Review

- Language paradigms
  - Imperative
    - C, Pascal, Fortran
  - OO
    - C++, Java, Smalltalk, JavaScript
  - Functional
    - ML, Haskell, LISP, Scheme
  - Logic
    - Prolog, Datalog
OO languages

- Simula67
  - Kristen Nygaard, Ole-Johan Dahl
  - objects, classes, inheritance, virtual methods, coroutines
- Smalltalk
  - 1972-80 Alan Kay, Dan Ingalls, et al. (Xerox PARC)
  - pure OO (everything is an object)
- C++
  - 1985 Bjarne Stroustrup (Bell Labs)
  - C with Simula67 classes
- Java
  - 1995 James Gosling, Guy Steele, Bill Joy, ... (Sun)
- JavaScript
  - 1995 Brendan Eich (Netscape)
Why is Java your favorite language?
Java

- 1995 James Gosling et al. (Sun)
- C++-like syntax
- Portable ("write once, run anywhere")
- Type-safe
- GC
- rich libraries
History

- Java 1.0 - 1996
- Java 1.1 - 1997
  - inner classes, reflection
- Java 1.2 - 1998
  - collections library
- Java 1.3 - 2000
  - HotSpot VM
- Java 1.4 - 2002
  - assert, NIO
- Java 5 - 2004
  - generics, autoboxing, annotations, enums, enhanced for, varargs
- Java 6 - 2006
  - concurrency libraries
- Java 7 - 2011?
  - closures, method handles, dynamic language support
Language definition

- **Syntax** defines the structure of a program
  - set of rules defining which symbols are a legally structured program

- **Semantics** defines the “meaning” of a program
  - without semantics, programs are just sequences of characters
Java syntax

- Basically C++ with a few changes
  - same unary and binary operations (+, -, *, /, <<, &&)
  - same field access syntax
  - same control constructs (while, for, etc.)
  - same class declaration syntax
  - . vs. ::
- no #includes, #defines
- class members declared within class body in same file
Types

• What’s a type?

• A **predicate** on what values might occur at run time

• i has type int means \( i \in [-2^{31}, 2^{31}) \)

• b has type boolean means \( b \in \{\text{false, true}\} \)

• s has type String means \( s \in \{\text{"", "0", "1", ..., "00", ...} \} \)
Types

• Can a type be an arbitrary predicate?
  • e.g., x : an integer equal to gcd(a, b)

• In principle, yes, but:
  • might be undecidable or at least intractable (theorem proving)
  • but automatic theorem provers are getting better and better

• Types should be **efficiently decidable**
Type systems

- Languages have **type systems**

- Type system describes:
  - what are the types
  - what is the type of a given expression
  - is the type of a given expression allowed
Types
Types

• What types does Java have?
Types

• What types does Java have?
  • primitives: boolean, char, byte, short, int, long, float, double
Types

• What types does Java have?
  • primitives: boolean, char, byte, short, int, long, float, double
  • reference types: class types, interface types, array types
Types

• What types does Java have?
  • primitives: boolean, char, byte, short, int, long, float, double
  • reference types: class types, interface types, array types
  • any others?
Types

• What types does Java have?
  • primitives: boolean, char, byte, short, int, long, float, double
  • reference types: class types, interface types, array types
  • any others?
    • What is the type of null?
Types

• What types does Java have?
  • primitives: boolean, char, byte, short, int, long, float, double
  • reference types: class types, interface types, array types
  • any others?
    • What is the type of null?
    • What about void?
Types

• What types does Java have?
  • primitives: boolean, char, byte, short, int, long, float, double
  • reference types: class types, interface types, array types
  • any others?
    • What is the type of null?
    • What about void?
    • What about methods?
Types

• What types does Java have?
  • primitives: boolean, char, byte, short, int, long, float, double
  • reference types: class types, interface types, array types
  • any others?
    • What is the type of null?
    • What about void?
    • What about methods?
    • Generics?
Type safety

• Type errors:
  • Improper, type-inconsistent operations during program execution

• **Type safety** = absence of run-time type errors

• Operations unsupported by a value will **never** occur at run time
  • cannot multiply a string and an int
  • cannot call a function that is not defined
  • cannot access a field that does not exist
  • cannot access outside the bounds of an array
Strong vs. weak typing

• A language is **strongly typed** if it ensures no type errors occur at run time (i.e., it is **type safe**)

• Contrast with: **weakly typed**
Strong vs. weak typing

- Is Java strongly typed or weakly typed?
- C?
- Python?
- Other examples?
C is weakly typed

- int x = 100;
- printf("%s", x);
Static vs. dynamic typing

- **statically typed languages** rule out type errors at compile time
  - requires programmers to write type declarations

- **dynamically typed languages** rule out type errors at run time
  - perform dynamic type checks before each operation

- Many statically typed languages allow some dynamic type checks
  - e.g., casts, instanceof in Java
Static vs. dynamic typing

• Is Java statically typed or dynamically typed?
• C?
• Python?
• Other examples?
<table>
<thead>
<tr>
<th>Type systems</th>
<th>static</th>
<th>dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>strong</td>
<td>Java, C#, Scala</td>
<td>Smalltalk, JavaScript, Scheme</td>
</tr>
<tr>
<td>weak</td>
<td>C, C++</td>
<td>Perl, Forth, assembly code</td>
</tr>
</tbody>
</table>
Java’s type system

• Strongly typed
  • Cannot use a value at the “wrong” type
• Statically typed
  • Type errors detected at compile type
  • Except when the programmer does an explicit run-time check
Casts in Java

• Can cast an object reference to an object type

• Compiler will perform a run-time type test:
  • Bird b = new Penguin();
  • (Penguin) b   // ok
  • (Vulture) b   // throws ClassCastException

• Compiler can reject program if cast will always fail

• (Bird) new Elephant()}
Primitive casts in Java

• Casting from one primitive type to another **converts** the value to the target type

• `int x = 65536;`
• `byte b = (byte) x;`
• `assert b == 0;`
Computing types

• How to determine what type an expression has?
  • e : T means “e has type T”

• 1 : int
• true : boolean
• new C() : C
Typing rules

- Language specification defines rules for computing types:
  - $x + y : \text{int}$
    - if $x$ and $y$ both have type byte, short, char, or int
  - $x + y : \text{long}$
    - if $x$ and $y$ both have type byte, short, char, int, or long and at least one is long
  - ...  

- Can define rules for every expression in the language
Typing rules

• e.f : T  if
  • e has type C, and
  • C has a field f of type T

• a[0] : T  if
  • a has type T[]
Type checking

• Compiler computes types and then checks them

  • while (c) S
    • c must have type boolean

  • int[] x = y;
    • y must have type int[]

  • a[i]
    • a must be an array type
    • i must have type int
Design choices

- Why did the designers of Java make these choices?
  - Object references, not pointers
  - Reference types and primitive types are different
  - Single class inheritance, multiple interface inheritance
Object references

• Can only have references to objects, not arbitrary pointers
• Cannot have a pointer into the middle of an array
• Cannot have a pointer to a variable on the stack
• Cannot do pointer arithmetic like in C, C++:
  • int *p = &a;
  • p++;
Primitive types

- Java distinguishes between **primitive types** and **reference types**
- Done for efficiency reasons
  - int is more efficient than java.lang.Integer
  - int[] is much much more efficient than Integer[]
- More on this next week
Inheritance

• Java has
  • single class inheritance
  • multiple interface inheritance

• Avoids most of the complications with multiple inheritance, while preserving some of the usefulness

• More on this over the next week
How is Java implemented?
Java

- Java code is compiled to **Java bytecode**

- The **Java Virtual Machine (JVM)** *interprets* bytecode
Java: Language vs. platform

• Many different languages compile to Java bytecode
• JVM can execute code for many different languages
  • Java
  • Scala
  • Python (Jython)
  • Ruby (JRuby)
  • Groovy
  • Scheme (Kawa)
  • JavaScript (Rhino)
  • Thorn
  • X10
Language implementation

• Compilers
• Interpreters
Compilers

• A program that translates an executable program in one language into an executable program in another language

• Usually higher-level to lower-level
  • e.g., C to x86, Java to JVM bytecode, C++ to C
  • “translator” - target is a similar language
  • “decompiler” - target is a higher-level language
  • “assembler” - source is assembly code, target is machine code
Interpreters

- A program that reads an executable program and produces the results of running that program.
- Reads code into a data structure.
- Evaluates the data structure.
Representing programs

• Data structure that represents a program is called the intermediate representation (IR)

• Examples:
  • abstract syntax trees
  • stack code

• IRs are used by both compilers and interpreters
Abstract syntax trees

• Represent program as a tree of objects
• One class for each kind of expression

```java
abstract class Exp {
    abstract int eval();
}
class Int extends Exp {
    int value;
    int eval() { return value; }
}
class Add extends Exp {
    Exp x, y;
    int eval() { return x.eval() + y.eval(); }
}
```
Stack-based interpreters

- Many interpreters use a stack-based IR
  - Simpler representation of the program
  - Faster to evaluate

- Example: Java bytecode

  - \((x+1)*2\)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>iload x</td>
<td>x</td>
</tr>
<tr>
<td>ldc 1</td>
<td>x</td>
</tr>
<tr>
<td>iadd</td>
<td>(x+1)</td>
</tr>
<tr>
<td>ldc 2</td>
<td>(x+1)</td>
</tr>
<tr>
<td>imul</td>
<td>(x+1)*2</td>
</tr>
</tbody>
</table>
Simple bytecode interpreter

while (true) {
    ins = readNextInstruction();
    switch (ins.opcode) {
    case LDC: // push constant
        stack.push(ins.operand);
        break;
    case ILOAD:
        stack.push(vars.get(ins.operand));
        break;
    case IADD: // pop 2 operands, add, push result
        int arg2 = stack.pop();
        int arg1 = stack.pop();
        stack.push(arg1 + arg2);
        break;
    ... 199 more cases ...
    }
}
Virtual machines

- A software execution engine that provides a machine-independent language implementation

- interpreter + language runtime system services

Java
- javac – compiles Java to Java bytecode
- Java virtual machine interprets Java bytecode
- VM provides other services used by interpreter
  - garbage collector
  - thread scheduler
  - class loader
  - bytecode verifier
Basic VM structure

- Program/Bytecode
  - Class loader, Verifier, etc.
  - Interpreter / dynamic compiler
- Executing Program
- Heap
- Thread Scheduler
- Garbage Collector
Dynamic compilation

• Bytecode interpreter is slow
  • ~100x compiled code

• **Just-in-time** compilation
  • compile to native code at first invocation
  • rather than interpret, compile the IR into native code **in memory**, then just run the native code

• JITs are also slow! – *Why?*
Just-in-time is too often

- JITs compile all code

- Compile time (CT)
  - > interpretation time (IT)
  - > native execution time (NT)

- Suppose code is executed $n$ times
  - Interpreter takes time $n \times IT$
  - JIT takes time $CT + n \times NT$

- If $n$ is small, $n \times IT < CT + n \times NT$
Dynamic optimization

- At first **interpret** code
- But also, collect profiling information
  - identify where most time is spent
    - **hot** methods and loops
- Then, dynamically compile just the hot code
  - typically 2-10% of the code
Dynamic optimization

• Pioneered in the language *Self*  
  [David Ungar, Randall Smith 1986]
• Popularized with the HotSpot JVM [Sun 1999]
• Now used in most Java and CLR implementations and for many dynamic languages: Python, JavaScript, ...
Dynamic optimization

- Since we don’t waste time compiling cold code
- We can spend more time compiling hot code
  - Can do a better job optimizing hot code
  - Can also use the profiling information to do better than an ahead-of-time compiler
As fast as C?

• Conventional wisdom
  • To make code fast, write it in C

• Not so true
  • One data point: Java virtual machines
    • HotSpot, J9, Apache DRLVM – all written in C, C++
    • Jikes RVM – JVM written in Java
  • 1999: they all performed within 10%
  • 2009: Jikes RVM performs the same as HotSpot & J9 on DaCapo benchmarks
  • Aside: GCC code 2-10% slower than product compilers
Questions?