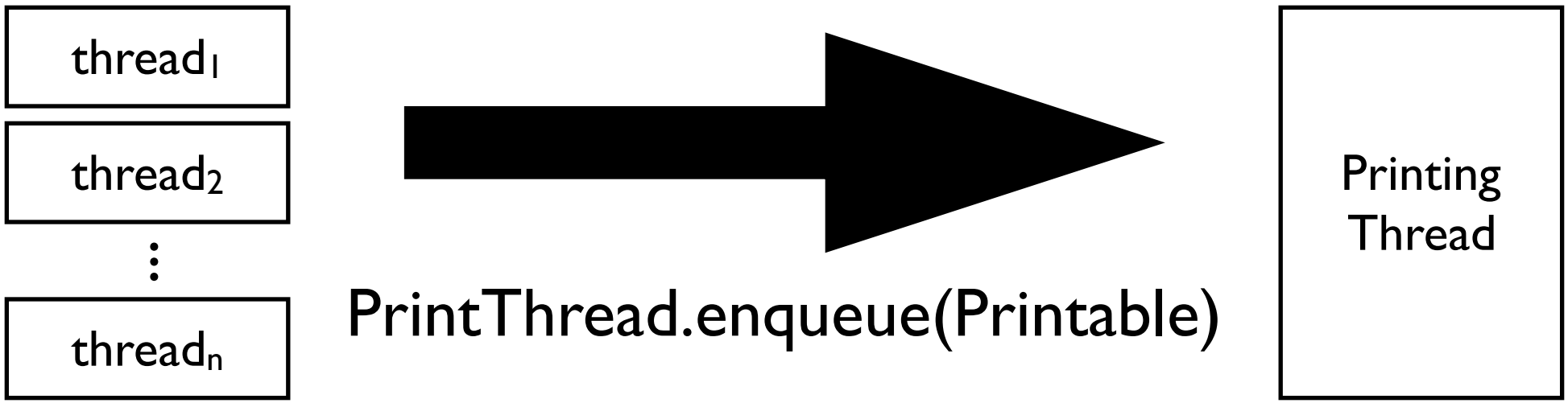
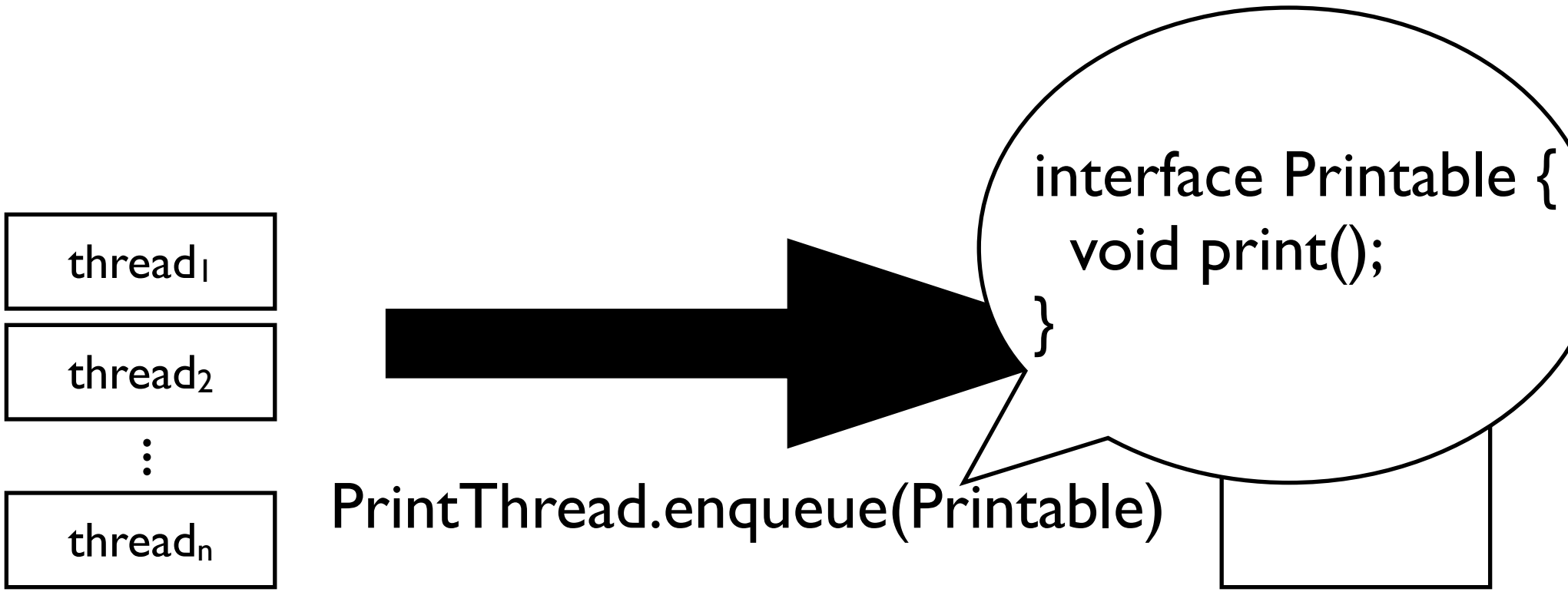
Statically Scoped Object Adaptation with Expanders

Alessandro Warth, Milan Stanojevic, Todd Millstein
UCLA

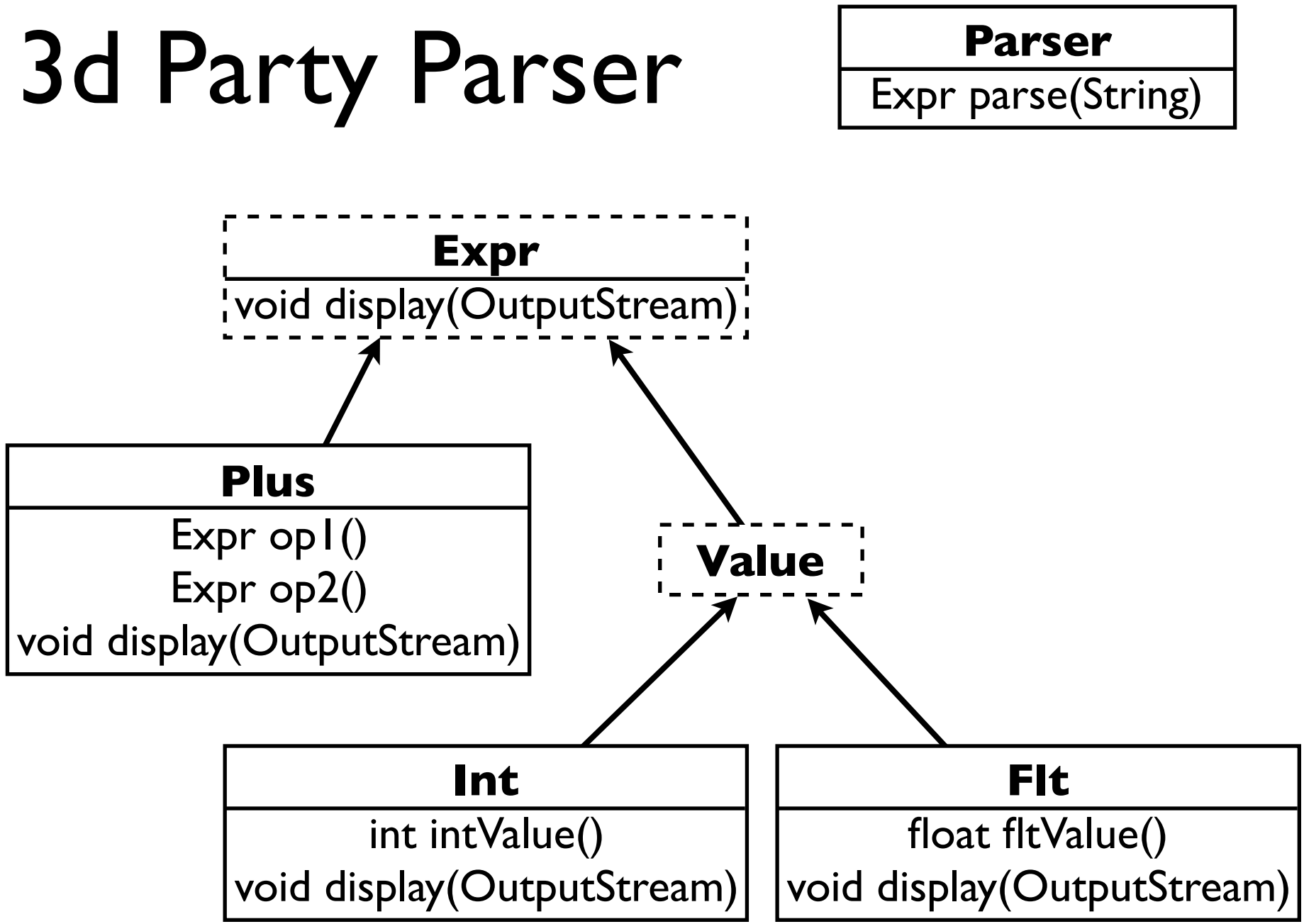
A Fancy Multithreaded Number Crunching App

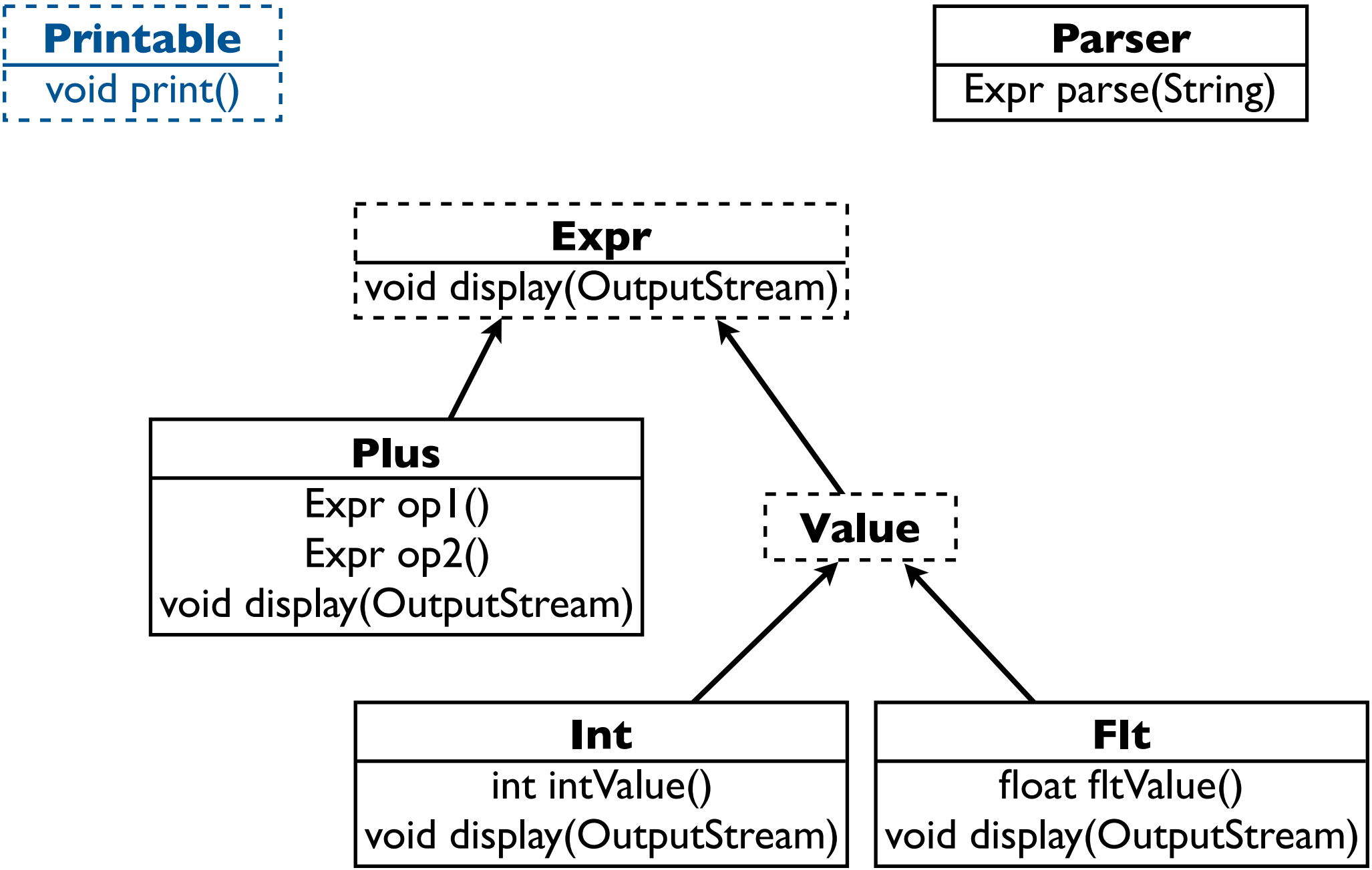


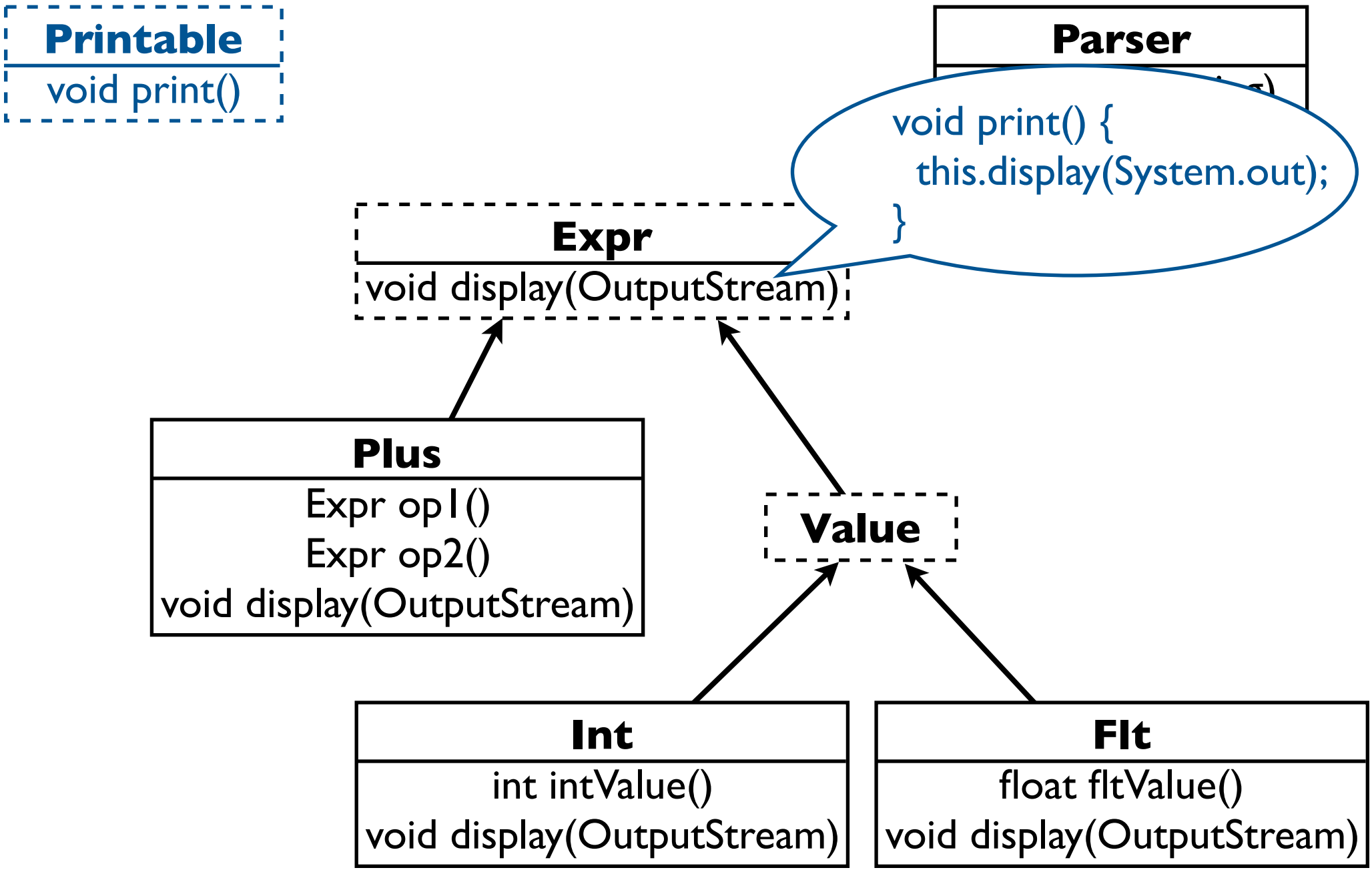
A Fancy Multithreaded Number Crunching App

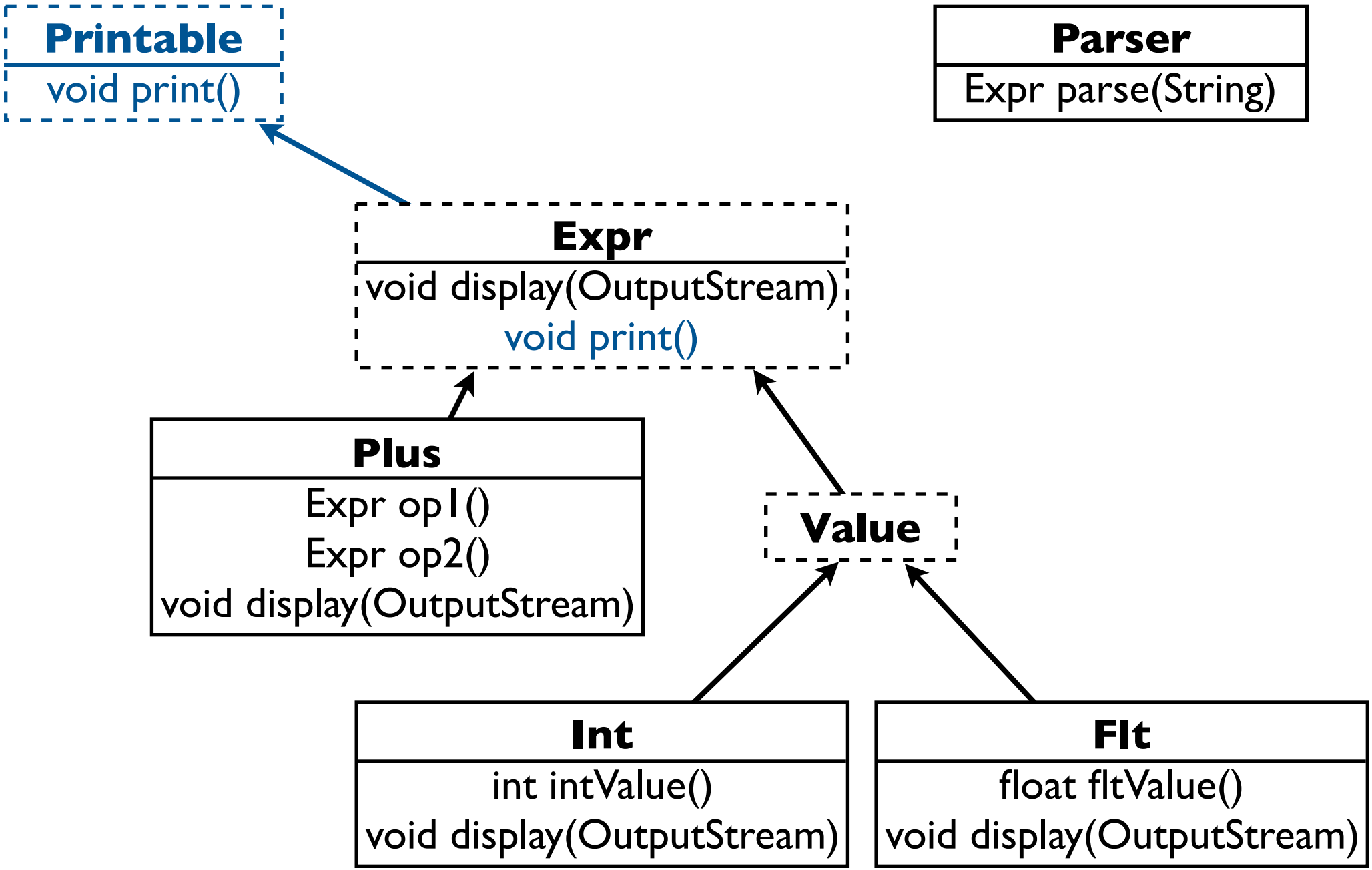


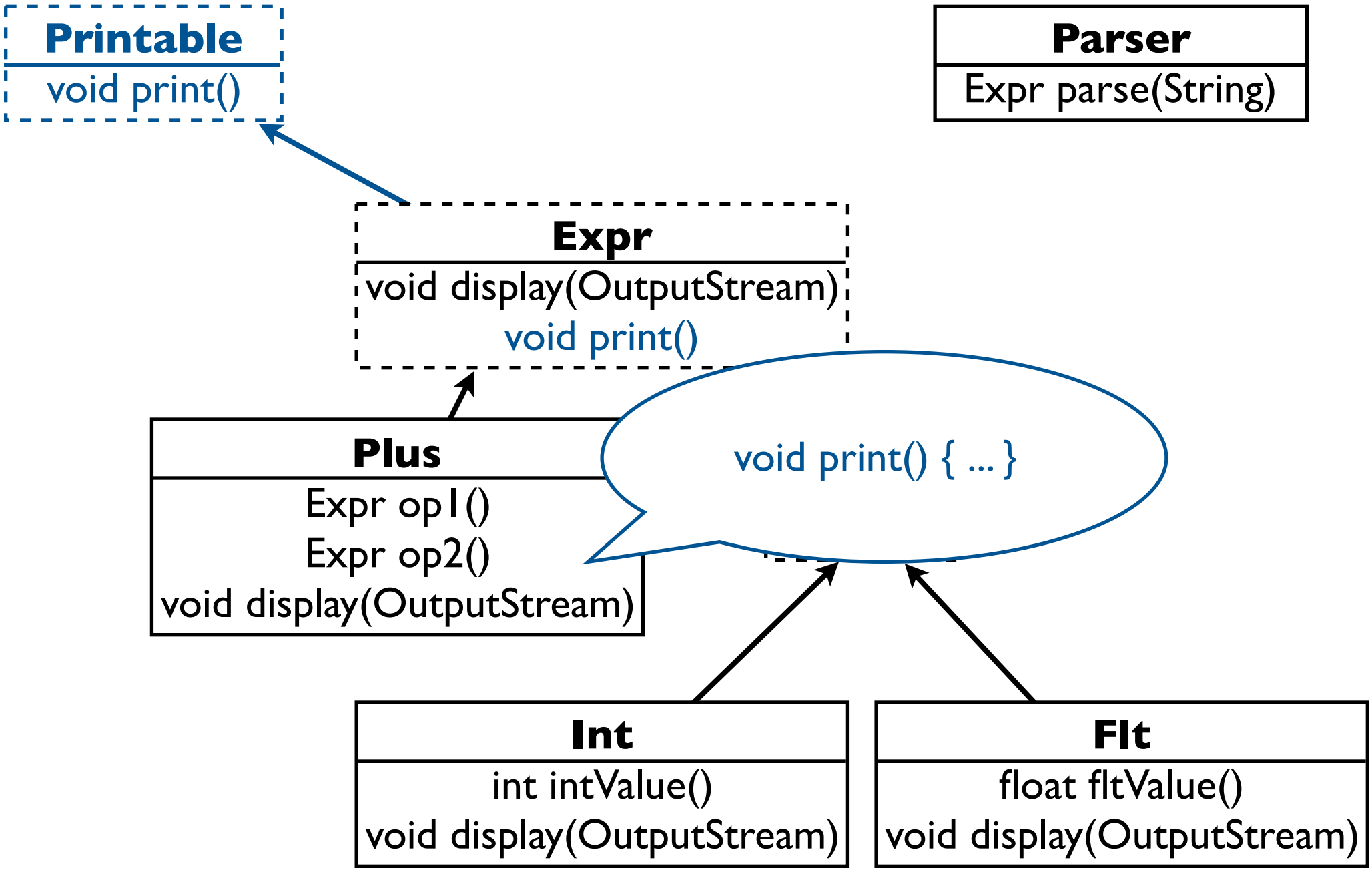
A 3d Party Parser

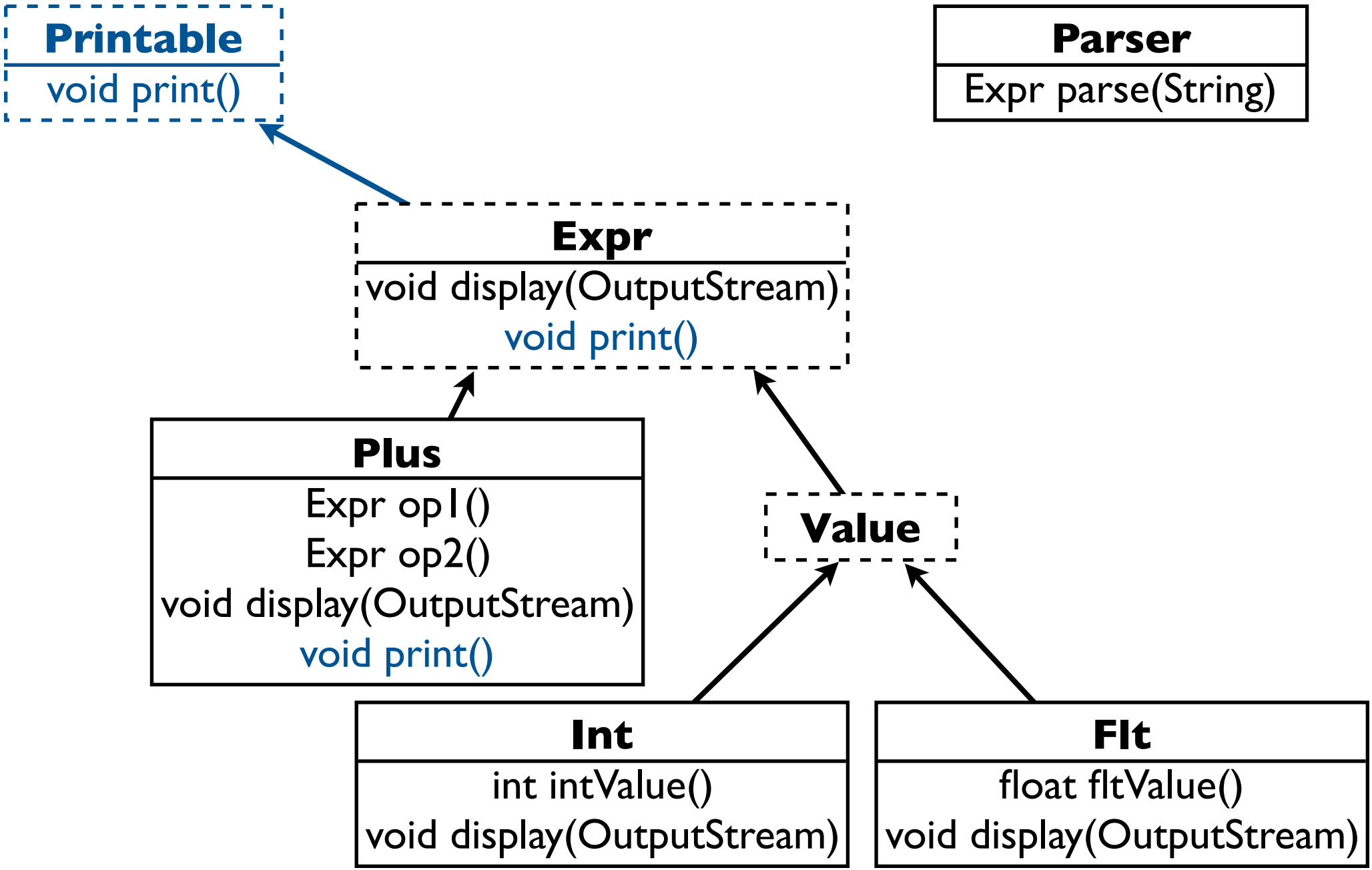



Printable
void print()
Parser
Expr parse(String)
Expr
void display(OutputStream)
Plus
Expr op1()
Expr op2()
void display(OutputStream)
Value
Int
int intValue()
void display(OutputStream)
Flt
float fltValue()
void display(OutputStream)


Printable
void print()
Parser
void print() {
this.display(System.out);
}
Expr
void display(OutputStream)
Plus
Expr op1()
Expr op2()
void display(OutputStream)
Value
Int
int intValue()
void display(OutputStream)
Flt
float fltValue()
void display(OutputStream)

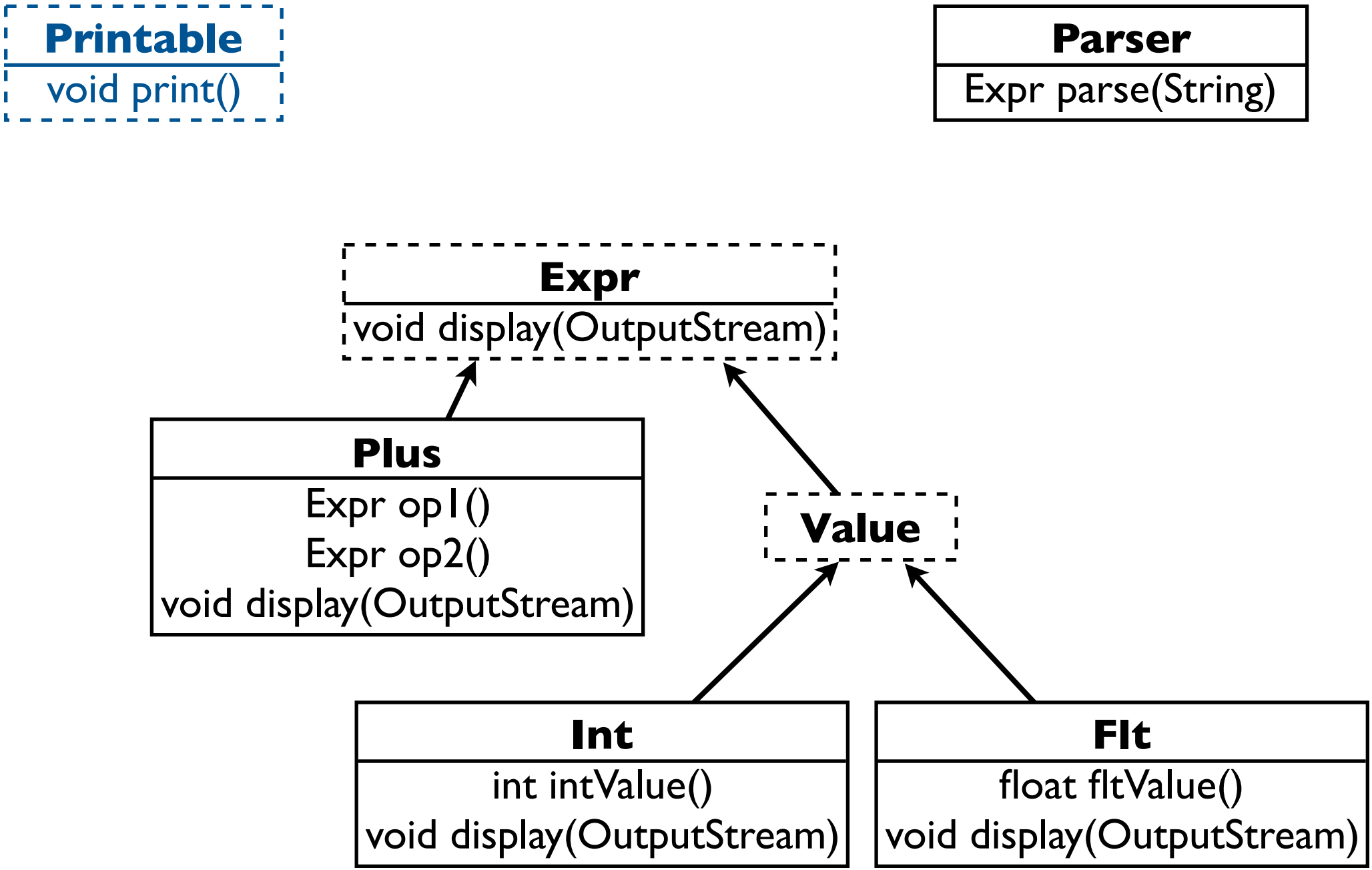

Printable
void print()
Parser
Expr parse(String)
Expr
void display(OutputStream)
void print()
Plus
Expr op1()
Expr op2()
void display(OutputStream)
Value
Int
int intValue()
void display(OutputStream)
Flt
float fltValue()
void display(OutputStream)

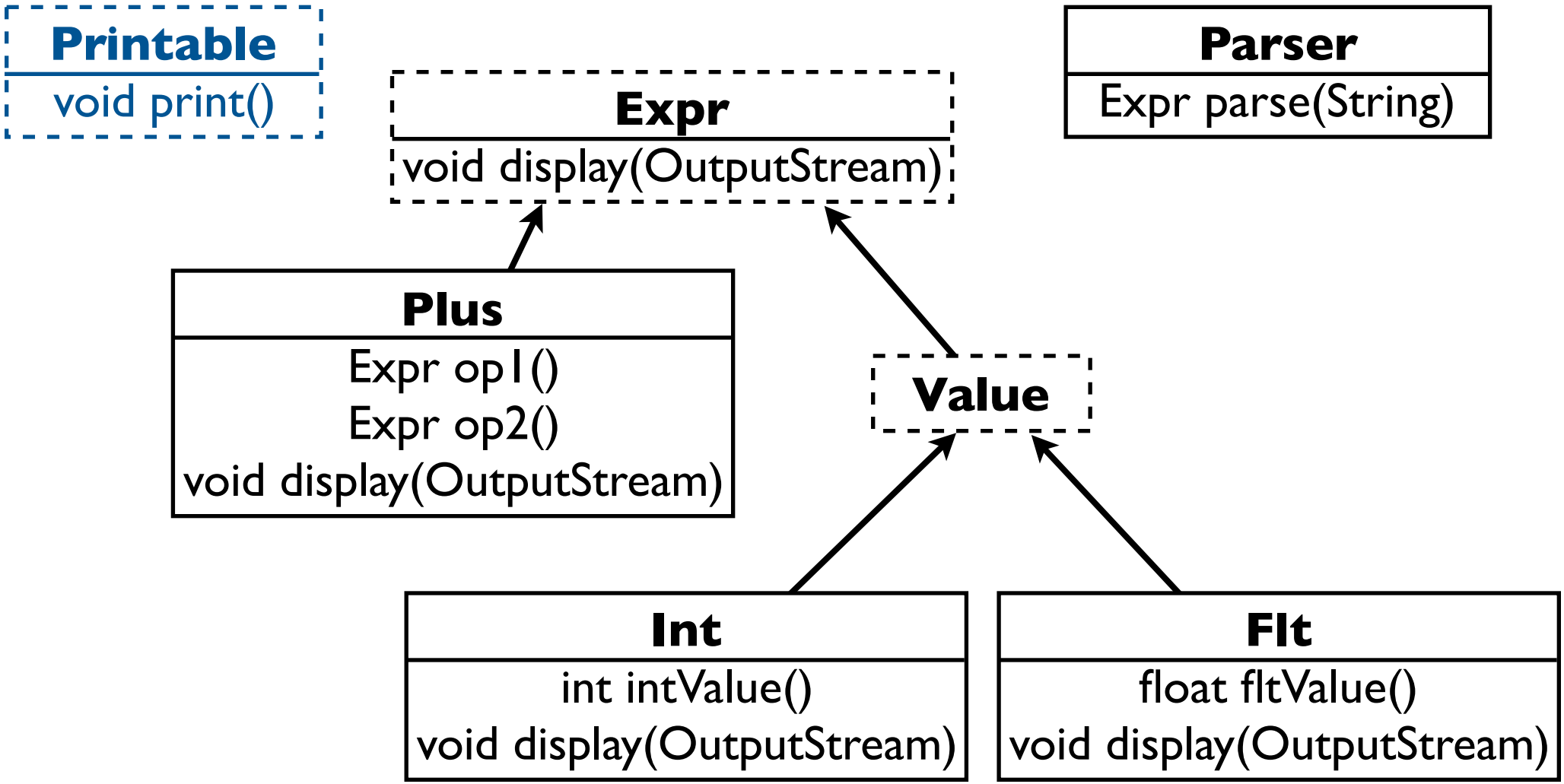

Printable
void print()
Parser
Expr parse(String)
Expr
void display(OutputStream)
void print()
Plus
Expr op1()
Expr op2()
void display(OutputStream)
void print() { ... }
Int
int intValue()
void display(OutputStream)
Flt
float fltValue()
void display(OutputStream)

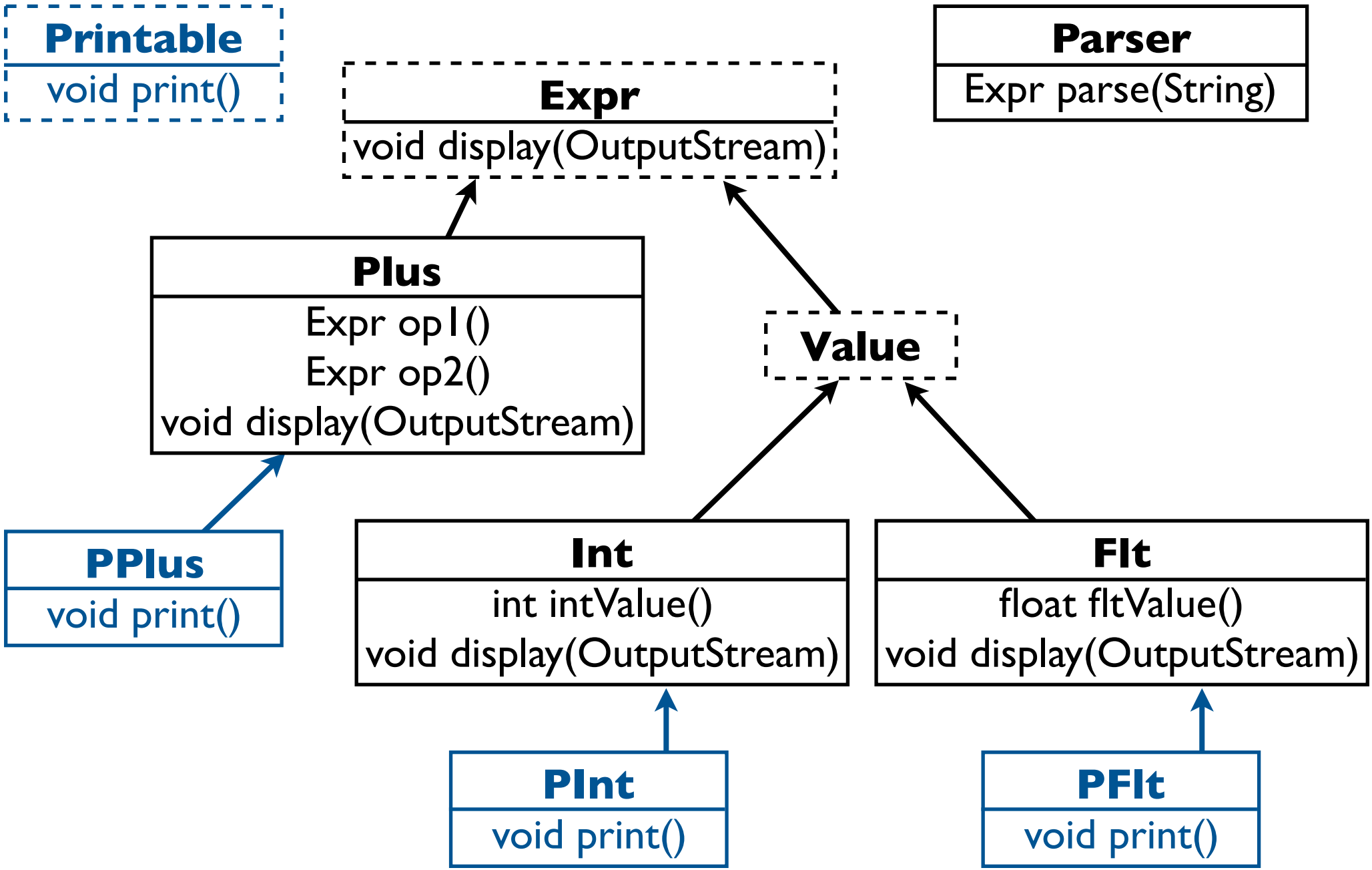

Printable
void print()
Parser
Expr parse(String)
Expr
void display(OutputStream)
void print()
Plus
Expr op1()
Expr op2()
void display(OutputStream)
void print()
Value
Int
int intValue()
void display(OutputStream)
Flt
float fltValue()
void display(OutputStream)

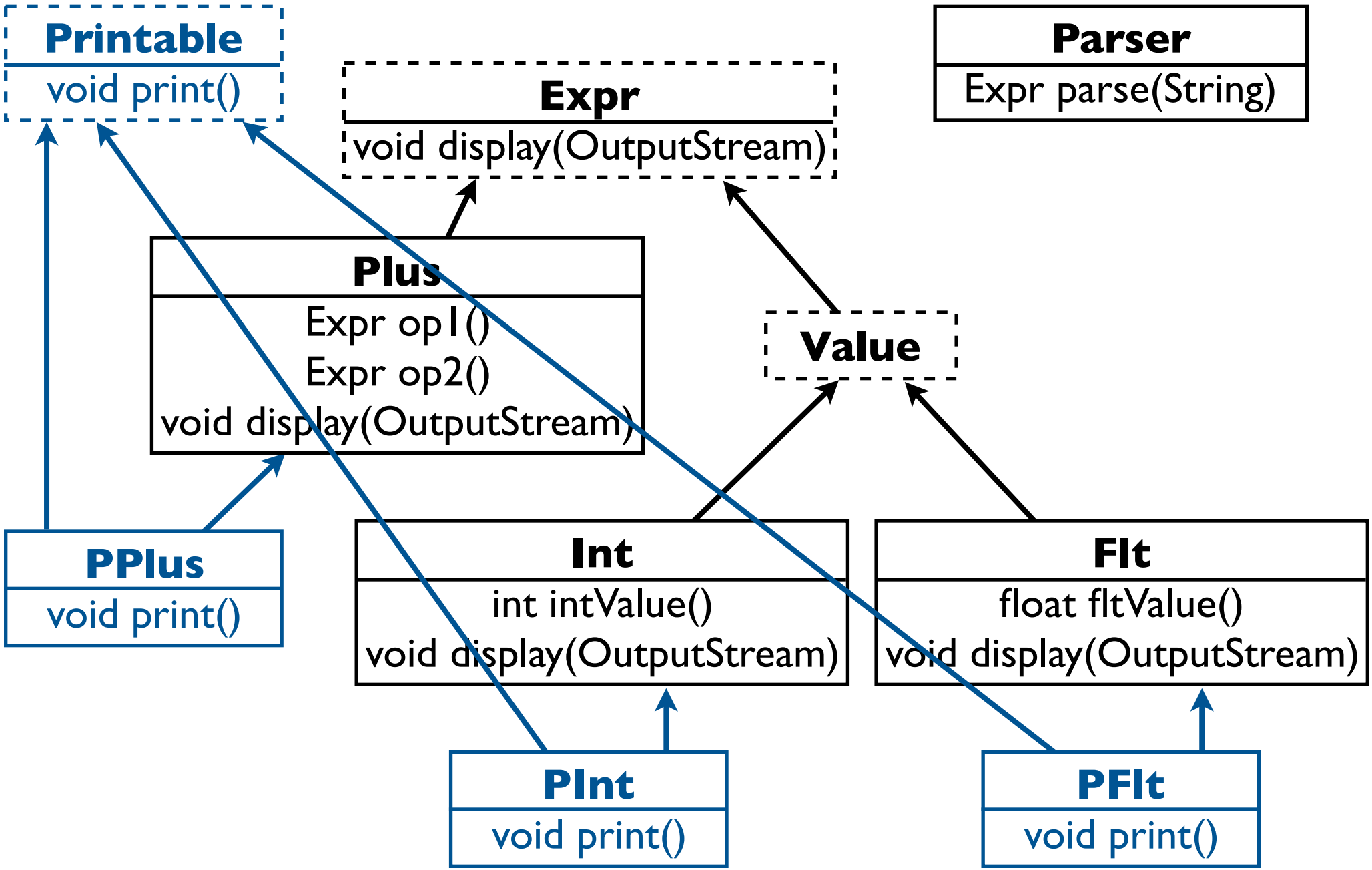
In-Place Modification

- Requires access to source
  - May not have it
  - Tricky to incorporate “their” later changes
- Others may not want to print this way
- Others may not want to print at all


Printable
void print()
Parser
Expr parse(String)
Expr
void display(OutputStream)
Plus
Expr op1()
Expr op2()
void display(OutputStream)
Value
Int
int intValue()
void display(OutputStream)
Flt
float fltValue()
void display(OutputStream)

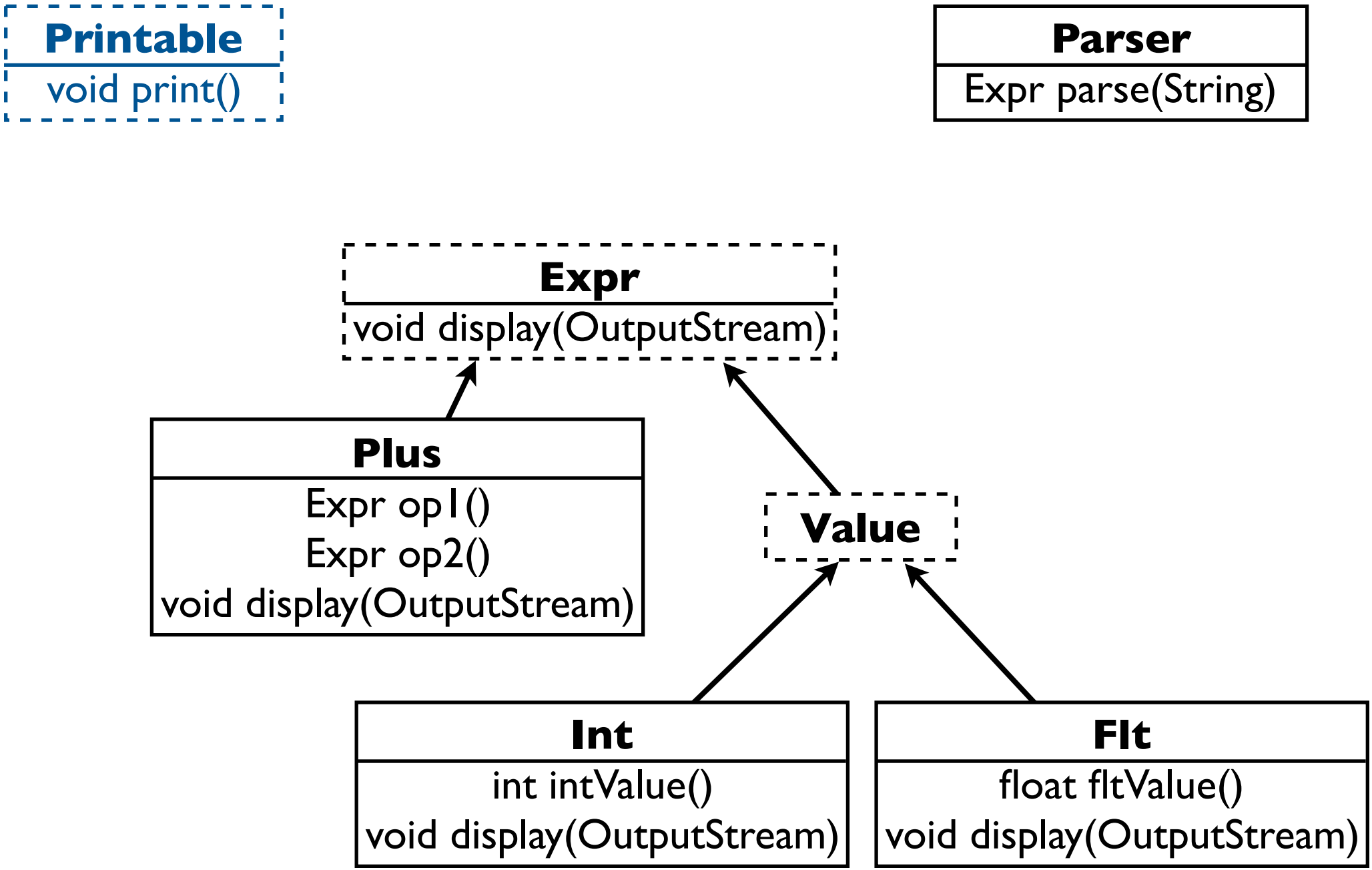

Printable
void print()
Expr
void display(OutputStream)
Parser
Expr parse(String)
Plus
Expr op1()
Expr op2()
void display(OutputStream)
Value
Int
int intValue()
void display(OutputStream)
Flt
float fltValue()
void display(OutputStream)


Printable
void print()
Expr
void display(OutputStream)
Parser
Expr parse(String)
Plus
Expr op1()
Expr op2()
void display(OutputStream)
Value
PPlus
void print()
Int
int intValue()
void display(OutputStream)
Flt
float fltValue()
void display(OutputStream)
PInt
void print()
PFlt
void print()


Printable
void print()
Expr
void display(OutputStream)
Parser
Expr parse(String)
Plus
Expr op1()
Expr op2()
void display(OutputStream)
Value
PPlus
void print()
Int
int intValue()
void display(OutputStream)
Flt
float fltValue()
void display(OutputStream)
PInt
void print()
PFlt
void print()

Inheritance

- Parser doesn't instantiate our classes
  - PPlus, PInt, PFlt not used
- Alternatives to inheritance (mixins, traits, ...) suffer from same problem
  - Good for creating new classes
  - Want to adapt existing instances!


Printable
void print()
Parser
Expr parse(String)
Expr
void display(OutputStream)
Plus
Expr op1()
Expr op2()
void display(OutputStream)
Value
Int
int intValue()
void display(OutputStream)
Flt
float fltValue()
void display(OutputStream)

Adapter Design Pattern

```
class ExprAdapter implements Printable {
  Expr e;
  ExprAdapter(Expr e) { this.e = e; }
  void print() {
    if (e instanceof Plus) {
      Plus p = (Plus) e;
      ...
    }
    else
      e.display(System.out);
  }
}
```

Adapter Design Pattern

```
class ExprAdapter implements Printable {
  Expr e;
  ExprAdapter(Expr e) { this.e = e; }
  void print() {
    if (e instanceof Plus) {
      Plus p = (Plus) e;
      ...
    }
    else
      e.display(System.out);
  }
}
```

Adapter Design Pattern

```
class ExprAdapter implements Printable {
  Expr e;
  ExprAdapter(Expr e) { this.e = e; }
  void print() {
    if (e instanceof Plus) {
      Plus p = (Plus) e;
      ...
    }
    else
      e.display(System.out);
  }
}
```

Adapter Design Pattern

```
class ExprAdapter implements Printable {
  Expr e;
  ExprAdapter(Expr e) { this.e = e; }
  void print() {
    if (e instanceof Plus) {
      Plus p = (Plus) e;
      ...
    }
    else
      e.display(System.out);
  }
}
```

Adapter Design Pattern

```
class ExprAdapter implements Printable {
  Expr e;
  ExprAdapter(Expr e) { this.e = e; }
  void print() {
    if (e instanceof Plus) {
      Plus p = (Plus) e;
      ...
    }
    else
      e.display(System.out);
  }
}
```

Adapter Design Pattern

```
class ExprAdapter implements Printable {
  Expr e;
  ExprAdapter(Expr e) { this.e = e; }
  void print() {
    if (e instanceof Plus) {
      Plus p = (Plus) e;
      ...
    }
    else
      e.display(System.out);
  }
}
```

Adapter Design Pattern

```
class ExprAdapter implements Printable {
  Expr e;
  ExprAdapter(Expr e) { this.e = e; }
  void print() {
    if (e instanceof Plus) {
      Plus p = (Plus) e;
      ...
    }
    else
      e.display(System.out);
  }
}
```

- Problems:
  - Manual dispatch

Adapter Design Pattern

```
class ExprAdapter implements Printable {
  Expr e;
  ExprAdapter(Expr e) { this.e = e; }
  void print() {
    if (e instanceof Plus) {
      Plus p = (Plus) e;
      ...
    }
    else
      e.display(System.out);
  }
}
```

- Problems:
  - Manual dispatch

```
Plus x = ...
ExprAdapter ea =
  new ExprAdapter(x);
printer.enqueue(ea);
```

Adapter Design Pattern

```
class ExprAdapter implements Printable {
  Expr e;
  ExprAdapter(Expr e) { this.e = e; }
  void print() {
    if (e instanceof Plus) {
      Plus p = (Plus) e;
      ...
    }
    else
      e.display(System.out);
  }
}
```

- Problems:
  - Manual dispatch

```
Plus x = ...
ExprAdapter ea =
  new ExprAdapter(x);
printer.enqueue(ea);
Expr y = x.op1();
```

Expanders

- Language support for **object adaptation**
- Add methods, fields, interfaces "from the outside"
- Different clients can use different expanders for the same objects
- eJava = Java 1.4 + Expanders

Key Technical Contribution!

- Modular type checking
  - different clients can safely use different expanders without conflict
  - **no surprises** guarantee
  - formalized and proven in core eJava subset
  - key advance over previous work
- Modular compilation
  - no modifications to existing source / bytecode

Declaring an Expander

```
expander PX of Expr implements Printable {
  void print() {
    display(System.out);
  }
}
```

Declaring an Expander

```
expander PX of Expr implements Printable {
  void print() {
    display(System.out);
  }
}
```

Declaring an Expander

```
expander PX of Expr implements Printable {
  void print() {
    display(System.out);
  }
}
```

Declaring an Expander

```
expander PX of Expr implements Printable {
  void print() {
    display(System.out);
  }
}
```

Declaring an Expander

```
expander PX of Expr implements Printable {
  void print() {
    display(System.out);
  }
}
```

Declaring an Expander

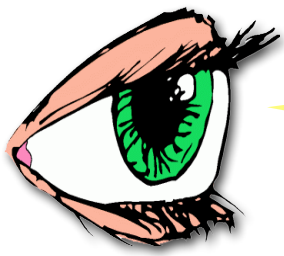
```
expander PX of Expr implements Printable {
  void print() {
    display(System.out);
  }
}
expander PX of Plus {
  void print() {
    Expr x = op1(), y = op2();
    ...
  }
}
```

Declaring an Expander

```
expander PX of Expr implements Printable {
  void print() {
    display(System.out);
  }
}
expander PX of Plus {
  void print() {
    Expr x = op1(), y = op2();
    ...
  }
}
```

Printable

void print()

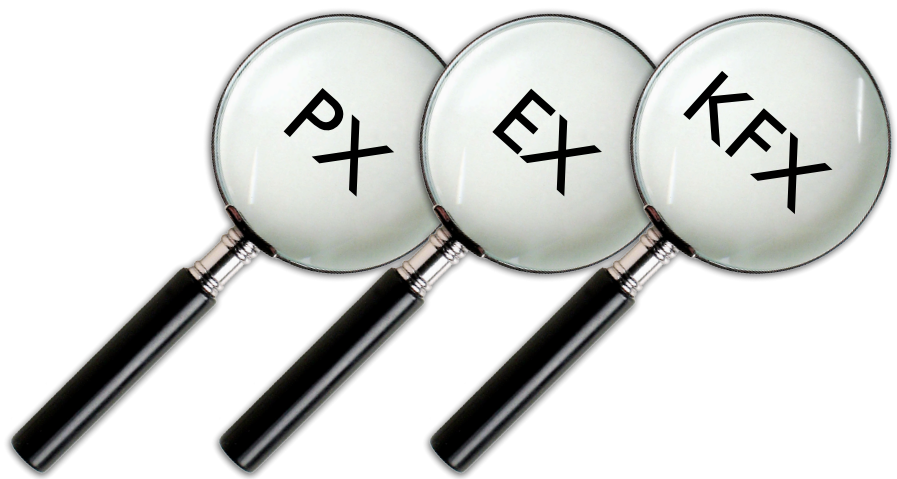


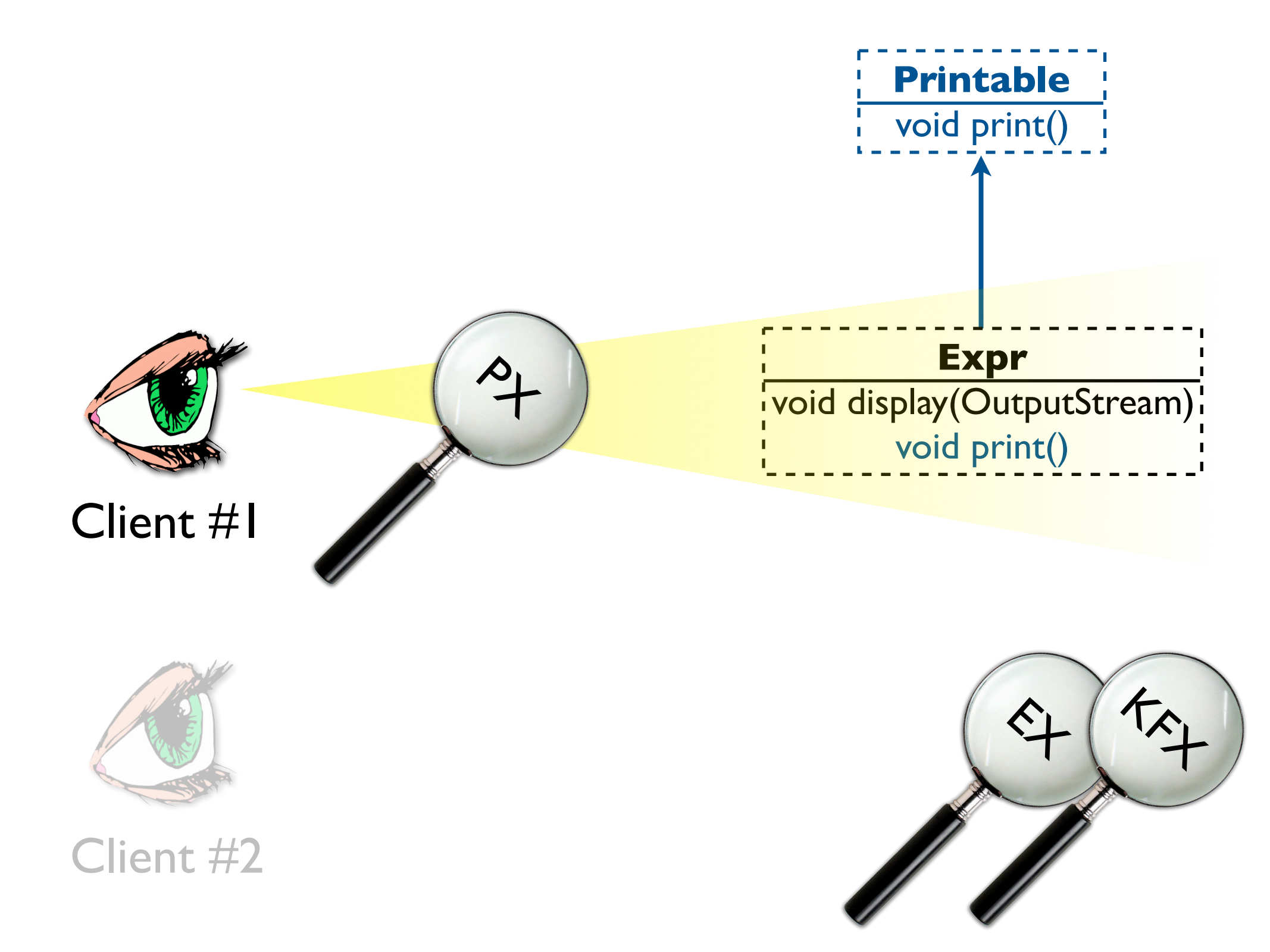
Client #1

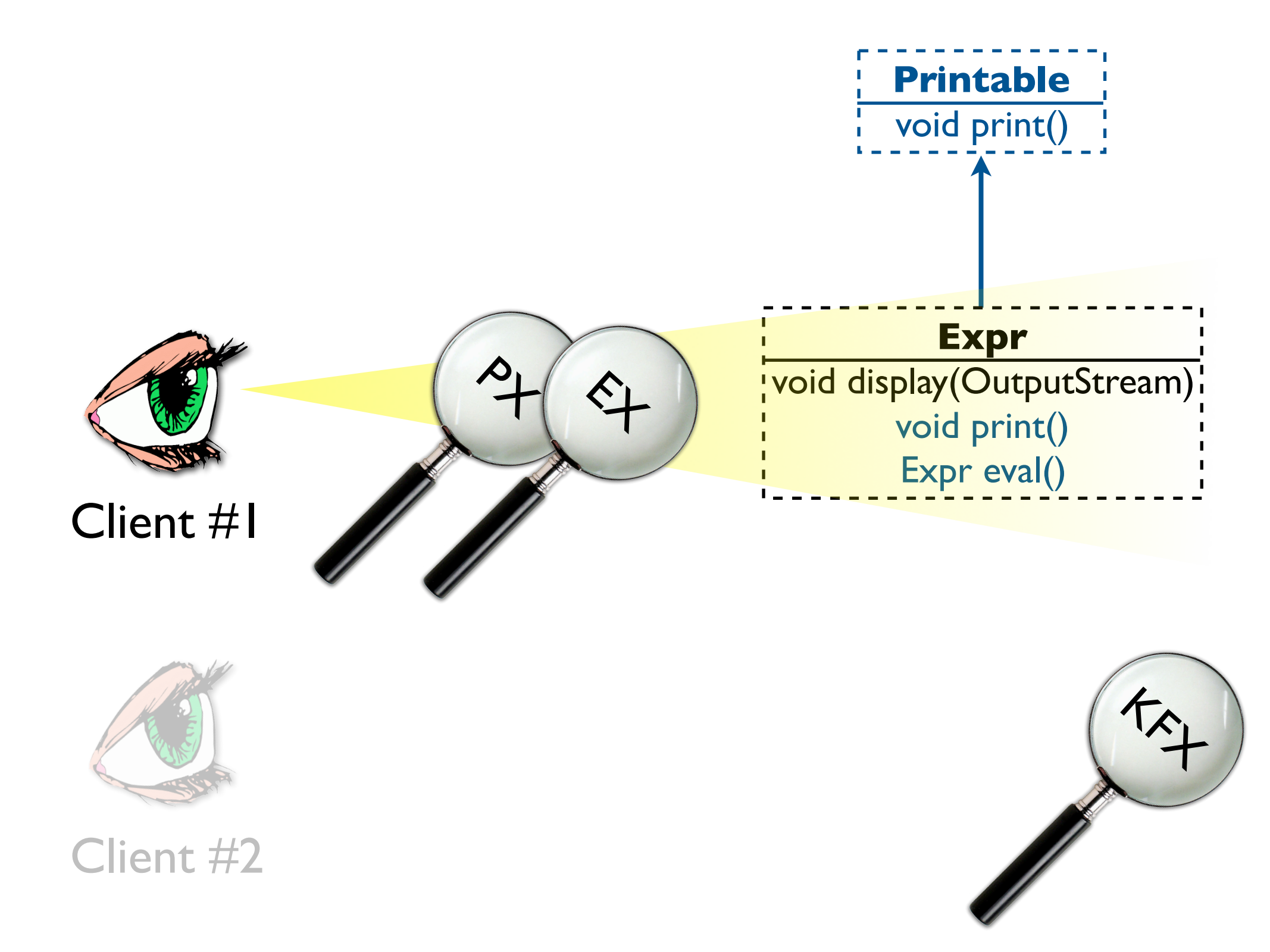
Expr

void display(OutputStream)

Client #2




Printable
void print()
Expr
void display(OutputStream)
void print()
PX
EX
KFX
Client #1
Client #2


Printable
void print()
Expr
void display(OutputStream)
void print()
Expr eval()
PX
EX
KFX
Client #1
Client #2

Printable

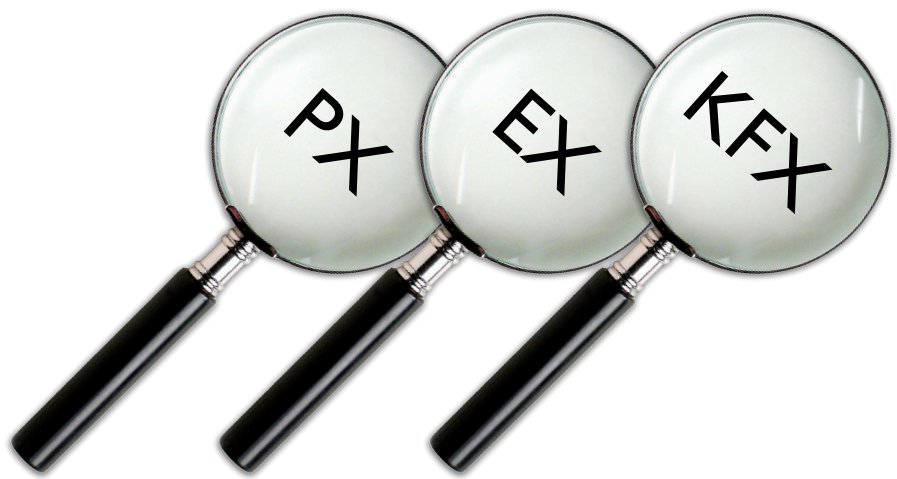
void print()

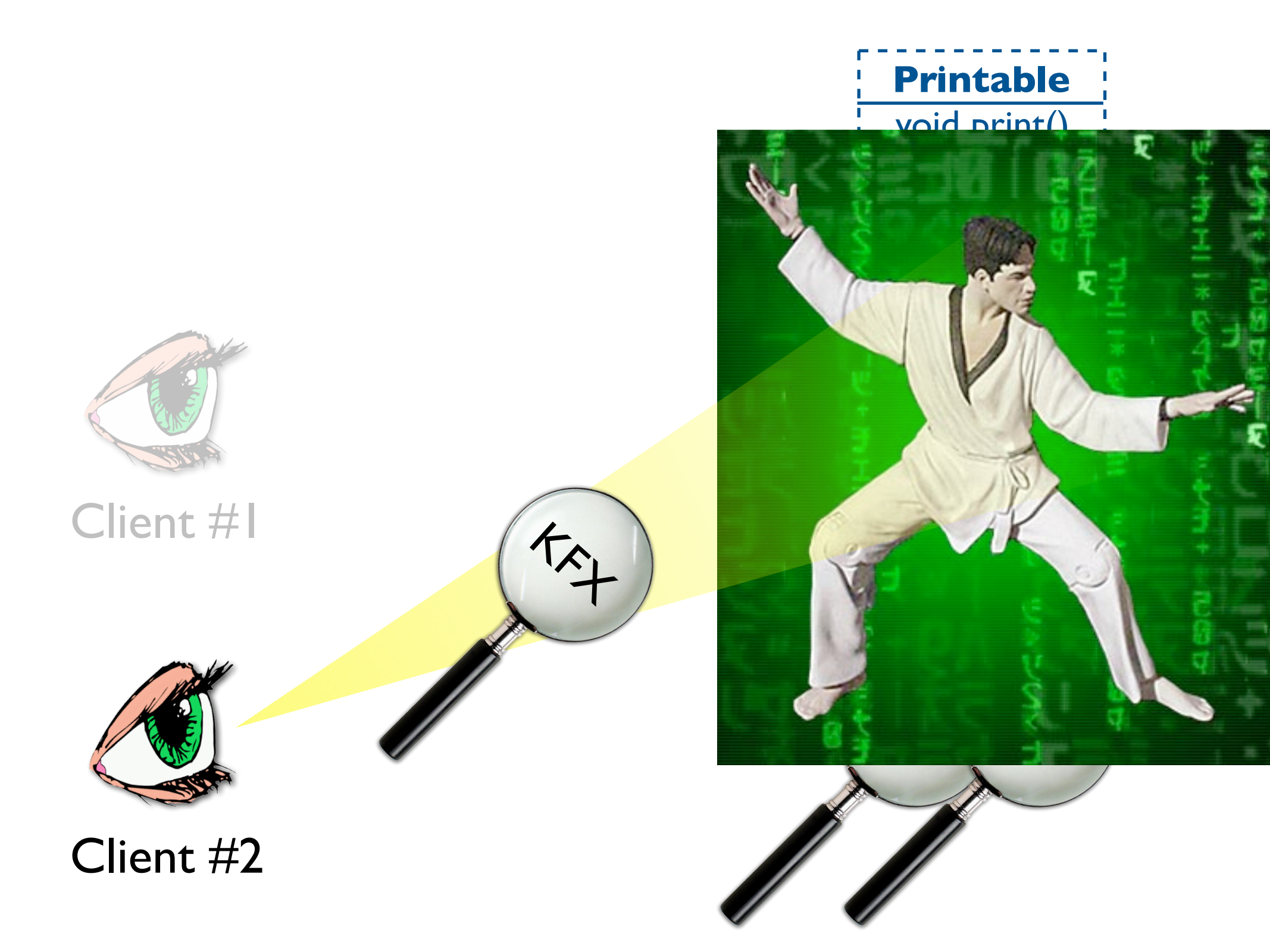
Client #1

Expr

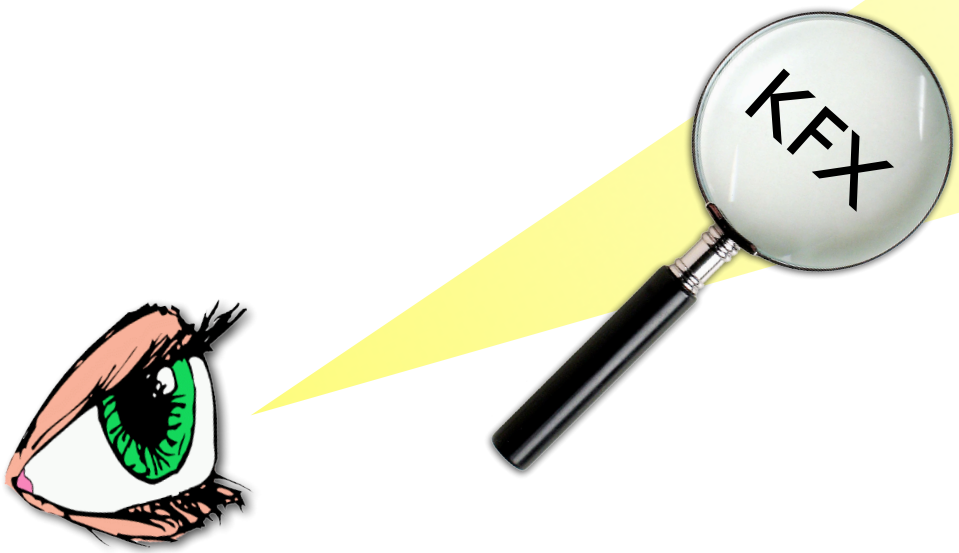
void display(OutputStream)

Client #2




Printable
void print()
Client #1
KFX
Client #2

- Client #2 = “The Matrix”
- Expr = Neo
- KFX = program downloaded to Neo’s brain
- Neo knows Kung-Fu inside the matrix, but not outside!



Client #2

Explicit import

- A client must explicitly import the expanders it wants to use
  - other expanders cannot affect its behavior
- Easy to understand
- Allows an object to be expanded in different ways by different clients
  - no conflicts

```
import print.PX;
class WorkerThread {
  void work() {
    Expr e;
    ...
    printThread.enqueue(e);
  }
}
```

Adding State with Expanders

```
expander TX of Expr {
    Type type = null;
    Type type() {
        if (type == null) typeCheck();
        return type;
    }
    void typeCheck() { type = new ErrorType(); }
}
```

Adding State with Expanders

```
expander TX of Expr {
    Type type = null;
    Type type() {
        if (type == null) typeCheck();
        return type;
    }
    void typeCheck() { type = new ErrorType(); }
}
```

Adding State with Expanders

```
expander TX of Expr {
    Type type = null;
    Type type() {
        if (type == null) typeCheck();
        return type;
    }
    void typeCheck() { type = new ErrorType(); }
}
```

Adding State with Expanders

```
expander TX of Expr {
    Type type = null;
    Type type() {
        if (type == null) typeCheck();
        return type;
    }
    void typeCheck() { type = new ErrorType(); }
}
expander TX of Int {
    void typeCheck() { type = new IntType(); }
}
```

Adding State with Expanders

```
expander TX of Expr {
    Type type = null;
    Type type() {
        if (type == null) typeCheck();
        return type;
    }
    void typeCheck() { type = new ErrorType(); }
}
expander TX of Int {
    void typeCheck() { type = new IntType(); }
}
expander TX of Plus {
    void typeCheck() {
        Type t1 = op1().type(), t2 = op2().type();
        type = ...
    }
}
```

(Similarly for Flt)

Method Resolution

- Same as in Java:
  - Static types of the receiver and arguments determine which **method family** is being called
    - **MF:** collection of methods that all override a common top method
  - Dynamic dispatch finds most specific implementation for receiver within method family

Resolving Ambiguities

```
expander X1 of C {
  void m1() { ... }
  void m2() { ... }
}
expander X1 of D {
  void m1() { ... }
}
```

Resolving Ambiguities

```
expander X1 of C {
  void m1() { ... }
  void m2() { ... }
}
expander X1 of D {
  void m1() { ... }
}
```

```
expander X2 of C {
  void m2() { ... }
}
```

Resolving Ambiguities

```
expander X1 of C {
  void m1() { ... }
  void m2() { ... }
}
expander X1 of D {
  void m1() { ... }
}
```

```
expander X2 of C {
  void m2() { ... }
}
```

```
import p.X1;
import p.X2;
class Test {
  void test {
    C x = new D();
    x.m1();
    x.m2();
  }
}
```

Resolving Ambiguities

```
expander X1 of C {
  void m1() { ... }
  void m2() { ... }
}
expander X1 of D {
  void m1() { ... }
}
```

```
expander X2 of C {
  void m2() { ... }
}
```

```
import p.X1;
import p.X2;
class Test {
  void test {
    C x = new D();
    x.m1();
    x.m2();
  }
}
```

Resolving Ambiguities

```
expander X1 of C {
  void m1() { ... }
  void m2() { ... }
}
expander X1 of D {
  void m1() { ... }
}
```

```
expander X2 of C {
  void m2() { ... }
}
```

```
import p.X1;
import p.X2;
class Test {
  void test {
    C x = new D();
    x.m1();
    x.m2();
  }
}
```

Resolving Ambiguities

```
expander X1 of C {
  void m1() { ... }
  void m2() { ... }
}
expander X1 of D {
  void m1() { ... }
}
```

```
expander X2 of C {
  void m2() { ... }
}
```

```
import p.X1;
import p.X2;
class Test {
  void test {
    C x = new D();
    x.m1();
    x.m2();
  }
}
```

Resolving Ambiguities

```
expander X1 of C {
  void m1() { ... }
  void m2() { ... }
}
expander X1 of D {
  void m1() { ... }
}
```

```
expander X2 of C {
  void m2() { ... }
}
```

```
import p.X1;
import p.X2;
class Test {
  void test {
    C x = new D();
    x.m1();
    x.m2();
  }
}
```

Resolving Ambiguities

```
expander X1 of C {
  void m1() { ... }
  void m2() { ... }
}
expander X1 of D {
  void m1() { ... }
}
```

```
expander X2 of C {
  void m2() { ... }
}
```

```
import p.X1;
import p.X2;
class Test {
  void test {
    C x = new D();
    x.m1();
    x.m2();
  }
}
```

Resolving Ambiguities

```
expander X1 of C {
  void m1() { ... }
  void m2() { ... }
}
expander X1 of D {
  void m1() { ... }
}
```

```
expander X2 of C {
  void m2() { ... }
}
```

```
import p.X1;
import p.X2;
class Test {
  void test {
    C x = new D();
    x.m1();
    (x with X2).m2();
  }
}
```

Preventing Accidental Overriding

```
class C {
  // does not have m()
}
```

Preventing Accidental Overriding

```
class C {
  // does not have m()
}
```

```
class D extends C {
  int m() { ... }
}
```

Preventing Accidental Overriding

```
class C {
  // does not have m()
}
```

```
class D extends C {
  int m() { ... }
}
```

```
expander E of C {
  int m() { ... }
}
```

Preventing Accidental Overriding

```
class C {
  // does not have m()
}
```

```
class D extends C {
  int m() { ... }
}
```

```
expander E of C {
  int m() { ... }
}
```

Different method families!

- Not aware of each other
- Happen to have same signature
- Probably have different behaviors

Preventing Accidental Overriding

```
class C {
  // does not have m()
}
```

```
class D extends C {
  int m() { ... }
}
```

```
expander E of C {
  int m() { ... }
}
```

```
import p.E;

class Test {
  void test() {
    C x = new D();
    x.m() * 5;
  }
}
```

Preventing Accidental Overriding

```
class C {
  // does not have m()
}
```

```
class D extends C {
  int m() { ... }
}
```

```
expander E of C {
  int m() { ... }
}
```

```
import p.E;

class Test {
  void test() {
    C x = new D();
    x.m() * 5;
  }
}
```

Preventing Accidental Overriding

```
class C {
  // does not have m()
}
```

```
class D extends C {
  int m() { ... }
}
```

```
expander E of C {
  int m() { ... }
}
```

```
import p.E;

class Test {
  void test() {
    C x = new D();
    x.m() * 5;
  }
}
```

NOT Preventing Accidental Overriding

```
class C {
  // does not have m()
}
```

```
class D extends C {
  int m() { ... }
}
```

```
expander E of C {
  int m() { ... }
}
```

```
import p.E;

class Test {
  void test() {
    C x = new D();
    x.m() * 5;
  }
}
```

(dream sequence...)

NOT Preventing Accidental Overriding

```
class C {
  // does not have m()
}
```

```
class D extends C {
  int m() { ... }
}
```

```
expander E of C {
  int m() { ... }
}
```

```
import p.E;

class Test {
  void test() {
    C x = new D();
    x.m() * 5;
  }
}
```

(dream sequence...)

NOT Preventing Accidental Overriding

```
class C {
  // does not have m()
}
```

```
class D extends C {
  int m() { ... }
}
```

```
expander E of
  int m() { ... }
}
```

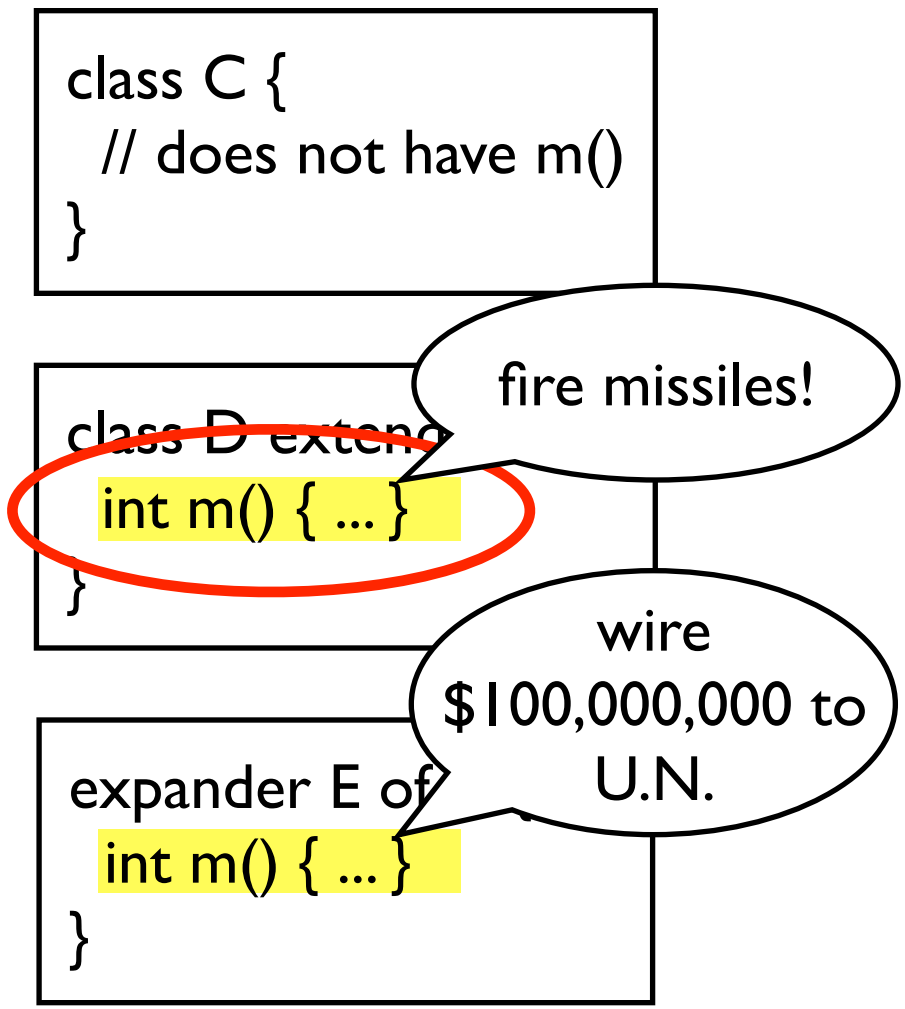
wire $100,000,000 to U.N.

```
import p.E;

class Test {
  void test() {
    C x = new D();
    x.m() * 5;
  }
}
```

(dream sequence...)

NOT Preventing Accidental Overriding




```
import p.E;

class Test {
  void test() {
    C x = new D();
    x.m() * 5;
  }
}
```

(dream sequence...)

NOT Preventing Accidental Overriding

```
class C {
  // does not have m()
}
```

```
class D extends C {
  int m() { ... }
}
```

```
expander E of C {
  int m() { ... }
}
```

```
import p.E;

class Test {
  void test() {
    C x = new D();
    x.m() * 5;
  }
}
```

(dream sequence...)

NOT Preventing Accidental Overriding

```
class C {
  // does not have m()
}
```

```
class D extends C {
  String m() { ... }
}
```

```
expander E of C {
  int m() { ... }
}
```

```
import p.E;

class Test {
  void test() {
    C x = new D();
    x.m() * 5;
  }
}
```

(dream sequence...)

Local Expander Refinement

```
import print.PX;
import new.Parser;

class Test {
  void test {
    Expr e = Parser.parse(...);
    e.print();
  }
}
```

Local Expander Refinement

```
import print.PX;
import new.Parser;

class Test {
  void test {
    Expr e = Parser.parse(...);
    e.print();
  }
}
```

expander PX of Times { ... }

Local Expander Refinement

```
import print.PX;
import new.Parser;

expander PX of Times { ... }

class Test {
  void test {
    Expr e = Parser.parse(...);
    e.print();
  }
}
```

Local Expander Refinement

```
import print.PX;
import new.Parser;

expander PX of Times { ... }

class Test {
  void test {
    Expr e = Parser.parse(...);
    e.print();
  }
}
```

Implementation

- Implemented eJava with Polyglot (Nystrom et al. '03)
- Add'l methods, fields, interfaces → wrapper
  - created lazily
  - cached (necessary for state)
  - weak references for proper GC'ing

Experience

- Case Study: Adapting business objects to Swing
  - Wrote Java and eJava programs
  - eJava version more easily extensible, less error prone
- Exploratory Study of Eclipse
  - Identified natural places where expanders could be used for several of Eclipse's extensibility idioms

Related Work

- AspectJ [Kiczales et al., '01] and other AOP langs
  - Different goals:
    - Aspects have global scope
    - Expanders have limited scope
  - Can't TC or compile aspects in isolation
    - Possible bad interactions between aspects
  - Aspects can do many, **many** things beyond the scope of expanders

Related Work (cont'd)

- Classboxes [Bergel et al., '05]
  - Can add methods, fields, and interfaces
  - Add'l methods, fields, interfaces are treated as if they were declared in the original class
  - Accidental method overriding (and type hole)
  - Non-modular compilation (weaving)
  - A client can only import one version of a class

Related Work (cont'd)

- MultiJava's *open classes* [Clifton et al., '00]
  - can add methods, but not interfaces or fields
    - explicit import
  - modular TC and compilation
- Half & Half [Baumgartner et al., '02]
  - *retroactive abstraction:* can only add interfaces
  - can't add methods or fields

Ongoing and Future Work

- Generic expanders
- Expander inheritance
- Using expanders for class composition, unifying
  - expander-style object extensibility
  - *Traits**-style class extensibility [Scharli et al. '03]

Expanders, a new approach to extensibility

- Support for object adaptationn
  - Add methods, fields, and interfaces to existing class hierarchies
  - Fully modular TC and compilation
- **Expressiveness w/o surprises!**
- Comprehensible to "mere mortals"

Questions?