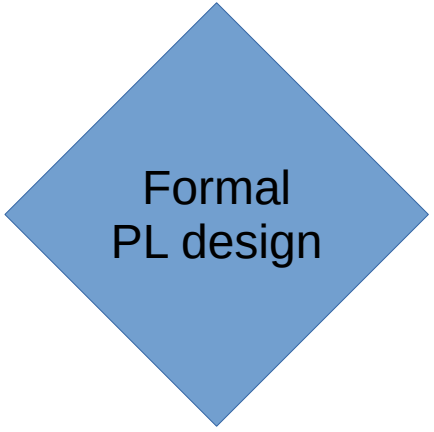
SWEN423 - Featherweight Java



Featherweight Java

- A core of Java that capture the minimal OO features. Many variations of FJ exists in literature
- To begin, we present a version with interfaces but without subclassing and casts

- We will formally define it using metarules:
  the mathematical metalanguage used by the research community
  “formal design of programming languages”

FJ Grammar, Memorize

$e ::= x \mid e_0.m(e_1..e_n) \mid \text{new } C(e_1..e_n) \mid e.f$

$cd ::= \text{class } C \text{ implements } C_1..C_n\{ Fs\ K\ Ms \}$
$\quad \mid \text{interface } C \text{ extends } C_1..C_n\{ MH_1; .. MH_k; \}$

$Fs ::= F_1..F_n$

$Ms ::= M_1..M_n$

$K ::= C(T_1\ x_1 .. T_n\ x_n)\{this.x_1=x_1; .. this.x_n=x_n;\}$

$F ::= C\ f;$

$M ::= MH\{return\ e;\}$

$MH ::= C\ m(T_1\ x_1 .. T_n\ x_n)$

FJ Grammar, Memorize

e ::= x //variable. “this”: a special kind of variable
| e_0.m(e_1..e_n) //method call. e0 is the receiver
| new C(e_1..e_n) //object instantiation
| e.f //field access

cd ::= class C implements C_1..C_n{ Fs K Ms } //class declaration
| interface C extends C_1..C_n{ MH_1; .. MH_k;} //interface declaration

K ::= C(T_1 x_1 .. T_n x_n){this.x_1=x_1; .. this.x_n=x_n;}
//canonical constructor. We can assume it to be always present

F ::= C f; //field declaration

M ::= MH{return e;} //method declaration

MH ::= C m(T_1 x_1 .. T_n x_n) //method header

FJ Grammar, Details of notation

Terminals: { } ( ) , ; . = class interface extends implements return
Non Terminals: e cd K F M MH C x m f
We assume “C x m f” to be identifiers from different syntactic categories

$e_0.m(e_1..e_n)$ //we use subscript indexes,
new $C(e_1..e_n)$ //another grammar case can reuse them
//new C(e..e) it would instead mean many repetitions of the same “e”

interface C extends $C_1..C_n\{ MH_1; .. MH_k;\}$
//note how we use 1..n before and 1..k after.
//interface C extends $C_1..C_n\{ MH_1; .. MH_n;\}$ would instead require
//the number of method headers to be the same number of the extended
//interfaces
//Also, note the use of the semicolon in “$MH_1; .. MH_k;$”:
//They are part of what the “..” repeats. The “..” is not an
//informal notation; it has precise semantic.
//For simplicity, we omit comma separators in sequences.

Example of programs in FJ

- How can we code anything? we do not have number, we do not have booleans, no statements, like if-while-for.
  No System.out.println, and.... no static methods either! How can we code like this?

- In this context we are forced to actually code in a true pure object oriented style.

- Static method can be emulated by instance methods of a class with no fields:

```
static N foo(N n1,N n2){return ..;}     //called as ClassName.foo(n1,n2)
       ==
class Foo{ N of(N n1,N n2){return ..;}} //called as new Foo(),of(n1,n2)
```

Example of programs in FJ

```
interface N{}
class Z implements N{}
class S implements N{
  N pred;S(N pred){this.pred=pred;}
  }
class Example{
  N three(){return new S(new S(new S(new Z())));}
  }
```

- (N) Peano numbers: (Z) zero and (S) successor.
- Can we write methods for those numbers?
- successor, predecessor, sum, subtraction, multiplication

```
interface N{
  N succ();




  }
class Z implements N{
  N succ(){return new S(this);}




  }
class S implements N{  N pred;  S(N pred){this.pred=pred;}
  N succ(){return new S(this);}




  }
```

```
interface N{
  N succ();
  N pred();




  }
class Z implements N{
  N succ(){return new S(this);}
  N pred(){return this.pred();}//to go in error




  }
class S implements N{  N pred;  S(N pred){this.pred=pred;}
  N succ(){return new S(this);}
  N pred(){return this.pred;}




  }
```

```
interface N{
  N succ();
  N pred();
  N add(N n); //this+n



  }
class Z implements N{
  N succ(){return new S(this);}
  N pred(){return this.pred();}//to go in error
  N add(N n){return n;}



  }
class S implements N{  N pred;  S(N pred){this.pred=pred;}
  N succ(){return new S(this);}
  N pred(){return this.pred;}
  N add(N n){return this.pred.add(n).succ();}



  }
```

```
interface N{
  N succ();
  N pred();
  N add(N n); //this+n
  N minus(N n);//this-n //inductively defined on the right argument
  N rightMinus(N n);//n-this

  }
class Z implements N{
  N succ(){return new S(this);}
  N pred(){return this.pred();}//to go in error
  N add(N n){return n;}
  N minus(N n){return n.rightMinus(this);}
  N rightMinus(N n){return n;}

  }
class S implements N{  N pred;  S(N pred){this.pred=pred;}
  N succ(){return new S(this);}
  N pred(){return this.pred;}
  N add(N n){return this.pred.add(n).succ();}
  N minus(N n){return n.rightMinus(this);}
  N rightMinus(N n){return this.pred.rightMinus(n.pred());}

  }
```

```
interface N{
  N succ();
  N pred();
  N add(N n); //this+n
  N minus(N n);//this-n
  N rightMinus(N n);//n-this
  N times(N n);//this*n
  }
class Z implements N{
  N succ(){return new S(this);}
  N pred(){return this.pred();}//to go in error
  N add(N n){return n;}
  N minus(N n){return n.rightMinus(this);}
  N rightMinus(N n){return n;}
  N times(N n){return this;}
  }
class S implements N{  N pred;  S(N pred){this.pred=pred;}
  N succ(){return new S(this);}
  N pred(){return this.pred;}
  N add(N n){return this.pred.add(n).succ();}
  N minus(N n){return n.rightMinus(this);}
  N rightMinus(N n){return this.pred.rightMinus(n.pred());}
  N times(N n){return this.pred.times(n).add(n);}}//(a+1)*b == a*b + b
  }
```

```
interface N{
  N succ();
  N pred();
  N add(N n); //this+n
  N minus(N n);//this-n
  N rightMinus(N n);//n-this
  N times(N n);//this*n
  }
class Z implements N{
  N succ(){return new S(this);}
  N pred(){return this.pred();}//to go in error
  N add(N n){return n;}
  N minus(N n){return n.rightMinus(this);}
  N rightMinus(N n){return n;}
  N times(N n){return this;}
  }
class S implements N{  N pred;  S(N pred){this.pred=pred;}
  N succ(){return new S(this);}
  N pred(){return this.pred;}
  N add(N n){return this.pred.add(n).succ();}
  N minus(N n){return n.rightMinus(this);}
  N rightMinus(N n){return this.pred.rightMinus(n.pred());}
  N times(N n){return this.pred.times(n).add(n);}}//(a+1)*b == a*b + b
  }
```

Why?? Just Why!

- Using minimal features to encode well known concepts makes us learn about the full power of those minimal features
- What is the simplest usable language?
- Do we need numbers in the language semantic or it would be sufficient to have a nicer syntax over method calls?
- In scala a + b, '+' is just a method name, and a + b == a.+(b)
  (also a c b == a.c(b), even in c is an identifier and not an operator)
- In python a + b is compiled mostly as a.__add__(b) //We will discuss __radd__ another day

- The compiler can recognize those methods and optimize the code, so it is not a performace problem.
- Advantages of having primitive numbers instead of encoding numbers?
- Advantages of encoding numbers instead of having them as primitive?

Booleans

```
interface B{
  B not();



  }
class T implements B{
  B not(){return new F();}



  }
class F implements B{
  B not(){return new T();}



  }
```

Booleans

```
interface B{
  B not();
  B and(B b);


  }
class T implements B{
  B not(){return new F();}
  B and(B b){return b;}


  }
class F implements B{
  B not(){return new T();}
  B and(B b){return this;}


  }
```

Booleans

```
interface B{
  B not();
  B and(B b);
  B or(B b);

  }
class T implements B{
  B not(){return new F();}
  B and(B b){return b;}
  B or(B b){return this;}

  }
class F implements B{
  B not(){return new T();}
  B and(B b){return this;}
  B or(B b){return b;}

  }
```

Booleans

```
interface B{
  B not();
  B and(B b);
  B or(B b);
  ProducerN thenElse(ProducerN n1,ProducerN n2);
  }
class T implements B{
  B not(){return new F();}
  B and(B b){return b;}
  B or(B b){return this;}
  ProducerN thenElse(ProducerN n1,ProducerN n2){return n1;}
  }
class F implements B{
  B not(){return new T();}
  B and(B b){return this;}
  B or(B b){return b;}
  ProducerN thenElse(ProducerN n1,ProducerN n2){return n2;}
  }
interface ProducerN{ N get(); }
```

Booleans

```
interface B{
  B not();
  B and(B b);
  B or(B b);
  ProducerN thenElse(ProducerN n1,ProducerN n2);
  }
class T implements B{
  B not(){return new F();}
  B and(B b){return b;}
  B or(B b){return this;}
  ProducerN thenElse(ProducerN n1,ProducerN n2){return n1;}
  }
class F implements B{
  B not(){return new T();}
  B and(B b){return this;}
  B or(B b){return b;}
  ProducerN thenElse(ProducerN n1,ProducerN n2){return n2;}
  }
interface ProducerN{ N get(); }
```

Numbers -> Booleans

```
interface N{ ...
  B isZero(); // this==0




  }
class Z implements N{ ...
  B isZero(){return new T();}




  }
class S implements N{  N pred;  S(N pred){this.pred=pred;} ...
  B isZero(){return new F();}




  }
```

Numbers -> Booleans

```
interface N{ ...
  B isZero(); // this==0
  B eq(N n);  // this==n



  }
class Z implements N{ ...
  B isZero(){return new T();}
  B eq(N n){return n.isZero();};



  }
class S implements N{  N pred;  S(N pred){this.pred=pred;} ...
  B isZero(){return new F();}
  B eq(N n){return this.pred.eq(n.pred);};//is it correct?



  }
```

Numbers -> Booleans

```
interface N{ ...
  B isZero(); // this==0
  B eq(N n);  // this==n
  B eqS(S n); // this==n but we know n is not zero


  }
class Z implements N{ ...
  B isZero(){return new T();}
  B eq(N n){return n.isZero();};
  B eqS(S n){return new F();};


  }
class S implements N{  N pred;  S(N pred){this.pred=pred;} ...
  B isZero(){return new F();}
  //B eq(N n){return this.pred.eq(n.pred());};//wrong
  B eq(N n){return n.eqS(this);};
  B eqS(S n){return this.pred.eq(n.pred);};//or n.pred.eq(this.pred)


  }
```

Numbers -> Booleans

```
interface N{ ...
  B isZero(); // this==0
  B eq(N n);  // this==n
  B eqS(S n); // this==n but we know n is not zero
  B geq(N n); // this>=n
  B leqS(S n);// this<=n but we know n is not zero
  }
class Z implements N{ ...
  B isZero(){return new T();}
  B eq(N n){return n.isZero();};
  B eqS(S n){return new F();};
  B geq(N n){return new T();};
  B leqS(S n){return new T();};
  }
class S implements N{  N pred;  S(N pred){this.pred=pred;} ...
  B isZero(){return new F();}
  B eq(N n){return n.eqS(this);};
  B eqS(S n){return this.pred.eq(n.pred);};//or n.pred.eq(this.pred)
  B geq(N n){return n.leqS(this);};
  B leqS(S n){return n.pred.geq(this.pred);};
  }
```

Numbers -> Booleans

```
interface N{ ...
  B isZero(); // this==0
  B eq(N n);  // this==n
  B eqS(S n); // this==n but we know n is not zero
  B geq(N n); // this>=n
  B leqS(S n);// this<=n but we know n is not zero
  }
class Z implements N{ ...
  B isZero(){return new T();}
  B eq(N n){return n.isZero();};
  B eqS(S n){return new F();};
  B geq(N n){return new T();};
  B leqS(S n){return new T();};
  }
class S implements N{  N pred;  S(N pred){this.pred=pred;} ...
  B isZero(){return new F();}
  B eq(N n){return n.eqS(this);};
  B eqS(S n){return this.pred.eq(n.pred);};//or n.pred.eq(this.pred)
  B geq(N n){return n.leqS(this);};
  B leqS(S n){return n.pred.geq(this.pred);};
  }
```

Usage

```
class Example{
  N operation(N n1,N n2){
    return n1.geq(n2).thenElse( //if(n1>n2){
      new DoAdd(n1,n2),         //  return n1+n2;}
      new DoTimes(n1,n2)        //else{return n1*n2;}
    ).get();
    }
  }

class DoAdd implements ProducerN{
  N l; N r; DoAdd(N l, N r){this.l=l;this.r=r;}
  public N get(){
    return this.l.add(this.r);
    }
  }
class DoTimes implements ProducerN{
  N l; N r; DoTimes(N l, N r){this.l=l;this.r=r;}
  public N get(){
    return this.l.times(this.r);
    }
  }
```

```
n1.geq(n2).thenElse(
  ()->n1.add(n2),
  ()->n1.times(n2)
  ).get();
//you see the lambdas if you
//squint just a little
```

Stack

```
interface Stack{
  N top();
  Stack pop();



  }
class EmptyStack implements Stack{
  N top(){return this.top();}
  Stack pop(){return this.pop();}



  }
class NonEmptyStack implements Stack{
  N t; Stack p; NonEmptyStack(N t,Stack p){this.t=t;this.p=p;}
  N top(){return t;}
  Stack pop(){return p;}



  }
```

Stack

```
interface Stack{
  N top();
  Stack pop();
  B isEmpty();


  }
class EmptyStack implements Stack{
  N top(){return this.top();}
  Stack pop(){return this.pop();}
  B isEmpty(){return new T();}


  }
class NonEmptyStack implements Stack{
  N t; Stack p; NonEmptyStack(N t,Stack p){this.t=t;this.p=p;}
  N top(){return t;}
  Stack pop(){return p;}
  B isEmpty(){return new F();}


  }
```

Stack

```
interface Stack{
  N top();
  Stack pop();
  B isEmpty();
  Stack push(N n);

  }
class EmptyStack implements Stack{
  N top(){return this.top();}
  Stack pop(){return this.pop();}
  B isEmpty(){return new T();}
  Stack push(N n){return new NonEmptyStack(n,this);}

  }
class NonEmptyStack implements Stack{
  N t; Stack p; NonEmptyStack(N t,Stack p){this.t=t;this.p=p;}
  N top(){return t;}
  Stack pop(){return p;}
  B isEmpty(){return new F();}
  Stack push(N n){return new NonEmptyStack(n,this);}

  }
```

Stack

```
interface Stack{
  N top();
  Stack pop();
  B isEmpty();
  Stack push(N n);
  Stack concat(Stack s);
  }
class EmptyStack implements Stack{
  N top(){return this.top();}
  Stack pop(){return this.pop();}
  B isEmpty(){return new T();}
  Stack push(N n){return new NonEmptyStack(n,this);}
  Stack concat(Stack s){return s;}
  }
class NonEmptyStack implements Stack{
  N t; Stack p; NonEmptyStack(N t,Stack p){this.t=t;this.p=p;}
  N top(){return t;}
  Stack pop(){return p;}
  B isEmpty(){return new F();}
  Stack push(N n){return new NonEmptyStack(n,this);}
  Stack concat(Stack s){return this.pop().concat(s).push(this.t);}
  }
```

Stack

```
interface Stack{
  N top();
  Stack pop();
  B isEmpty();
  Stack push(N n);
  Stack concat(Stack s);
  }
class EmptyStack implements Stack{
  N top(){return this.top();}
  Stack pop(){return this.pop();}
  B isEmpty(){return new T();}
  Stack push(N n){return new NonEmptyStack(n,this);}
  Stack concat(Stack s){return s;}
  }
class NonEmptyStack implements Stack{
  N t; Stack p; NonEmptyStack(N t,Stack p){this.t=t;this.p=p;}
  N top(){return t;}
  Stack pop(){return p;}
  B isEmpty(){return new F();}
  Stack push(N n){return new NonEmptyStack(n,this);}
  Stack concat(Stack s){return this.pop().concat(s).push(this.t);}
  }
```

Stack ~ Number

```
interface Stack{
  N top();
  Stack pop();            //N.pred()
  B isEmpty();            //N.isZero()
  Stack push(N n);        //N.succ()
  Stack concat(Stack s); //N.add()
  }
class EmptyStack implements Stack{
  N top(){return this.top();}
  Stack pop(){return this.pop();}
  B isEmpty(){return new T();}
  Stack push(N n){return new NonEmptyStack(n,this);}
  Stack concat(Stack s){return s;}
  }
class NonEmptyStack implements Stack{
  N t; Stack p; NonEmptyStack(N t,Stack p){this.t=t;this.p=p;}
  N top(){return t;}
  Stack pop(){return p;}
  B isEmpty(){return new F();}
  Stack push(N n){return new NonEmptyStack(n,this);}
  Stack concat(Stack s){return this.pop().concat(s).push(this.t);}
  }
```

Foundations

- It is not crazy to state that the presented implementation of peano numbers is the logic foundation
  behind most datastructures

- In the same way,
  It is not crazy to state that the presented implementation of booleans is the logic foundation
  behind most programming patterns