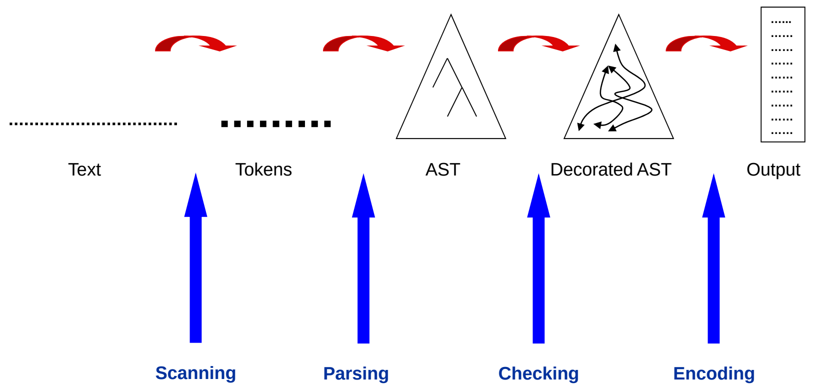
SSE2-PLDE_3

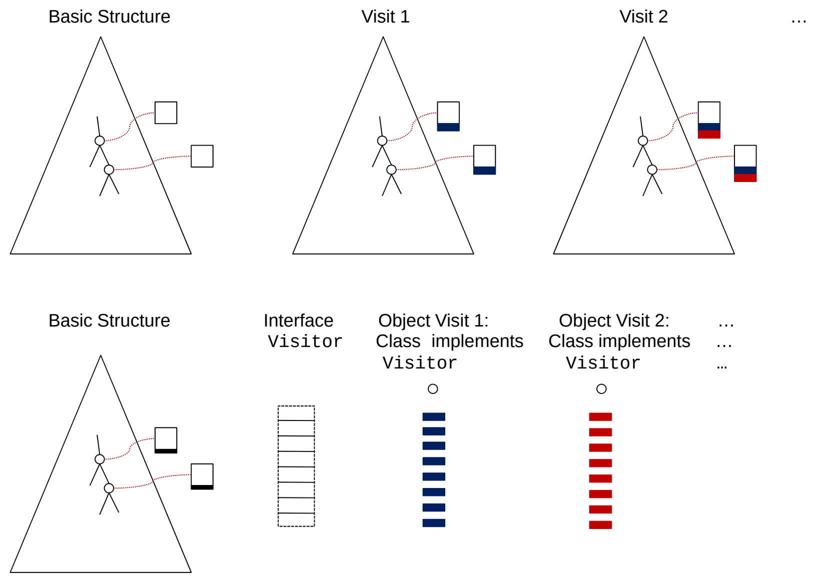
Contents:
- Checking, Encoding, Visitor Design Pattern
- Run-Time Organization
- Code Generation

Literature:
- Watt & Brown:
  - 5.3
  - 6.4.2, 6.7-6.8
  - 7.2

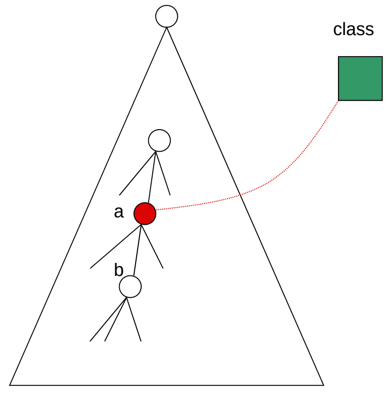
Translation



Visit of Structure

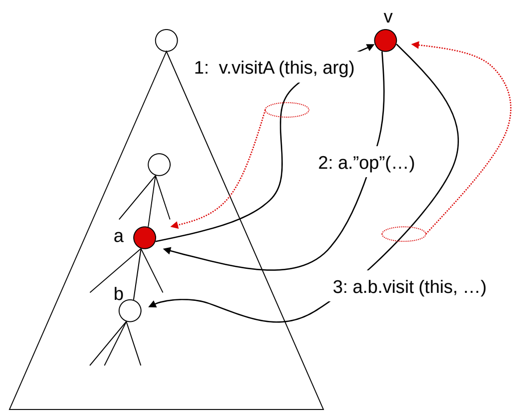


Visit of Structure



1. Method **visit(…)** of object **a** invokes whatever methods “op”(…) of object **a**
2. Method **visit(…)** of **a** invokes any substructure, say object **b**, of object a by invoking method **b.visit(…)** of object **b**

VISITOR



• Assume that method visit (...) of object a is invoked with object v as parameter

• Then method visit(…) of object a invokes visitA(…) of object v with a as parameter

•Then method visitA(…) of object v invokes whatever methods "op"(…) of object a

•Then method visitA(…) of v invokes any subtree, say object b, of object a by invoking method visit(…) of object b (through a.b) with object v as parameter

VISITOR General

Visitor interface: For each concrete AST subclass *A*: visit*A*

```
public interface Visitor {
   …
   public object visitA (A a, Object arg);
   …
}
```

Each concrete AST subclass *A* implements visit method

```
public class A extends … {
  …
  public Object visit (Visitor v, Object arg) {
    return (v.visitA(this, arg);
  }
}
```

Each signature of Visitor is implemented in e.g. Traverse

```
public final class Traverse implements Visitor {
   …
   public object visitA (A a, Object arg){
      … a.op(…); …
      … a.b.visit(this, …); …
      return …
   };
   …
}
```

Mini Triangle VISITOR

Visitor interface:
For each concrete AST subclass *A*: `visitA`

```
public interface Visitor {
   public object visitProgram (Program prog, Object arg);
   …
   public object visitAssignCommand (AssignCommand com, Object arg);
   public object visitCallCommand (CallCommand com, Object arg);
   public object visitSequentialCommand (SequentialCommand com, Object arg);
   public object visitIfCommand (IfCommand com, Object arg);
   public object visitWhileCommand (WhileCommand com, Object arg);
   public object visitLetCommand (LetCommand com, Object arg);
   …
   public object visitIntegerExpression (IntegerExpression expr, Object arg);
   …
   public object visitConstDeclaration (ConstDeclaration decl, Object arg);
   …
}
```

Mini Triangle VISITOR

```
public interface Visitor {
   public object visitProgram (Program prog, Object arg);
   …
}
```

```
public class Program extends AST {
  …
  public Object visit (Visitor v, Object arg) {
    return (v.visitProgram(this, arg);
  }
}
…
```

```
public final class Checker implements Visitor {
   …
   public void Check(Program prog) {
      …
      prog.visit(this, null);
   }
}
```

```
public final class Encoder implements Visitor {
   …
   public void Encode(Program prog) {
      …
      prog.visit(this, null);
   }
}
```

Mini Triangle VISITOR

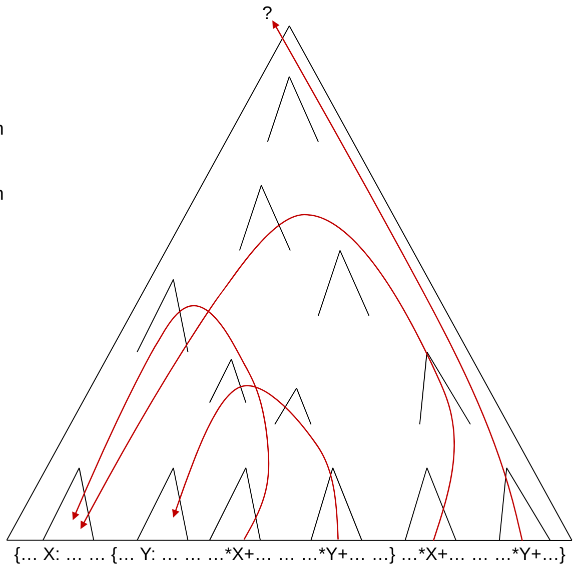
Each concrete AST subclass *A* implements `visit` method

```
public abstract class AST {
  …
  public abstract Object visit (Visitor v, Object arg);
}
```

```
public class Program extends AST {
  …
  public Object visit (Visitor v, Object arg) {
    return (v.visitProgram(this, arg);
  }
}

public class AssignCommand extends Command {
  …
  public Object visit (Visitor v, Object arg) {
    return (v.visitAssignCommand (this, arg);
  }
}

public class IfCommand extends Command {
  …
  public Object visit (Visitor v, Object arg) {
    return (v.visitIfCommand (this, arg);
   }
}
```

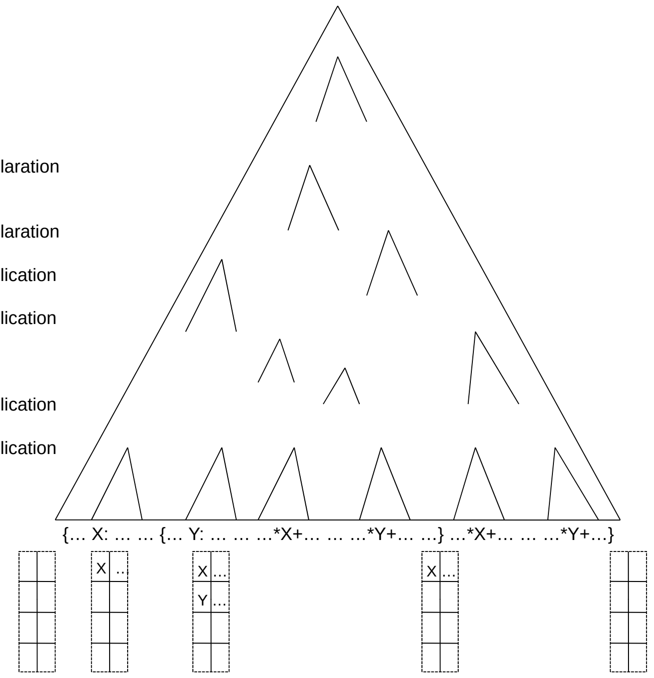
Checking

```
{ …
 X: …                Declaration
 …
         { …
          Y: …       Declaration
          …
          …*X+…      Application
          …
          …*Y+…      Application
          …
         }
 …
 …*X+…               Application
 …
 …*Y+…               Application
 …
}
```



Checking

```
{ …
 X: …                Declaration
 …
         { …
          Y: …       Declaration
          …
          …*X+…      Application
          …
          …*Y+…      Application
          …
         }
 …
 …*X+…               Application
 …
 …*Y+…               Application
 …
}
```



Mini Triangle: Checker

```
public Object visitAssignCommand (AssignCommand com, Object arg) {
   Type vType = (Type) com.V.visit (this, null);
   Type eType = (Type) com.E.visit (this, null);

   … check vType and eType …

   return null;
}

public Object visitSequentialCommand (SequentialCommand com, Object arg) {
   com.C1.visit (this, null);
   com.C2.visit (this, null);
   return null;
}

public Object visitIfCommand (IfCommand com, Object arg) {
   Type eType = (Type) com.E.visit (this, null);

   … check that eType is boolean …

   com.C1.visit (this, null);
   com.C2.visit (this, null);
   return null;
}
```

Mini Triangle: Checker

```
public interface Visitor {
  …
}
```

```
public final class Checker implements Visitor {
   public object visitProgram (Program prog, Object arg){
      …
      return …
   };
   …
   public Object visitSequentialCommand (SequentialCommand com, Object arg) {
      com.C1.visit (this, null);
      com.C2.visit (this, null);
      return null;
   }
   public Object visitIfCommand (IfCommand com, Object arg) {
      Type eType = (Type) com.E.visit (this, null);

      … check that eType is boolean …

      com.C1.visit (this, null);
      com.C2.visit (this, null);
      return null;
   }
   …
   public void Check(Program prog) {
      …
      prog.visit(this, null);
   }
}
```

Mini Triangle: Encoder

```
public Object visitAssignCommand (AssignCommand com, Object arg) {
   com.E.visit (this, arg);

   … encode assignment to com.V …

   return null;
}

public Object visitSequentialCommand (SequentialCommand com, Object arg) {
   com.C1.visit (this, arg);
   com.C2.visit (this, arg);
   return null;
}
```

Mini Triangle: Encoder

```
public interface Visitor {
  …
}
```

```
public final class Encoder implements Visitor {

   public object visitProgram (Program prog, Object arg){
      prog.C.visit(this, arg);
      emit(Instruction.HALTop, 0, 0, 0);
      return null;
   };
   …
   public Object visitSequentialCommand (SequentialCommand com, Object arg) {
      com.C1.visit (this, arg);
      com.C2.visit (this, arg);
      return null;
   }
   …
   public void Encode(Program prog) {
      prog.visit(this, null);
   }
}
```

Mini Triangle: Encode (Backpatching)

```
while E do C

j: JUMP h
g: execute C
h: evaluate E
   JUMPIF(1) g
```

```
repeat C until E

g: execute C
   evaluate E
   JUMPIF(0) g
```

```
while E do C

h: evaluate E
   JUMPIF(0) g
   execute C
j: JUMP h
g:
```

```
(loop C1 exit (E) C2 loop)

g: execute C1
   evaluate E
   JUMPIF(0) h
   execute C2
j: JUMP g
h:
```

Mini Triangle: Encode (Backpatching)

```
public Object visitWhileCommand (WhileCommand com, Object arg) {
   short j = nextInstAddr;
   emit(Instruction.JUMPop, 0, Instruction.CBr, 0);
   short g = nextInstAddr;
   com.C.visit (this, arg);
   short h = nextInstAddr;
   patch(j, h);
   com.E.visit (this, arg);
   emit(Instruction.JUMPIFop, 0, Instruction.CBr, g);
   return null;
}
```

```
while E do C

j: JUMP h
g: execute C
h: evaluate E
   JUMPIF(1) g
```

```
private void patch (short addr, short d)  {
   code[addr].d = d;
}
```

Mini Triangle: Encode (Backpatching)

```
public Object visitIfCommand (IfCommand com, Object arg) {
   com.E.visit (this, arg);
   short i = nextInstAddr;
   emit(Instruction.JUMPIFop, 0, Instruction.CBr, 0);
   com.C1.visit (this, arg);
   short j = nextInstAddr;
   emit(Instruction.JUMPop, 0, Instruction.CBr, 0);
   short g = nextInstAddr;
   patch(i, g);
   com.C2.visit (this, arg);
   short h = nextInstAddr;
   patch(j, h);
   return null;
}
```

```
if E then C1 else C2

   evaluate E
i: JUMPIF(0) g
   execute C1
j: JUMP h
g: execute C2
h:
```

```
private void patch (short addr, short d)  {
   code[addr].d = d;
}
```

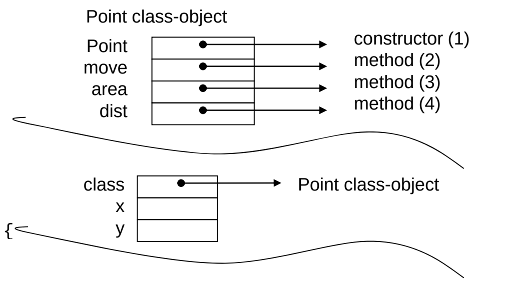
```
public Object visitIfCommand (IfCommand com, Object arg) {
   com.E.visit (this, arg);
   short i = nextInstAddr;
   emit(Instruction.JUMPIFop, 0, Instruction.CBr, 0);
   com.C1.visit (this, arg);
   short j = nextInstAddr;
   emit(Instruction.JUMPop, 0, Instruction.CBr, 0);
   short g = nextInstAddr;
   patch(i, g);
   com.C2.visit (this, arg);
   short h = nextInstAddr;
   patch(j, h);
   return null;
}
```

```
   …
   …code for E
   …
i: JUMPIF(0) g
 …
   …code for C1
   …
j: JUMP h
g: …
   …code for C2
   …
h:
```

Run-time Organization

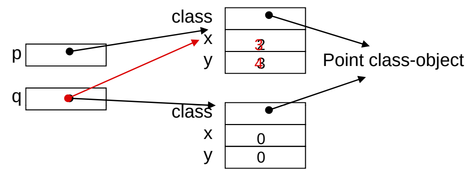
(Object-Oriented Languages)

```
Class Point  {
   protected int x, y;

1  public Point (int x, int y)  {
      this.x = x; this.y = y;
   };

2  public void move (int dx, int dy)  {
      this.x += dx; this.y += dy;
   };

3  public float area ()  {
      return 0.0;
   };

4  public float dist (Point that)  {
      int dx = this.x – that.x;
      int dy = this.y – that.y;
      return Math.sqrt(dx*dx + dy*dy);
   }
}
```

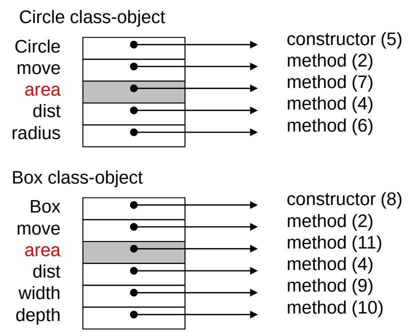


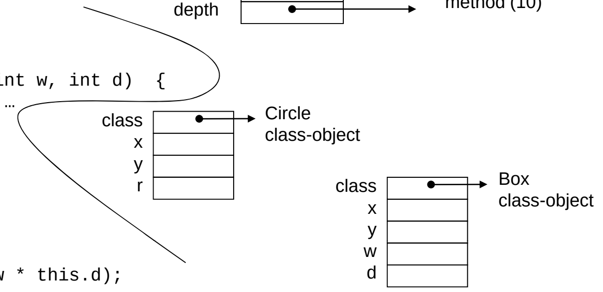
```
Point p = new Point (2, 3);
Point q = new Point (0, 0);
p.move (1, 1);
q.dist(p);
q = p;
```

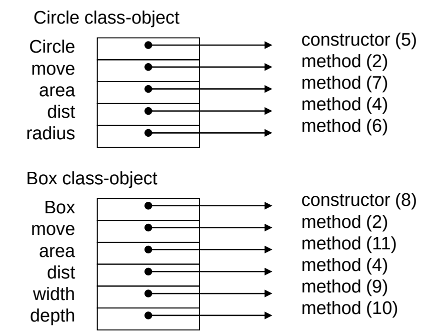
5

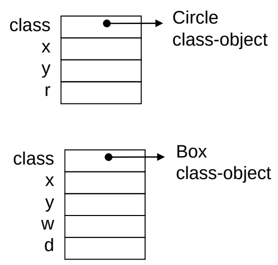


```
Class Circle extends Point  {
   protected int r;

5  public Circle (int x, int y, int r)  {
      this.x = x; this.y = y; this.r = r;
   };

6  public int radius ()  {
      return this.r;
   };

7  public double area ()  {
      double pi = 3.1416;
      return pi * this.r * this.r;
   }
}

Class Box extends Point  {
   protected int w, d;

8  public Box (int x, int y, int w, int d)  {
      this.x = x; this.y = y; …
   };

9  public int width ()  {
      return this.w;
   };
   …
11 public double area ()  {
      return  (double) (this.w * this.d);
   }
}
```

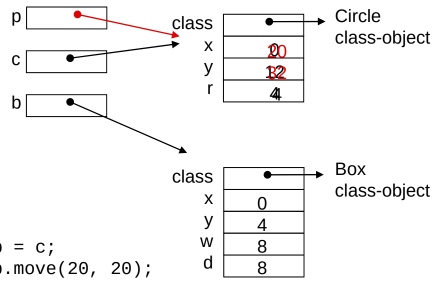




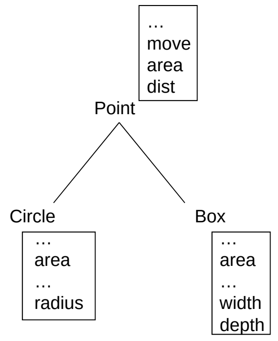




```
Int s = 4;
Point p = null;
Circle c = new Circle (0, 3*s, s);
Box b = new Box (0, s, 2*s, 2*s);
```

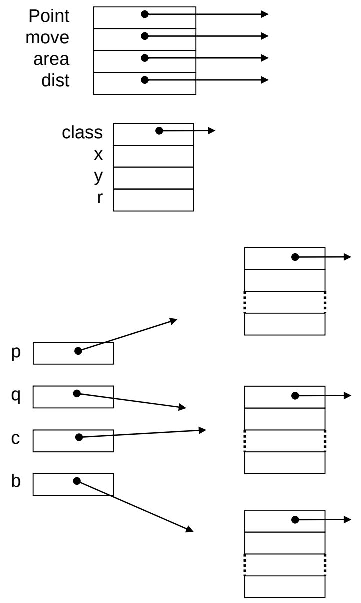


```
p = c;
p.move(20, 20);
```



```
Point p = …
Point q = …

Circle c = …
Box b = …
```

```
p.x;
p.r;
…
p = c;
b = q;
…
c.dist(b);
…
p.area()
p.radius()
```



Point
move
area
dist
class
x
y
r
p
q
c
b