CS614: Advanced Compilers

Front-End Recap. JavaCC and JTB

Manas Thakur
CSE, IIT Bombay

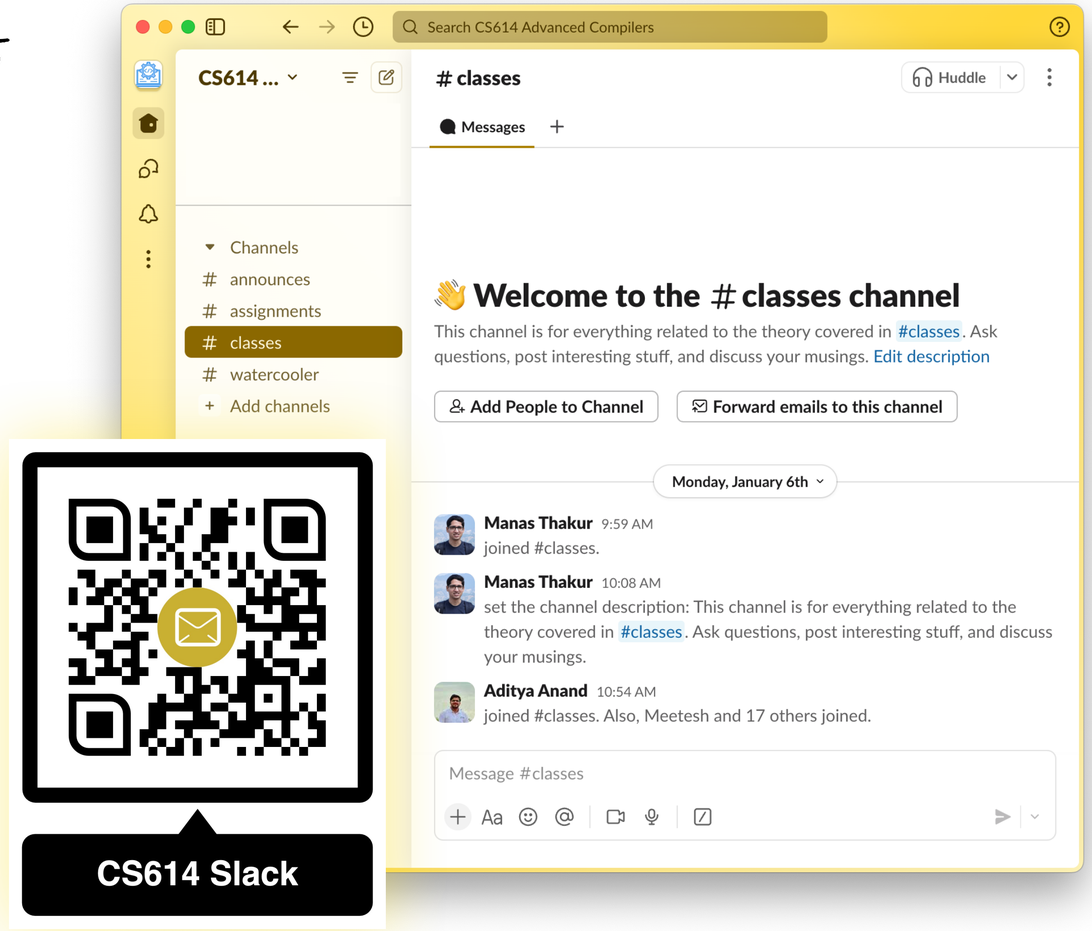
Spring 2025

Things we learnt in the last class

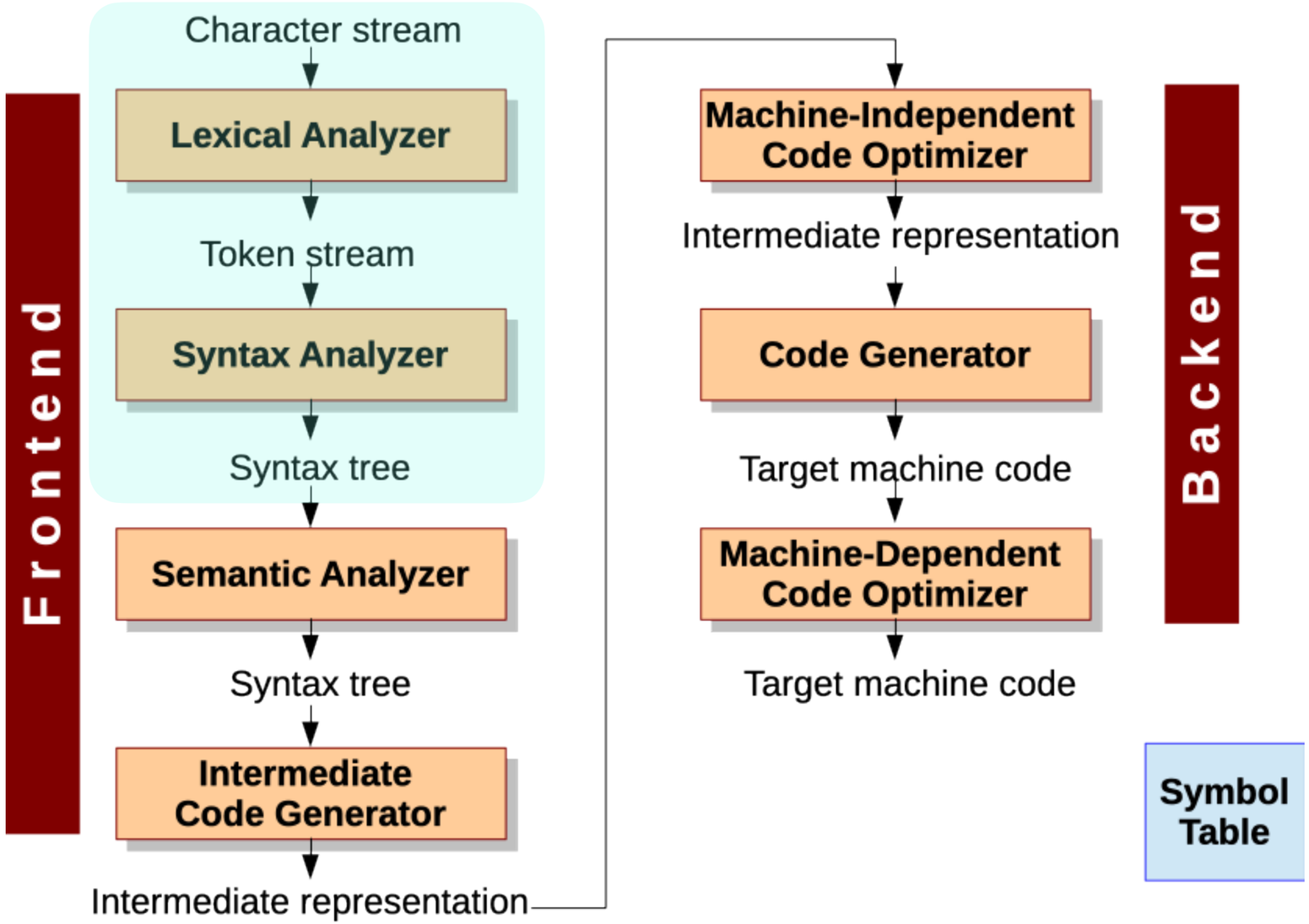
- The close relationship between Turing Awards and Compilers
- Growing importance of compilers in 2025
- Why compilers are called compilers?

C may not be the fastest!
Interpreted Python likely the slowest!

- Evaluation and logistics
- Learning resources
- Slack (only 24 members yet!)
- Typical components of a compiler



Typical components of a compiler



Lexical Analysis

- **Goal:**
  - Partition character stream to tokens
  - Record information in the symbol table

Recollection
Regexes and DFAs

- **Methodology:**
  - Represent patterns using regular expressions
  - Use lexer generators to generate scanners
  - If found then throw errors
  - Return a stream of tokens

Syntax Analysis

- **Goal:**
  - Check if the tokens are arranged as per the language's grammar
  - Construct a parse tree for future phases

Recollection
Context-Free Grammars

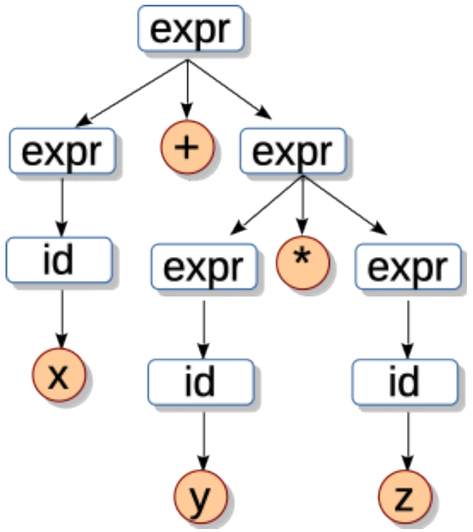
- **Methodology:**
  - Represent syntactic constructs using context-free grammars
  - Use parser generators to generate parsers
  - If found then throw errors
  - Return a parse tree (or *abstract syntax tree*, AST)

Parse Tree

- A pictorial representation of program derivation/reduction.
- Consider the following grammar:

```
expr → expr + expr | expr * expr | id
id   → a | b | ... | z
```

- A parse tree for `x + y * z`:



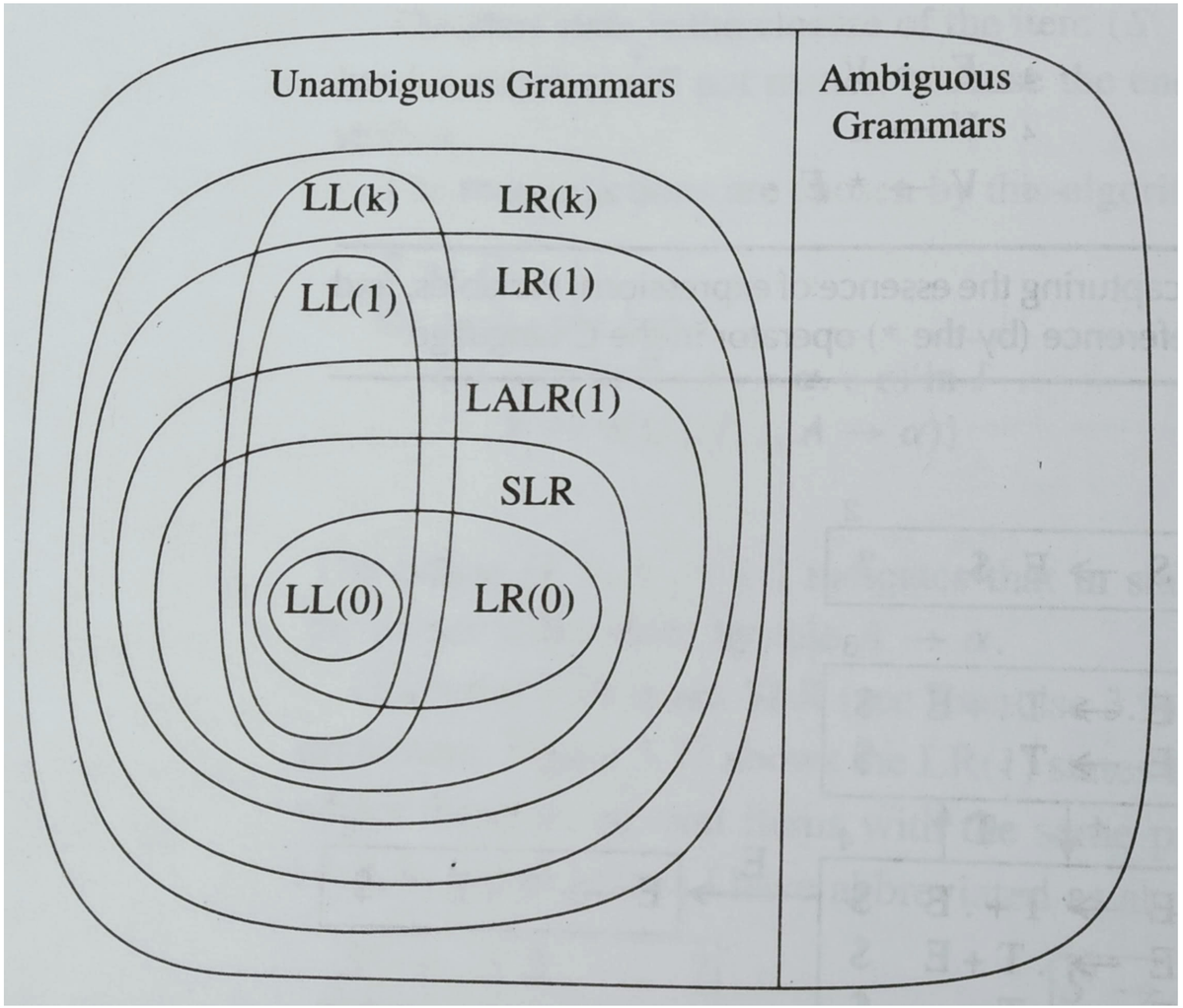
Parsing (Subset of) Java Programs

- **Practice:** Look at the MiniJava grammar (zoomed version!) on the screen and try to create parse trees for sample Java programs.

```
Goal ::= MainClass ( TypeDeclaration )* <EOF>
MainClass ::= "class" Identifier "{" "public" "static" "void" "main" "(" "String" "[" "]" Identifier ")" "{" PrintStatement "}" "}"
TypeDeclaration ::= ClassDeclaration
ClassDeclaration ::= "class" Identifier "{" ( VarDeclaration )* ( MethodDeclaration )* "}"
ClassExtendsDeclaration ::= "class" Identifier "extends" Identifier "{" ( MethodDeclaration )* "}"
VarDeclaration ::= Type Identifier ";"
MethodDeclaration ::= "public" Type Identifier "(" ( FormalParameterList )? ")" "{" ( VarDeclaration )* ( Statement )* "return" Expression ";" "}"
FormalParameterList ::= FormalParameter ( FormalParameterRest )*
FormalParameter ::= Type Identifier
FormalParameterRest ::= "," FormalParameter
Type ::= ArrayType
       | BooleanType
       | IntegerType
       | FloatType
       | Identifier
ArrayType ::= "int" "[" "]"
       | "float" "[" "]"
BooleanType ::= "boolean"
IntegerType ::= "int"
FloatType ::= "float"
Statement ::= Block
       | AssignmentStatement
       | ArrayAssignmentStatement
       | IfStatement
       | WhileStatement
       | PrintStatement
Block ::= "{" ( Statement )* "}"
AssignmentStatement ::= Identifier "=" Expression ";"
ArrayAssignmentStatement ::= Identifier "[" Expression "]" "=" Expression ";"
IfStatement ::= IfthenElseStatement
       | IfthenStatement
IfthenStatement ::= "if" "(" Expression ")" Statement
IfthenElseStatement ::= "if" "(" Expression ")" Statement "else" Statement
WhileStatement ::= "while" "(" Expression ")" Statement
PrintStatement ::= "System.out.println" "(" Expression ")" ";"
Expression ::= CompareExpression
       | NeqExpression
       | PlusExpression
       | MinusExpression
       | TimesExpression
       | DivExpression
       | MessageSend
       | PrimaryExpression
AndExpression ::= PrimaryExpression "&&" PrimaryExpression
CompareExpression ::= PrimaryExpression "<=" PrimaryExpression
NeqExpression ::= PrimaryExpression "!=" PrimaryExpression
PlusExpression ::= PrimaryExpression "+" PrimaryExpression
MinusExpression ::= PrimaryExpression "-" PrimaryExpression
TimesExpression ::= PrimaryExpression "*" PrimaryExpression
DivExpression ::= PrimaryExpression "/" PrimaryExpression
ArrayLookup ::= PrimaryExpression "[" PrimaryExpression "]"
ArrayLength ::= PrimaryExpression "." "length"
MessageSend ::= PrimaryExpression "." Identifier "(" ( ExpressionList )? ")"
ExpressionList ::= Expression ( ExpressionRest )*
ExpressionRest ::= "," Expression
PrimaryExpression ::= TrueLiteral
       | FalseLiteral
       | IntegerLiteral
       | Identifier
       | ThisExpression
       | AllocationExpression
       | NotExpression
       | BracketExpression
IntegerLiteral ::= <INTEGER_LITERAL>
TrueLiteral ::= "true"
FalseLiteral ::= "false"
Identifier ::= <IDENTIFIER>
ThisExpression ::= "this"
ArrayAllocationExpression ::= "new" "int" "[" Expression "]"
AllocationExpression ::= "new" Identifier "(" ")"
NotExpression ::= "!" Expression
BracketExpression ::= "(" Expression ")"
IdentifierList ::= Identifier ( IdentifierRest )*
IdentifierRest ::= "," Identifier
```

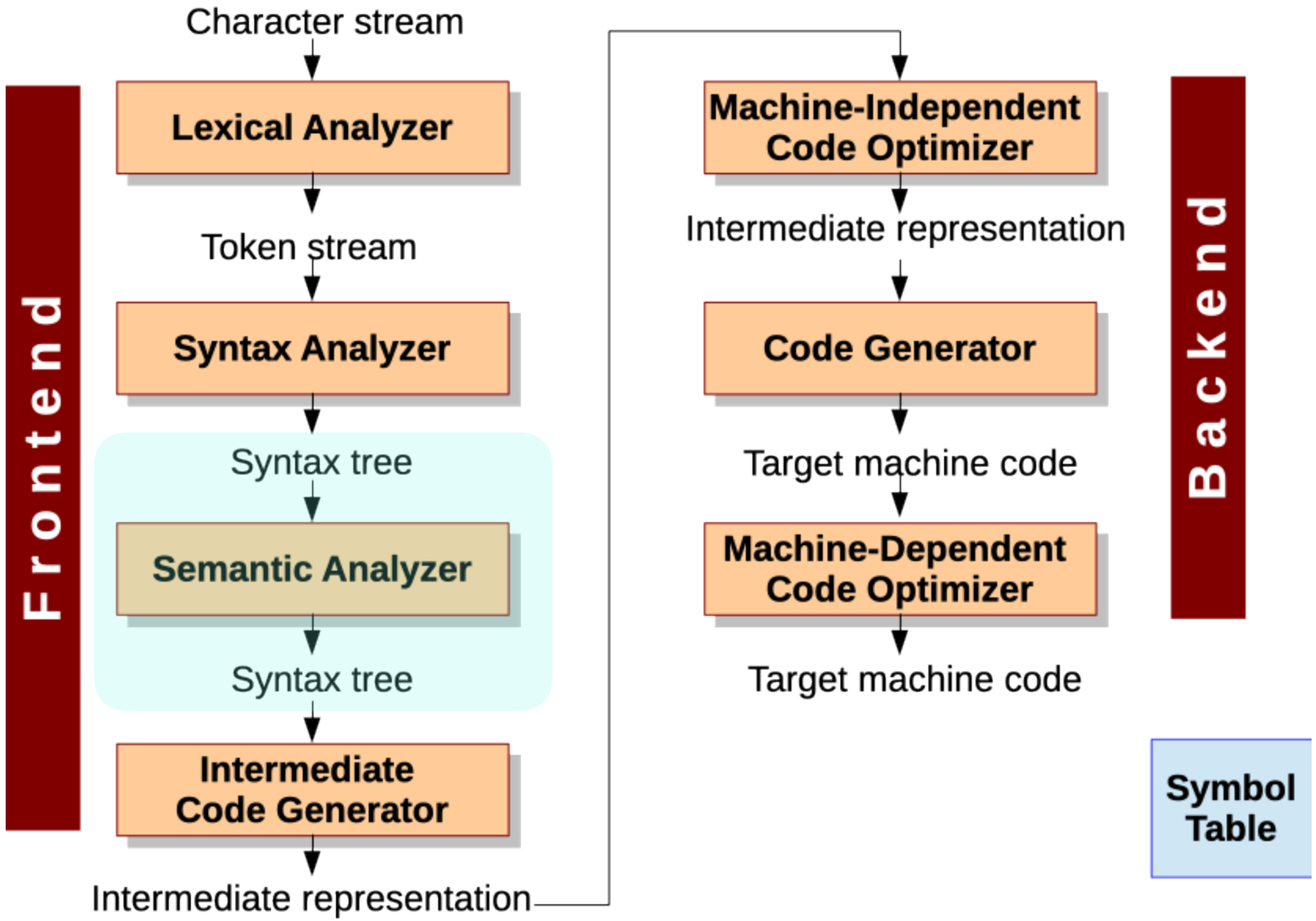
```
class Factorial {
    public static void main(String[] args) {
        System.out.println(new Fac().ComputeFac(10));
    }
}
class Fac {
    public int ComputeFac(int num) {
        int num_aux;
        if (num <= 1)
            num_aux = 1;
        else
            num_aux = num * (this.ComputeFac(num-1));
        return num_aux ;
    }
}
class MFac extends Fac {
    int f;
    public int foo(int b) {
        return b;
    }
}
```

Parser Power



Clicked from “Modern Compiler Implementation in Java” by Andrew W. Appel

Typical components of a compiler



Find the bugs!

- What's wrong with the following code?

```
bar(int a, char* s) {...}

int foo() {
  int f[3];
  int i, j, k;
  char q, *p;
  float k;
  bar(f[6], 10, x);
  break;
  i->val = 5;
  q = k + p;
  printf("%s, %s.\n", p, k);
  goto label2;
}
```

- This is all syntactically correct!
- Syntax analysis can only do so much.
- These errors come under the world of **semantics**!
- Semantic analysis identifies such errors and gives out [an *annotated*] syntax tree.

Now for Programming (Assignments)

Java Primer Level 0: Examples on Board

- Variables and methods can be defined only inside classes
- Classes can be instantiated to create objects
- A class can extend another class and inherit its fields
- A class can extend another class and override its methods
- Parent class references can point to child class objects, but not vice versa
- A class can extend only one class but implement any number of interfaces
- All methods are virtual by default — resolved based on the “receiver” object
- In this course, we would need the above, and knowledge of basic collection classes (Vector/ArrayList, HashSet, HashMap)

Let's look at an OO problem

- **Goal:** We want to sum the elements of an integer linked list:

```
interface List {}

class Nil implements List {}

class Cons implements List {
    int head;
    List tail;
}
```

Approach 1: Instance checks and Typecasts

```
List l; ...
int sum = 0;
boolean proceed = true;
while (proceed) {
    if (l instanceof Nil) {
        proceed = false;
    else if (l instanceof Cons) {
        sum = sum + ((Cons) l).head;
        l = ((Cons) l).tail;
    }
}
```

- Bad: Typecasts and instanceof checks are ugly (and expensive).
- Good: No need to touch `Nil` and `Cons` (abstraction principles respected).

Approach 2: Utilize OO style

```
interface List {
   int sum();
}
class Nil implements List {
   public int sum() { return 0; }
}
class Cons implements List {
   int head;
   List tail;
   public int sum() {
      return head + tail.sum();
   }
}
```

- Good: No typecasts and instanceofs.
- Bad: Original classes to be recompiled for each new operation.

Approach 3: Visitor Pattern

- Divide code into an object structure and a `Visitor`.
- Insert an `accept` method in each structure class. Each `accept` method takes a `Visitor` as an argument.
- A `Visitor` contains a `visit` method for each class.

```
interface List {
   void accept(Visitor v);
}
interface Visitor {
   void visit(Nil x);
   void visit(Cons x);
}
```

Approach 3: Visitor Pattern

- The purpose of `accept` methods is to invoke the `visit` method in the `Visitor` that can handle the current object.

```
class Nil implements List {
    public void accept(Visitor v) {
        v.visit(this);
    }
}
class Cons implements List {
    int head;
    List tail;
    public void accept(Visitor v) {
        v.visit(this);
    }
}
```

Approach 3: Visitor Pattern

- The control flows back and forth between the `visit` methods in the `Visitor` and the `accept` methods in the object structure.

```
class SumVisitor implements Visitor {
    int sum = 0;
    public void visit(Nil x) {}
    public void visit(Cons x) {
        sum = sum + x.head;
        x.tail.accept(this);
    }
}
List l;
...
SumVisitor sv = new SumVisitor();
l.accept(sv);
System.out.println(sv.sum);
```

Visitor Pattern: Everything together

```
interface List {
   void accept(Visitor v);
}
interface Visitor {
   void visit(Nil x);
   void visit(Cons x);
}
```

```
class Nil implements List {
   public void accept(Visitor v) {
      v.visit(this);
   }
}
class Cons implements List {
   int head;
   List tail;
   public void accept(Visitor v) {
      v.visit(this);
   }
}
```

```
class SumVisitor implements Visitor {
   int sum = 0;
   public void visit(Nil x) {}
   public void visit(Cons x) {
      sum = sum + x.head;
      x.tail.accept(this);
   }
}
List l;
...
SumVisitor sv = new SumVisitor();
l.accept(sv);
System.out.println(sv.sum);
```

Several compilers use visitor pattern to perform **syntax-directed translation.**

Demo: ASTs and Visitor Pattern

- Getting token values.
- Returning values back to parents.
- Supplying arguments from parents to children.
- Using global data structures.

Interesting ungraded assignment

- Implement lists with the three kinds of approaches seen previously.
- Populate the lists with N elements.
- Print the sum of elements.
- Convince yourself about the programmability with Visitor Pattern.
- Find out which of the three approaches is more efficient:
  - Vary N: 10, 100, 1000, 10000, 100000, 1000000, 10000000.
  - Make a table and list the numbers.
  - Reason about your observations and post on Slack (#classes).
- **Best answer(s) earn *k* PCs.**