Announcements

- HW0 is now posted
  - Clone repo and create Eclipse project
  - I'll open the page in Submitty tonight

- Any problems with this setup, ask at csci2600@cs.lists.rpi.edu or LMS Discussion board

Outline

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

Java: Differences with C++

- Model for variables
  - Java uses 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.

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

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!
Types and Type Checking

Java and C/C++ are statically typed
- Statically typed languages typically require type annotations: perform substantial amount of type checking before runtime
- Expressions have static (compile-time) types
- ... and they have dynamic (run-time) types

Alternative is dynamically typed
- Perform substantial type checking during runtime

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
- This is because C++ is a superset of C

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., if 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?

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

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(int) on an object (A) that does not support foo(int)
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

Pure Interpretation

- **Source program**
- **Input data**
- **Interpreter**
- **Results**

- **e.g. BASIC**
  - REM
  - COMMENT
  - LET X = 5
  - LET Y = 6
  - PRINT X
  - PRINT Y
  - LET Z = X
  - PRINT Z

Hybrid Interpretation

- **Compiler**
- **Interpreter**

- **e.g. Java byte code**
  - 09 AB 19 29
  - 09 73
  - 09 AB

- **Also Perl...**

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

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
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 {...

Outline

- **Java**
  - Intro to reasoning about code (informal account)
    - Specifications
    - Preconditions and postconditions
    - Forward reasoning and backward reasoning

Reasoning About Code

- Determines what facts hold during program execution
  - `0 <= index < names.length`
  - `x > 0`
  - array `names` is sorted
  - `x > y`
  - `x != 0`

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
  - Validates that code works **correctly**
  - Finds errors in code
  - Helps understand errors
  - E.g., what input caused division by zero?

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!)

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, when the code finishes execution, the postcondition holds. To do this, we must reason about code.
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?

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
- Type signature is not enough! Why?
  - We need reasoning about behavior and effects (deeper properties)

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

What is Wrong With this Code?

```java
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;
    }
}
```

Aside: Type Signature is a Specification

- Type checker verifies that caller and callee obey the contract
- If language is type safe we can "trust" the type checker
- But if language is type unsafe it would be possible for a caller to pass an argument of the wrong type (or a callee to return a wrong type)

Aside: Type Signature is a Specification

- Type checker verifies that caller and callee obey the contract
- If language is type safe we can "trust" the type checker
- But if language is type unsafe it would be possible for a caller to pass an argument of the wrong type (or a callee to return a wrong type)

What Inputs Cause Wrong Output?

```java
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 would produce `StringIndexOutOfBoundsException`?
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

Forward Reasoning

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

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

Strongest Postcondition

- Many postconditions hold from this precondition and code!

```plaintext
// 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
```

Forward Reasoning Example

```plaintext
// precondition: x > y
z = x;
x = y;
y = z;
// What is the postcondition ??
```

Backward Reasoning

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

```plaintext
// precondition: ??
```
Forward Reasoning: Putting Statements Together

Precondition: \( x \geq 0 \);

\[ z = 0; \quad \{ x \geq 0 \&\& z = 0 \} \]

\[ \text{if } (x \neq 0) \{ \]
  \[ z = x; \quad \{ x > 0 \&\& z = x \} \]
\[ \text{else } \{ \]
  \[ z = z+1 \quad \{ x = 0 \&\& z = 0 \} \]
\[ \} \]

Postcondition: \( z > 0 \);

Therefore, postcondition holds!

Forward Reasoning With a Loop

Precondition: \( x \geq 0 \);

\[ i = x; \quad \{ x \geq 0 \&\& i = x \} \]

\[ z = 0; \quad \{ x \geq 0 \&\& i = x \&\& z = 0 \} \]

\[ \text{while } (i \neq 0) \{ \]
  \[ z = z+1; \quad \text{Yes, it holds. Key is to guess a loop invariant. Then prove by induction over the number of iterations of the loop.} \]
  \[ i = i-1; \quad \text{postcondition holds!} \]
\[ \} \]

Postcondition: \( x = z \);

Next Time

- Reasoning about code, more formally:
  - Hoare Logic
  - Hoare Triples
  - Rules for assignment, sequence, if-then-else
  - Loops