Compiler Construction
Winter 2020

Recitation 2:
Visitor Pattern & Resolving Method Names

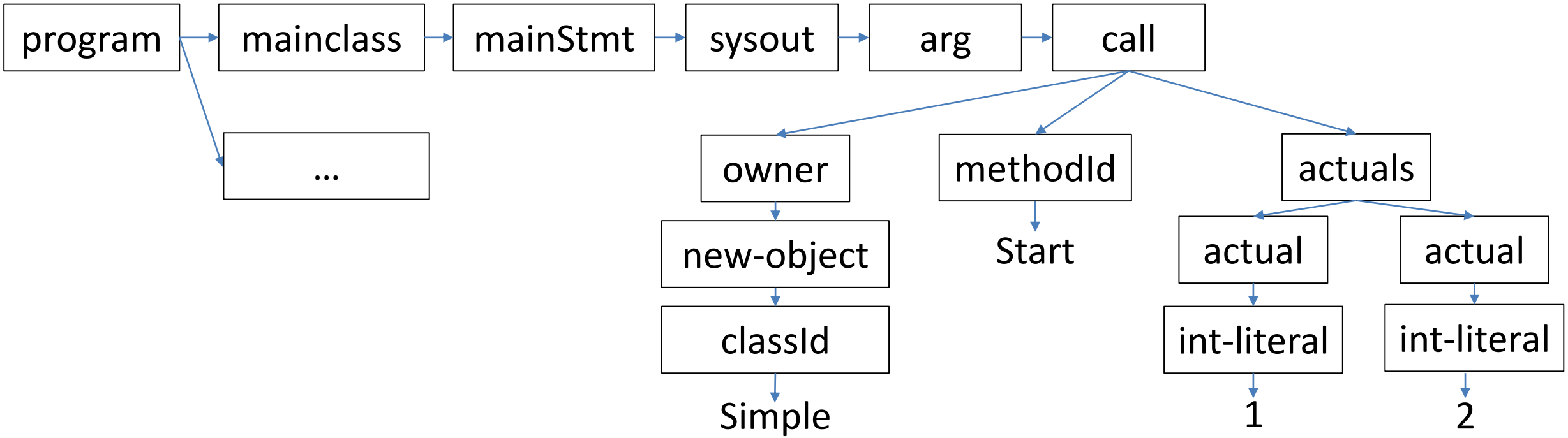
Yotam Feldman

Simple Example.java

```
class Main {
    public static void main(String[] a) {
        System.out.println(new Simple().Start(1, 1));
    }
}

class Trivial {
        int f;
}
```

```
class Simple extends Trivial {
        public int Start(int a, int b) {
                int x;
                int y;

                x = a;
                y = b + 3;

                if (true) {
                        f = 0;
                } else {
                        f = 1;
                }

                return x + y + f;
        }
}
```

Simple Example AST (partial)




```
class Main {
    public static void main(String[] a) {
        System.out.println(new Simple().Start(1, 2));
    }
}
…
```

The AST in a Class Hierarchy

```
public abstract class AstNode {
        …
}
public class Program extends AstNode {
        private List<ClassDecl> classdecls;
        …
}
public class ClassDecl extends AstNode {
        private List<MethodDecl> methoddecls;
        …
}
public class MethodDecl extends AstNode {
        private List<Statement> statements;
        …
}
```

```
public abstract class Statement extends AstNode
{
        …
}
public class IfStatement extends Statement {
        private Expr cond;
        private Statement thencase;
        private Statement elsecase;
        …
}
…
```

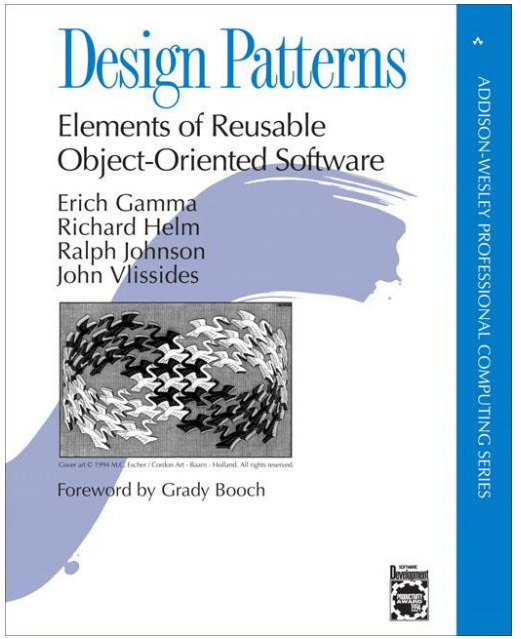
Passing on AST: Naïve Solution

```
public abstract class AstNode {
        …
        public abstract void print();
}
public class Program extends AstNode {
        private List<ClassDecl> classdecls;
        …
        public void print() {…}
}
public class ClassDecl extends AstNode {
        private List<MethodDecl> methoddecls;
        …
        public void print() {…}
}
public class MethodDecl extends AstNode {
        private List<Statement> statements;
        …
        public void print() {…}
}
```

```
public abstract class Statement extends AstNode
{
        …
}
public class IfStatement extends Statement {
        private Expr cond;
        private Statement thencase;
        private Statement elsecase;
        …
        public void print() {…}
}
…
```

Visitor Pattern

"**Intent**: Represent an operation to be performed on the elements of an object structure.
Visitor lets you define a new operation without changing the classes of the elements on which it operates.
Motivation: Consider a compiler that represents programs as abstract syntax trees. … " (p. 366)



- Each operation (pass) may be implemented as separate visitor
- Use double-dispatch to find the right method for each object

Visitor Interface

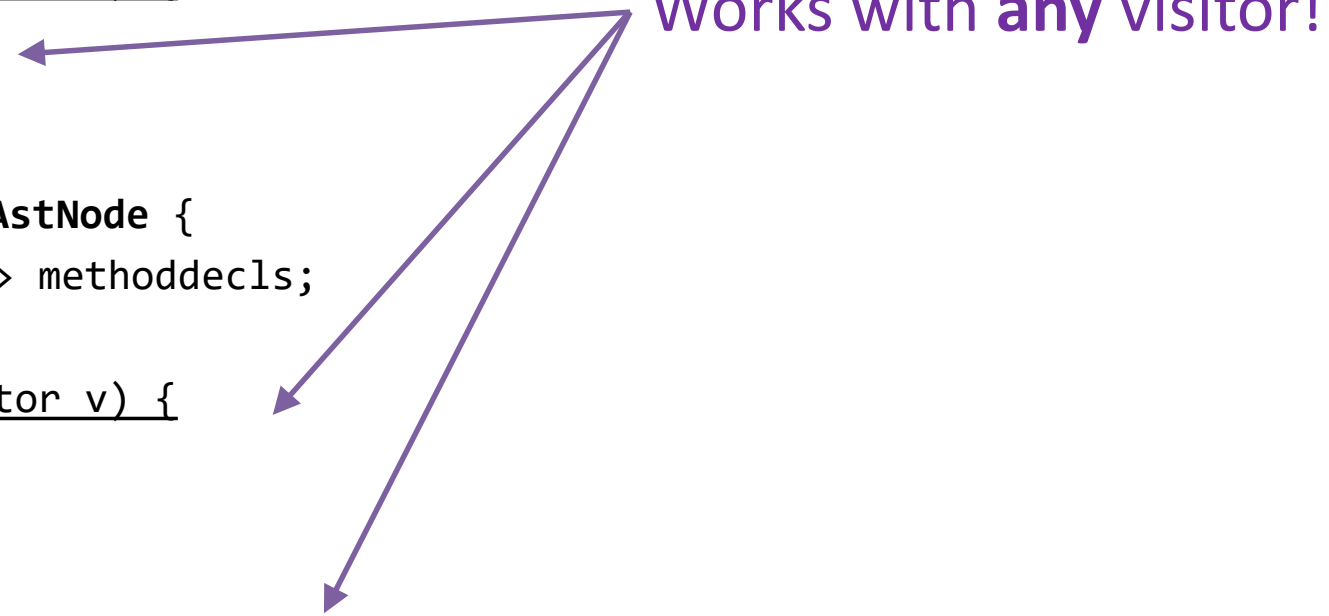
```
public interface Visitor {
        public abstract void visit(Program p);
        public abstract void visit(ClassDecl c);
        public abstract void visit(MethodDecl c);
        public abstract void visit(IfStatement c);
        …
}
```

(Warning: don't write visit methods for classes that have subclasses!)

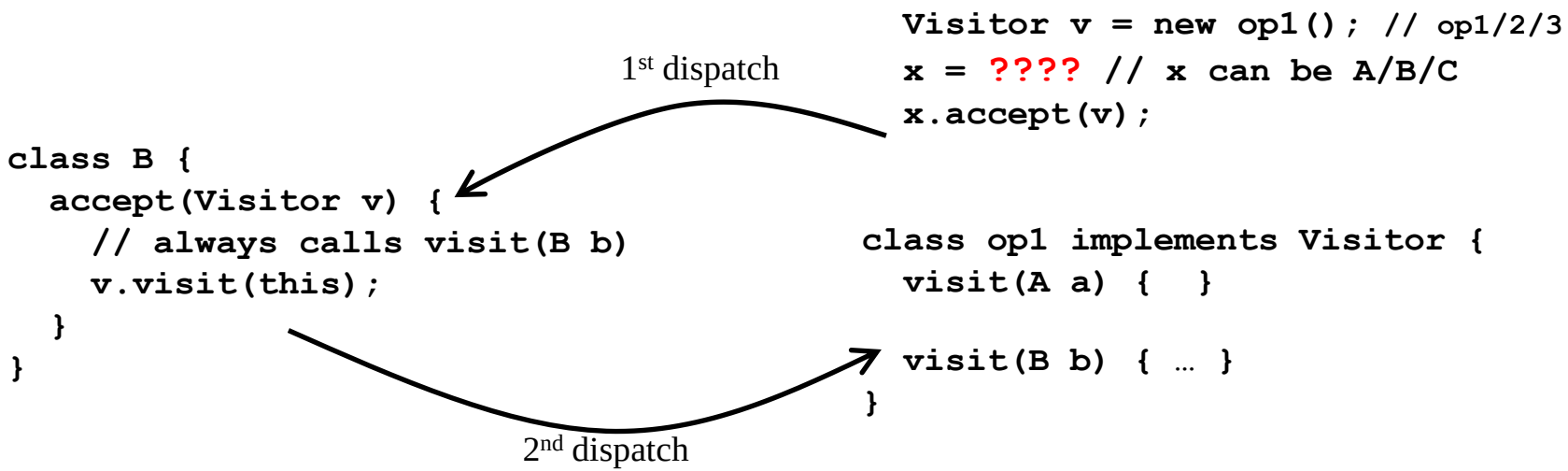
```
public class PrintVisitor implements Visitor {
        public void visit(Program p) {
                …
        }
        public void visit(ClassDecl c) {
                …
        }
        public void visit(MethodDecl c) {
                …
        }
        public void visit(IfStatement c) {
                …
        }
        …
}
```

Acceptor Methods

```
public abstract class AstNode {
        …
        public abstract void accept(Visitor v);
}
public class Program extends AstNode {
        private List<ClassDecl> classdecls;
        …
        public void accept(Visitor v) {
                v.visit(this);
        }
}
public class ClassDecl extends AstNode {
        private List<MethodDecl> methoddecls;
        …
        public void accept(Visitor v) {
                v.visit(this);
        }
}
…
```



Double Dispatch



Example: MiniJava AST to Java Code

Symbol Resolution

Resolving Instance Methods

```
class A {
      public int bar(int x, int y)
      {…}
}
class B {
      public int bar(int x, int y)
      {…}
}
```

```
…
x.bar(1, 2)

…
```

Resolving Instance Methods

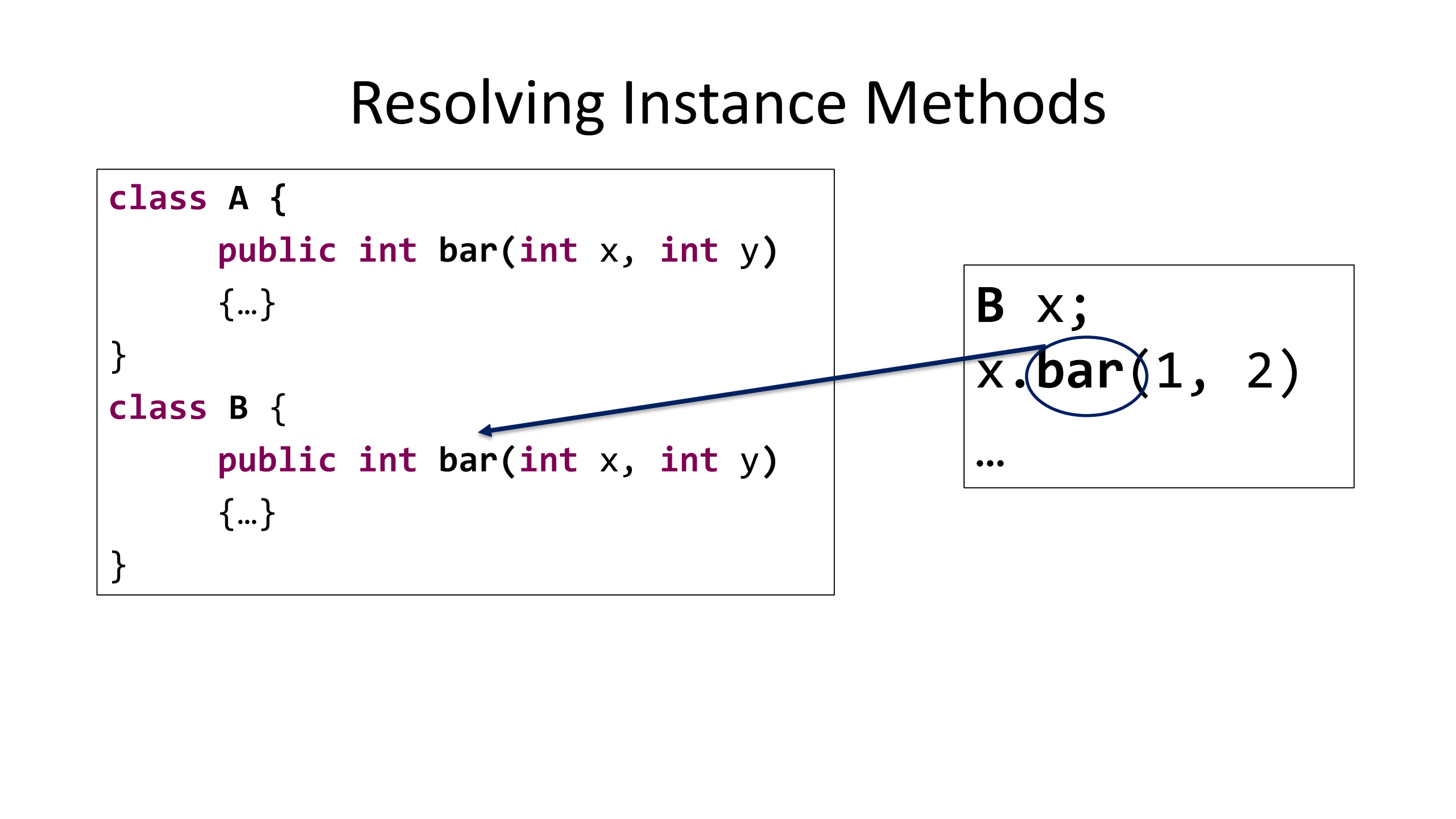
```
class A {
      public int bar(int x, int y)
      {…}
}
class B {
      public int bar(int x, int y)
      {…}
}
```

```
A x;
x.bar(1, 2)

…
```

Resolving Instance Methods

```
class A {
      public int bar(int x, int y)
      {…}
}
class B {
      public int bar(int x, int y)
      {…}
}
```

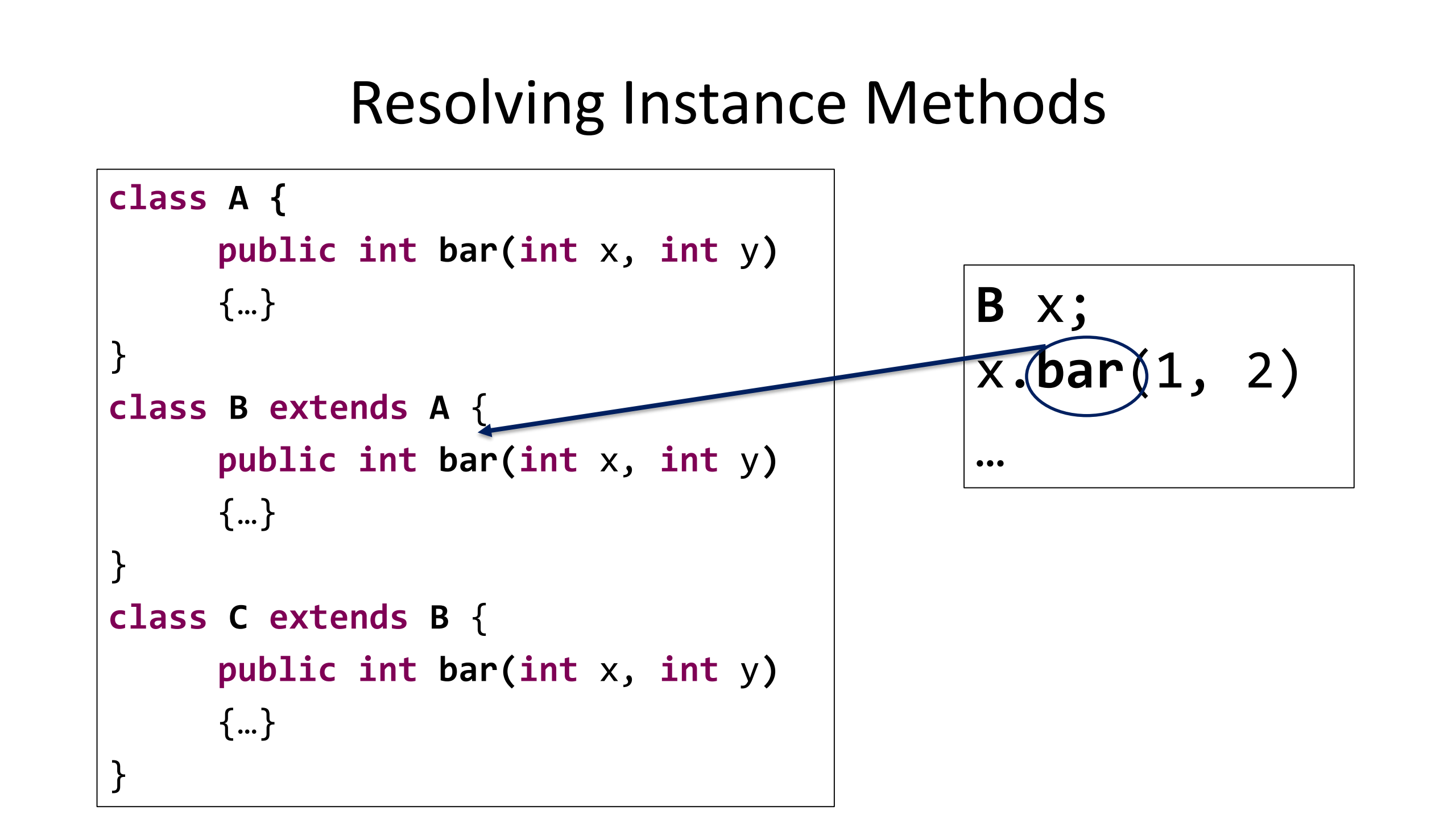
```
B x;
x.bar(1, 2)

…
```



Resolving Instance Methods

```
class A {
      public int bar(int x, int y)
      {…}
}
class B extends A {
      public int bar(int x, int y)
      {…}
}
class C extends B {
      public int bar(int x, int y)
      {…}
}
```

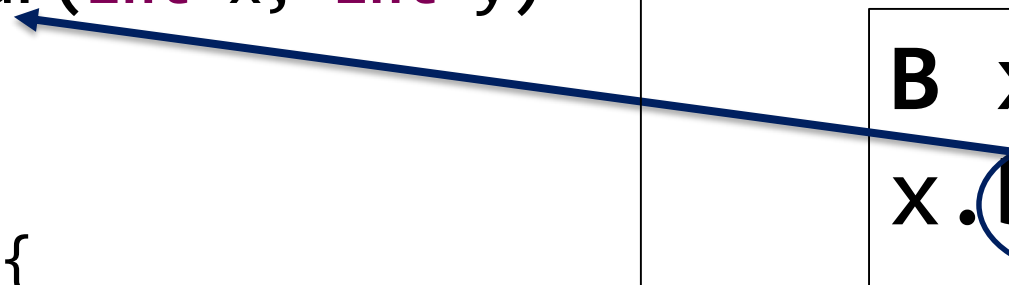
```
B x;
x.bar(1, 2)

…
```



Resolving Instance Methods

```
class A {
      public int bar(int x, int y)
      {…}
}
class B extends A {


      …
}
class C extends B {
      public int bar(int x, int y)
      {…}
}
```

```
B x;
x.bar(1, 2)

…
```



Determining the Static Type

```
[??? x]
x.bar(1, 2)
```

1. Variable / formal parameter / field:

```
A x; x.bar(1, 2); -> A
```

2. new:

```
new A().bar(1,2); -> A
```

3. this:

```
this.bar(1,2); -> enclosing class
```

Determining the Static Type Beyond MiniJava

```
e1.bar(e2, e3)
```

1. `e1` could be more complex, requiring more elaborate type analysis (later in the course)

```
new B().getField().bar(1, 2);
```

2. Overloading: need to know the number and the static type of the actual parameters

Exercise #1

- **Start early**
- Get accustomed to writing AST (XMLs) as tests
  - Some larger AST examples in https://github.com/yotamfe/compilers-project-2020-public/tree/master/examples/ast
- Submission instructions in the website
- Next week: resolving variable names