# Program Representations

## Overview

- Abstract Syntax Tree
  - Eclipse JDT
  - Java Model
  - Eclipse JDT AST
- Control Flow Graph
- Program Dependence Graph
- Points-to Graph
- Call Graph



## Abstract Syntax Tree (AST)

- Created by the compiler at the end of syntax analysis phase
- A tree representation for the abstract syntactic structure of source code
  - Node: construct, such as statement, loop
  - Edge: containment relationship
- Different compilers can define different AST representations



## Eclipse JDT

- The Eclipse Java Development Tools project (JDT) provides
  - tools to develop Java application
  - APIs to access, create, and manipulate Java projects' source code
- It provides access to Java source code via two ways: Java Model and Abstract Syntax Tree

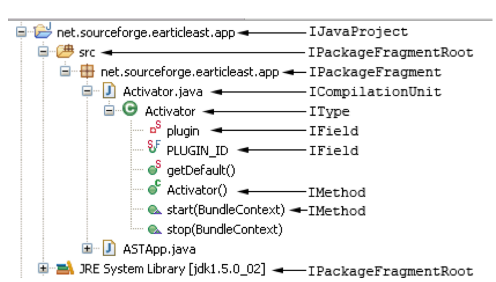


## Java Model

- It is defined in the org.eclipse.jdt.core plug-in
- Each Java project is internally represented in Eclipse as a Java model
- It has a tree structure to represent hierarchical components in a Java project



## The Tree Structure of Java Project[2]



# How do we use Java Model?

- Programmatically parse information from Java Projects
- Create new Java elements
- Automatically manipulate Java source code



# Programmatically Parse Information

```
public Object execute(ExecutionEvent event) throws ExecutionException {
  // Get the root of the workspace
  IWorkspace workspace = ResourcesPlugin.getWorkspace();
  IWorkspaceRoot root = workspace.getRoot();
  // Get all projects in the workspace
  IProject[] projects = root.getProjects();
  // Loop over all projects
  for (IProject project : projects) {
    try {
      printProjectInfo(project);
    } catch (CoreException e) {
      e.printStackTrace();
    }
  }
  return null;
}
```

```
private void printIMethodDetails(IType type) throws JavaModelException {
  IMethod[] methods = type.getMethods();
  for (IMethod method : methods) {

    System.out.println("Method name " + method.getElementName());
    System.out.println("Signature " + method.getSignature());
    System.out.println("Return Type " + method.getReturnType());

  }
}
```



# Create New Java Elements

```
private void createPackage(IProject project) throws JavaModelException {
  IJavaProject javaProject = JavaCore.create(project);
  IFolder folder = project.getFolder("src");
  // folder.create(true, true, null);
  IPackageFragmentRoot srcFolder = javaProject
      .getPackageFragmentRoot(folder);
  IPackageFragment fragment = srcFolder.createPackageFragment(project.getName(), true, nu
ll);
 }
}
```

```
private void changeClasspath(IProject project) throws JavaModelException {
  IJavaProject javaProject = JavaCore.create(project);
  IClasspathEntry[] entries = javaProject.getRawClasspath();
  IClasspathEntry[] newEntries = new IClasspathEntry[entries.length + 1];

  System.arraycopy(entries, 0, newEntries, 0, entries.length);

  // add a new entry using the path to the container
  Path junitPath = new Path("org.eclipse.jdt.junit.JUNIT_CONTAINER/4");
  IClasspathEntry junitEntry = JavaCore
      .newContainerEntry(junitPath);
  newEntries[entries.length] = JavaCore
      .newContainerEntry(junitEntry.getPath());
  javaProject.setRawClasspath(newEntries, null);
}
```

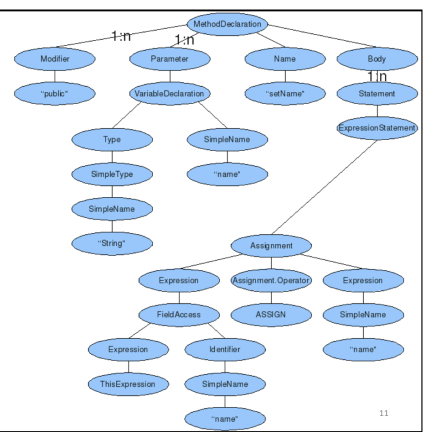


# Why is Java Model important?

- The basis for quick fix and code generation feature in Eclipse
  - generate equals() and hashcode()
  - declare a new class to resolve unresolved type reference
- APIs support structure change, but not statement
- Enabler for automatic programming!



# Eclipse AST[3]






# How do we generate Eclipse AST from source code?

```
protected CompilationUnit parse(ICompilationUnit unit) {
        ASTParser parser = ASTParser.newParser(AST.JLS3);
        parser.setKind(ASTParser.K_COMPILATION_UNIT);
        parser.setSource(unit); // set source
        parser.setResolveBindings(true); // we need bindings later on
        return (CompilationUnit) parser.createAST(null /* IProgressMonitor */); // parse
}
```

## How do we use Eclipse AST?

- Use ASTVisitor to parse any source code information from the AST
- Conduct program analysis based on the AST information
- Manipulate AST to insert/delete code



## Parse Information

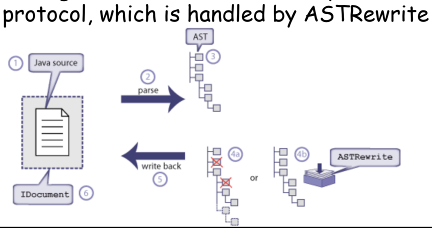
- To get information about AST, you only need to declare a visitor which extends ASTVisitor to define how to visit each AST element

```
public class MethodVisitor extends ASTVisitor {
  List<MethodDeclaration> methods = new ArrayList<MethodDeclaration>();

  @Override
  public boolean visit(MethodDeclaration node) {
    methods.add(node);
    return super.visit(node);
  }

  public List<MethodDeclaration> getMethods() {
    return methods;
  }
}
```



## AST Manipulation[2]

- Two ways to manipulate AST:
  - Directly modifying the AST
  - Noting the modifications in a separate protocol, which is handled by ASTRewrite






## Why is AST important?

- Makes it possible to apply all kinds of syntax-directed translation/ transformation
- Combined with Java model, enable automatic programming
- When mining software repository to understand program changes, program analysis based on AST is the key to automate the process



## Control Flow Graph (CFG)

- A representation, using graph notation, of all paths that might be traversed through a program during its execution



## Formal Representation[5]

- CFG = <V, E, Entry, Exit>, where
  - V = vertices or nodes, representing an instruction or basic block (a group of instructions)
  - E = edges, potentially flow of control, $E \subseteq V \times V$
  - $Entry \in V$, unique program entry $(\forall v \in V)[Entry \xrightarrow{*} v]$
  - $Exit \in V$, unique program exit $(\forall v \in V)[v \xrightarrow{*} Exit]$

## Basic Block

- A maximal sequence of consecutive instructions such that inside the basic block, an execution can only proceed from one instruction to the next
- Single entry, single exit



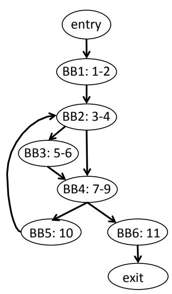
## CFG Example

```
1        A = 4
2        t1 = A * B

3  L1:   t2 = t1/C
4         if t2 < W goto  L2

5        M = t1 * k
6        t3 = M + I

7  L2:   H = I
8         M = t3 – H
9         if t3 >= 0 goto L3

10        goto L1

11 L3:        halt
```

- What are the basic blocks?
- What are the edges between them?



## CFG Example



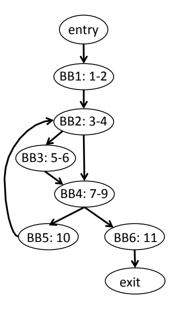



## Why is CFG important?

- A lot of program analysis and abstract representations are built based on it
- In testing scenario, CFG is leveraged to design test cases in order to have enough path/statement coverage



## CFG Used for Selective Testing




- Basic Path Testing
  - Cyclomatic complexity V(G)
    - number of simple decisions + 1
    - number of enclosed areas + 1
  - What are the paths to test?

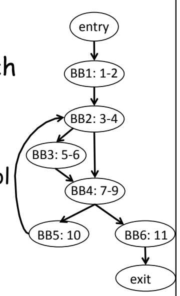


## Program Dependence Graph (PDG)

- A directed graph representing dependencies among code
  - Control dependence
    - A **control depends on** B if B's execution decides whether or not A is executed
  - Data dependence
    - A **data depends on** B if A uses variable defined in B

## Control Dependence Example

- BB3 control depends on BB2 because whether or not BB3 is executed depends on the branch taken at BB2
  - Every block control depends on entry block
  - In most cases, statements control depend on their AST container constructs, such as loop, switch, if. Can you think about cases violating this observation?

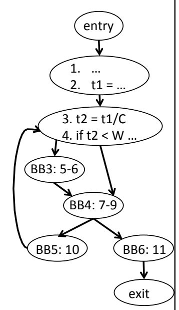



25

## Data Dependence Example

- BB2 data depends on BB1 because BB2 uses the variable t1, whose value is defined by instruction(s) in BB1
  - Which statement does "sum = sum + i" data depend on?

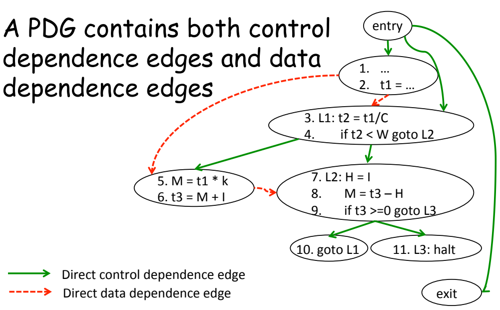
```
sum = 0;
i = 1;
while (i < N) {
   i = i + 1;
   sum = sum + i;
}
```




26

## PDG

- A PDG contains both control dependence edges and data dependence edges




27

## Why is PDG important?

- It demonstrates some program semantics and facilitates program comprehension
  - find bugs, program slicing
- Guide safe program transformations/ optimizations which modify code without compromising dependency relations
  - Automatic parallelism, common subexpression elimination, code motion

28

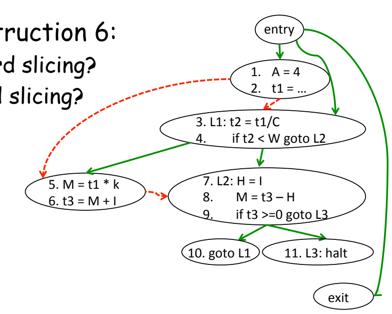
## Program Slicing

- Set of statements that may affect the values at some point of interest
  - data/control dependence relationship
- Backward slicing
  - The statements the current value is dependent on
- Forward slicing
  - The statements which depend on the current value

29

## Example

- t3 at instruction 6:
  - Backward slicing?
  - Forward slicing?

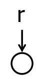



30

## Points-to Graph

- For a program location, for any object reference/pointer, calculate all the possible objects/variables it may/must refer/point to

```
→ r = new C();
  p.f = r;
  t = new C();
  if (…)
     q=p;
  r->f = t;
```

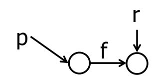



31

## Points-to Graph

- For a program location, for any object reference/pointer, calculate all the possible objects/variables it may/must refer/point to

```
  r = new C();
→ p.f = r;
  t = new C();
  if (…)
     q=p;
  r->f = t;
```

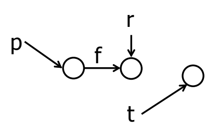



32

## Points-to Graph[4]

- For a program location, for any object reference/pointer, calculate all the possible objects/variables it may/must refer/point to

```
  r = new C();
  p.f = r;
→ t = new C();
  if (…)
     q=p;
  r->f = t;
```

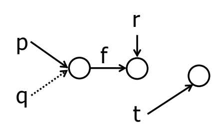



33

## Points-to Graph

- For a program location, for any object reference/pointer, calculate all the possible objects/variables it may/must refer/point to

```
  r = new C();
  p.f = r;
  t = new C();
  if (…)
     q=p;
→ r.f = t;
```

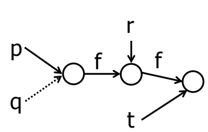



34

## Points-to Graph

- For a program location, for any object reference/pointer, calculate all the possible objects/variables it may/must refer/point to

```
  r = new C();
  p.f = r;
  t = new C();
  if (…)
     q=p;
→ r->f = t;
```




p.f.f and t are aliases

35

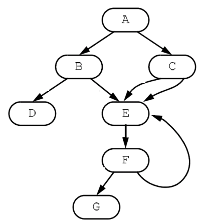
## Why is Points-to Graph important?

- Connect together analyzed program semantics for individual methods
  - Essential to expand intra-procedural analysis to inter-procedural
- Detect consistent usage of resources
  - File open/close, lock/unlock, malloc/free
- Garbage collection

36

## Call Graph

- A directed graph representing caller-callee relationship between methods/ functions
  - Node: methods/functions
  - Edges: calls






## Why is Call Graph important?

- Facilitate program comprehension and optimization
  - When a program crashes, what is the possible calling context?
  - Detect anomalies of program execution