Software Engineering

14. Compilation and Static Analysis | Thomas Thüm | May 17, 2022

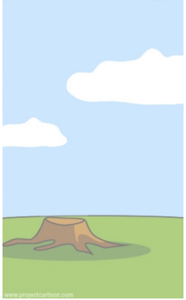
Software Engineering
Programming Languages

ulm university universität uulm

Compilation and Static Analyses

how patches were applied

software evolution

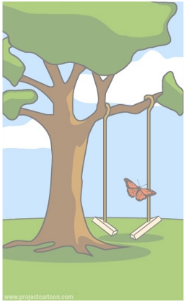


how it was supported

software maintenance

when it was delivered

configuration management



how it performed under load

compilation and static analyses

Lessons Learned

- System building: 3 platforms, requirements on tooling
- Continuous integration: tools, steps
- DevOps
- Further Reading: Sommerville, Chapter 25 Configuration Management and Krypczyk/Bochkor, Chapter 9.3 DevOps

Practice

- Form groups of 2–3 students
- 1. Discuss flavors of continuous integration
- 2. Report one flavor in Moodle

Lecture Overview

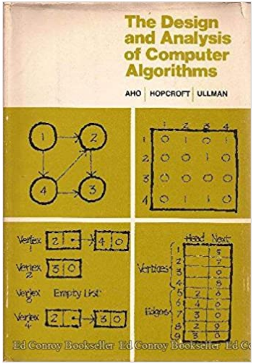
1. Fundamentals on Compilation
2. Misunderstandings on Performance
3. Test-Driven Development & Design by Contract

Lecture Contents

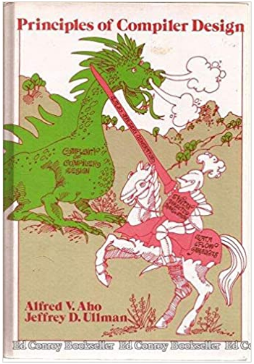
And the 2020 Turing Award Goes To [awards.acm.org]

Alfred V. Aho

Jeffrey D. Ullman



1974



"Dragon Book", 1977

Recap: History of Programming Languages

Languages

[Jones + Krypczyk/Bochkor]

- 1945: first high-level language Plankalkül by Konrad Zuse (compiler written in 1998)
- 1954: first professional high-level language FORTRAN (Formula Translator) by IBM
- 1963: Basic as general-purpose language
- 1959: functional language Lisp
- 1970: first object-oriented lang. Smalltalk-80
- 1970: declarative langugage SQL
- 1971: Pascal by Niklaus Wirth for teaching
- 1974: very common procedural language C
- 1977: logical language Prolog
- 1980: C++ as object-oriented extension of C
- 1990: object-oriented language Java
- 1990: functional language Haskell
- 1991: multi-paradigm language Python
- 1995: scripting language JavaScript

Milestones

[Jones]

- controlling behavior of mechanical devices by wiring or with punchcards (Lochkarten)
- machine languages used during World War II
- assembly languages: distinction between human-readable instructions (source code) and executable instructions (object code)
- birth of compilers and interpreters having a one-to-many mapping between source and object code (opposed to one-to-one mapping in assemblers)
- structured programming pioneered by David Parnas and Edsger Dijkstra
- high-level programming languages: high number of executable for each human-readable instruction
- domain-specific languages, later general-purpose programming languages

Compilation vs Interpretation

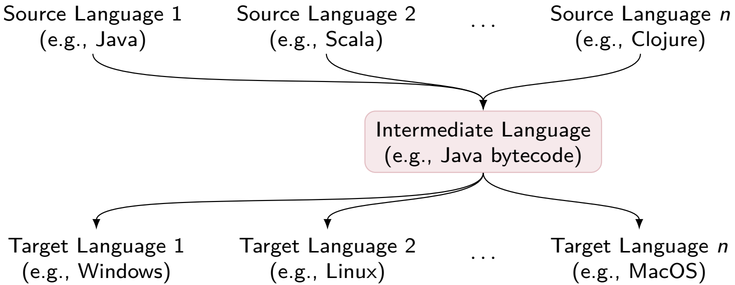
Compilation

- C/C++/Go/Rust/Swift to machine code
- Java/Groovy/Kotlin/Scala/Clojure to Java bytecode

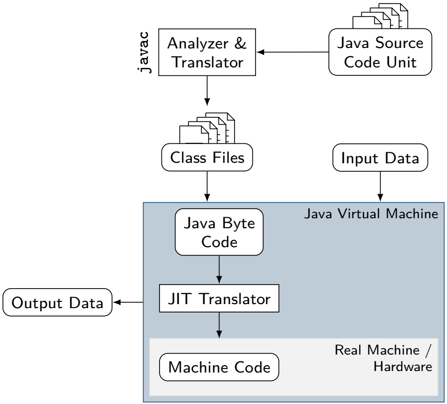
Interpretation

- of source code: Ruby/Python/Perl/PHP/Matlab
- of bytecode: Java Virtual Machine (JVM)

The Power of Intermediate Languages



Compilation and Performance



Goals of Compiler Optimizations

- fast execution
- low memory / energy consumption
- small binaries (fast start/download/updates)
- desired for both compiler (compile time) and compiled program (run time)

Compile Time vs Run Time

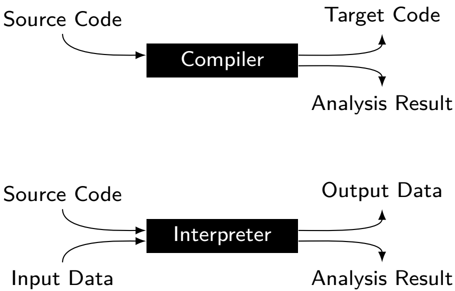
run time: when program or software is executed

compile time: during (ahead-of-time) compilation

Just-in-Time Compilation

- often executed code is compiled at run time
- warm-up time: execution is slower when new code is executed

What Happens for Incorrect Programs?



Warnings and Errors

- errors indicate that compilation failed (target code is incomplete)
- warnings indicate potential problems or that compilation may fail in the future (cf. deprecated methods)

Common Types of Errors

lexical errors, parse error, type error, runtime errors, logical errors

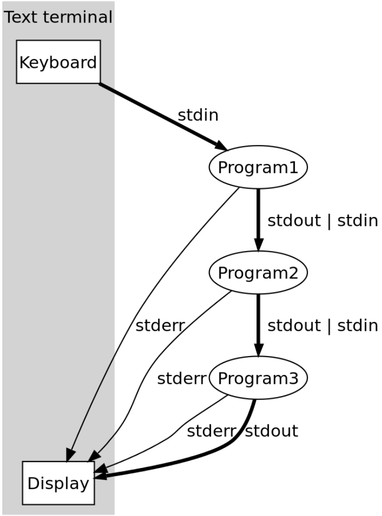
Recap: Pipe-and-Filter Architecture

Pipe-and-Filter Architecture [Sommerville]

- **Problem**: data is processed in numerous processing steps, which are prone to change
- **Idea**: modularization of each processing step into a component
- filter components process a stream of data continously
- pipes transfer data unchanged from filter output to filter input

Pipe Operator in UNIX

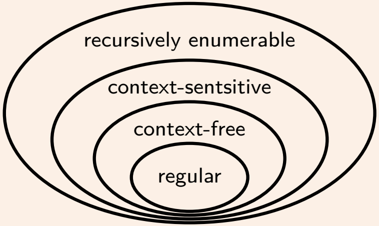
"`ls -al | grep '2020' | grep -v 'Nov' | more`" searches files in a folder from the year 2020 except those from November and delivers the results in pages.

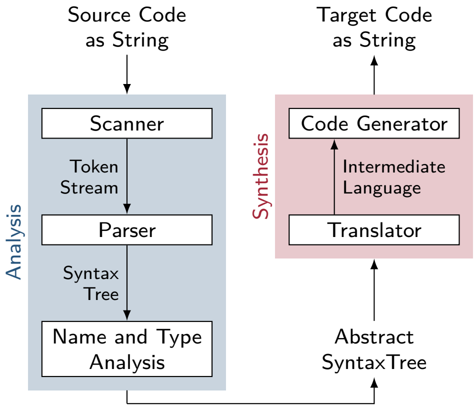


Compiler Architecture

Application of the Chomsky Hierarchy

- scanning/lexing: regular expression for each token class
- parsing: context-free grammars (e.g., a class has fields, methods, and inner classes)
- name and type analysis: context-sensitive analysis (e.g., lookup type for identifier)





Type Safety and Type Correctness

Type Safety

A **type** characterizes properties of program elements, for example:

- a variable can only store particular values
- an expression returns particular values
- an object has a method with a certain signature (name and parameter types)

A **type error** occurs if properties are not met. A program is **type safe** if its execution cannot not lead to type errors.

Type Errors

undefined identifier, assignment of incompatible type, method call with incompatible parameter

Type Correctness

- The language specification defines type rules checked by the compiler (**statically typed language**) or by the interpreter (**dynamically typed language**).
- All type rules together constitute the **type system**.
- A program is **type correct** if it satisfies the type rules.
- A programming language is **strongly typed** if every type correct program is type safe (**weakly typed** otherwise).

In Practice

continuum between strongly (e.g., Java) and weakly typed (e.g., JavaScript) languages

Fundamentals on Compilation

Lessons Learned

- 2020 Turing Award for high-level programming languages
- Compilation vs interpretation vs just-in-time compilation
- Compile time vs run time: goals of optimizations
- Compiler architecture and intermediate languages
- Type safety and correctness

Practice

- Form groups of 2–3 students
- Discussion: What are chances and risks of high-level programming languages (i.e., an increasing gap between high-level instructions and low-level machine code)?
- Add one argument to Moodle

Lecture Contents

Common Misunderstandings

Misunderstanding #1

"I've heard array access is slow.
Now I try to avoid them."

Misunderstanding #2

"I've heard loops are slow.
Now I try to avoid them."

Misunderstanding #3

"I've heard method calls are slow.
Now I try to avoid them."

Misunderstanding #4

"I've heard objects are slow.
Now I try to avoid them."

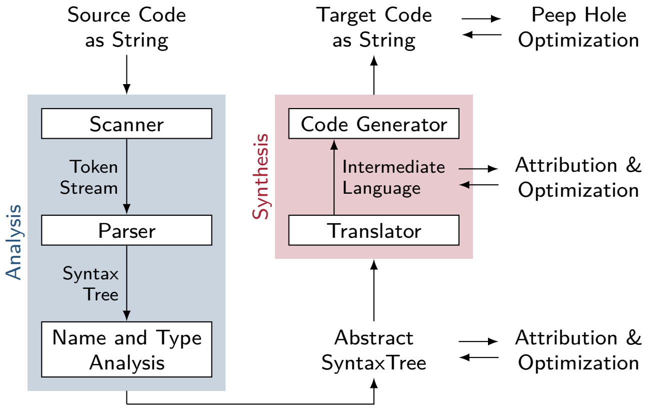
Misunderstanding #5

"I've heard garbage collection is slow.
Now I try to avoid it."

Misunderstanding #6

"I'm new to programming and think that every instruction takes equally long."

Compiler Optimizations



Kinds of Optimizations

- machine-dependent optimizations: exploit properties of a particular machine
- machine-independent optimizations: applicable to several machines
- local optimizations: e.g., switch order of two statements
- intra-procedural optimizations: affect only one method
- inter-procedural optimizations: affect several methods or require global knowledge

No Array Access?

Misunderstanding #1

"I've heard array access is slow.
Now I try to avoid them."

Tasks for Array Access a[b]

- evaluate the expression b (could be arbitrarily complex)
- compute offset = b * size of each field
- compute memory location = position of a + offset
- access memory location
- only for objects: use value as memory location

Example

```
a[b.n()] = a[b.n()-1] * a[b.n()-1];
```

Simplified Example

Assumption 1: method n has no side-effects
Assumption 2: method n cannot be overridden (e.g., a private method)

```
int n = b.n();
a[n] = a[n-1] * a[n-1];
```

Note

arrays are very common, compilers and hardware have plenty of optimizations

No Loops?

Misunderstanding #2

"I've heard loops are slow.
Now I try to avoid them."

Tasks for Loops

- create and initialized loop variable
- check loop condition
- **run loop body**
- increment loop variable
- repeat from check loop condition

Example Loop

for (int i = 0; i++; i < 3) { a[i] = i; }

Loop Unrolling

a[0] = 0; a[1] = 1; a[2] = 2;

Do Not Avoid Loops!

smells: duplicated code, long method
compiler optimization: loop unrolling (if number of runs is statically known and small enough)

No Method Calls?

Misunderstanding #3

“I’ve heard method calls are slow.
Now I try to avoid them.”

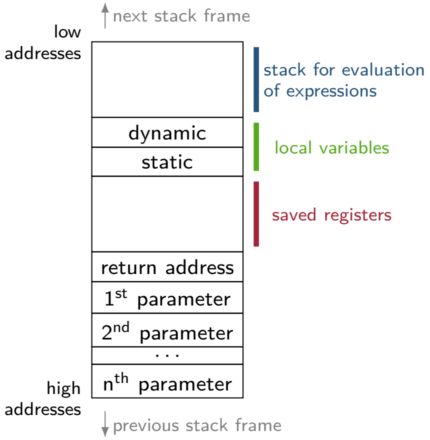
Tasks for Each Method Call

- pass parameters and return address
- save registers
- run method body, pass return value
- restore registers, return to caller

Do Not Avoid Method Calls!

smells: duplicated code, long method
refactoring: extract method
compiler optimization: method inlining

Memory Layout: The Method Stack



Method Inlining in Practice

What is the output of this program?

```
class A {
    public static void main(String[] args) {
        A x = new B();
        Object y = new String();
        System.out.print(x.m(y));
    }
    int m(Object o) { return 1; }
    int m(String s) { return 2; }
}
class B extends A {
    int m(Object o) { return 3; }
    int m(String s) { return 4; }
}
```

Hint: Java uses static dispatch for overloading (y) and dynamic dispatch for overriding (x – required for inheritance)

```
class A {
    public static void main(String[] args) {
        System.out.print(3);
    }
}
```

No Objects?

Misunderstanding #4

"I've heard objects are slow.
Now I try to avoid them."

Memory Layout for Objects

- if life time not bound to a method, cannot be stored on method stack
- stored in free position on the heap
- pointer and derefence required
- need to store class information
- pointer to virtual method table for dynamic dispatch
- all fields, even inherited fields

Do Not Avoid Objects!

unless performance or memory consumption is a problem, then identify which objects (a) consume most memory and (b) have the shortest life time

Thomas' Experience with LinkedList in Java

LinkedList can be problematic, as there is a list object for every entry in the list and list manipulations lead to new list objects

solution: use arrays or ArrayList instead

large speed-ups in FeatureIDE: github.com

also reported by others: stackoverflow.com

No Garbage Collection?

Misunderstanding #5

"I've heard garbage collection is slow.
Now I try to avoid it."

Garbage Collection

- find objects not referenced anymore
- free memory space assigned by them
- algorithms: reference counting, mark-and-sweep, copying collection
- causes random delays

Reasons for Automatic Memory Management

- simplified programming, programs
- fewer memory leaks
- improved safety, security, reliability

Avoiding Garbage Collection

- switch to a language with manual memory allocation (e.g., C/C++)
- deactivate garbage collection completely (only for programs with short runtime and low memory consumption)
- web services: do garbage collection when service is idle

Speed = Instructions per Minute?

Misunderstanding #6

"I'm new to programming and think that every instruction takes equally long."

Misunderstanding #7

"I'm new to computers / smartphones and think that every instruction takes equally long."

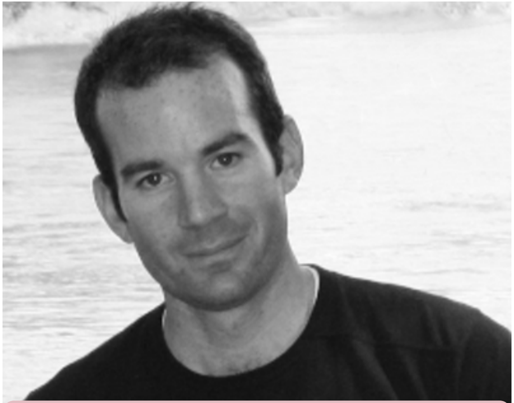
Hint

Use all language constructs and let compilers do their job.

If performance is not sufficient, inspect bottleneck.

Only reduce code quality in favor of performance where necessary.

Recap: Simplicity over Performance



Wes Dyer [twitter.com]

"Make it correct, make it clear, make it concise, make it fast. In that order."

Joshua J. Bloch (born 1961) [twitter.com]

"The cleaner and nicer the program, the faster it's going to run. And if it doesn't, it'll be easy to make it fast."

Misunderstandings on Performance

Lessons Learned

- Compiler optimizations
- What are common misunderstandings about performance?
- Why are array, loops, method calls, objects, and garbage collection slow?
- What is the connection between compiler optimizations, smells, and refactorings?

Practice

- Thomas decides about your opinion
- What is better large or small classes?
- Long or short methods?
- One- or multidimensional arrays?
- Clean or fast programs?

Lecture Contents

Type Error

Is the program type correct?

```
class Node {}
class Edge {
  Node first, second;
  Edge(Node first, Node second) {
    this.first = first;
    this.second = second;
  }
  boolean equals(Edge e) {
    return first.equals(e.first) && second.equals(e.first);
  }
  public static void main(String[] args) {
    Node a = new Node();
    Node b = new Node();
    System.out.println(new Edge(a, b).equals());
  }
}
```

Error message by the Java compiler

"The method equals(Edge) in the type Edge is not applicable for the arguments ()"

No – type error for a method call

```
class Node {}
class Edge {
  Node first, second;
  Edge(Node first, Node second) {
    this.first = first;
    this.second = second;
  }
  boolean equals(Edge e) {
    return first.equals(e.first) && second.equals(e.first);
  }
  public static void main(String[] args) {
    Node a = new Node();
    Node b = new Node();
    System.out.println(new Edge(a, b).equals());
  }
}
```

Runtime Error

NullPointerException

not detected by the Java compiler, but when executing the program

Is the revised program type correct?

```
class Node {}
class Edge {
  Node first, second;
  Edge(Node first, Node second) {
    this.first = first;
    this.second = second;
  }
  boolean equals(Edge e) {
    return first.equals(e.first) && second.equals(e.first);
  }
  public static void main(String[] args) {
    Node a = new Node();
    Node b = new Node();
    System.out.println(new Edge(a, b).equals(null));
  }
}
```

Yes, it is type correct

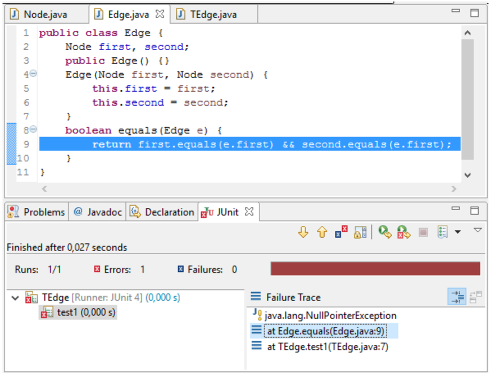
```
class Node {}
class Edge {
  Node first, second;
  Edge(Node first, Node second) {
    this.first = first;
    this.second = second;
  }
  boolean equals(Edge e) {
    return first.equals(e.first) && second.equals(e.first);
  }
  public static void main(String[] args) {
    Node a = new Node();
    Node b = new Node();
    System.out.println(new Edge(a, b).equals(null));
  }
}
```

Unit Testing with JUnit

Test-driven development: write a test first

```
class Node {}
class Edge {
  Node first, second;
  Edge(Node first, Node second) {
    this.first = first; this.second = second;
  }
  boolean equals(Edge e) {
    return first.equals(e.first) && second.equals(e.first);
  }
}
import org.junit.Test;
public class TEdge {
  @Test
  public void test1() {
    Node a = new Node();
    Node b = new Node();
    System.out.println(new Edge(a, b).equals(null));
  }
}
```

JUnit in Eclipse: test is supposed to fail

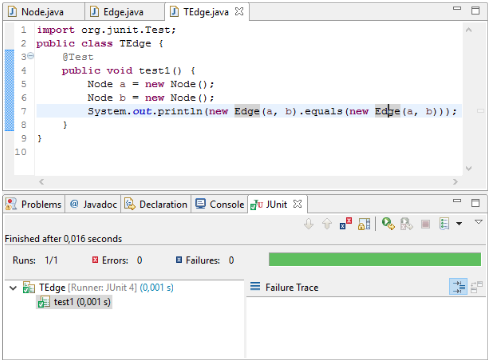


Unit Testing with JUnit

Test-driven development: fix code afterwards

```
class Node {}
class Edge {
  Node first, second;
  Edge(Node first, Node second) {
    this.first = first; this.second = second;
  }
  boolean equals(Edge e) {
    return first.equals(e.first) && second.equals(e.first);
  }
}
import org.junit.Test;
public class TEdge {
  @Test
  public void test1() {
    Node a = new Node();
    Node b = new Node();
    System.out.println(new Edge(a, b).equals(new Edge(a, b)));
  }
}
```

JUnit in Eclipse: check whether test still fails



Logical Error

Oops: unexpected output false

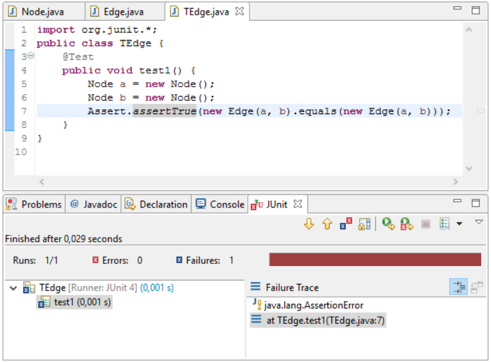
```
class Node {}
class Edge {
  Node first, second;
  Edge(Node first, Node second) {
    this.first = first; this.second = second;
  }
  boolean equals(Edge e) {
    return first.equals(e.first) && second.equals(e.first);
  }
}
import org.junit.Test;
public class TEdge {
  @Test
  public void test1() {
    Node a = new Node();
    Node b = new Node();
    System.out.println(new Edge(a, b).equals(new Edge(a, b)));
  }
}
```

Using assertions in JUnit tests

```
class Node {}
class Edge {
  Node first, second;
  Edge(Node first, Node second) {
    this.first = first; this.second = second;
  }
  boolean equals(Edge e) {
      return first.equals(e.first) && second.equals(e.first);
  }
}
import org.junit.*;
public class TEdge {
  @Test
  public void test1() {
    Node a = new Node();
    Node b = new Node();
    Assert.assertTrue(new Edge(a, b).equals(new Edge(a, b)));
  }
}
```

Runtime Assertions

JUnit failure in Eclipse



Violated assertion

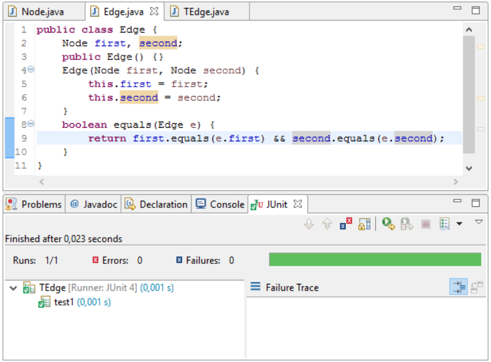
```
class Node {}
class Edge {
  Node first, second;
  Edge(Node first, Node second) {
    this.first = first; this.second = second;
  }
  boolean equals(Edge e) {
    return first.equals(e.first) && second.equals(e.first);
  }
}
import org.junit.*;
public class TEdge {
  @Test
  public void test1() {
    Node a = new Node();
    Node b = new Node();
    Assert.assertTrue(new Edge(a, b).equals(new Edge(a, b)));
  }
}
```

Runtime Assertions

All type and runtime errors fixed?

```
class Node {}
class Edge {
  Node first, second;
  Edge(Node first, Node second) {
    this.first = first; this.second = second;
  }
  boolean equals(Edge e) {
    return first.equals(e.first) && second.equals(e.second);
  }
}
import org.junit.*;
public class TEdge {
  @Test
  public void test1() {
    Node a = new Node();
    Node b = new Node();
    Assert.assertTrue(new Edge(a, b).equals(new Edge(a, b)));
  }
}
```

JUnit is happy again. What does it mean?



Design by Contract

Motivation

- code pollution by defensive programming
- specification far away from code: in tests or separate documents
- informal specification (e.g., JavaDoc) often outdated, only limited automated checks

Design by Contract

- code-level specification, formal specification
- assertions on methods: preconditions, postconditions, assignable locations
- assertions on classes: class invariants
- for example: Java Modeling Language (JML)
- generation of documentation (jmldoc), runtime assertions (jmlc), unit tests (jmlunit), for formal verification (KeY), ...

Class invariants, preconditions, and postconditions

```java
class Node {}
class Edge {
  //@ invariant first != null && second != null;
  Node first, second;
  Edge(Node first, Node second) {
    this.first = first;
    this.second = second;
  }
  /*@ requires e != null;
    @ ensures \result <==> first.equals(e.first) &&
    @ second.equals(e.second); @*/
  boolean equals(Edge e) {
    return first.equals(e.first) && second.equals(e.first);
  }
  public static void main(String[] args) {
    Node a = new Node();
    Node b = new Node();
    System.out.println(new Edge(a, b).equals(null));
  }
}
```

Generated Assertions and Blame Assignment

precondition checked at runtime

```
/*@ requires e != null; @*/
```

callee

```
boolean equals(Edge e) {
  return first.equals(e.first)
    && second.equals(e.first);
}
```

caller

```
static void main(String[] args) {
  Node a = new Node();
  Node b = new Node();
  System.out.println(
    new Edge(a, b).equals(null));
}
```

```
/*@ ensures \result <==>
  @ first.equals(e.first) &&
  @ second.equals(e.second); @*/
```

postcondition checked at runtime

Behavioral Subtyping

```
class Edge {
  //@ invariant first != null && second != null;
  Node first, second;
  /*@ requires e != null;
    @ ensures \result ==> first.equals(e.first) &&
    @   second.equals(e.second); @*/
  boolean equals(Edge e) {
    return first.equals(e.first) &&
      second.equals(e.second);
  }
  [...]
}
```

```
new Edge([...]).equals([...]);
```

```
Edge e = new WeightedEdge([...]);
e.equals([...]);
```

```
new WeightedEdge([...]).equals([...]);
```

```
class WeightedEdge extends Edge {
  Integer weight = 0;
  /*@ also
    @ requires e != null && weight != null;
    @ ensures \result ==> weight == e.weight; @*/
  boolean equals(WeightedEdge e) {
    return super(e) && weight.equals(e.weight);
  }
  [...]
}
```

Test-Driven Development & Design by Contract

Lessons Learned

- Test-driven development
- Examples for type, runtime, and logical errors
- Examples for JUnit and assertions
- Design by contract: class invariants, preconditions, postconditions
- Blame assignment, behavioral subtyping
- Java Modeling Language

Practice

- Quiz with examples for parse error, type errors, runtime errors, logical errors
- Post questions to Moodle, if any