple interfaces but classes are not allowed to inherit multiple classes. There are of course ways to deal with this as we will also see.

The payoff for using interfaces is that you can then define methods for any class that meets the requirements of a given interface. In mathematical terms that means we can write for example a `Ring`-algorithm and it will work with all `Ring`-implementing classes. So we want for example `Fields` to implement the `Ring` interface and then we can run `Ring`-algorithms on them.

When we start to look at the interfaces we will at first glance recognize a little trouble with checked methods.

3.2 Checked Methods

When we look at any interface (for example `jalgebra/intf/set/set.java`) then with a big probability there are two methods that appear to be the same. They will look something like "`public void mymethod(Object o, ...)`" and "`public void _mymethod(Object o, ...)`". Did you notice the difference? The little underscore in front of the second function means that this function is the faster unchecked version of `mymethod`. Unchecked here means, that we will not check that mathematical objects given to this method are really in the set we are operating on. In other words, for efficiencies sake we don't check whether we do makes any sense. This requires a great deal of trust to the programmer. These methods should actually only be used by methods inside the jalgebra package.

The other checked method performs the check typically with the `in()` method, that has to be implemented by any set. Checking an implementation of the interface we have observed we will see lot's of similar code. This is because the way these checked functions work is always the same! It's do the standard check throw an error or call the unchecked version depending on the outcome. Something like the code snippet

```
if(in(o)) { _mymethod(o); }
else { throw new MathException("Error!"); }
```

always does the trick! However java will not allow you to generalize these checked functions in any way. A real shame! The generalization is possible though by using a self-programmed preprocessor. The only challenge would be to write the general error message in such a fashion that it becomes clear where the error occurred. The methods name and the name of the file that contains it should suffice for this.

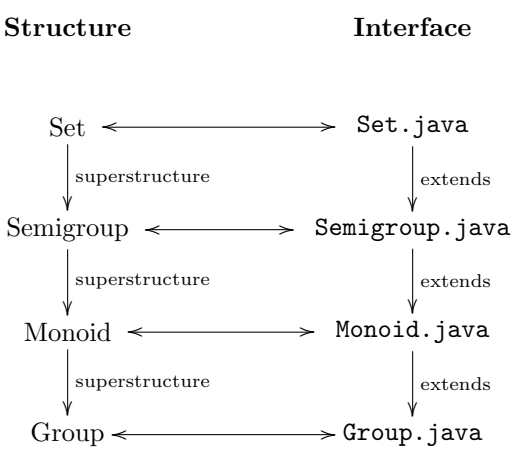
Once we overcome this little technicality we can discuss what became the big trouble.

3.3 Interface Inheritance Problem

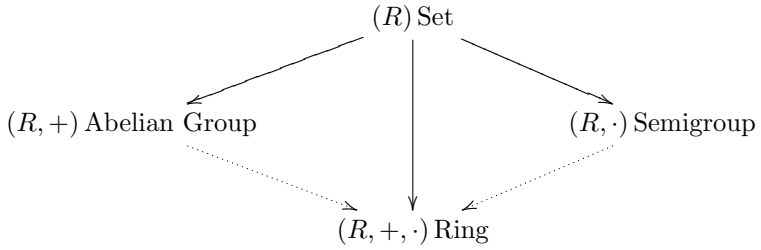
What we cannot see anymore is how the jalgebra package looked for the first couple of months. So here comes a partial description and the reasons it was changed.

It all stared with multiple inheritance of interfaces and that it does not really work in the fashion that it should. The fashion that would make the transfer from axiom to interface almost seamless. Let us take a glance at the

way it should be. Then we can discuss the way it is. In the following diagram you will see the way interface-structures were imagined to begin with.



This far there is no problem in the world. Everything is simple and all interfaces have exactly one heir. But once we write these interfaces we will also want to write one for a Ring. This is where the problems begin. The schematics look like this:



So we would actually want to say the Ring interface guarantees two similar but different sets of properties for two different binary operations, namely plus and times. In a way Java lets you do this using declarations, but not smoothly. It is what we will call the 'Inheritance Solution using Declaration'.

However we also realize at this stage that from `Semigroup` onwards all our interfaces only make sense when you tell them the operation they are supposed to be using on the set. So we might make lots of different (and we can already see it get messy here) interfaces corresponding to each operation. We will call this the 'Inheritance Solution using many Interfaces'.

We will now explore both of these solutions to our problem and see which one works better in what areas. We shall see that none of the two is perfect, however [Jalgebra] is meanwhile completely written in the first style, since it

has some advantages. The other one is easier to explain however so we start with that.

3.3.1 Solution using many Interfaces

For this solution we first create an interface of every operation that we will use. Typically we would want one for addition and multiplication, but there might also be need for composition or any other binary operation really. These interfaces would only contain the operation they are supposed to guarantee. They should extend the set interface because otherwise the operation might not be implementable. Addition would for example look like this

```
package jalgebra.intf.binop;
import java.lang.Object;
import jalgebra.intf.structure.Set;

public interface Addition extends Set{
   public void add(Object x, Object y);
   public void _add(Object x, Object y);
}
```

Then we go on by creating multiple version of the Semigroup interface. `AddSemigroup` would then extend `Addition`, `MulSemigroup` would extend `Multiplication` and so on. Furthermore we write these multiple interface versions for **every** interface that extends `*-Semigroup`. These would be for example `AddAbelianGroup`, `MulMonoid` and things of this kind. We can already see, that these could potentially be a lot.

Also notice, that we start at Semigroup because that is the least structure we wish to consider. So should we want to be even more basic like for example have a non-associative operation, we would have to start at `Groupoid`.

A `Ring` interface would now be easy to do. It just extends `AddAbeliangroup` and `MulMonoid/Semigroup` (depending again on how basic we want to get). However we could strictly speaking only have `Rings` with the operation + and ·! There would be no Ring over a set of functions where composition is used instead of multiplication. This is already the second drawback.

The third one is even worse. It is that we could **not** write any generic `Group`-algorithms. We would need to write the same algorithm for every version of a `Group` that we have. The same goes naturally for `Semigroups` or `AbelianGroups`. Also we could not trick around here with a little casting, but we would literally have to copy and paste the code and then change our "`add()`" entries within the algorithm to "`mul()`" or whichever operation we

need.

We see that this solution, while working fine, involves a lot of 'unnecessary' and repetitive coding, which is exactly what we do not want. Let's see what the alternative does.

3.3.2 Solution using Declaration

In this solution we still want a single generic `Operation` interface, however the operation method required to implement this interface will be something non-committing like "`op()`" and this will suffice for all later tasks.

Writing the interfaces within the `Group`-Hierarchy (`Semigroup`, `Monoid`,...) works very nicely now, as we need just one interface for each structure and that is a minimum of work that we will always be required.

Our `Ring` interface would ideally work on a `Set` and then *delegate* all operation calls. An implementation would look something like this.

```
package jalgebra.intf.structure;
import jalgebra.intf.structure.Set;
import jalgebra.intf.binop.Distributive;

public interface Ring extends Set, Distributive {
   public AbelianGroup add();
   public Monoid mul();
}
```

So to make an Addition on such a Ring implementation you would have the command "`R.add().op(a,b);`", where `R` is the `Ring` and `a,b` are the `Objects`.

Notice that in this setting compared with the other solution we could have a Ring that does composition on the elements, when it's "`mul()`" method is called. In fact any `Monoid` and `Abeliangroup` on the same `Set` could become this `Ring`, which is great. So there goes drawback number two.

Still no Problems yet, well here it comes. We want to implement anything that extends a `Ring` next, so let's say a `Field`. And it should look something like this.

```
package jalgebra.intf.structure;
import jalgebra.intf.structure.Ring;

public interface Field extends Ring {
   public AbelianGroup add();
```

```
    public AbelianGroup mul();
}
```

This however, as nice and logical as it would be, will **not** compile. The reason being, that this interface, in the way it is written, will not extend the `Ring` interface.

We have come to a great drawback within Java. The `Field` multiplication method

```
public AbelianGroup mul();
```

will **not** extend the `Ring` multiplication

```
public Monoid mul();
```

despite the fact that `AbelianGroup` **is** an extension of `Monoid`. This is because Java internally sees these two as different methods, which have the same name. By the way we would encounter the same problem within most OOP-languages, for example C++.

There is no *nice* and satisfactory way to work around this problem. [Jalgebra] does it by going back to the `Ring` interface and letting both `add()` and `mul()` return `Semigroups`. If it is understood by the programmer, that there is really more structure to the `Objects` returned by these methods, then we 'just' have to cast a lot later on.

This has numerous little negative sides. For example the Field, SkewField, DivisionRing, Ring interfaces now all look the same, which is ridiculously against the point of them. The casting that has to be done **very** often now, in just about all areas, can be quite time consuming. It appears in almost every algorithmic loop within [Jalgebra] now and is drawing down overall performance of the package.

However other than that there are no problems with this solution. We are able to write nice generic algorithms for `Groups`, as well as `Rings` or `Fields` and everything works the way it should, with the help of *lots* of casting.

4

Preliminary Regulations

In this chapter we introduce and explain some *regulations* about how to implement Mathematics in Java. As these were developed while writing [Jalgebra] it conforms with all these regulations. A detailed account why these regulations were made is included in the according sections. At the end of the chapter we will discuss a big mistakes that was made while writing on [Jalgebra].

We must realize, that the following regulations are *not* meant to make the life of the programmer any easier. They are supposed to speed up the running of algorithms and generally the usage of [Jalgebra], bear this in mind. They are largely inspired by the [MIPS] assembler programming language. Don't worry if you never heard of this, it is not required to be known. If you are interested check the reference section.

4.1 Implementations of Mathematical Objects are Mutable

So far these include only integers and polynomials, but there could also be matrices or graphs and basically all element-objects of a set you wish to create.

The big advantage of mutable objects is, that once they are created, you can modify them without the need to create a *new* object. This very mutability of objects however will usually cause problems in multi task programs. If two threads want to modify an object at the same time it is never quite clear who gets it first. Also the reading of mutable objects in multi task environments has to be done with care. You need to be sure that every 'good' modifying thread is finished before you start reading and the 'bad' modifiers, the ones that should run after you read the object, haven't started yet. In short it's a mess. This is the reason the original Java objects tend to be the opposite - immutable. So I recommend not to use [Jalgebra] if you plan to do multi task programming. On the other hand if your program is linear or single tasking

then mutable objects will make it much faster. To illustrate this here is an example of the simple add operation used in the additive group of the *Integers*:

Example 4.1.1. *Unchecked Add Operation with Mutable Objects*

```
/** Additive Group Operation on Integers.
 * Input Object x changes to x+y, Object y is unchanged
 */
public void _op(Object _x, Object _y) {
   Integer x = (Integer)_x;
   Integer y = (Integer)_y;

   x.set(x.get()+y.get());
}
```

Unchecked Add Operation with Immutable Objects

```
/** Additive Group Operation on Integers.
 * Input Object x is unchanged, Object y is unchanged
 * Returns Object x+y
 */
public Object _op(Object _x, Object _y) {
   Integer x = (Integer)_x;
   Integer y = (Integer)_y;

   return new Integer(x.get()+y.get());
}
```

We notice that the immutable objects require you to create a new object every time we add two integers. This will of course happen in numerous other places and ultimately kill the speed in almost any algorithm. It also takes up huge amounts of memory and puts a good deal of work to the Java Garbage Collector, which is designed to free up memory from objects that are no longer used. The garbage collector is very slow at this.

Also notice that you do not *need* to change mutable objects. You can use them just like immutable ones. But this does require a little extra coding. Every time you want to change an object you need to create a new one copying the old object and then change this one. Since this technique is useful either way there is a method required by the set interface that does just this. It looks as follows:

Example 4.1.2. *Unchecked Copying of Objects*

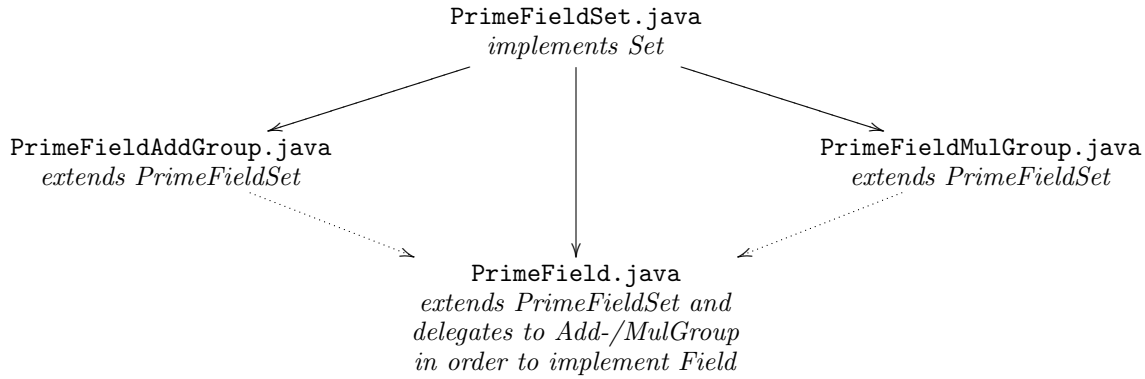
```
/** Copy for Integers.
 * Input Object o is unchanged
 * Returns new Object copy_of_o
 */
```

```
public Object _copy(Object o) {
   return new Integer((Integer)o);
}
```

Some objects though being mutable will be restricted of any change. They are the constant objects. You wouldn't want to change the object representing *one* or *zero* for example.

4.2 'Constant' Objects are created in the Construction

When implementing a non-trivial class like a prime field, we often want to use objects whose value is fixed. Those are the constants. Their purpose is to be handy for several methods of the class. All these constants should be made ready at construction time, since this process is slow anyway and it is unlikely that this will ever happen in the loop of an algorithm. So if in some method you require a constant, that equals twice the multiplicative neutral element, i.e. *two*, build it at construction time and *not* when the method is called. Construction time here means the time when the lowest, i.e. the *set*-component is built, which is then extended by all the fancier classes. A little diagram makes this clear.



We make the constants `protected` so we may use them throughout this structural hierarchy. As an example we will look at the constants and the constructor of the PrimeFieldSet class that was discussed here.

Example 4.2.1. *Constants and constructor of* `PrimeFieldSet.java`

```
public class PrimeFieldSet implements Set {

   //we need integer arithmetic all over the place
   protected Integers Z = new Integers();
   protected AbelianGroup intadd = (AbelianGroup)Z.add();
   protected Monoid intmul = (Monoid)Z.mul();
```

```
    //these are some real constants
    protected Integer zero = (Integer)intadd.neutral();
    protected Integer one = (Integer)intmul.neutral();
    protected Integer two = new Integer(2);
    protected Integer order;
    protected Integer negorder;
    protected Object lowbound;
    protected Object uprbound;


    public PrimeFieldSet(int p) {
       ...//check that p is prime
       order = new Integer(p);
       negorder = (Integer) Z._copy(order);
       intadd.inv(negorder);
       ...//set lowbound and uprbound constants
     }
     ...
}
```

All these constant Objects will be used by numerous methods within the PrimeField implementation and the constructor, as we have seen here, is really just making sure all the constants do have the right values in place. Notice that most of these objects are not constants to java. It is understood that they should not be changed after the construction is complete.

4.3 Methods change Objects instead of returning them

This rule is closely related to the last one. Since the objects are mutable the methods we write will certainly work on the given objects changing them. It is a question of style whether one should then return a reference of the object that has changed, but I decided against it. This leaves the option of returning something open for when a really new object is created.

It is generally the first parameter of a method that is changed. If the method for example describes a binary operation the result of a call should be found afterwards in the first given object and the other one should be left untouched. This will make code look something like this.

Example 4.3.1. *Fibonacci Numbers*

```
/** Fibonacci Numbers.
 * Input Integers Z is unchanged, int n
```

```
 * Returns new Object the_nth_fib_number
 */
public static Object fib(Integers Z, int n) {
   Integer a = Z._copy(Z.mul().neutral()); // a = 1
   Integer b = Z._copy(a); // b = a

   for(int i=0;i<n;i++) {
      if(i%2==0) Z.add()._op(a,b); // a = a + b;
      else Z.add()._op(b,a); // b = b + a;
   }

   return n%2==0 ? b : a; // returns the bigger one
}
```

This whole style of programming is very much like an assembler language, which is not surprising since it is meant to speed things up, but it does take some getting used to. It also *requires* the programmers to read and write clear specifications for each method to make sure the user understands the usage. For the really basic operations like adding and multiplying integers this regulation may not seem worth the effort but if we observe the following example, then the whole algorithm used will play into our hands:

Example 4.3.2. *Polynomial Division*

$$\begin{array}{rl} (\quad X^3 + X^2 \qquad - 2) \div (X-1) = X^2 + 2X + 2 \\ \underline{-\,X^3 + X^2} \\ \qquad\quad 2X^2 \\ \qquad \underline{-\,2X^2 + 2X} \\ \qquad\qquad\quad 2X - 2 \\ \qquad\qquad \underline{-\,2X + 2} \\ \qquad\qquad\qquad\quad 0 \end{array}$$

Notice that the algorithm starts with two polynomials. These are naturally Polynomial objects in [Jalgebra]. Let us call the first one $p(X) = X^3 + X^2 - 2$ *and the second* $d(X) = X - 1$*. We know the usual polynomial division algorithm is made by a loop that corresponds to a three-line set above. After the first three lines we see that the solution up to this point is* $X^2 + \frac{r(X)}{d(X)}$ *where* $r(X) = 2X^2$ *is the remainder up to this point.*

Now we do nothing but redo the same loop with the $r(X)$ *instead of* $p(X)$ *until the degree of* $r(X)$ *is smaller then that of* $d(X)$*. But this means that in the second loop the information "what is* $p(X)$*" has become redundant as we only need to know* $r(X)$*. So instead of wasting memory by having two variables we change* $p(X)$ *into the remainder* $r(X)$ *after every loop. This also has the positive effect that after the algorithm is done the variable* $p(X)$ *automatically becomes* $p(X)$ *mod* $d(X)$*. So the algorithm did really play into our hands this time.*

4.4 Implementations are slim

In correspondence with the Three Way System (see chapter 2) the implementations in [Jalgebra] are written very closely tied to their interfaces. They should have no more non-private methods then necessary to fulfill the requirements of their corresponding interfaces. All the fancy stuff that can be done for example with the integers should be implemented as an algorithm. This algorithm however should not use the integers directly but the least necessary interfaces it requires. In this way the algorithm must never be written again for a different structure that has all the necessary interfaces. In other words it will work on all the structures it could work on. This also lets us make the algorithms a little smart sometimes. I will show this with the example of the simple power function:

Example 4.4.1. *The simple power function decides how much it can do. Taken from* `GroupAlgorithm.java`

```
/** Simple Power Function for Semigroups up to Groups.
 * Input Object x changes to x^y, Object y is unchanged
 * SX semigroup contains x, RY ordered ring contains y
 */
public static void _pow(Semigroup SX, Object x,
                    ...OrderedRing RY, Object y) {

   Object zero = ((Monoid)RY.add()).neutral();
   Object one = ((Monoid)RY.mul()).neutral();

   if(RY._slt(y,zero)) {  // needs a group
      if(!(SX instanceof Group))
         throw new MathException("Can't take powers < 0
                              ...without a group.");

      ((Group)SX)._inv(x);
      Object ycopy = RY._copy(y);
      ((Group)RY.add())._inv(ycopy);
      y = ycopy; // original y object remains unchanged
    } else if (RY._eq(y,zero)) { //need a monoid
       if(!(SX instanceof Monoid))
          throw new MathException("Can't take powers = 0
                               ...without a monoid.");

       ((Monoid)SX)._setNeutral(x);
       return;
   }

   Object xcopy = RY._copy(x);
```

```
    for(Object i = RY._copy(zero); RY.sle(i,y);
                                  ...RY.add()._op(i,one))
        SX._op(x,xcopy);
}
```

This method decides at runtime whether the given input makes sense, i.e. whether the power can be calculated with the given structure. This makes the power function very user friendly.

This ends the regulations we wanted to see and now we go on to learn about a mistake that should really be avoided.

4.5 A big Mistake

In fact the biggest mistake I have made while writing the [Jalgebra] package and the one that took the longest time to correct is described here. Too see what happened we start by following a line of thought from algebra.

The way the structures we want to implement here were discovered or rather described and defined are not uncorrelated. For example any sensible teacher would tell you all about the integers $\mathbb{Z}$ before he starts to explain a prime modulo-ring like $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$. This is because they are so very similar. Loosely speaking the operations on the primefield work in a truncated way and the multiplicative inverses can be defined via Euclid's algorithm but that's all there is to it.

When you think about it the very same thing will apply to a primepower field $\mathbb{F}_{p^q}$ and the polynomial ring $\mathbb{F}_p[X]$ over its prime ground field. It is just another modulo structure with the irreducible polynomial instead of the prime. Consider the ring $\mathbb{F}_p[X]$ and the ring of the integers $\mathbb{Z}$. Any integer $i \neq 0$ has a unique prime factorization $i = u \cdot p_1^{q_1} p_2^{q_2} \cdots p_n^{q_n}$ where p's are prime, q's are integers and $u \in \mathbb{Z}^\times$ is a ring unit.

Now the same thing happens in $\mathbb{F}_p[X]$, it just looks a little different. A polynomial $f(x) \in \mathbb{F}_p[X]$ has a unique factorization in irreducible polynomials. It looks like $f(x) = u \cdot p_1(x)^{q_1} \cdot p_2(x)^{q_2} \cdots p_n(x)^{q_n}$. Remember that the units in this ring $\mathbb{F}_p[X]^\times$ are the constant polynomials so since u is in this set we didn't need to write $u(x)$. The $p(x)$'s here are now irreducible polynomials and the q's again integers. So the irreducible polynomials do somewhat correspond to the prime numbers in these polynomial rings, but there is more.

We denote an ideal in $\mathbb{Z}$ generated by a number n as follows $n\mathbb{Z} = \{n \cdot z | z \in \mathbb{Z}\}$. Factorizing $\mathbb{Z}$ by such an ideal gives us a field if and only if n is prime. We use the notation $\mathbb{Z}/p\mathbb{Z} = \mathbb{F}_p$. Note that this field is isomorphic to the following set $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z} \cong \{0, 1, \ldots p-1\}$. To understand this let us call the later set

R. Since $\mathbb{Z}$ is a euclidean ring, any integer a can be uniquely written in the form $a = p \cdot m + r$ where $r \in R$ and m is an integer. But in $\mathbb{Z}/p\mathbb{Z}$ the first summand becomes 0.

So this R was my first choice as representatives for the elements in $\mathbb{F}_p$, but there is an even better one. We shall see which this is later when we talk about the actual prime field implementation in section 5.2.

Analoguesly we denote an ideal in $\mathbb{F}_p[X]$ generated by a polynomial $f(x)$ as follows $f(x)\mathbb{F}_p[X] = \{f(x) \cdot g(x) | g(x) \in \mathbb{F}_p[X]\}$ (sometimes these ideals are denoted shorter: $(f(x)) = f(x)\mathbb{F}_p[X]$). The analogy continues here because we get a field when factorizing by such an ideal if and only if the generating polynomial $f(x)$ was irreducible. Let us consider such a case. So $f(x)$ is now irreducible in $\mathbb{F}_p[X]$. We may say its degree is q for some $q \geq 1$, since polynomials of lesser degree, i.e. constants are units in the ring and not considered irreducible.

The field we get is $\mathbb{F}_{p^q} = \mathbb{F}_p[X]/f(x)\mathbb{F}_p[X]$. This field has p^q elements and is in fact isomorphic to the following set of all polynomials in the ring up to but not including degree q. We can say $\mathbb{F}_{p^q} = \mathbb{F}_p[X]/f(x)\mathbb{F}_p[X] \cong \{a_0 + a_1X + a_2X^2 + \cdots a_{q-1}X^{q-1} | a_i \in \mathbb{F}_p\}$. Let us call the last set R again. The argument for the isomorphism corresponds very much to the integer case. Since $\mathbb{F}_p[X]$ is a euclidean ring, any element $a(x)$ can be uniquely written in the form $a(x) = f(x) \cdot m(x) + r(x)$ where $r(x) \in R$, $m(x) \in \mathbb{F}_p[X]$ and $f(x)$ is the irreducible polynomial of degree q. In $\mathbb{F}_p[X]/f(x)\mathbb{F}_p[X]$ however the first summand in the equation becomes 0 again.

To find out how many elements R has consider the following. It has q coefficients a_i which can each be any element in $\mathbb{F}_p$, so $|\mathbb{F}_{p^q}| = |\mathbb{F}_p| \cdot |\mathbb{F}_p| \cdots |\mathbb{F}_p| = |\mathbb{F}_p|^q = p^q$. The set R we have seen here is also my choice of representatives for the elements of $\mathbb{F}_{p^q}$.

This marks the end of the little algebra line of thought we followed. We have now seen that in both cases the representative elements of the fields $\mathbb{F}_p$ and $\mathbb{F}_{p^q}$ we wish to implement are really the same elements that were used in the rings $\mathbb{Z}$ and $\mathbb{F}_p[X]$ of which the fields are modulo structures. So realizing this you might think the following. Why don't we use Java's special feature allowing us to extend a single class in order to save ourselves some coding time.

Sadly enough the thing that goes so wrong with this idea happens after you write almost all the code. It does save a lot of lines though to write for example the primefield as extended integers. We just rewrite the addition and multiplication and add an inverse method and we're done. However things go wrong because Java tries to be smart for us without us telling it to or even

being able to control it.

There is a feature (not a bug) within Java that will ruin things for us here. We will see what it does illustrated in this little example:

Imagine you wrote a simple power function inside your integers for later use, which just repeatedly calls a multiplication. It would look something like this. A totally unchecked internal power method for integers:

```
protected void _pow(Object x, Object y) {
   Object xcopy = _copy(x);
   for(Object i = _copy(zero); sle(i,y); _add(i,one))
      _mul(x,xcopy);
}
```

Now you want to write any method within the primefield, which extends the integers, that uses a call to this integer power. For example let us write an inverse implementation using Fermat's Little Theorem (see in Section 5.2.3). A simple implementation of this would be:

```
public void _inv(Object x) {
   Object orderminustwo = _copy(one);
   _addinv(orderminustwo);
   _add(orderminustwo, orderminustwo);
   _add(orderminustwo, order);

   _pow(x, orderminustwo);
}
```

This method however would not do what we might think. What happens is Java will call the old integer power function but use the new primefield multiplication within it. Whenever Java executes a method within a subclass, like the integer power, and within this method others are called which have been redefined, like the multiplication in a primefield, java will always and without asking use the highest (with respect to the class hierarchy) implementation it can. In our case this means the wrong multiplication will be used and the whole result will be wrong. So inverses cannot be calculated without a workaround in this fashion.

We see that despite the fact that our prime field extends the integers we cannot use the methods implemented there without being very careful! If we insist on using Integers methods we need to create an Integers object within the PrimeField object to provide the arithmetic that has been overwritten. But this would mean that creating a PrimeField would always create two Integers objects. One that has been extended to become a PrimeField and the other for the lost arithmetic. This is just horribly inefficient in terms of time

and memory.

Note that the problems would get far worse as we move on to more complicated structures that build upon the ones we've already got. Like the Polynomial Ring, which works over a Ring and the Primepower Field that was once supposed to extend a Polynomial Ring over a Prime Field. Java will not make it so easy for us.

So the actual version of [Jalgebra] doesn't use these ideas. An `Integers` class is created whenever a `PrimeFieldSet` or `PrimePowerFieldSet` is created, but they do not extend each other, the more complicated structures just borrow the arithmetic where it is needed.

5

Implementations

In this chapter we will discuss various classes that are implemented in the [Jalgebra] package. We begin with the most basic implemented Ring class, the *Integers*. From there we make our way up to more complicated structures like *Prime Fields*, *Polynomial Rings* and even *Primepower Fields*.

5.1 Integers

The Integers are a fairly easy thing to do, because they are so basic, that we have to cheat a little bit. The rules of only working with other components from our package doesn't apply here. All further structure implementations should however use the Integers instead of Java's own `int`.

At the moment the Integers are just a wraparound of Java's simple datatype int, however they are also an Object that can be replaced by something more complicated. In the future we could replace the methods within the Integers by something much more complicated. We could make the Object Integer a Vector of int's, for example. This would then not only allow much bigger Integers but also bigger Prime Fields, Polynomial Rings over these Fields, and much much more.

However to go on we will not dwell on the Integers. They are at the moment not smart or interesting in any way other then that they use Java's `int`-arithmetic in the [Jalgebra] fashion and give us a nice wraparound to work with for the other classes. A typical Integers method looks like this.

```
public void _mod(Object _x, Object _y) {
   int x = ((Integer)_x).get();
   int y = ((Integer)_y).get();
   ((Integer)_x).set(x%y);
}
```

The Integers class should be the only one that uses the Integer Objects `get()` and `set()` methods. Once the Integers class exist any manipulation of Integer Objects should be done using just this class. This is to ensure compatibility once the Integers are replaced by something more complicated. Let us move on to the much more interesting Prime Fields though.

5.2 Prime Fields

There are a more points of interest that we come across when looking at the implementation of the Prime Fields and here they are.

5.2.1 Is p prime

The first thing we want to think about, when implementing a Prime Field, is how to write a method that checks whether a given integer p is really a prime. A good, polynomial-time algorithm for this has been devised by Manindra Agrawal, Neeraj Kayal and Nitin Saxena. They have written a paper [AKS02] about their AKS-Algorithm and after reading it we can be sure it could be implemented in [Jalgebra], although this has not been done yet.

For now a different probability based check by java's own BigInteger package is used. This check is potentially faster in finding if the given number is composite, but not completly reliable, as it does have an error probability by nature. So once it tells us that p is prime, we still have to check it more thoroughly before moving on.

This is now done by simple trail-and-error method that goes from 2 to $\sqrt{p}$ to find a divisor. Here the AKS-Algorithm would do much faster work, but since this is only used when a `new` PrimeField is created, time is not an important factor here.

5.2.2 Representatives

Another issue is not quite so obvious. Since $\mathbb{Z}/p\mathbb{Z}$ is really a modulo structure and its p elements can be represented by any p successive numbers in $\mathbb{Z}$ it is not clear which ones are the best choice. The most common representatives for the cosets are of course the numbers $0, \ldots p-1$. These were the ones I used first, but there is a slight drawback here. They are not the most intuitive representatives. Let us take for example $p = 17$. Then our representatives would be the integers $0, \ldots 16$. If we do multiplications with the last three of

these the outcome is often quite *hard* to see (at least without a calculator). What is $16^4 \cdot 15^2$? Well usually quite a big number, but in this ring we can just take other representatives and it becomes obvious.

So let is take 17 numbers around zero, i.e. $-8, \ldots 0, \ldots 8$. With these the same calculation becomes $(-1)^4 \cdot (-2)^2 = 4$. This was much easier. To implement this idea in a Prime Field we introduce two constant Integers: l for *lowerbound* and u for *upperbound*. Now all elements of the set are between those two and all operations must take care, that when they finish the objects which they change are back in there. If $p \neq 2$ then $l = -(p+1)/2$ and $u = (p+1)/2$. Since the Integers in [Jalgebra] for now are just a wraparound for the simple data type `int` in java this representation has another benefit. Since we are now actually using the sign bit of this data type, we can do some calculations that would have produced an overflow before. Much like the one with the big powers in the example.

These same representatives can also be defined in a different way. Remember the *norm* on the integers used for example in Euclid's Algorithm is the absolute value function. An element $a \in \mathbb{Z}/n\mathbb{Z}$ is a coset of $n\mathbb{Z}$, i.e. $a = b + n\mathbb{Z}$ for some $b \in \mathbb{Z}$. We want our representatives to be minimal in terms of the norm, i.e. the absolute value. In the case of even n this will not be unique for the $n/2 + n\mathbb{Z}$ element. So in these cases we take the positive integer $+n/2$ instead of $-n/2$ to be our representative. We define the representative to be

Definition 5.2.1. *Let the coset $(a + n\mathbb{Z})$ be an element of the modulo ring $\mathbb{Z}/n\mathbb{Z}$, then its representative is the integer $k \in a + n\mathbb{Z}$ which has minimal norm.*

$$(\forall x \in a + n\mathbb{Z})\, |k| \leq |x|$$

Should there be two k's of minimal norm, we chose the positive one. This makes the representative unique.

Note that the representative definition makes sense for all integer modulo rings $\mathbb{Z}/n\mathbb{Z}$, not only the ones where n is prime. As all ideals in $\mathbb{Z}$ have the form $n\mathbb{Z}$ for some n these are all the factor structures of $\mathbb{Z}$.

Note that using smart representatives in the PrimeField will also effect the way polynomials over this field look, i.e. it will also change the way representatives of a Primepower Field will look and act.

5.2.3 Inverses via Fermat

There are two ways that come to mind to calculate inverses in finite prime fields. We will see that both of them work fine in [Jalgebra] and that one is

definitely faster in most cases.

We start with Fermat's approach, or rather the approach that uses Fermat's Little Theorem (a proof is given in Section A.1.1). It states, that for all a coprime to p the following equation holds.

$$a^{p-1} \equiv_p 1 \pmod{p} \tag{5.1}$$

Let p be the prime that generated the field, then the multiplicative inverse any non-zero element a (they are all coprime to p, since p is prime) is its $(p-2)$th power. As an example let us look at $\mathbb{F}_5$ and find the inverse of 2.

The order $p = 5$ so $p - 2 = 3$ and $2^3 = 8$. Now if Fermat is right then

$2 \cdot 8 = 2^{p-1} \equiv_p 1$, and indeed
$2 \cdot 8 = 16 \equiv_5 1$

So equation 5.1 can really help us to find inverses within arbitrary prime fields but we need to take a potentially big power (p could be huge) of the element we wish to invert.

This method as we will see can still be optimized for a computer by an algorithm called Repeated Squaring (RS for short). Repeated Squaring uses the fact that we are in a modulo ring (in this case $\mathbb{Z}/5\mathbb{Z}$ and so to calculate 2^{100} in this ring we do not really need to calculate $2 \cdot 2 \cdots 2$ all the way before we check what is this big number modulo 5. It works as follows.

Let a be an element in $\mathbb{Z}/n\mathbb{Z}$ (notice n need not be prime for the Repeated Squaring algorithm) and $k \in \mathbb{N}$. Then a^k can be found by the following iteration.

$a_0 = 1$, we think of k in binary from now on.
iteration over i, starts at $i = 1$:

`case one`: the i-th digit from the front of k is 0, then
$a_{i+1} = (a_i^2 \bmod n)$
`case two`: the i-th digit from the front of k is 1, then
$a_{i+1} = (a_i^2 \cdot a \bmod n)$

Redo until the last digit of k is reached. The result is the last calculated $a_{\lfloor \log_2 k \rfloor + 1}$.

This takes $\lfloor \log_2 k \rfloor + 1$ (length of k in binary) steps. We will see how good this works in an example. Let's calculate 3^{10} in $\mathbb{Z}_7$. Let $a = 3$, $k = 10$ and $n = 7$.

$a_0 = 1$ and $k = 1010_2$

$a_1 = 1 \cdot 3 \bmod 7 = 3^{0_2} \cdot 3^{1_2} \bmod 7 = 3$, as the first digit is one.
$a_2 = 3^2 \bmod 7 = 3^{10_2} \bmod 7 = 2$, as the 2nd digit is zero.
$a_3 = 2^2 \cdot 3 \bmod 7 = 3^{100_2} \cdot 3^{1_2} \bmod 7 = 5$, as the 3rd digit is one.
$a_4 = 5^2 \bmod 7 = 3^{1010_2} \bmod 7 = 4$, as the 4th digit is zero.

So the result is $3^{10} = 4 \bmod 7$. In the integers $3^{10} = 59049$ which is indeed equal to 4 modulo 7. We took $4 = \lfloor \log_2 10 \rfloor + 1$ steps.

Notice the first step in this algorithm is the same for any $k \neq 0$. So it is in general omitted and we start directly with $a_0 = a$ and need then exactly $\lfloor \log_2 k \rfloor$ steps.

The same algorithm is implemented within [Jalgebra] for arbitrary Euclidean Rings. Indeed we will see that inverses in a Primepower Field, which uses polynomials and not numbers as representatives, can be calculated in the same way. The file that contains the algorithm is `ringalgorithm.java`.

5.2.4 Inverses via Euclid

Euclid's approach is more subtle and uses only Euclids own famous algorithm. As p is a prime all non-zero elements of the field $\mathbb{F}_p$ are coprime to p. Euclids Algorithm is used to calculate the greatest common divisor (in short gcd) of two elements in a euclidean ring (i.e. a normed ring in which a Euclidean Algorithm can be defined). So for any element $a \in \mathbb{F}_p^\times$ this $\gcd(a, p)$ is equal to a unit of the ring $\mathbb{Z}$. Euclid also showed that by isolating the gcd and going through the algorithm backwards we can find two ring elements s, t such that

$$\gcd(a, p) = s \cdot a + t \cdot p$$

Within the field $\mathbb{F}_p$ this simplifies to the equation.

$$\gcd(a, p) = s \cdot a$$

$$1 = (\gcd(a, p))^{-1} \cdot s \cdot a$$

As the $\gcd(a, p)$ in our case can only be 1 or -1 which are self-inverse, we may write.

$1 = \gcd(a, p) \cdot s \cdot a$, so the inverse of a in general is

$$a^{-1} = \gcd(a, p) \cdot s$$

Let us make an example to understand this better. Let $p = 5$ then our representative elements of $\mathbb{F}_5$ are $-2, -1, \ldots, 1, 2$. Calculating the gcd of -2 with Euclids Algorithm gives us the equations.

$5 = -2 \cdot -2 + 1$
$-2 = -2 \cdot 1 + 0$, so $\gcd(-2, 5) = 1$

But we already knew that much as 5 is a prime. Now there is a method which will allow us to find coefficients n, m in $\mathbb{Z}$ such that

$$1 = \gcd(-2, 5) = n \cdot (-2) + m \cdot 5$$

So let us find these coefficients. We start with the equation where the last term is equal to the gcd and then rewrite it to isolate the gcd on the rhs.

$$5 - (-2 \cdot -2) = 1$$

In our simple example this already did the trick. In general there is usually more re-inserting of equations required. Here we see.

$$1 = 2 \cdot -2 + 5$$

The inverse of -2 in $\mathbb{F}_5$ is therefore 2.

So how is this interesting coefficient s calculated in general. Let us assume we did the normal Euclid Algorithm on a, b till the end. Let $a = b_1$ and $p = b_2$ then the normal algorithm works like this

$b_1 = a_1 \cdot b_2 + b_3$
$b_2 = a_2 \cdot b_3 + b_4$
$\cdots$
$b_{n-2} = a_{n-2} \cdot b_{n-1} + b_n$
$b_{n-1} = a_{n-1} \cdot b_n + b_{n+1}$
$b_n = a_n \cdot b_{n+1} + \gcd(a, p)$

We solve every line for its last summand and re-insert the result into the one before. As we do this we get a representation of the gcd in terms of two b_is with ever smaller growing indices.

$$\gcd(a, p) = b_n + b_{n+1} \cdot (-a_n)$$

$b_{n+1} = b_{n-1} + b_n \cdot (-a_{n-1})$
$\gcd(a, p) = b_n + (b_{n-1} + b_n \cdot (-a_{n-1})) \cdot (-a_n)$
$\gcd(a, p) = b_{n-1} \cdot (-a_n) + b_n \cdot (1 + a_n \cdot a_{n-1})$

$$b_n = b_{n-2} + b_{n-1} \cdot (-a_{n-2})$$

$\gcd(a,p) = b_{n-1} \cdot (-a_n) + (b_{n-2} + b_{n-1} \cdot (-a_{n-2})) \cdot (1 + a_n \cdot a_{n-1})$
$\gcd(a,p) = b_{n-2} \cdot (1 + a_n \cdot a_{n-1}) + b_{n-1} \cdot (-a_n - a_{n-2} \cdot (1 + a_n \cdot a_{n-1}))$

The iteration emerging here is the following. Observe the sequence of coefficients where s_n is the coefficient of b_n in the first line and s_{n-1} the coefficient for b_{n+1}. In the next line the three lines we have calculated should have the following sequence members

$\gcd(a,p) = b_n \cdot s_n + b_{n+1} \cdot s_{n-1}$
$\gcd(a,p) = b_{n-1} \cdot s_{n-1} + b_n \cdot s_{n-2}$
$\gcd(a,p) = b_{n-2} \cdot s_{n-2} + b_{n-1} \cdot s_{n-3}$

Notice that the reuse of s_{n-1}, s_{n-2} is consistent with our calculation. Indeed if we look at the calculation it can be seen that

$s_n = 1$ and $s_{n-1} = -a_n$ as well as the iteration

$s_i := s_{i+2} - a_{i+1} \cdot s_{i+1}$ covers what we did.

$s_{n-2} = 1 + a_n * a_{n-1}$ and
$s_{n-3} = -a_n - a_{n-2} * (1 + a_n * a_{n-1})$ both work in this fashion.

If we follow the re-inserting through to the end we get.

$\gcd(a,p) = b_1 \cdot s_1 + b_2 \cdot s_0$, inserting a and p again gives
$\gcd(a,p) = a \cdot s_1 + p \cdot s_0$

So the s we were looking for is s_1 in our iteration. Now we shall do another more complicated example to use this knowledge. Let $a = 151$ and $p = 127$ then the usual algorithm gives

$151 = 1 \cdot 127 + 24$
$127 = 5 \cdot 24 + 7$
$24 = 3 \cdot 7 + 3$
$7 = 2 \cdot 3 + 1$, done $\gcd(151, 127) = 1$ (no surprise as 127 is a prime).

We did 4 steps so we start with $s_4 = 1$ and $s_3 = -a_4 = -2$.

$s_2 = s_4 - a_3 \cdot s_3 = 1 + 3 \cdot 2 = 7$
$s_1 = s_3 - a_2 \cdot s_2 = -2 - 5 \cdot 7 = -37$
$s_0 = s_2 - a_1 \cdot s_1 = 7 + 1 \cdot 37 = 44$

We could have stopped at s_1 but just to check we did the right thing we need s_0. So a quick check is

$\gcd(a, p) = s_1 \cdot a + s_0 \cdot p$
$1 = -37 \cdot 151 + 44 \cdot 127 = -5587 + 5588$, so it works!

The part of the [Jalgebra] package, that does all these calculations for us is also in `ringalgorithm.java`. It is again implemented for arbitrary Euclidean Rings. So it also can be used to calculate inverses in Primepower Fields.

5.2.5 Fastest Inverse

So now that we have seen both of these ways to implement inverses in Primefields which one is faster. This is particularly interesting because the very same algorithms can be used to calculate the inverses within the Primepower Fields as well. So they are double useful.

The steps taken in a repeated squaring is as we have seen always $\lfloor \log_2 k \rfloor$, where k is the number we actually wish to invert. With Euclid the number of steps taken cannot be given in general. However we know that Euclids Algorithm works worst for two successive Fibonacci Numbers. Consider the Fibonacci sequence

$f_0 := 1$, $f_1 := 1$ and
$f_n := f_{n-2} + f_{n-1}$

So it starts like this: $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610,$
$987, 1597, 2584 \ldots$

As the second number in our inverse finding case will always be a prime we might ask ourselves whether the case of two successive Fibonaccis will ever occur. A quick check reveals that Fibonacci primes are $2, 3, 5, 13, 89, 1597, \ldots$ so there are some of those to start with. However there is a catch. Let us assume we take one of these numbers to be our p and let us assume we didn't take 2 as this is really not interesting. Then the Fibonacci predecessor of p will be just a little bigger then $p/2$ by the way the Fibonacci Numbers are defined.

A quick proof by contradiction. Assume p_{-1} is the Fibonacci predecessor of $p > 2$ and is smaller or equal to $p/2$ then p_{-1}'s predecessor p_{-2} is even smaller as the Fibonacci sequence is strictly increasing. But now

$$p = p_{-1} + p_{-2} < p_{-1} + p_{-1} \leq p/2 + p/2 = p$$

This is of course a contradiction as $p < p$.

So we see that p_{-1} is really bigger then $p/2$ and would therefore be taken $-p$ by our representation system of $\mathbb{F}_p$. The resulting representative would

probably be a small negative number. So the case we are about to consider will never really occur, but let us do it nonetheless, since the steps in Euclids Algorithm are hardly predictable in other cases. Keep in mind though that this is a worse then worst case prediction for Euclid.

Fibonacci Number	**Steps taken by Euclid**	**Steps taken by Repeated Squaring**
3	2	0
5	3	1
8	4	2
13	5	3
21	6	4
34	7	5
55	8	5
89	9	6

In the ith row the entry in the first column here is the $i+3$rd Fibonacci Number. If the first column has value k, the second and third column contains the highest number of steps to calculate any 'inverse' in $\mathbb{Z}_k$.

In the case of Euclids Algorithm this would be the amount of steps to calculate the inverse of the Fibonacci predecessor of k. These 'inverses' always exist, since two adjacent Fibonacci Numbers are coprime, i.e. have gcd = 1, so the coefficient we calculate is a multiplicative inverse in this ring, whether k is prime or not. If k is the ith Fibonacci Number, then this number of steps is $i-2$, so inverting 5 (the 5th F-Number) in $\mathbb{Z}_8$ takes 3 Steps.

The RS case is easier. The number of steps taken here is independent of the element you wish to invert, if it's not one or zero where the outcome is clear from the start. So the worst case will take $\lfloor \log_2(k-2) \rfloor$ steps as we have seen in the introduction to RS earlier.

We can see the Repeated Squaring algorithm is better then Euclid in these cases and we can even prove that this will be so as we go on. We can see and it is clear by the way Euclid works, that the steps taken by Euclid will increase by one for each increase of the Fibonacci Number.

The steps taken by RS are $\lfloor \log_2(k-2) \rfloor$, where $k = f_i$ the ith F-Number is the first entry. Let us forget the floors and just consider the term inside. Let us also assume we started later then row 3, then the difference between two successive F-Numbers is greater then 2. We see that

$$f_i - 2 = f_{i-1} + f_{i-2} - 2 < 2 \cdot f_{i-1} - 4$$
$$f_i - 2 < 2 \cdot (f_{i-1} - 2)$$

$$\log_2(f_i - 2) < \log_2(2) + \log_2(f_{i-1} - 2)$$
$$\log_2(f_i - 2) < 1 + \log_2(f_{i-1} - 2)$$

Notice the RHS is just the number of steps taken by RS on the previous F-Number plus one. As the LHS is smaller this tells us the term inside the floor brackets will never increase by as much as one for a single step. We Remember Euclid does this for every step. It is clear that RS will always be faster in this table.

The table shows that RS does increase by one however quite often and this is due to the fact that we omitted the floor brackets. So we don't increase by as much as one *compared* to the last term *without* the floor brackets. The argument that RS is faster is however still valid, as with the floor brackets RS never increases by as much as two and sometimes it doesn't increase at all as we have seen, so it will stay below Euclid.

So now that we are convinced of these cases, let us think practical. We have are already seen that they will never occur for our inverse problem, so let us find out that Euclid is really better then RS on average. Let's invert say $9, 10$ and 11 in $\mathbb{Z}_{23}$ and just count the steps. The result is devastating for RS.

Numbers to invert	**Steps taken by Euclid**	**Steps taken by Repeated Squaring**
9	3	4
10	2	4
11	1	4

These numbers where quite random, so we see on average Euclid has the upper hand. Euclid is therefore implemented in [Jalgebra] to invert elements of a Prime- / PrimePower Field.

5.3 Polynomial Rings

The implementation of a Polynomial Ring is a bit more tricky then the Integers. We will see a few of the problems that this poses here.

5.3.1 Degrees and the Null

One of the most basic problems is to realize and keep track of the degree of a polynomial all the time. The way the Polynomial Object is written in [Jalgebra] is the following. It extends a Vector Object (a dynamically growing array) from Java and uses this to store references for all the coefficients. This is very useful for polynomials with lots of entries.

We might also fill this vector with pairs of the form `(degree, reference to coefficient)` for every non-zero coefficient. This would obviously take much less space for polynomials of high degree with lots of zeros. It could however in the worst case be twice the size we need for a complete polynomial (all coefficients are non-zero). Depending on what we wish to do in an algorithm or an application we should always be able to switch between those two.

Notice that [Jalgebra] at the moment only features the first of these polynomial implementations as there was no need for the other one *yet*.

One thing we have to decide in either case is what should represent the zero polynomial and what the one, i.e. the neutral elements of the Ring we implement. For the zero we choose in both cases the empty vector. Especially zero and one should be objects, whose creation takes as little time as possible, as these will be used whenever *new* elements of the ring are created anywhere. The empty vector is the fastest possible for a Polynom Object and the degree should be the lowest Integer possible (a public static constant Integer MINUS_INFINITY should be available).

The one is then straightforward. In the first case it is a Vector filled with a reference to the ground Ring one and in the other it is simply the pair (zero, reference to ground Ring one).

Note that when you try to read the coefficients of the zero you will get null-references by default and not the ground Ring zero object. In fact the way we made these polynomials in the first case is very much assuming, that null-references mean the ground Ring zero anyway. If the Vector is bigger then the degree, it contains null-references, which should then mean zeros. The empty Vector only contains null-references which should also stand for zero, so we might as well say it is a convention.

This will make things go a lot faster because we do **not** need to create a *new* zero object for every coefficient we have. In fact this convention makes the first way to implement Polynomials almost as good as the second, even for polynomials with lots of zeros, since the null-references representing the zeros do not take up space.

We must keep in mind however to make our algorithms understand both a coefficient being a ground zero object and a coefficient being a null-reference, for everything to work smoothly.

A nice trick would also have been to say that the Integer zero is the null-reference to begin with, since that would have made the way to implement these polynomials obvious and smoother, however this might be dangerous in

other areas and was therefore not done in [Jalgebra].

5.3.2 Addition and Multiplication

As easy as it sounds and is in mathematics, the addition and multiplication of polynomials is no laughing matter for Java. Especially as we have to be careful with the null-references meaning zero now.

Addition for example is now implemented as a nice four step algorithm, but it took some time to get it right. The steps are:

1. *solve trivial cases, where one summand is zero*
2. *find which one has bigger degree and make f have that degree*
   f here is the first argument of the operation, i.e. the polynomial we want to change to become the outcome.

3. *add g to f where it has objects and not null-references*
   At this point this is merely a careful delegation to the ground Ring taking care of the nulls.

4. *correct the degree of the outcome*

```
for(int i=fd;i>=0;i--){   //fd = degree of f
   fco = f.getCo(i);      //fco = coefficient of f
      if(fco!=null && !Radd._isNeutral(fco))
         if (i==fd) { break; }
         else { f.setDegree(i); break; }
   if(i==0) { f.setDegree(Polynomial.MINUS_INFINITY); }
}
```

   Here we go through the coefficients of f and look for one that isn't zero to set the correct degree.

The multiplication is not much more complicated, but it is not so nice and organised and does the steps 3 and 4 combined, so we may omit it here. Note however, that since we will work over Fields a lot, step 4 is not even needed most of the time. The product of the leading coefficient of both polynomials has to be zero for the degree correction to take effect.

5.3.3 Working on just a Ring

The problem here is that while we write the code for an arbitrary Polynomial Ring we can make no assumptions about the Ring we work over. This is a problem because for example Polynomial Rings over Fields are Euclidean. So

should we include the EuclideanRing interface or not. The truth is we cannot know.

The way we circumvent this is the following. We include the interface in every Polynomial Ring and whenever someone calls a method, that uses the properties or is even part of this interface, we check whether this call makes sense. This is done using Javas `instanceof` operation. For example the EuclideanRing interface is implemented in PolynomialRing.java, but every method it has contains the following line in the implementation.

```
if(!(R instanceof Field))
   throw new MathException("This ring is not euclidean.");
```

The R in this case is a reference for the Ring we are working over. For efficiency reasons this check is only done on the 'safe' methods without the '_' in front.

The same circumvention would also work if we wanted to create a lexicographic order in the PolynomialRing class. Before any order-related method is called there we would have to check of course, that the Ring we are working over is also ordered, i.e. an instance of the OrderedRing interface.

Notice that the 'unsafe' methods here are even unsafer then usual as they might call methods that are not implemented in the class. These should really only be used by those who know they need the timecut. However if what we do works with the safe method then everything must be in order and we may switch to the unsafe one. However if we change something afterwards switching we might get complicated errors, so best do this when the real work is over.

5.3.4 An extended Ring or a Ring over a Ring

The last item of interest for Polynomial Rings is actually just a though experiment. At the moment we are trying to create the structure $R[X]$. We are just about done with it and [Jalgebra] already has all the work done there.

We will get a bit ahead of ourselves now. So the way a Primepower Field will work is the following. We give it an irreducible (in $\mathbb{F}_p$) polynomial f of degree q and it simply realizes the structure $R[X]/(f)$. The way this is implemented however it could work with any kind of polynomial not just the irreducible ones. Now imagine we had an implementation of $\mathbb{Q}$, which should also be possible with a lot of help from the Integers. Then we could already realize all structures of the form $\mathbb{Q}[X]/(f)$.

'Imagine' we took $f = X^2 + 1$, then we would get something like $\mathbb{Q}[i]$. In fact we could attach all the interesting reals we wanted to our rationals.

Well not all of course, π and e for example would still need us to factor by an infinite series not a polynomial (which doesn't work), but still it would be very nice.

But why stop there, we could create $\mathbb{Q}[X]/(f)$ and then form the Rolynomial Ring over that and factor by another polynomial to get another radical extension in some cases. We could create $\mathbb{Q}[i, \sqrt{2}]$ for example in this fashion. This could really be a future project on [Jalgebra]. A nice way to get some real and even complex numbers implemented.

5.4 Primepower Fields

This section about Primepower Fields will be a lot like the one we just had about Prime Fields (section 5.2). They are implemented in a very similar way, one over the Integers and the other over a Polynomial Ring (as we have also discussed in section 4.5). We even face almost the same problems here and the solutions are quite similar sometimes.

5.4.1 Checking an Irreducible Polynomial

This is done at the moment with the most basic method. It comes directly from the definition. A polynom f is irreducible if and only if

$f = a \cdot b \Rightarrow a$ is a unit or b is a unit

We can simply check if we can find a proper divisor. The divisor polynomial has to have a lower degree then f to be a proper divisor. It also has to have degree greater then 0 otherwise it's just a unit or zero. By the degree formula we can do even better. If f has a proper divisor a, then

$f = a \cdot b$
$\deg(f) = \deg(a) + \deg(b)$

So the degrees of the two divisors add up to the degree of f. Which tells us to find any divisor of f we only have to check on the polynomials of degree d such that $1 \leq d \leq \lfloor \deg(f)/2 \rfloor$.

In terms of degree that is as much as we can tell so far. But we can also make a restriction on the polynomials a we are trying to divide by. It suffices to try the monics, i.e. the ones with leading coefficient one. This is because multiplying with the inverse of the leading coefficient, which is a unit, will not change the divisibility of f by a.

So to see whether a polynomial f is reducible or not we check all the monic polynomials of degree d, where $1 \leq d \leq \lfloor \deg(f)/2 \rfloor$. If all the remainders of these divisions are non-zero we can say that f is irreducible.

5.4.2 Representatives Revival

Before we start working on this make sure you have read the section about representatives in Prime Fields (section 5.2.2), as we will model these Primepower Field representatives after them.

Let $\mathbb{F}_{p^q} = \mathbb{F}_p[X]/(f)$ be the field we are working over. Then we could choose the following set to represent the elements.

$$S = \left\{ a_0 + a_1 X + a_2 X^2 + \cdots + a_{q-1} X^{q-1} : (\forall\ i) a_i \in \mathbb{F}_p \right\}$$

These elements suffice because of the following argument. The irreducible polynomial used to generate this field has degree q. In this modulo structure it is equal to 0 so we can rewrite this to get an expression for X^q in terms of Xes of lower degree.

$$f(X) = a_0 + a_1 X + a_2 X^2 + \cdots + a_{q-1} X^{q-1} + a_q X^q = 0$$
$$-a_q X^q = a_0 + a_1 X + a_2 X^2 + \cdots + a_{q-1} X^{q-1}$$

$$X^q = -a_q^{-1} \cdot \left(a_0 + a_1 X + a_2 X^2 + \cdots + a_{q-1} X^{q-1} \right)$$

So this works fine and it allows us to say that every coset in the Primepowerfield has exactly one representative of degree strictly less then q. So S could indeed represent all these cosets. However there is again a much better way.

We will in a way still be using S in the way it is defined up there, but we will not write the elements as polynomials, but integers. We will use the following nice function.

$$\phi : \mathbb{F}_p[X] \to \mathbb{Z} : f(X) \mapsto f^*(p)$$

where f^* is an element of $\mathbb{Z}[X]$ whose coefficients are the Prime Field representatives of the corresponding coefficients of f. We will see in the next section that this function is invertible and thus bijective (section 5.4.3). Let's make this clearer with an example.

Let $\mathbb{F}_{3^3} = \mathbb{F}_3[X]/(f)$ be the field we are working with. Let us take an element a of this set.

$$a(X) = 5 + 7X + 2X^2 + X^4$$

The problem is that this notation is not quite accurate because this element in the field is a coset as well as all it's coefficients. Let's write this out clearly.

$$a(X) = (5 + 3\mathbb{Z}) + (7 + 3\mathbb{Z})X + (2 + 3\mathbb{Z})X^2 + (1 + 3\mathbb{Z})X^3 + f(X)\mathbb{F}_3[X]$$

So this is much more confusing and therefore not used, but it is more accurate and will help us to see what we are doing. Let us start by getting the degree below $q = 3$. Well to do that we have to know what f our irreducible polynomial really looks like.

$$f(X) = 1 + 2X + X^3$$

In our field this will be in the coset of 0 so we may set it equal to zero and solve for X^3.

$$X^3 = -1 - 2X$$

So we can now replace our X^3 by $-1 - 2X$ within a to get the same $f(X)\mathbb{F}_3[X]$-coset looking nicer.

$$a(X) = (5+3\mathbb{Z})+(7+3\mathbb{Z})X+(2+3\mathbb{Z})X^2+(1+3\mathbb{Z})(-1-2X)+f(X)\mathbb{F}_3[X]$$
$$a(X) = (4 + 3\mathbb{Z}) + (5 + 3\mathbb{Z})X + (2 + 3\mathbb{Z})X^2 + f(X)\mathbb{F}_3[X]$$

Now we change the look of the coefficient by choosing the first summand to be the representative of the Prime Field element it was. Notice this does not change the $3\mathbb{Z}$-coset.

$$a(X) = (1 + 3\mathbb{Z}) + (-1 + 3\mathbb{Z})X + (-1 + 3\mathbb{Z})X^2 + f(X)\mathbb{F}_3[X]$$

This is what we wanted! Now to get the correct representative of a we just forget all the cosets again and evaluate it at $p = 3$.

$$a^*(3) = 1 - 3 - 9 = -11$$

So how do we let [Jalgebra] do this procedure for us. Well if we have the right polynomials in our set S, then all it has to do is evaluate at p. In fact by the way [Jalgebra] is written it will always have the right coefficients to begin with. We wrote the Prime Field such that it really calculates in $\mathbb{Z}$ with the correct representatives and then converts the result back to the correct representative.

So we get this for free and it turns out that we get the other bit as well. We have already programmed the *remainder division* for arbitrary Euclidean Rings because we needed it to implement the `modulo` and `div` methods on a Polynomial Ring. Be sure to read the proof appendix if you want to see that a Polynomial Ring over a field really is euclidean (section A.2).

We can use this remainder division to find our polynomials of degree less then q in this way. Remember we had our element a and the irreducible polynomial f. In the Euclidean Ring $\mathbb{F}_p[X]$ there exist by definition polynomials q and r such that.

$a = q \cdot f + r$, with $\deg(r) < \deg(f)$

In $\mathbb{F}_p[X]/(f)$ the first summand on the RHS is the same as 0 so r and a are in the same coset. This r is just what we wanted and we get it is the *remainder* of the division a/f.

Let see how nice these representatives are by listing them for a small field like $\mathbb{F}_{3^2}$.

Elements of S	**Representative**
$-X-1$	-4
$-X$	-3
$-X+1$	-2
-1	-1
0	0
1	1
$X-1$	2
X	3
$X+1$	4

Notice that polynomials with a negative leading coefficient become negative integers. The elements of the ground field (which are already the right representatives in their respective field) remain unchanged. Our representatives 'again' stay between $\pm(p^q-1)/2$ except for the case $p = 2$ that is still special. And this gives us a nice integer order on $\mathbb{F}_p[X]$.

A lot of nice properties and all came from this ϕ, but as much as one might suspect it ϕ is not a homomorphism.

5.4.3 A non-existent Homomorphism

We are discussing in this section the aforementioned function

$\phi : \mathbb{F}_p[X] \to \mathbb{Z} : f(X) \mapsto f^*(p)$

Let us start by looking at it's inverse, which was not given yet, but which exists and is also interesting.

$\psi : \mathbb{Z} \to \mathbb{F}_p[X]$

Let k be an integer. Then it has a p-ary representation, where the digits are between $\pm(p-1)/2$ for $p > 2$ and between 0 and 1 for $p = 2$, i.e. they are the representatives we chose earlier for $\mathbb{F}_p$. Let this representation be

$k = (d_n d_{n-1} \dots d_1 d_0)_p$

ψ maps this integer to the polynomial

$\psi(k) = f_k(X) = d_0 + d_1 X + \dots + d_{n-1} X^{n-1} + d_n X^n$

Note that the degree of f_k can easily be determined.

$\deg(f_k) = \lfloor \log_p(k) \rfloor$

Check that ψ is indeed the inverse function of ϕ and that we can use it to get bijectively from the set of integers $Z = \{-(p^q-1)/2, \dots, 0, \dots, (p^q-1)/2\}$ to our set S of representatives from $\mathbb{F}_{p^q}$ (see the last section 5.4.2), given that $p > 3$ and $q \geq 0$.

ψ is also implemented within [Jalgebra] and all it does to get the coefficients it needs to make up f_k is a looped remainder division of k with p in order to get the p-ary representation. We will not go into this in detail as the idea of this chapter was for something else.

Observe that with the mapping ϕ for $p = 2$. Let's run it on some irreducible polynomials.

Degree	**Irreducible Polynomial** f	$\phi(f)$
1	$X + 1$	3
2	$X^2 + X + 1$	7
3	$X^3 + X + 1$	11
3	$X^3 + X^2 + 1$	13
4	$X^4 + X + 1$	19
4	$X^4 + X^3 + 1$	25
4	$X^4 + X^3 + X^2 + X + 1$	31

So these are a lot of primes and we seem to have an error with the 25. Of course we also missed lots of primes, but it is still interesting. If we go on to the next degree we see there are more multiples of 5 and more misses.

Degree	**Irreducible Polynomial** f	$\phi(f)$
5	$X^5 + X^2 + 1$	35
5	$X^5 + X^3 + 1$	41
5	$X^5 + X^3 + X^2 + X + 1$	45
5	$X^5 + X^4 + X^2 + X + 1$	55
5	$X^5 + X^4 + X^3 + X + 1$	60
5	$X^5 + X^4 + X^3 + X^2 + 1$	61

Still the images of ϕ that are not divisible by 5 are prime. What could that mean? Well let us assume that a function $\Phi : \mathbb{F}_p[X] \to \mathbb{Z}$ was a multiplicative homomorphism, so $(\forall\ a, b \in \mathbb{F}_p[X])\ \ \Phi(a \cdot b) = \Phi(a) \cdot \Phi(b)$. Then the images of the irreducible polynomials would have to be the prime numbers. Also the image of $\mathbb{F}_p^\times$ would have to be $\{\pm 1\}$, as ring-units have to be mapped to ring-units.

The unit observation would only work for ϕ with $p = 2$ or $p = 3$. So for all other p ϕ will not be a homomorphism. For $p = 2$ we get a bijection to the monoid $(\mathbb{N}_0, \cdot)$, which would still mean that the images of irreducible polynomials have to be primes. But the table we did on the last page already shows that ϕ is no multiplicative homomorphism for $p = 2$. However it got close, except for the screw-ups with 5.

We have shown earlier that for $p = 3$ the mapping ϕ is a bijection. However if we observe the first few irreducible polynomials it becomes apparent, that it is not as good as the $p = 2$ case. We will only observe the monics (leading coefficient is one) here, as it is clear that if p is prime so is $-p$ and if a is irreducible so is $-a$ within $\mathbb{F}_3[X]$.

Degree	Irreducible Polynomial f	$\phi(f)$
1	$X-1$	2
1	$X+1$	4
2	X^2-X-1	5
2	X^2+1	10
2	X^2+X-1	11
3	X^3-X^2-X-1	14
3	X^3-X^2+1	19
3	X^3-X^2+X+1	22
3	X^3-X-1	23
3	X^3-X+1	25
3	X^3+X^2-X+1	34
3	X^3+X^2-1	35
3	X^3+X^2+X-1	38

So this looks a lot worse. We have to leave out the multiples of 2 and the multiples of 5 to get only primes.

So ϕ is just not a homomorphism and we cannot distinguish a rule when the number we are going to get from a polynomial will be prime for the cases $p = 2$ and $p = 3$

5.4.4 Easy Inverses

The work of this section is actually done in the proof appendix. There we prove, that every Polynomial Ring over a Field is indeed a Euclidean Ring with the degree function as it's norm (see section A.2).

This means for the inverses in Primepower Fields we can use the same Euclid inverse idea that was explained in detail for the Prime Field case (section 5.2.4). Note however, that the coefficients, i.e. the inverse we calculate here will naturally be polynomials and not a number. In fact in all places where the Prime Field Euclid used integers we now have polynomials and the absolute value norm is replaced by the degree norm, but that is all.

There is one more subtlety to this. The greatest common divisor of two coprime polynomials will not necessarily be ± 1 but it will be a Ring unit, i.e. invertible in $\mathbb{F}_p[X]$. The Ring units of $\mathbb{F}_p[X]$ are the invertible elements of $\mathbb{F}_p$ so everything of the ground field except zero.

So let f, g be coprime in $\mathbb{F}_p[X]$, meaning there is no polynomial of degree 1 or higher that divides both f and g. Let the unit $u = \gcd(f, g)$. We do Euclids algorithm and calculate the polynomial-coefficients a, b such that

$$u = f \cdot a + g \cdot b$$

Let's now assume g was our irreducible (in $\mathbb{F}_p[X]$) polynomial of degree q. This is coprime with all non-unit, non-zero polynomials of lower degree anyway. Then the coefficient a that we calculated is almost the inverse of f. In $\mathbb{F}_p[X]/(g)$ the equation above becomes this.

$$u = f \cdot a$$

But since u is a unit in $\mathbb{F}_p[X]$ it is invertible and so the following works.

$$1 = f \cdot a \cdot u^{-1}$$

This means the inverse of f in $\mathbb{F}_p[X]/(g)$ is $a \cdot u^{-1}$, where u^{-1} is just an integer and a is a polynomial.

6

Conclusion

6.1 The End

This section concludes the Jalgebra project. We have seen that implementing our mathematical ideas in java is possible but not trivial. We have to understand though that the things we chose here, were fitted for a computer to begin with. We deliberately implemented *finite* fields. We should ask ourselves now is the framework, that is outlined by the Three Way System and Regulation (Sections 2 and 4) good enough to work for more complicated mathematics.

6.1.1 Finite vs. Infinite

The simplest countably infinite field, the rationals $\mathbb{Q}$ could be added easily, since we already have the countable integers and can just take two of those to make a rational $\frac{p}{q}$. However we already cheated a bit on these integers, since we implemented them on based on javas `int`s, which are very finite. We would have to rewrite our `Integers` using javas more complicated arbitrary-precision '`BigIntegers`' instead of the simple data type `int`. In order to do this, we would have to give up our timesaving-immutably regulation though, as `BigIntegers` are immutable by design.

To implement an uncountable set, like $\mathbb{R}$ would be much more complicated. Since the computer cannot really say whether an Object you give it is within this uncountable set. So already the `in()`-method of the `Set`-interface would cause trouble. We could implement sequences of rationals, but could not get a concept like limits fully working, because we could not check for *all* $\epsilon > 0$. Nevermind finding a rational sequence for a given real such that it converges to this number. So perhaps this is not a direction we should take.

In general, when we do go beyond finite mathematics, we have to make approximations and keep track of how precise we need our results to be. But

finite mathematics can be really interesting too and there is a lot more that would work in our jalgebra environment. Such as graphs, and generally some coding theory, which uses vectorspaces over finite fields all the time. A finite dimensional vetorspace is easily do-able.

If we wanted to do more algebra, we could implement some finite group theoretical algorithms and for example the dihedral groups or the permutation and alternating groups using our integers. These should all be possible to implement and some even simple. Afterwards we could try to make a link back to the fields we already have with the fundamental theory of Galois.

So finite mathematics still has lots of challenges in store for the willing programmer.

6.1.2 Diophantine Equations

A good example for how different even the countable integers are from the finite fields we have implemented are diophantine equations. Take for example the following one:

$$x^2y^3 + 2xy^2 - 3 = 0$$

Were x and y in this equation an element of any finite field $\mathbb{F}_{p^q}$, all solutions could simply be found by a computer trying out all the values that x and y can take. This will take p^{2q} steps of inserting the values and checking whether we get a zero. But after these potentially very many steps we know for sure everything about this equation in the specific field we chose. In particular we know which values solve the equation and how many different solutions there are.

If on the other hand x and y where integers, then not much is known about these equations. We don't know if there are infinitely many solutions, and we cannot simply write a program, that tries out values and is guaranteed to find something.

What we can do however at this point, is introduce multiple threads to our programming. Should the need to solve an equation like this for integers x and y arise we start a thread that goes through the elements of $\mathbb{Z} \times \mathbb{Z}$ in a sensible way.

The thread would keep trying out integers and would never be guaranteed to finish. But all the while our program goes on and could do other things. Then if the thread does find something the results could be reported back to the main program and the user could decide whether to search on or

give up. So while we never have a guarantee to find something we can keep trying as long as we want to and the normal program can go on while we look.

This approach would of course require us to give up the mutable-regulation, since as we have mentioned before multiple threads and mutable objects are bound to cause trouble and unforeseen errors.

A mathematician who has been studying diophantine equations for a while has a much better chance of finding solutions to a given problem here then a program that we create though. After some study humans tend to get an intuition about the solvability and the forms of solutions of these equations. That's something we could not program into a computer unless a general rule were discovered of course.

6.1.3 Euclid for Humans and Computers

Interestingly there are also times when the computer has an easier time adapting to a problem then humans.

We have learned in *Section 5.2.4: Inverses via Euclid* how the Euclidean Algorithm works. Later in *Section A.2: A Polynomial Ring over a Field is Euclidean* we proved that indeed the same algorithm can be used on polynomials over fields, as these are elements of a Euclidean Ring.

Now humans usually have an easy time performing Euclids Algorithm on integers. The way you would do it when searching for inverses in a prime field. However they find it more difficult to do the same thing for a primepower field with polynomials. Since you need to do more arithmetic in each step.

For the Jalgebra package there is no difference here! The way the algorithm is implemented helps java to do it on **any** Euclidean Ring we care to implement. Be it the integers or a polynomial ring over a field or a prime field, which is itself a euclidean ring. We cannot get the computer out of concept, or confuse him by asking to do an algorithm it knows on similar structures.

That is one of the things that Jalgebra and the framework it uses is really good for, because the general statement, about using the same algorithm on similar structures, wouldn't be true for any old implementation we care to write. It is true precisely because we followed the Three-Way-System and we understood how it works!

6.1.4 A Computers View and Communal Projects

So how does mathematics look to our computer now? After some trouble we have discovered in *Section 3.3: Interface Inheritance Problem*, that Java gives us a very nice tool - delegation. We can use delegation to pretty accurately 'emulate' for the computer the axioms of mathematical structures that build upon two or more other structures, which are themselves *similar*. Remember that without delegation we could not thoroughly explain a ring, which builds upon a group and a semigroup, to the compiler without tricking - *Section 3.3.1: Solution using many Interfaces*. But even with delegation things are not as nice as they could be, because object-oriented programming in general does not make its objects fit together as smoothly as mathematical axioms do - *Section 3.3.2: Solution using Delegation*.

What we can ask ourselves now is: Are interfaces and delegation enough to explain to java all the structural axioms there are in algebra? Or even all in general? The former question is a yes for many examples, and that means our system to explain axioms to the computer works well! But of course the Three Way System can do even more. It helps many people to write *independent* work that is still *compatible*.

So could we have lots of people around the world working together without even knowing it? A big java-community working independently to implement all the different parts of algebra for the computer? If everyone of them looks at Jalgebra and this write-up, then chances are good this would work. Because as we have discussed in *Section 2: The Three Way System*, the work is *compatible by design*. Of course we would need someone to compile all the different packages that would be written into a big one, but since they are all compatible this is just a bit of copy-and-paste work.

So Jalgebra has an open ending. We have developed a solid groundwork, which many people can build upon if they wish to make computers understand mathematics better. There is also a nice side effect, because ultimately making somebody else, who is very skeptical, understand something is the best way to know that you understand it completely yourself.

A

Proofs

This appendix chapter is used as a reference for the mathematically interested reader. The proofs here stand free of the rest of the text in this project. The following proofs are neither very deep nor complicated, they are just too winding or technical for the main text. The first proof is taken from a big internet site were many proofs are collected. The link there is given as a reference ([Fermat's Little Theorem]). The second proof was done by me from the definitions.

A.1 Fermat's Little Theorem

Like the 'big' and famous Last Theorem of Fermat (only fairly recently proved by Andrew Wiles in 1995), this little brother was also left open to proof by the great man. This one however was as easy as Fermat claimed and the first proof was given by Leibniz. Then a little later Euler proved it independently and published it before Leibniz who hadn't bothered. Both of them argumented like this:

Theorem A.1.1. ***Fermat's Little Theorem***
Let p be a prime and $a \in \mathbb{Z}$, then

$$a^{p-1} \equiv_p 1 \pmod{p} \tag{A.1}$$

Proof. List the first $p-1$ positive multiples of a:

$$a,\ 2a,\ 3a,\ \ldots,\ (p-1)a$$

Note that $na \equiv_p ma \pmod{p}$ iff $n \equiv_p m \pmod{p}$, since a is coprime to p and thus invertible in $\mathbb{Z}_p$. So these $p-1$ elements must be congruent to

$$1,\ 2,\ 3,\ \ldots,\ (p-1) \text{ in some order.}$$

We form both ordered products and they must be congruent modulo p because we can rearrange the order on the LHS by commutativity of the multiplication in $\mathbb{Z}_p$.

$$a \cdot 2a \cdot 3a \cdots (p-1)a \equiv_p 1 \cdot 2 \cdot 3 \cdots (p-1) \pmod{p}$$

$$a^{p-1} \cdot (p-1)! \equiv_p (p-1)! \pmod{p}$$

Now we can divide both sides by $(p-1)!$, as this is also coprime to p, to get the result. □

A.2 A Polynomial Ring over a Field is Euclidean

Definition A.2.1. ***Euclidean Ring***
An Integral Domain R is a Euclidean Ring if and only if we can define a norm function

$$N : R \setminus 0 \to \mathbb{N}$$

with the following properties.

$(N1)$ $(\forall\ a, b \in R \setminus 0)\ \ N(a) \leq N(ab)$
$(N2)$ $(\forall\ a, b \in R \setminus 0)\,(\exists\ q, r \in R)\ \ a = b \cdot q + r$ *and* $N(r) < N(b)$ *or* $r = 0$

Note that both $(N1)$ and $(N2)$ work nicely on the Integers with the absolute value norm $N : \mathbb{Z} \setminus 0 \to \mathbb{N} : a \mapsto |a|$. So $\mathbb{Z}$ really is a Euclidean Ring just like we have implemented it.

Definition A.2.2. ***Degree of a Polynomial***
Let $f \neq 0$ be a polynomial over the Ring R, then

$$f(X) = a_0 + a_1 X + a_2 X^2 + \cdots + a_n X^n$$

for some sequence $(a_i)_{i \in \mathbb{N}} \in R^{\mathbb{N}}$ with $a_n \neq 0$. We define the degree of this polynomial to be the highest power of X with a non-zero coefficient. The zero polynomial has degree minus infinity by convention.

$$\deg(f) := n \quad \deg(0) := -\infty$$

Lemma A.2.3. *A Polynomial Ring $R[X]$ over an Integral Domain R is itself an Integral Domian.*

Proof. Let R be an Integral Domain, and $R[X]$ the Polynomial Ring over R. Take two elements f, g from $R[X] \setminus 0$. Then both f and g have a non-zero

leading coefficient. Their product p is defined as.

$$\begin{aligned} p(X) &= f(X) \cdot g(X) = \sum_{i=0}^{n} \left(a_i X^i\right) \cdot \sum_{j=0}^{m} \left(b_j X^j\right) \\ &= \sum_{k=0}^{n+m} \left(\sum_{i+j=k} a_i \cdot b_j \right) X^k \\ &= a_0 b_0 + (a_0 b_1 + a_1 b_0)\, X + \cdots + (a_{n-1} b_m + a_n b_{m-1})\, X^{n+m-1} + a_n b_m X^{n+m} \end{aligned}$$

So we see that the leading coefficient of the product p is just the product of the two leading coefficients of f and g. As these were non-zero, so is their product, since R is an Integral Domain. So p is not the zero polynomial which completes our proof. □

Theorem A.2.4. ***A Polynomial Ring over a Field is Euclidean***
Let $\mathbb{F}$ be a Field and $\mathbb{F}[X]$ the Polynomial Ring over this Field. Then $\mathbb{F}[X]$ is a Euclidean Ring and the following norm satisfies the properties (N1) and (N2).

$$N : \mathbb{F}[X] \setminus 0 \to \mathbb{N} : f(X) \mapsto \deg(f)$$

More precisely for any two element f, g of $\mathbb{F}[X] \setminus \{0\}$ these hold

$(N1)$ $N(f) \leq N(fg)$
$(N2)$ $(\exists\ q, r \in \mathbb{F}[X])$ $f = g \cdot q + r$ *and* $N(r) < N(g)$ *or* $r = 0$

Proof. By the previous Lemma A.2.3 it suffices to show that N satisfies $(N1)$ and $(N2)$ for $\mathbb{F}[X]$ to be Euclidean.

We can show $(N1)$ using the degree formula. Let f, g be in $\mathbb{F}[X] \setminus \{0\}$, then

$N(f) = \deg(f) \leq \deg(f) + \deg(g) = \deg(f \cdot g) = N(fg)$, so $(N1)$ holds.

In order to show $(N2)$ we take a closer look at f and g.

$f(X) = a_0 + a_1 X + a_2 X^2 + \cdots + a_n X^n$, so $\deg(f) = n$
$g(X) = b_0 + b_1 X + b_2 X^2 + \cdots + b_m X^m$, and $\deg(g) = m$

We first take care of some simple cases.
Assume that $\deg(g) = m = 0$, then

$$f = b_0 \cdot \underbrace{(1/b_0 \cdot f)}_{=:q} + \underbrace{0}_{=:r} \text{ and } r = 0$$

So this special case is solved and we may assume $m = \deg(g) \geq 1$ for the rest of the proof. Another special case we wish to observe is when

$\deg(f) < \deg(g)$ because then

$$f = g \cdot \underbrace{0}_{=:q} + \underbrace{f}_{=:r} \text{ and } N(r) < N(g)$$

Which solves this case and tells us we may assume that $\deg(f) \geq \deg(g)$ or equally $n \geq m$ form now on.

These were enough special cases, so we start an induction on n, the degree of f. We start with the base case $n = \deg(f) = 0$. But this case must already be solved as we assume $n \geq m \geq 1$ at this point.

Let's do the induction step then and assume we have shown $(N2)$ for all polynomials of degree less then n.

We define the polynomial h as follows.

$h := f - g \cdot \left(\frac{a_n}{b_m} X^{n-m}\right)$, this has $\deg(h) \leq n - 1$ and solves
$f = g \cdot \left(\frac{a_n}{b_m} X^{n-m}\right) + h$

As the degree of h is less then n, our induction hypothesis gives us two polynomials $q_h, r_h \in \mathbb{F}[X]$ such that

$$h = g \cdot q_h + r_h \text{ and } N(r_h) < N(g) \text{ or } r_h = 0$$

We insert this into a previous equation

$$\begin{aligned}
f &= g \cdot \left(\tfrac{a_n}{b_m} X^{n-m}\right) + h \\
f &= g \cdot \left(\tfrac{a_n}{b_m} X^{n-m}\right) + g \cdot q_h + r_h \\
f &= g \cdot \underbrace{\left(\frac{a_n}{b_m} X^{n-m} + g_h\right)}_{=:q} + \underbrace{r_h}_{=:r} \text{ and } N(r_h) = N(r) < N(g) \text{ or } r_h = r = 0
\end{aligned}$$

This proves the induction step and thus the general case for $(N2)$, which in turn finishes the whole proof. □

References

[Jalgebra] Richard Lindner, *The jalgebra package*.
Available on the web at `http://mat140.bham.ac.uk/~lindner/` or `http://havelock.esmartstudent.com/`.

[Wal91] Jim Waldo, *Controversy: The case for multiple inheritance in C++*. USENIX Computing Systems, 4(2):157-172, Spring 1991.

[AG96] Ken Arnold and James Gosling, *The Java Programming Language*. The Java Series. Addison-Wessley, 1996.

[TB98] Ewan Tempero and Robert Briddle, *Simulating Multiple Inheritance in Java*. Technical Report CS-TR-98/1. School of Mathematics and Computer Science, Victoria University of Wellington.

[MIPS] Matthias Kegelmann, *Handout 6: An Introduction to MIPS Assembler*.
Available on the web at `http://www.mathematik.tu-darmstadt.de/Math-Net/Lehrveranstaltungen/Lehrmaterial/SS2002/Introduction_to_Computer_Science_II/`

[AKS02] Manindra Agrawal, Neeraj Kayal and Nitin Saxena, *PRIMES is in P*. Department of Computer Science & Engineering, Indian Institute of Technology Kanpur, Kanpur-208016, INDIA.

[1000 Primes] The Prime Pages - A prime number fans web site.
`http://www.utm.edu/research/primes/`

[Irreducible Polynomials] Chris Lomont, *May 10th 2000 - List of irreducible polynomials that generate different finite fields*.
Available on the web at `http://math.purdue.edu/~clomont/Main.htm`

[Fermat's Little Theorem] Chris K. Caldwell, *Proof of Fermat's Little Theorem*
`http://www.utm.edu/research/primes/notes/proofs/FermatsLittleTheorem.html`