# CS 320: Principles of Programming Languages

Mark P Jones and Andrew Tolmach
Portland State University

Spring 2019

Week 7: Tradeoffs in language design: Java, C/C++, and static analysis

---

## A short history of the automobile

Hybrid

- Modern cars are:
  - Faster
  - Safer
  - More comfortable
  - More efficient
  - More reliable
  - More capable
  - …
- Unsurprisingly, most drivers today drive modern cars

ectric

Capacity

Speed

1900 1920 1940 1960 1980 2000 2020

(Images via Wikipedia, subject to Creative Commons and Public Domain licenses)



---

## A short history of programming languages

Simula Java

Lisp

Fortran

COBOL

- Modern programming languages are:
  - Higher-level
  - Feature rich
  - Type safe
  - Memory safe
  - Less error prone
  - Well-designed
  - Well-defined
  - …
- Surprisingly, most systems programmers today are still using C …

Swift

Go

PHP

1955 1965 1975 1985 1995 2005 2015



---

## C is great … what more could you want?

- Programming in C gives systems developers:
  - Good (usually predictable) performance characteristics
  - Low-level access to hardware when needed
  - A familiar and well-established notation for writing imperative programs that will get the job done
- What can you do in modern languages that you can't already do with C?
- Need objects? Then you can use C++ or Objective C!
- What could possibly go wrong?



---

Impact: An application may be able to execute arbitrary code with kernel privileges

Description: Multiple memory corruption issues were addressed through improved input validation.

Impact: An application may be able to exec

Description: A use after free issue was add management.

Impact: An application may be able to exec

Description: A null pointer dereference was addressed through improved input validation.

Impact: A local user may be able to gain root privileges

Description: A type confusion issue was addressed through improved memory handling.

Impact: An application may be able to execute arbitrary code

Description: An out-of-bounds write issue was addressed by removing the vulnerable code.

Could a different language make it **impossible** to write programs with errors like these ?



---

## From C/C++ to Java

THE C PROGRAMMING LANGUAGE

Java

## Java

- Conceived as an improved version of C++
  - Safe
  - Portable
  - Simple object model
- Microsoft's C# is very similar
- JavaScript is scarcely related (despite the name)
- Initial enthusiasm about running "Java applets" in webpages
- Widely used today in many areas, from Android smart phones to Big Data server farms



## A little history

- Originally developed by James Gosling at Sun Microsystems (now Oracle) beginning in 1995, open-source since ~2006
- Originally called "oak" and designed for use in "interactive TV"
- The language has had many revisions, identified by JDK version numbers
  - For this course, any version ≥ 8 should be fine
  - Confusingly, version n is sometimes called version 1.n



## Resources for learning about Java

- Oracle documentation (for latest version):
  - https://docs.oracle.com/en/java/javase/11/
  - Library API specification
  - Tools reference
  - Java Language Specification (not an easy read!)
  - Tutorials (not great for beginners)
- Books
  - Eckel, Thinking in Java, 4th ed. (free on web)
  - Arnold, Gosling, Holmes, The Java Programming Language, 4th ed.
- Many, many other books and tutorials (check the web …)



strong static typing

**Safety** no unchecked runtime errors

garbage collection

**Simplicity**

reference semantics for objects

primitive type behaviors are fully specified

**Portability**

bytecode intermediate representation



## Program safety: different kinds of errors

- Many bugs involve invalid use of the programming language
- Useful categories of language errors:
  - Static errors are checked before execution begins
    - syntax errors, type errors, etc.
  - Checked runtime errors are detected during execution
  - Unchecked runtime errors might not be detected at all
- Highly desirable to detect bugs
  - as early as possible
  - as precisely as possible
- Java has no unchecked runtime errors (but C/C++ do!)



## Why is safety is so important?

- Example: **Buffer overflow attacks** occur when malicious software exploits buggy software that writes beyond the intended bounds of an array or object
  - For example, depending on the location of the object, an attacker might fool the program into interpreting data as instructions to be executed
- Even decades after the danger of buffer overflows became apparent, these attacks are still one of the most common causes of exploits on today's systems
- One simple solution: use safe languages (like Java, Python, Haskell, …) instead of unsafe ones like C or C++

## Question

syntax vs. semantics

```
static void f(int n) {
  int a[] = {0,1,2,3,4};
  a[n] = n;
}
```

What happens if we evaluate the function call `f(5)` ?

Answer: It depends on what language this code is in!

In Java: this raises an `arrayIndexOutOfBoundsException` which, if not caught, will halt the program with a polite message and a stack traceback.

In C/C++, this is an "Undefined Behavior" and so the program might do…*anything at all!*



## Java primitive types: numerics

A small collection of types are completely "built-in." Unlike in C/C++, sizes are the same on **all** platforms

`byte` (8 bits, signed)

`short` (16 bits, signed)

`int` (32 bits, signed)

`long` (64 bits, signed)

`char` (16 bits, unsigned — uses Unicode representations)

`float` (32 bits — IEEE format)

`double` (64 bits — IEEE format)



## Arithmetic and conversions

- Integer arithmetic is always performed in 32 bits, unless a long operand is involved, in which case it is done in 64 bits
- Values of a type are automatically promoted to a type with a larger range when needed, e.g.,

```
byte b = 32;
long i = b;
double d = i;
```

- But conversions to a type with a smaller range require explicit casts, e.g.,

```
b = (byte) (i+1);
i = (long) d;
```

- … and these may produce an approximate (but well-defined!) result



## Primitive type: `boolean`

- Booleans are *not* integers! They form a distinct type `boolean` with:
  - two literal values `true` and `false`
  - operators: `&&  ||  &  |  ^  !` (the same as in C/C++)
- You cannot do arithmetic on `boolean` values, nor cast between them and numbers
- Booleans are used to govern `if`, `for`, `do`, and `while` statements, as in C/C++



## C/C++ types are less portable and less safe

- Range and precision of basic types is implementation-dependent and varies among hardware platforms
- Coercions between types may happen silently even when the overall magnitude of the number is lost
- Many more casts are allowed, e.g., between integers and pointers, with implementation-dependent results
- Some operations are deliberately under-specified
  - e.g., signed integer overflow is an Undefined Behavior
  - not only is the result unspecified — the compiler is allowed to assume that this kind of overflow will never happen!



## Objects

- Every value in Java that does not belong to a primitive type is an **object**
- Each object is an instance of some **class**, which is much like a C++ class
  - Class definitions can contain **fields** and **methods** (i.e., associated functions that can refer to a specific objects fields via the special `this` parameter)
  - **Constructors** are a special kind of method used to create new instances; they typically initialize the field values
- Each instance object contains its own copy of each field (except for static fields — more to come ...)

## Access to object fields

- As in C++, methods (including constructors) can refer to the fields of the object on which they were invoked
- Unless explicitly restricted, fields can also be read or written from outside the class definition, using “dot notation”
  - There are possible several kinds of restrictions; the details are similar but not identical to C++
- Like structs in C and structs/classes in C++, the Java class is a mechanism for gathering together a labeled group of data items of various types. (i.e., it is a product / record type)



## Example using objects

```
class Point { // represents points in the plane
   int x; int y; // coordinates of the point

   Point(int xi, int yi){ // constructor
     x = xi; y = yi;
   }

   void trx (int dx) {// method
     x += dx;
   }
}
…
Point p = new Point(3,4);  // create new point p
p.trx(7);               // modify p’s x value
int z = p.x + p.y       // extract current values
                        // returns z = 14
```



## Heap management is automated

- Object are always* **heap-allocated**, i.e., `new` acts much as it does in C++ or like `malloc()` in C
- Objects are *never* explicitly deallocated
  - From the programmer’s perspective, they live forever!
  - Underneath, the Java runtime system performs **garbage collection**, which is the automatic deallocation of objects when they are no longer pointed to from anywhere in the running program
- Object addresses are **abstract**; you cannot do pointer arithmetic on them, convert them to/from other types, etc.

*Well, almost always. Clever compilers may be able to avoid the cost of heap allocation in certain special cases. Since addresses are abstract, you can’t really tell.



## Heap safety

- The combination of garbage collection and pointer abstraction makes Java a **heap-safe** language, unlike C/C++
  - Programmers do not have to worry about when to deallocate objects.
  - It is impossible to have a **dangling pointer** (a pointer that still points to an address even after the object there has been deallocated)
  - **Space leaks** (objects that are still allocated but no longer pointed to) are also impossible
    - Although it is still possible for programmers to hold onto pointers, and hence objects, longer than necessary
- This has *huge* benefits for debugging



## Object values are references

- Variables or fields of object type always contain references (i.e., pointers) to objects, rather than the objects themselves
- The Java declaration `Point p`

  ... is like the C++ declaration `Point *p`
- Similarly, the Java notation `p.x`

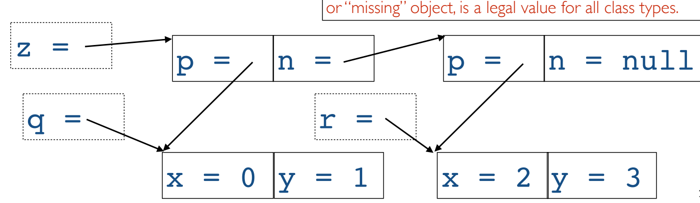
  ... is like the C++ expressions `(*p).x` or `p->x`
- Unlike in C++, there is simply no way to declare storage for the object itself (e.g., on the stack or inside another object)



## Object values are references

```
class Link {
  Point p;
  Link n;
  Link (Point pi, Link ni) { p = pi; n = ni;}
}
Point q = new Point(0,1);
Point r = new Point(2,3);
Link z = new Link(q, new Link(r, null));
```

Note: The special value `null`, representing an “empty” or “missing” object, is a legal value for all class types.



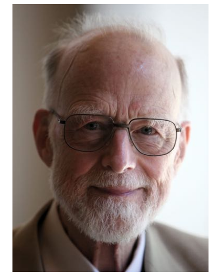
## Null pointer dereferences

- C/C++: a value of type `T*` is a pointer to an object of type `T`
  Java: a value of type `T` is a reference to an object of that type
- But this may be a lie!
- In C++/Java, any such value can be `null`, in which case it does NOT point to an object of type `T`
- Attempting to read or write the value pointed to by a `null` pointer is called a "**null pointer dereference**":
  - In C/C++: undefined behavior, often results in system crashes, vulnerabilities, or memory corruption
  - In Java: a runtime exception, typically terminates program
  - Other languages have distinct types for pointers (which can be null) and references (which cannot): null pointer dereferences can then be detected during static analysis!



## The "Billion Dollar Mistake"

"I call it my billion-dollar mistake…At that time, I was designing the first comprehensive type system for references in an object-oriented language. My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."



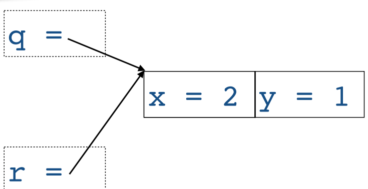
Tony Hoare, reflecting (in 2009) on his invention of the null pointer in ALGOL W (in 1965)



## Copying is shallow

Assigning an object variable just assigns a pointer!

```
Point q = new Point(0,1);
Point r = q;
q.x = 2;
// now q.x == r.x == 2
```




To copy the **contents** of an object, we must **clone** it (i.e., copy its fields one at a time)



## Java always uses call-by-value

- Java method parameters are *always* passed "by value"
  - At entry to the method, a new variable is created for each parameter and initialized to contain a copy of the argument's value
  - But remember: if the parameter is an object, its value will actually be a *pointer* to the object, and copying the value just gives us another pointer to that object
- Unlike C++, there are no by-reference (`&`) parameters
- Unlike C/C++, there is no way to take the address of a local variable and pass it as a pointer value



## Object memory model is simpler than C++

- No requirement to think about object "ownership"
- No need for `delete()` or `free()`
- No need for destructors, copy constructors, move constructors (C++11), etc.
- Parameters and local variables are truly local: their values can only be changed by code within the method
- Downsides:
  - GC may be less efficient than manual memory management (typically ~10% slower)
  - We have less precise control over memory layout and pointer manipulation



## Static members and methods

- Both fields and methods of a class can be declared static

```
class Stuff {
  static int counter = 0;
  static sqr(int x) { return x*x; }
}
Stuff.counter++;
int z = Stuff.sqr(33);
```

- A static field has only one copy, no matter how many objects of the class are created (so it is like a global variable)
- A static method has no associated object; it operates only on its arguments and on static fields
- Static fields and methods are referenced using dot notation, where the thing before the dot is a class name rather than an expression denoting an object

## Strings

- Strings are (almost ordinary) objects of library class `String`
- They are **immutable**, i.e., their contents never changes
- Strings are *not* arrays (or lists) of characters
- We can use class methods to examine characters in the string
- There is another library class `StringBuilder` for handling mutable sequences of characters
- There is some special language-level support for strings:
  - Literal constructors: `"abc"` creates a new `String` object
  - Concatenation: `"2+" + (1+1)` evaluates to `"2+2"`



## Arrays

- Arrays are (slightly special) objects!
- Each array contains elements of some primitive type or class, e.g., `int[]`, `char[]`, `String[]`, `int[][]`
- For compatibility with C/C++, can declare array variables in two equivalent ways: `int[] a; int b[];`
- As with other objects, an array variable is just a reference to an array; to create the actual array (with contents) we must use new: `int[] a = new int[10];`
- or an explicit initializer (which does a `new` under the hood):

```
int[] a = {0,1,2,3,4,5,6,7,8,9};
```



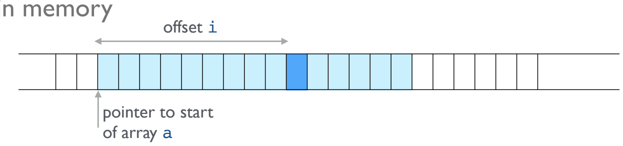
## More arrays

- The length of an array is permanently fixed when the array is created; it can be retrieved using the built-in final instance variable `length`
- As in C/C++:
  - all arrays are indexed from 0
  - multi-dimensional arrays are just one-dimensional arrays containing (pointers to) arrays as elements
- Each load or store on the array is checked against the array bounds; out-of-bounds indices cause an **exception** to be raised



## Arrays and out of bounds indexes:

- Arrays are collections of values stored in contiguous locations in memory




- Address of `a[i]` = start address of `a` + `i`*(size of element)
- Simple, fast, … and dangerous!
- If `i` is not a valid index (an "out of bounds index"), then an attempt to access `a[i]` could lead to a system crash, memory corruption, buffer overflows, …
- A common path to "arbitrary code execution"



## Array bounds checking

- The designers of C knew that this was a potential problem … but chose not to address it in the language design:
  - Performance concerns: We would need to store a length field in every array and check for valid indexes at runtime
- The designers of Java knew that this was a potential problem … and chose to address it in the language design:
  - Safety mechanisms: We can store a length field in every array and check for valid indexes at runtime
- Designers of other languages: can we design a language in which out of bounds errors are caught as compile time errors?
  - Efficient and safe ... but reduced expressivity



## Expression and statements

- These are mostly the same as in C/C++
- However, Java has no `goto` statement

But you never use this anyway, right??

- Java does have a labeled form of the `break` statement, which transfers control to the end of the labeled enclosing control structure (`for`, `do`, `while`, or `switch`)
  - Unlabeled `break` jumps to the end of the innermost enclosing control structure, as in C/C++

# Exceptions

- Mechanism for raising and handling conditions that deviate from "normal" control flow
- Exceptions are raised by
  - low-level failures (e.g., division by zero); or
  - from throw statements in library code (e.g., reading past end of file) or user code
- Exceptions can be caught and handled by user code
- In Java, uncaught exceptions halt the program with an error message and a stack traceback
- Java exceptions are similar to C++, but are used more consistently, for all checked runtime errors



# Exception example

```
class BadThingHappened extends Exception {};
int foo() throws BadThingHappened {
  if (…)
     throw new BadThingHappened();
}
void bar() {
  int x;
  try { … foo(); …
        x = a / b;  // implicitly throws if b==0
        …
  } catch (BadThingHappened e) {
    …report problem nicely and exit…
  } catch (ArithmeticException e){
    x = -99;
  }
}
```



# Packages

- The package is Java's top-level code structuring mechanism
- A package is just a name space containing class definitions
  - Similar to C++ namespaces
- Packages can include the standard libraries, libraries from other projects, or our own local code
  - Important library packages include java.lang, java.util, java.io
- By default, classes you define go into a default anonymous package, which is fine for small programs



# More on packages

- To refer to package elements, you can use fully qualified names, e.g.,

```
java.util.List x = new
java.util.LinkedList();
```

- Often better: you can import the package name (or specific class names) that you need:

```
import java.util.*;  // at top of file
List x = new LinkedList();
```

- Package `java.lang` is always implicitly imported, so you never need to qualify its class names



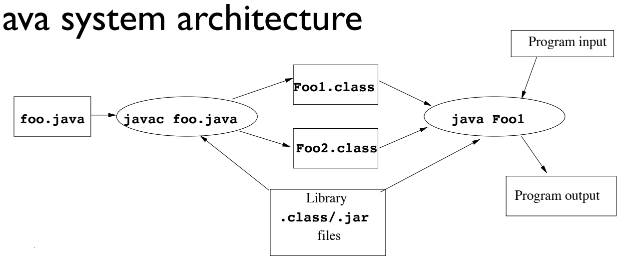
# A complete program

```
class MyApp {
  public static void main(String[] argv) {
    for (int i = 0; i < argv.length; i++)
      System.out.println(argv[i]);
  }
}
```

- Every program must define some class with a `main` method with the exact signature above (where the argument name must be present, but is arbitrary)
- When we run `java MyApp`, the `main` method is invoked with the argument set to an array containing the command line arguments (like `argc`/`argv` in C/C++)



# Java system architecture




- Each source file (`.java` extension) contain one or more class definitions
- Use the `javac` compiler to translate the source file into **bytecode** files (`.class` extension)
  - You get one `.class` file for each defined class
  - Name depends on class name, not on `.java` file name

## Bytecode improves portability

- Java aspires to be "write once, run anywhere"
  - Same bytecode works on any machine (x86, ARM, etc.) and OS (linux,windows,macos,etc.)
  - But must have a Java Virtual Machine (JVM) for the target environment ...
    - ... and precise behavior of built-in libraries can differ in subtle ways
- "Write once, run anywhere" does not come for free!



## Compiling and running the example

- Suppose file `myapp.java` contains the `MyApp` class described earlier
- This can be compiled to bytecode as follows:

```
$ java myapp.java
$
```

- This produces a file `MyApp.class` which can be executed thus:

```
$ javac MyApp.java p d x
p
d
x
$
```



## More to come

- We will explore Java in more detail in the labs and in the next lecture
- Some other important features we have not mentioned yet:
  - Class inheritance and dynamic method dispatch (but we have seen these already in Python in Lab 6 ...)
  - Interfaces
  - Standard library collection classes
  - Polymorphism and generics



## An Introduction to Static Semantics and Static Analysis



## Eliminating errors at compile-time

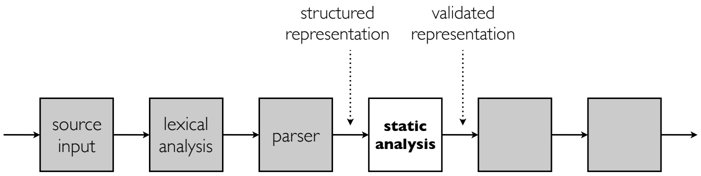
- Languages can be designed to ensure that certain common classes of error:
  - either cannot occur at all;
  - or else can be detected automatically at compile-time.
- These guarantees are typically reflected in the *static semantics* of a language, and enforced by the use of *static analysis* in its implementations.
- How does this work?
- How far can it take us?



$$\text{language} = \begin{cases} \text{syntax} = \begin{cases} \text{concrete} \\ \text{abstract} \end{cases} \\ \text{semantics} = \begin{cases} \text{static} \\ \text{dynamic} \end{cases} \end{cases}$$

**static semantics**: those aspects of a program's behavior/meaning that must be verified at compile time

## Static analysis




- Check that the program is reasonable:
  - no references to unbound variables
  - no type inconsistencies
  - etc...



## Uses of static analysis (1)

- *To ensure validity of input programs:*
  - Check that all variables and functions that are used in the program have been defined
  - Check that the correct number and type of arguments are passed to functions, operators, etc.
  - Check that all variables are initialized before they are used
- *Why?* If we don't make these tests at compile-time, the program might malfunction at run-time



## Uses of static analysis (2)

- *To clarify potential backend ambiguities:*
  - Distinguish between different uses of the same variable name (e.g., global and local)
  - Distinguish between different uses of the same symbol (e.g., arithmetic operators on different numeric types)
- *Why?* If we don't do these kinds of analysis, then it might be difficult to run, compile, or analyze input programs



## Uses of static analysis (3)

- *To justify backend optimizations:*
  - To allow run-time type checks to be omitted
  - To identify redundant computations
  - To identify repeated computations
  - To make good use of the machine's registers and other resources
- *Why?* If we don't do these kinds of analysis, then we might not get the best performance for compiled programs



## Specifying static semantics

- The static semantics of a language is part of each programming language's specification
- Some languages require strict static checking, others are more relaxed
- But, in any interesting language, there are aspects of program behavior that cannot be determined at compile-time:
  - Unknown values
  - Uncomputable problems



## Dealing with unknown values

- Some values are not know at compile-time
- In Java, we can find the current time and date using:

```
Date today = new Date();
```

- The actual result will depend on when we run the program, which isn't known at compile-time
- But we can be sure that the result will be a `Date`; this is the result of type checking

## Uncomputable problems

- A compiler cannot, in general, distinguish programs that may loop indefinitely from programs that are guaranteed to terminate
- Exercise: write a function
  `boolean halts(Program p, Input i)`
  that returns `true` if `p` halts on input `i`, and `false` if it doesn't
- This task is known as the "**halting problem**"
- Extra credit homework assignment?
- Don't try too hard: it can't be done!



If you were able to write `halts()` then you could use it to write this code

```
boolean twist(Program p, Input i) {
    if (halts(p,i)) {
        while (true) ;
    } else {
        return true;
    }
}
boolean test(Program p) {
    return twist(p, p);
}
```

What is `test(test)`?

| false? ✗ | true? ✗ | loops? ✗ |
|---|---|---|
| Can't occur because `twist()` only loops or returns `true` | So `twist(test,test)` returns `true`<br><br>So `halts(test,test)` returns `false`<br><br>So `test(test)` loops | So `twist(test,test)` loops<br><br>So `halts(test,test)` returns `true`<br><br>So `test(test)` terminates |

✗ conclusion: there is no way to write `halts`!



## Static approximates dynamic

- If we want to check the static semantics of a program at compile-time, then we will have to accept a conservative approximation
- Conservative: rejecting programs that might actually be ok, rather than risk allowing programs that might actually fail
- Approximation: Static semantics can tell us something about the result of a computation, but doesn't guarantee to predict every detail



## Some examples

- Given a program: `x = 6 * 7;`

  We know that the result will be `42`
  Static semantics can confirm that the result is an integer

- Given a program: `(true ? 16 : "Hello")`

  We know that the result will be `16`
  Static semantics might suggest that the program "could" cause a type error ...

- Given a program: `int y; if (x*x>=0) y=x;`

  We know that `y` will be initialized by this code
  Static semantics could suggest that `y` might not be initialized by this code … and it might be right too!



## Static analysis

- Static analysis refers to the phase(s) of a compiler that are responsible for checking that input programs satisfy the static semantics
- Static analysis comes **after** parsing:

  The structure of a program must be understood before it can be analyzed

- Static analysis comes **before** code generation:

  We need static analysis results to enable code generation
  There is no point generating code for a bad program



## What exactly does static analysis check?

```
class C extends D {
    T x = e;
    ...

    int f(S p, ...) {
        ...
    }

    void g() {
        V y;
        ...
    }
}
```

static analysis does **NOT** check for syntax errors, correct grammar, …

The preceding "syntax analysis" phases do that

but some "syntactic" properties are most easily checked after parsing

`return` statements here must specify a result expression…

`return` statements here must NOT specify a result expression …

examples

## What exactly does static analysis check?

```
class C extends D {
    T x = e;
    ...
    f(S p, ...) {
      ...
    }

    void g() {
      V y;
      ...
    }
}
```

is there a definition for D?

is D defined as a subclass of C?

is there another definition for C?

is there a definition for T?

is there a definition for S?

is there a definition for V?

**valid types?**

61

## What exactly does static analysis check?

```
class C extends D {
    T x = e;
    ...
    int f(S p, ...) {
      ...
    }
    ... ) {
      ...;
      ...
    }
}
```

are all the variables mentioned in e defined?

does e produce a value of type T?

is there another field called x in this class?

are there multiple parameters called p?

does this code return a value of type int?

is there another method called f in this class?

does this have the right access modifiers, if we are overriding a method from D?

**valid definitions?**

62

## What exactly does static analysis check?

```
class C extends D {
    T x = e;
    ...
    int f(S p, ...) {
      ...
    }

    void g() {
      V y;
      ...
    }
}
```

**valid statements and expressions?**

are all the variables used here defined?

do all the expressions here produce results of appropriate types?

is y initialized before it is used?

is y actually used?

63

## What exactly does static analysis check?

Is `p.q.r` a reference to:

- the package `p.q.r`?
- the class `r` in package `p.q`?
- the class `q.r` in package `p`?
- the inner class `r` in the class `q` in package `p`?
- the static field `r` in the class `q` in package `p`?
- the field `r` in the static field `q` of the class `p`?
- The `r` field of the `q` field in the object referenced by the local variable/parameter `p`?
- ...

the answer depends on the context in which `p.q.r` appears

static analysis can handle this using environments to keep track of which local variables, parameters, classes, packages, are in scope

One additional possible answer: `p.q.r` is not valid!

64

## What exactly does static analysis check?

- In general ... it depends:
  - Static analysis will typically check many properties, as determined by the language definition (imagine the previous examples had been written in Python ...)
  - Individually, many of these are easy to implement
  - But there are usually a lot of checks, and you may have to be careful about the order in which they are performed
- Two of the most common uses of static analysis:
  - Scope resolution: matching every use of an identifier with the corresponding definition
  - Type checking: ensuring that every expression will produce a value of the appropriate type.

65

## Summary

- Language design requires difficult tradeoffs between performance, safety, expressivity, ease of use, ......
- Java was designed to be an improvement on C++ in areas such as safety, portability, and simplicity
- The static semantics of a language determine the properties of programs that should be checked at compile-time
- Static analysis can be used to detect a broad range of common programming errors

  ... but it is necessarily conservative and approximate

66