

Java

Reasoning About Code

Announcements

- You should have access to your repositories and hw0
- Mistake in Grade Breakdown I posted with hw0
 - **RandomHello** is included in the grade!
 - **RandomHello**: 3pts
 - Problem 3 questions: 3pts
 - Problem 4 questions: 7pts
- Always check the **Announcements** page!

Fall 15 CSCI 2600, A Milanova

2

Outline

- Java
 - Types and type checking, type safety
 - Interpretation vs. compilation
- Reasoning about code

Fall 15 CSCI 2600, A Milanova

3

Java: Differences with C++

- Model for variables
 - Java uses the reference model for class types
 - No explicit pointers. All references `_are_` pointers
 - Must explicitly create object: `Foo f = new Foo();`
 - Two equalities: `==` and `equals`.
Remember: when comparing strings, use `equals`!
- Types and type checking, type safety
- Interpretation vs. Compilation
- Other: interfaces, inheritance, etc.

Fall 15 CSCI 2600, A Milanova

4

Types and Type Checking, Type Safety

- What is the role of types?
 - Data abstraction
 - Safety!
- **Types and type checking** prevent the program from going wrong. Disallow operations on objects that do not support those operations
 - E.g., `a+b` where `a` and `b` are `2DPoints` is rejected by the type checker
 - E.g., `a.substring(0,10)` where `a` is an `int` is rejected too

5

Type Safety

- **Type safety**: no operation is ever applied on object of the wrong type (i.e., object that does not support that operation)
- Java is **type safe** while C/C++ is **type unsafe**
 - In Java, the type system never allows operations on objects of the wrong type (i.e., in no execution, such erroneous operations occur)
 - In C++, the type system prevents most errors, but it is possible to write a program where operation on object of the wrong type occurs
- Goal: catch errors as early as possible!

6

Types and Type Checking

- Java and C/C++ are **statically typed**
 - A statically typed language typically requires type annotations; performs substantial amount of type checking before runtime
 - Expressions have static (compile-time) types
 - Objects have dynamic (run-time) types
- Alternative is **dynamically typed**
 - Perform substantial type checking during runtime

Fall 15 CSCI 2600, A Milanova

7

C++ is Type Unsafe

Note: In Java, all methods are virtual; use **final** keyword to force non-virtual

```
// C++:
void* x = (void *) new A;
B* q = (B*) x; //a safe downcast?
int case1 = q->foo(1); //what happens?
```

A virtual foo()
B virtual foo()
virtual foo(int)

Throws ClassCastException here!
Runtime never reaches bad call.

```
// Java:
Object x = new A();
B q = (B) x; //a safe downcast?
int case1 = q.foo(1); //what happens?
```

Fall 15 CSCI 2600, A Milanova

8

C++ is Type Unsafe

- Java: **B** q; ... q.foo(1);
- Java "honors" its promise that at q.foo(1) if q is not null, q is a B (or subclass of B).
- C++: **B*** q; ... q->foo(1);
- C++ does not "honor" its promise. q can be a B or an A or a Duck or whatever

Fall 15 CSCI 2600, A Milanova

9

Type Safety

- Java ensures type safety with a combination of compile-time (static) and runtime (dynamic) type checking
 - Compiler rejects plenty of programs. E.g., String s = 1 is rejected, String s = new Integer(1) is rejected
 - Some checks are left for runtime. Why?
 - E.g., at B b = (B) x; the Java runtime checks if x refers to a B, and throws an **Exception** if it doesn't
- Is Python type safe?

Fall 15 CSCI 2600, A Milanova

10

Java Throws Lots of Exceptions!

E.g.:

ArrayIndexOutOfBoundsException at **x[i]=0**; if i is out of bounds for array x

ClassCastException at **B q = (B) x**; if the runtime type of x is not a B

NullPointerException at **x.f=0**; if x is null

11

Java Throws Lots of Exceptions

- Exceptions are a good thing!
 - Tell us what went wrong
 - Prevent application of operation on wrong type --- stop program from doing harm down the road


```
Object x = new A();
B q = (B) x; // ClassCastException
// because A is not a B
... int case1 = q.foo(1);
```

 Exception prevents execution from reaching q.foo(1) and applying foo(1) on an object (A) that does not support foo(int)

12

Compilation vs. Interpretation

■ Compilation

- A "high-level" program is translated into executable machine code

■ **Compiler.** C++ uses compilation

■ Pure interpretation

- A program is translated and executed one statement at a time

■ **Interpreter**

■ Hybrid interpretation

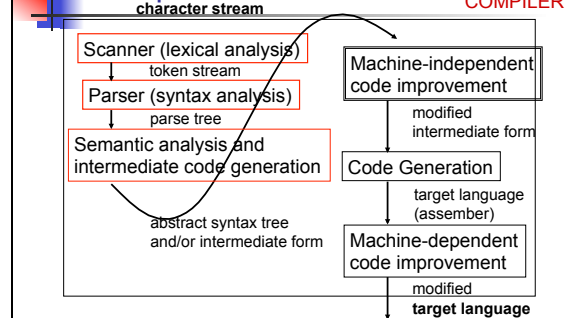
- A program is "compiled" into intermediate code; intermediate code is "interpreted"

■ Both a compiler and an interpreter. **Java**

Fall 15 CSCI 2600, A. Milanova

13

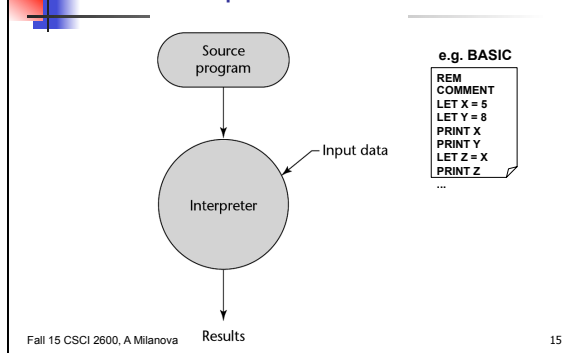
Compilation



Fall 15 CSCI 2600, A. Milanova

14

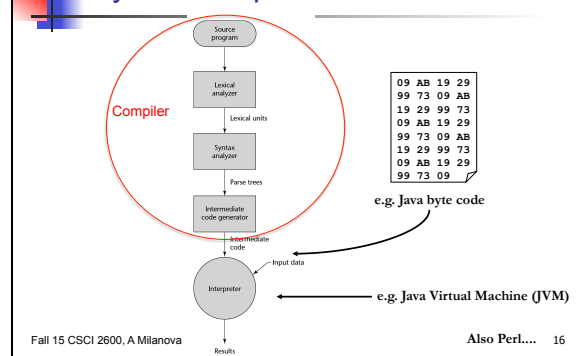
Pure Interpretation



Fall 15 CSCI 2600, A. Milanova

15

Hybrid Interpretation



Fall 15 CSCI 2600, A. Milanova

Also Perl.... 16

Compiling and Running Java

■ Command line:

javac HelloWorld.java produces HelloWorld.class

java HelloWorld // runs the interpreter

■ Eclipse:

Compiles automatically when you save!

Run -> Run runs the interpreter

Fall 15 CSCI 2600, A. Milanova

17

Compilation vs. Interpretation

■ Advantages of compilation?

- Faster execution

■ Advantages of interpretation?

- Greater flexibility
 - Portability, sandboxing, **dynamic semantic** (i.e., type) **checks**, other dynamic features are much easier

Fall 15 CSCI 2600, A. Milanova

18

Some Terminology

- C++: **Base class** and **derived class**
- Java: **Superclass** and **subclass**
- C++: **Member variable**, **member function**
- Java: **field** (instance or static), **method** (again instance or static)
- Java has **interfaces** (collections of method signatures)
 - Single class inheritance (`class B extends A {...}`),
 - Multiple interface inheritance (`class A implements I, J, K {...}`)
 - `class B extends A implements I, J, K {...}`

Fall 15 CSCI 2600, A. Milanova

20

Outline

- Java
 - **Intro to reasoning about code**
 - Specifications
 - Preconditions and postconditions
 - Forward reasoning and backward reasoning
- Reasoning about code, formally; Hoare logic
 - Hoare Triples
 - Rules for assignment, sequence, if-then-else

Fall 15 CSCI 2600, A. Milanova

20

Reasoning About Code

- Determines what facts hold during program execution

```
0 <= index < names.length
x > 0
array names is sorted
x > y
```

Fall 15 CSCI 2600, A. Milanova

21

Why Reason About Code

- Remember, our goal is to produce **correct** code! Two ways to ensure correctness
 - Testing
 - Reasoning about code. Known as verification
- Reasoning about code
 - Verifies that code works **correctly**
 - Finds errors in code
 - Helps understand errors
 - E.g., what input caused division by zero?

Fall 15 CSCI 2600, A. Milanova (based on slide by Michael Ernst, UW)

22

Specifications

- What does it mean for code to be **correct**?
 - (Informally) Code is correct if it conforms to its **specification**
- A specification consists of a **precondition** and a **postcondition**
 - Precondition: conditions that must hold before code executes
 - Postcondition: conditions that must hold after code finishes execution (if precondition held!)

Fall 15 CSCI 2600, A. Milanova

23

Specifications

Notation:
&& denotes logical AND
|| denotes logical OR

Precondition: `arr.length == len && len >= 0`
Postcondition: `result == arr[0] + ... + arr[arr.length-1]`

```
int sum(int[] arr, int len) {
    int result = 0;
    int i = 0;
    while (i < len) {
        result = result + arr[i];
        i = i+1;
    }
    return result;
}
```

To prove that `sum` is **correct**, we must prove that the implementation meets the specification. In other words, we must prove that if the precondition held, after code finishes execution, the postcondition holds. To do this, we must reason about code.

Fall 15 CSCI 2600, A. Milanova

24

Specifications

- The specification is a **contract** between the function and its caller. Both caller and function have obligations:
 - Caller must pass arguments that obey the precondition. If not, all bets are off --- function can break or return wrong result!
 - Function “promises” the postcondition
- In `sum`, how can the caller violate spec?
- How can `sum` violate spec?

25

Aside: Type Signature is a Form of Specification

- Type signature is a contract too!
E.g., `int sum(int[] arr, int len) {...}`
Precondition: arguments are an array of `ints` and an `int`
Postcondition: result is a `int`
- We need more than type signatures! Why?
 - We need reasoning about **behavior and effects** (deeper properties)

Fall 15 CSCI 2600, A Milanova

26

Aside: Type Signature is a Specification

- Type checker (among other things) verifies that the parties meet the contract
- If language is **type safe** we can “trust” the type checker
- But if language is **type unsafe** we it would be possible for a caller to pass an argument of the wrong type!

Fall 15 CSCI 2600, A Milanova

27

Why Reason About Code

- Ensure code works **correctly**
 - Ensure that code meets the specification
 - E.g., we can prove that `sum` is correct by proving that `sum` meets its specification
- Find errors in code
- Understand errors

Fall 15 CSCI 2600, A Milanova

28

What is Wrong With this Code?

```
class NameList {
    int index;
    String[] names;
    ...
    // Precondition: 0 ≤ index < names.length
    // Postcondition: 0 ≤ index < names.length
    void addName(String name) {
        index++;
        if (index < names.length)
            names[index] = name;
    }
}
```

Fall 15 CSCI 2600, A Milanova (example modified from Michael Ernst)

29

What Inputs Cause Wrong Output?

```
String[] parseName(String name) {
    int comma = name.indexOf(",");
    String firstName = name.substring(0,comma);
    String lastName = name.substring(comma+2);
    return new String[] { lastName, firstName };
}
```

What input produces array ["Doe","Jane"]?
What input produces array ["oe","Jane"]?
What input produces `StringIndexOutOfBoundsException`?

Fall 15 CSCI 2600, A Milanova (example due to Michael Ernst)

30

Types of Reasoning

- **Forward reasoning:** given a precondition, what is the postcondition?
 - Verify that code works correctly
- **Backward reasoning:** given a postcondition, what is the precondition?
 - Again, verify that code works correctly
 - What input caused an error

Fall 15 CSCI 2600, A Milanova (based on slide by Michael Ernst)

31

Forward Reasoning

- We know what is true before running the code. What is true after running the code?

```
// precondition: x is even
x = x + 3;
y = 2x;
x = 5;
// What is the postcondition here?
```

Fall 15 CSCI 2600, A Milanova (based on slide by Michael Ernst)

32

Strongest Postcondition

- Many postconditions hold from this precondition and code!

```
// precondition: x is even
```

```
x = x + 3;
```

```
y = 2x;
```

```
x = 5;
```

```
// postcondition: x = 5 && y % 4 = 2
```

```
// postcondition: x = 5 && y is even
```

```
// postcondition: x > 0 && y is even
```

x=5 && y%4 = 2 is the **strongest postcondition**. It implies all other postconditions. More on stronger and weaker conditions later.

33

Forward Reasoning Example

```
// precondition: x > y
```

```
z = x;
```

```
x = y;
```

```
y = z;
```

```
// What is the postcondition ??
```

One postcondition: $z = x_0 \ \&\& \ x = y_0 \ \&\& \ y = z \ \&\& \ x_0 > y_0$

(here x_0 denotes the initial value of x)

This postcondition implies $y > x$

Fall 15 CSCI 2600, A Milanova

34

Backward Reasoning

- We know what **we want to be true** after running the code. What must be true beforehand to ensure that?

```
// precondition: ??
```

```
x = x + 3;
```

```
y = 2x;
```

```
x = 5;
```

```
// postcondition: y > x
```

Fall 15 CSCI 2600, A Milanova (based on a slide by Michael Ernst)

35

Forward vs. Backward Reasoning

- Forward reasoning is more intuitive, just simulates the code
 - Introduces facts that may be irrelevant to the goal
 - Takes longer to prove task or realize task is hopeless
- Backward reasoning is usually more helpful
 - Given a specific goal, shows what must hold beforehand in order to achieve this goal
 - Given an error, gives input that exposes error

Fall 15 CSCI 2600, A Milanova (based on a slide by Michael Ernst)

36

Forward Reasoning: Putting Statements Together

Does the postcondition hold?

```

Precondition:  $x \geq 0$ ;

 $z = 0$ ;           {  $x \geq 0 \ \&\& \ z = 0$  }
if ( $x \neq 0$ ) {
   $z = x$ ;         {  $x > 0 \ \&\& \ z = x$  }
} else {
   $z = z + 1$       {  $x = 0 \ \&\& \ z = 1$  }
}
Postcondition:  $z > 0$ ;
  
```

Therefore, postcondition holds!

Fall 15 CSCI 2600, A Milanova (example due to Michael Ernst)

37

Forward Reasoning With a Loop

Does the postcondition hold?

```

Precondition:  $x \geq 0$ ;

 $i = x$ ;           {  $x \geq 0 \ \&\& \ i = x$  }
 $z = 0$ ;           {  $x \geq 0 \ \&\& \ i = x \ \&\& \ z = 0$  }
while ( $i \neq 0$ ) {
   $z = z + 1$ ;      ???
   $i = i - 1$ ;      ???
}
Postcondition:  $x = z$ ;
  
```

Yes. The key is to guess the **loop invariant**. Then prove by induction over the number of iterations of the loop.

Fall 15 CSCI 2600, A Milanova (based on slide by Michael Ernst)

38

Outline

- Intro to reasoning about code
 - Specifications
 - Preconditions and postconditions
 - Invariants
 - Forward reasoning and backward reasoning
- Reasoning about code, formally: Hoare Logic
 - Hoare Triples
 - Rules for assignment, sequence, if-then-else

Fall 15 CSCI 2600, A Milanova

39

Hoare Logic

- Formal framework for reasoning about code
- Sir Anthony Hoare (Sir Tony Hoare or Sir C.A.R. Hoare)
 - Hoare logic
 - Quicksort algorithm
 - Other contributions to programming languages
 - Turing Award in 1980

Fall 15 CSCI 2600, A Milanova

40

Hoare Triples

- A Hoare Triple: $\{ P \} \text{code} \{ Q \}$
 - P and Q are logical statements about program values, and **code** is program code (in our case, Java code)
- " $\{ P \} \text{code} \{ Q \}$ " means "if P is true and we execute **code**, then Q is true afterward"
 - " $\{ P \} \text{code} \{ Q \}$ " is a logical formula, just like " $0 \leq \text{index}$ "

Fall 15 CSCI 2600, A Milanova (based on slide by Michael Ernst)

41

Examples of Hoare Triples

```

{  $x > 0$  }  $x++$  {  $x > 1$  } is true
{  $x > 0$  }  $x++$  {  $x > -1$  } is true
{  $x \geq 0$  }  $x++$  {  $x > 1$  } is false. Why?

{  $x > 0$  }  $x++$  {  $x > 0$  } is ??
{  $x < 0$  }  $x = x + 1$  {  $x < 0$  } is ??
{  $x = a$  } if ( $x < 0$ )  $x = -x$  {  $x = |a|$  } is ??
{  $x = y$  }  $x = x + 3$  {  $x = y$  } is ??
  
```

Fall 15 CSCI 2600, A Milanova

42



Summary So Far

- Intro to reasoning about code. Concepts
 - Specifications, preconditions and postconditions, invariants, forward and backward reasoning
- Hoare triples



Next time

- Hoare logic
- Rules for backward reasoning
 - Assignment, sequence, if-the-else, method call
- Dafny lab at the end of class