CSEP 590 – Programming Systems
University of Washington

Lecture 6: Potpourri

Michael Ringenburg
Spring 2017

Course News

- Presentations
  - Start next week!
  - Schedule posted on course web
- Today – mix of interesting topics that we haven't covered yet
  - Type Checking
  - Loop Parallelism
  - JVM and JIT compilation
  - Query optimization, if time permits
- Final Homework posted today
  - Due end of quarter (June 2)

Types from the Compiler's Perspective

Types

- Types play a key role in most programming languages. E.g.,
  - Run-time safety
  - Compile-time error detection
  - Improved expressiveness (inheritance, overloading, etc)
  - Provide information to optimizer
    - Strongly typed languages – what data might be used where
    - Type qualifiers (e.g., const and restrict in C)

Type Checking Terminology

Static vs. dynamic typing

- static: checking done prior to execution (e.g. compile-time)
- dynamic: checking during execution

Strong vs. weak typing

- strong: guarantees no illegal operations performed
- weak: can't make guarantees

	static	dynamic
strong	Java, ML	Scheme, Ruby
weak	C	PERL

Type Systems

- Base Types
  - Fundamental, atomic types
  - Typical examples: int, double, char, bool
- Compound/Constructed Types
  - Built up from other types (recursively) via *constructors*
  - Constructors include arrays, records/structs/ classes, pointers, enumerations, functions, modules, …

Types vs ASTs

- Types are not typically AST nodes
  - AST nodes often have a type *field*, however
- AST = abstract representation of source program (including source program type info)
- Types = abstract representation of type semantics for type checking, inference, etc.
  - Can include information not explicitly represented in the source code, or may describe types in ways more convenient for processing
- Need a separate "type" class hierarchy in your compiler distinct from the AST

Base Types

- For each base type (int, boolean, etc), can create a single object to represent it
  - Base types in symbol table entries and AST nodes are direct references to these objects
  - Base type objects usually created at compiler startup
- Useful to create a type "void" object to tag functions that do not return a value
- Also useful to create a type "unknown" object for errors
  - ("void" and "unknown" types reduce the need for special case code in various places in the type checker)

Compound Types

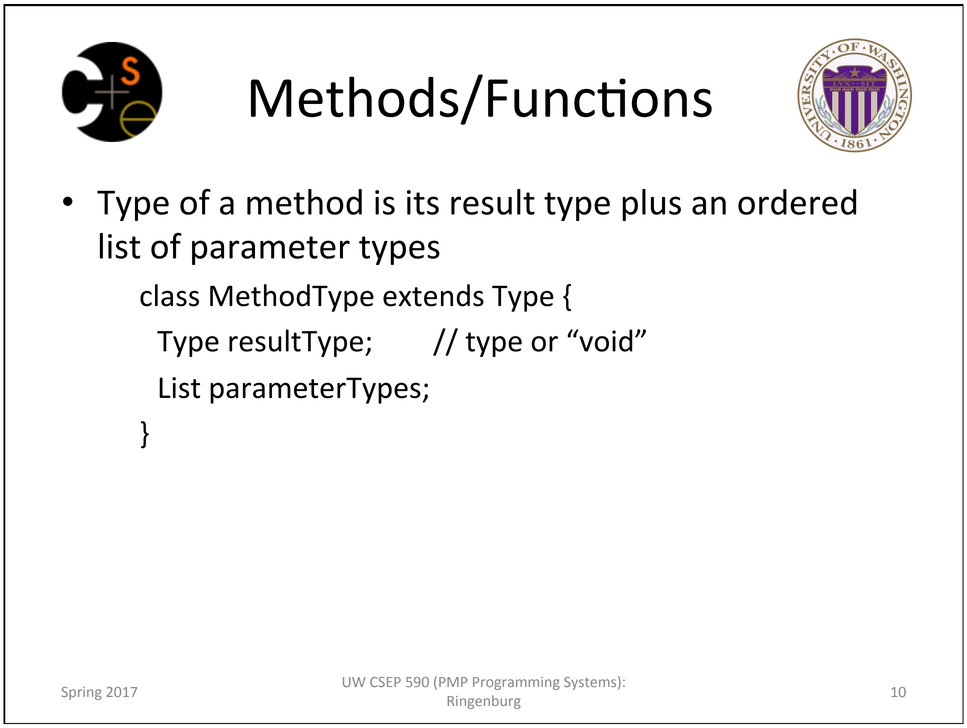
- Basic idea: use a appropriate “type constructor” object that refers to the component types
  - Limited number of these – correspond directly to type constructors in the language (record/struct/class, array, function,…)

Array Types

- For regular Java this is simple: only possibility is # of dimensions and element type

```
class ArrayType extends Type {
  int nDims;
  Type elementType;
}
```

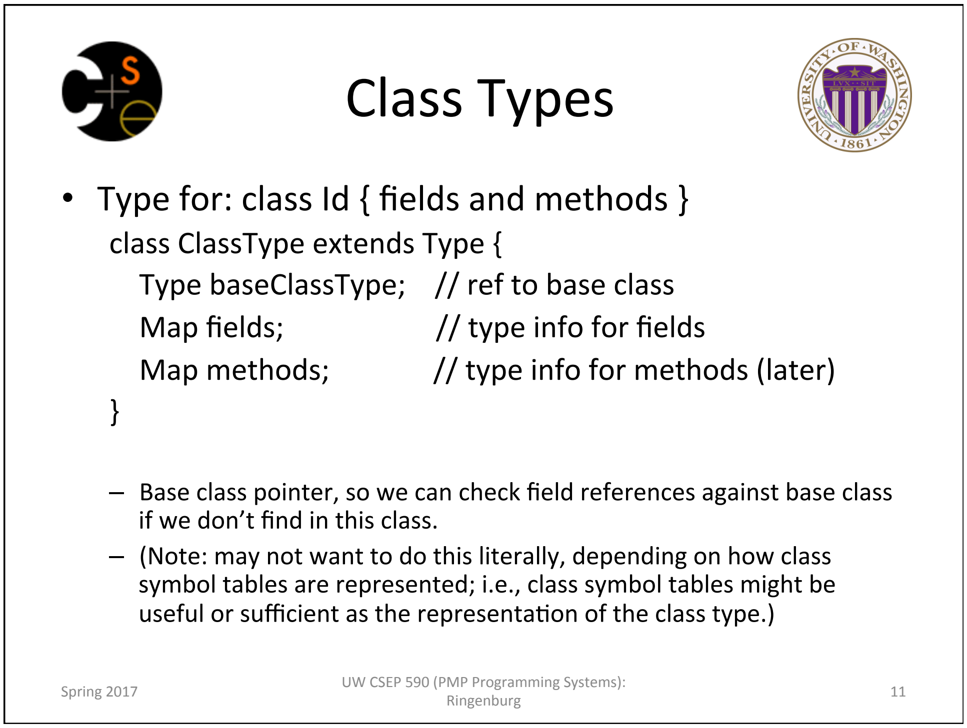
- Length *not* part of type
- More interesting in languages like Pascal (more complex array indexing – index types!)



Methods/Functions

- Type of a method is its result type plus an ordered list of parameter types

```
class MethodType extends Type {
  Type resultType;       // type or “void”
  List parameterTypes;
}
```



Class Types

- Type for: class Id { fields and methods }

```
class ClassType extends Type {
   Type baseClassType;   // ref to base class
   Map fields;           // type info for fields
   Map methods;          // type info for methods (later)
}
```

  - Base class pointer, so we can check field references against base class if we don’t find in this class.
  - (Note: may not want to do this literally, depending on how class symbol tables are represented; i.e., class symbol tables might be useful or sufficient as the representation of the class type.)

Type Equivalance

- For base types this is simple
  - – If you have just a single instance of each base type (as recommend), then types are the same if and only if they are identical
    - Pointer/reference comparison in the type checker
  - – Normally there are well defined rules for coercions between arithmetic types
    - Depending on language rules, compiler inserts these automatically or when requested by programmer (casts) – often involves inserting cast/conversion nodes in AST

Type Equivalence for Compound Types

- Two basic strategies
  - – *Structural equivalence*: two types are the same if they are the same kind of type and their component types are equivalent, recursively
    - E.g., two struct types, each with exactly two int fields
  - – *Name equivalence*: two types are the same only if they have the same name. If their structures match, but have distinct names, they are not equal.
- Different language design philosophies
  - – Mix is common, e.g., C/C++ name for structs/classes, structural otherwise.

Structural Equivalence

- Structural equivalence says two types are equal iff they have same structure
  - Identical base types clearly have the same structure
  - if type constructors:
    - same constructor
    - recursively, equivalent arguments to constructor
- Ex: atomic types, array types, pointer types
- Implement with recursive implementation of equals

Name Equivalence

- Name equivalence says that two types are equal iff they came from the same textual occurrence of a type constructor
  - Ex: class types, C struct types (struct tag name), datatypes in ML
  - special case: type synonyms (e.g. typedef in C) do not define new types – uses structural equivalence
- Implement with pointer/reference equality assuming appropriate representation of type info

Type Equivalence and Inheritance

- Suppose we have

```
class Base { … }
class Extended extends Base { … }
```

- A variable declared with type Base has a *compile-time type* of Base
- During execution, that variable may refer to an object of class Base or any of its subclasses like Extended (or can be null)
  - Sometimes called the *runtime type*
  - Subclasses guaranteed to have all fields/methods of base class, so typechecking as base class suffices

Type Casts

- In most languages, one can explicitly cast an object of one type to another
  - sometimes cast means a conversion (e.g., casts between numeric types)
  - sometimes cast means a change of static type without doing any computation (casts between pointer types or pointer and numeric types)
  - With class types, may also mean upcast (free) or downcast (runtime check)s

Type Conversions vs Coercions

- In Java, we can explicitly *convert* an value of type double to one of type int
  - Can represent as unary operator
  - Typecheck, generate code normally
- In Java, can implicitly *coerce* an value of type int to one of type double
  - Compiler must insert unary conversion operators, based on result of type checking

C and Java: type casts

- In C: safety/correctness of casts not checked
  - Allows writing low-level code that's type-unsafe
  - Result is often implementation dependent/undefined. Not portable, but sometimes useful.
- In Java: downcasts from superclass to subclass need run-time check to preserve type safety
  - Otherwise, might use field (or call method) that is not present in superclass
  - Static typechecker allows the cast
  - Code generator introduces run-time check
  - Java's main form of dynamic type checking

Final Note: Various Notions of Equivalance

- There are usually several relations on types that compiler needs to deal with:
  - "is the same as"
  - "is assignable to"
  - "is same or a subclass of"
  - "is convertible to"
- Be sure to check for the right one(s)

Useful Compiler Functions

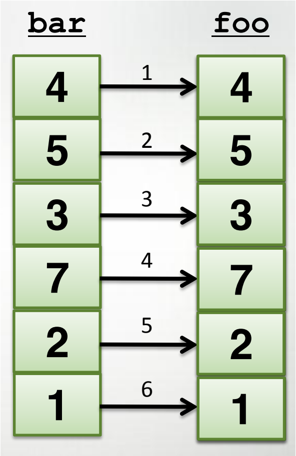
- Create a handful of methods to decide different kinds of type compatibility, e.g.:
  - Types are identical
  - Type t1 is assignment compatible with t2
  - Parameter list is compatible with types of expressions in the call
- Usual modularity reasons: isolates these decisions in one place and hides the actual type representation from the rest of the compiler
- Probably belongs in the same package with the type representation classes (package for dealing with types)

Loop Parallelism

What is loop parallelism?

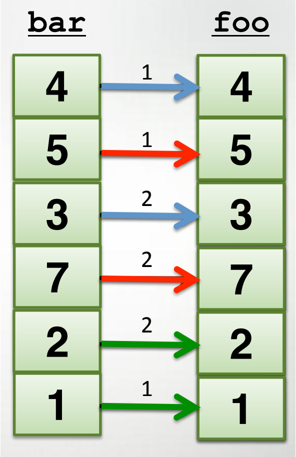
```
for(i=0; i<N; i++) {
   foo[i] = bar[i];
}
```

- A serial (non-parallelized) loop consists of a series of iterations that run one at a time (in order) on a single thread.



What is loop parallelism?

```
for(i=0; i<N; i++) {
  foo[i] = bar[i];
}
```



- A parallelized loop consists of a series of iterations that may run simultaneously on multiple threads.
- Every thread executes a distinct subset of the iterations
- Iterations not ordered.

Motivations for Loop Parallelization

- Take advantage of available parallelism in architecture
  - Multi-core and many-core processors
  - Vectorization instructions
  - Hyperthreading
- Latency hiding
  - Switch contexts rather than waiting

Conditions for Parallelization

- Typical necessary conditions for compiler to auto-parallelize a loop
  - It can figure out how to compute the number of iterations prior to executing the loop
  - It can prove that there are no dependences between iterations
  - There are no function calls with unknown side effects (e.g., output)
  - The loop has a simple structure (e.g., no multiple exits)
- Users may insert extra information to help the compiler establish that these conditions hold.
  - Compiler relies on info: if false, compiled program may behave unepectedly

Examples

- Parallelizable loop:

```
void foo(int n) {
  int i;
  int my_array[n];
  for (i = 0; i < n; i++) {
    my_array[i] = i;
  }
  return;
}
```

Examples

- Non-parallelizable loop:

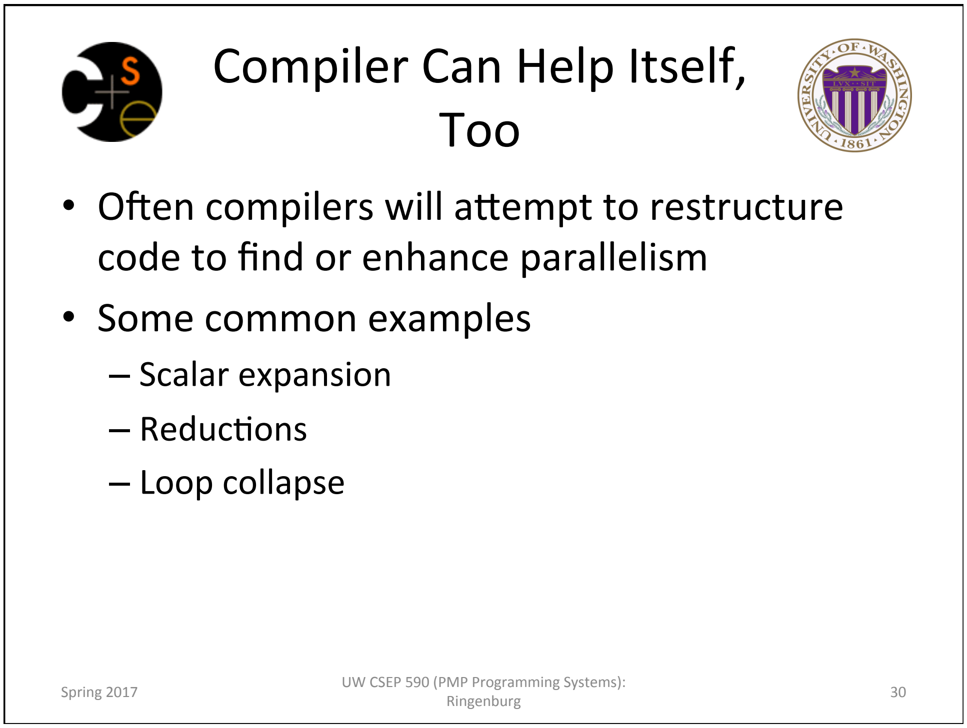
```
void foo(int *a, int *b) {
  int i;
  for (i = 0; i < 10000; i++) {
    a[i] = b[i];
   }
}
```

- a and b may point to overlapping memory:

```
foo(x, x+5000)
```

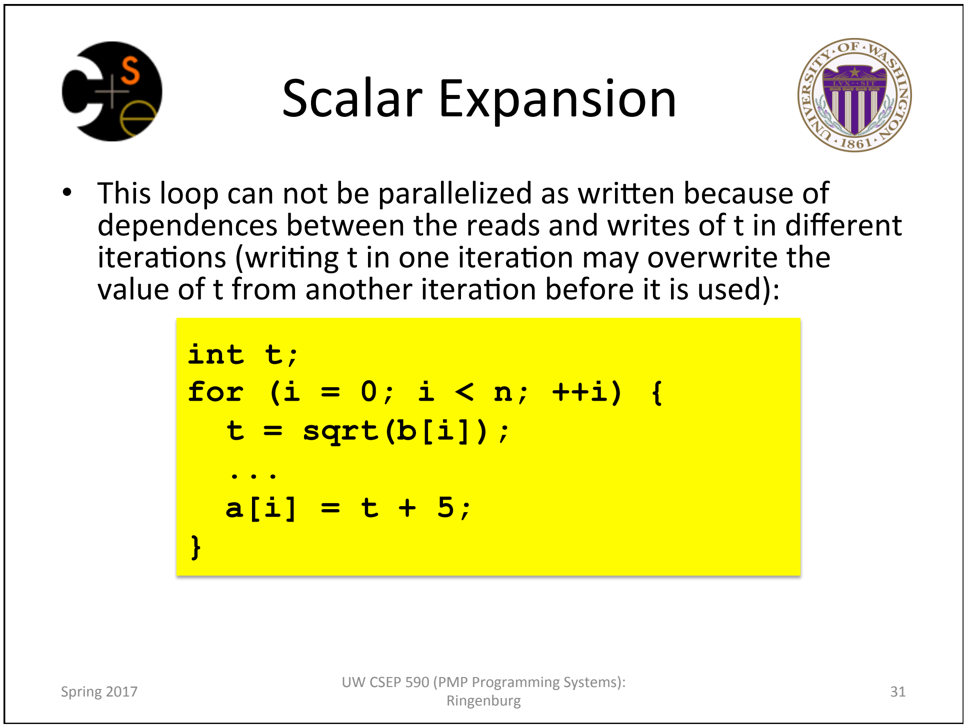
Helping the Compiler

- Common types of hints/information
  - This pointer is not aliased with any other pointers (doesn't point to data that overlaps with another pointer). E.g., restrict keyword in C
  - There are no dependencies between iterations (loop-carried dependencies) introduced by this pointer
  - Trust me, this loop can be parallelized
- Often better to give the compiler information about why it is safe to parallelize (allows more optimization – "trust me it's safe" only says safe as written)



Compiler Can Help Itself, Too

- Often compilers will attempt to restructure code to find or enhance parallelism
- Some common examples
  - Scalar expansion
  - Reductions
  - Loop collapse



Scalar Expansion

- This loop can not be parallelized as written because of dependences between the reads and writes of t in different iterations (writing t in one iteration may overwrite the value of t from another iteration before it is used):

```
int t;
for (i = 0; i < n; ++i) {
  t = sqrt(b[i]);
  ...
  a[i] = t + 5;
}
```

Scalar Expansion

- This loop can not be parallelized as written because of dependences between the reads and writes of t in different iterations (writing t in one iteration may overwrite the value of t from another iteration before it is used):

```
int t[n];
for (i = 0; i < n; ++i) {
  t[i] = sqrt(b[i]);
  ...
  a[i] = t[i] + 5;
}
```

- Compiler can solve this by converting the scalar integer t into an array of integers.

Reductions

- Compilers often attempt to recognize loops that calculate sums, products, minimums, and maximums over an array. E.g.:

```
int min = MAX_VAL;
for (i = 0; i < n; i++) {
  if (x[i] < min)
    min = x[i];
}
```

- The compiler can convert these to reductions
  - Each thread computes the min/max/sum/product over a sub-section of the array
  - Threads then combine results to determine the final value (can use tree-structure for efficiency)

Loop Collapse

```
void foo(int* restrict num_bars, int size_x,
         int* restrict x, int* restrict bar) {
  for (int i = 0; i < size_x; i++)
    for (int j = 0; j < num_bars[i]; j++)
      x[i] += bar[i + j];
}
```

- How do we handle nested parallel loops?
- Option 1: Go parallel for the outer loop, and then again for the inner loop.
  - Inefficient – there is a significant overhead to going parallel. If we nest, then every iteration of the outer loop has to pay that overhead.
  - May limit effectiveness of the load balancing obtained by some loop iteration scheduling methods.

Loop Collapse

```
void foo(int* restrict num_bars, int size_x,
         int* restrict x, int* restrict bar) {
  for (int i = 0; i < size_x; i++)
    for (int j = 0; j < num_bars[i]; j++)
      x[i] += bar[i + j];
}
```

- Option 2: Loop collapse. Convert the nested pair of parallel loops to a single parallel loop that simulates the execution of the nested loops.
- Manhattan style, when inner loop iterations may vary
  - Create a new parallel loop to calculate the total number of iterations of the inner loop (across all iterations of the outer loop).
  - Convert the pair of loops into a single loop where each iteration corresponds to a distinct outer/inner iteration pair.
- If inner loop iterations are fixed, a simple rectangular collapse suffices
  - M iterations nested in N iterations = M x N collapsed loop iterations
- Big performance win

Manhattan Collapse Psuedocode

```
// t[i] = total # of inner loop iterations
// in first i iterations of outer loop
t[0] = 0;
for (i = 0; i < size_x; i++)
  t[i + 1] = t[i] + num_bars[i];
for (k = 0; k < t[size_x]; k++) {
  // Set i to index of largest element of t
  // less than k (use binary search)
  // Optimization: Don’t recompute every time
  i = max_element_less_than(t, k);
  j = k - t[i];
  x[i] += bar[i + j]; // original loop body
}
```

Example

```
bool foo(int *a, int *b, int n,
             int sought, int *old_val) {
  int i;
  for (i = 0; i < n; i++) {
    if (b[i] == sought)
      break;
    a[i] = b[i];
  }
  return (i < n);
}
```

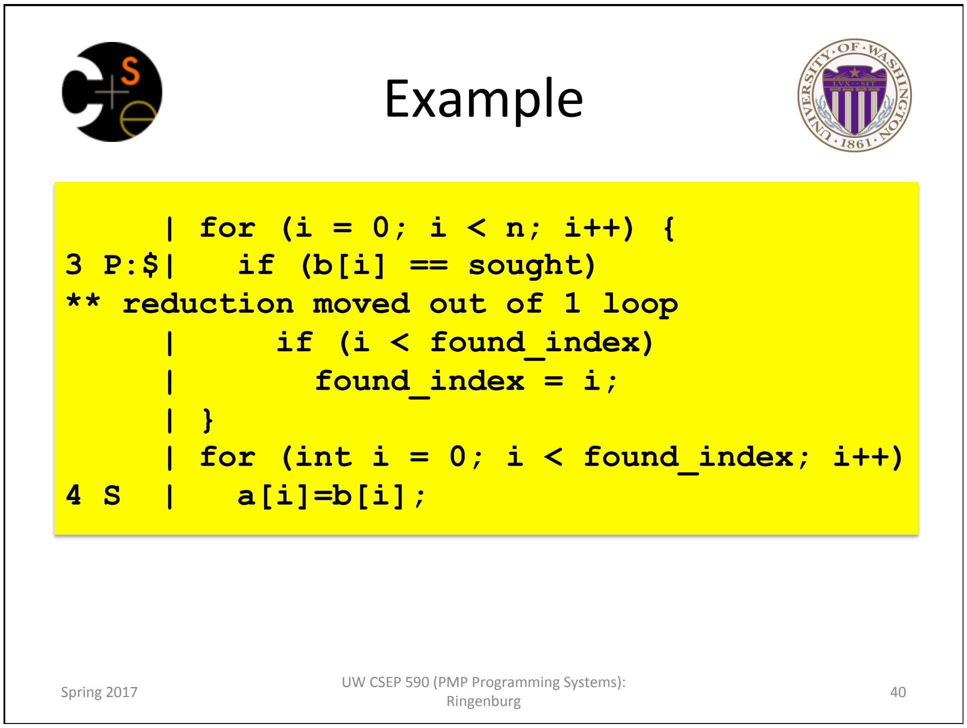
Example

```
    1 X |   for (i = 0; i < n; i++) {
** loop exit
** multiple exits
    1 X |     if (b[i] == sought)
        |       break;
    1 X |     a[i] = b[i];
        |   }
```

- Compiler feedback tools often provided to tell you where optimization/parallelization occurred or didn't, and why.

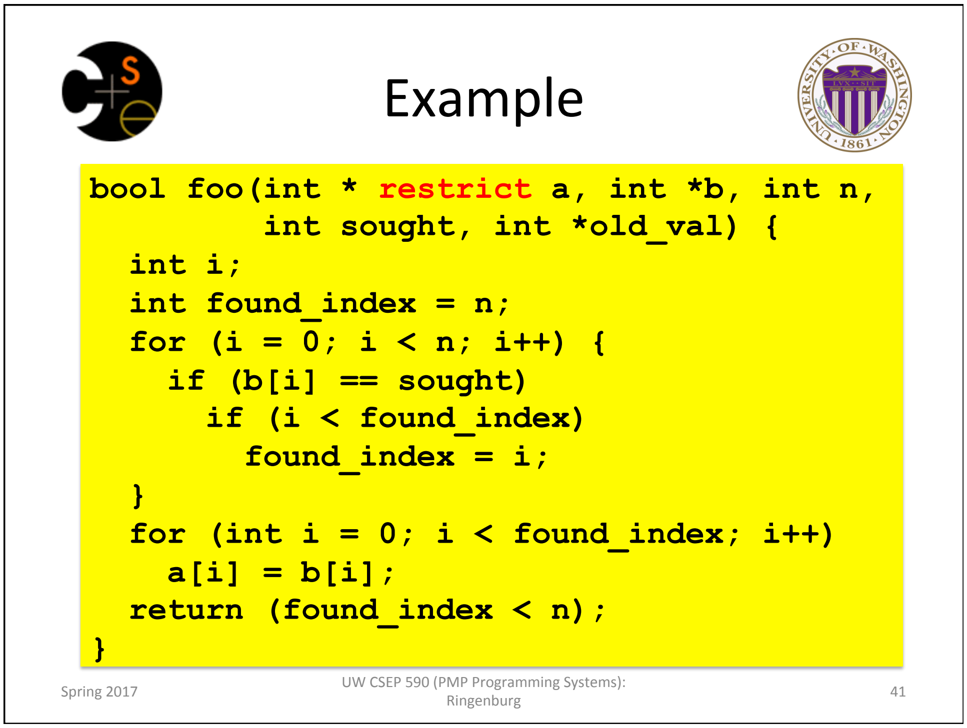
Example

```
bool foo(int *a, int *b, int n,
         int sought, int *old_val) {
  int i;
  int found_index = n;
  for (i = 0; i < n; i++) {
    if (b[i] == sought)
      if (i < found_index)
        found_index = i;
  }
  for (int i = 0; i < found_index; i++)
    a[i] = b[i];
  return (found_index < n);
}
```



Example

```
     | for (i = 0; i < n; i++) {
3 P:$|   if (b[i] == sought)
** reduction moved out of 1 loop
     |     if (i < found_index)
     |       found_index = i;
     | }
     | for (int i = 0; i < found_index; i++)
4 S  |   a[i]=b[i];
```



Example

```
bool foo(int * restrict a, int *b, int n,
         int sought, int *old_val) {
  int i;
  int found_index = n;
  for (i = 0; i < n; i++) {
    if (b[i] == sought)
      if (i < found_index)
        found_index = i;
  }
  for (int i = 0; i < found_index; i++)
    a[i] = b[i];
  return (found_index < n);
}
```

Example

```
      | for (i = 0; i < n; i++) {
3 P:$|   if (b[i] == sought)
** reduction moved out of 1 loop
      |     if (i < found_index)
      |       found_index = i;
      | }
      | for (int i = 0; i < found_index; i++)
4 P   |   a[i]=b[i];
```

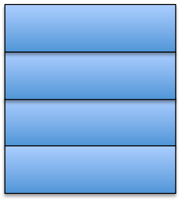
The JVM and JIT

Java Implementation Overview

- Java compiler (javac et al) produces machine-independent .class files
  - Target architecture is Java Virtual Machine (JVM) – simple stack machine
- Java execution engine (java)
  - Loads .class files (often from libraries)
  - Executes code
    - Either interprets stack machine code or compiles to native code (JIT)

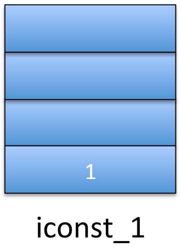
JVM Architecture

- Abstract stack machine
  - Bytecodes pop operands,
  - and push results
- Implementation not required to use JVM specification literally
  - Only requirement is that execution of .class files has specified effect
  - Multiple implementation strategies depending on goals
    - Compilers vs interpreters
    - Optimizing for servers vs workstations



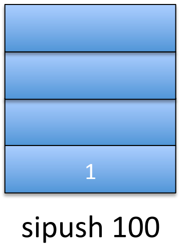
JVM Architecture

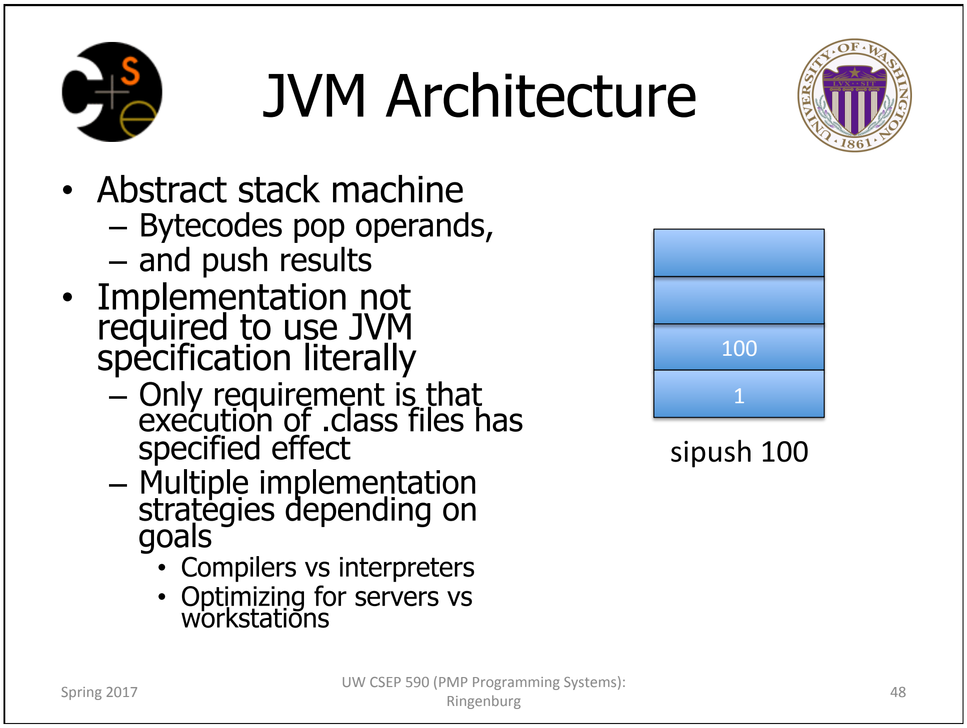
- Abstract stack machine
  - Bytecodes pop operands,
  - and push results
- Implementation not required to use JVM specification literally
  - Only requirement is that execution of .class files has specified effect
  - Multiple implementation strategies depending on goals
    - Compilers vs interpreters
    - Optimizing for servers vs workstations



JVM Architecture

- Abstract stack machine
  - Bytecodes pop operands,
  - and push results
- Implementation not required to use JVM specification literally
  - Only requirement is that execution of .class files has specified effect
  - Multiple implementation strategies depending on goals
    - Compilers vs interpreters
    - Optimizing for servers vs workstations

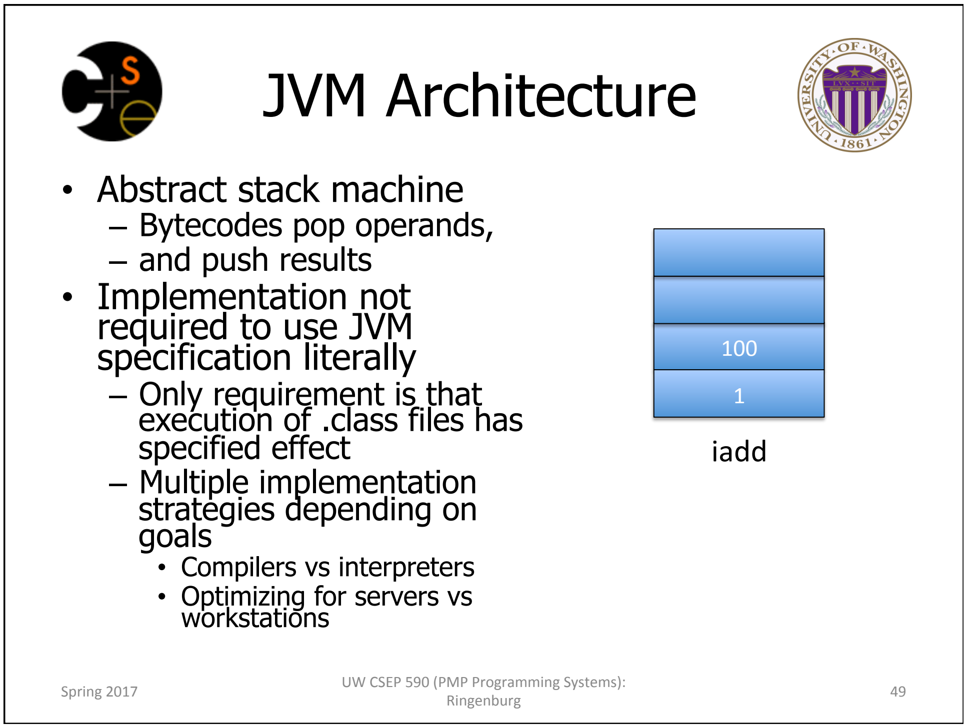




JVM Architecture

- Abstract stack machine
  - Bytecodes pop operands,
  - and push results
- Implementation not required to use JVM specification literally
  - Only requirement is that execution of .class files has specified effect
  - Multiple implementation strategies depending on goals
    - Compilers vs interpreters
    - Optimizing for servers vs workstations

sipush 100



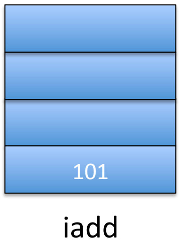
JVM Architecture

- Abstract stack machine
  - Bytecodes pop operands,
  - and push results
- Implementation not required to use JVM specification literally
  - Only requirement is that execution of .class files has specified effect
  - Multiple implementation strategies depending on goals
    - Compilers vs interpreters
    - Optimizing for servers vs workstations

iadd

JVM Architecture

- Abstract stack machine
  - Bytecodes pop operands,
  - and push results
- Implementation not required to use JVM specification literally
  - Only requirement is that execution of .class files has specified effect
  - Multiple implementation strategies depending on goals
    - Compilers vs interpreters
    - Optimizing for servers vs workstations



JVM Data Types

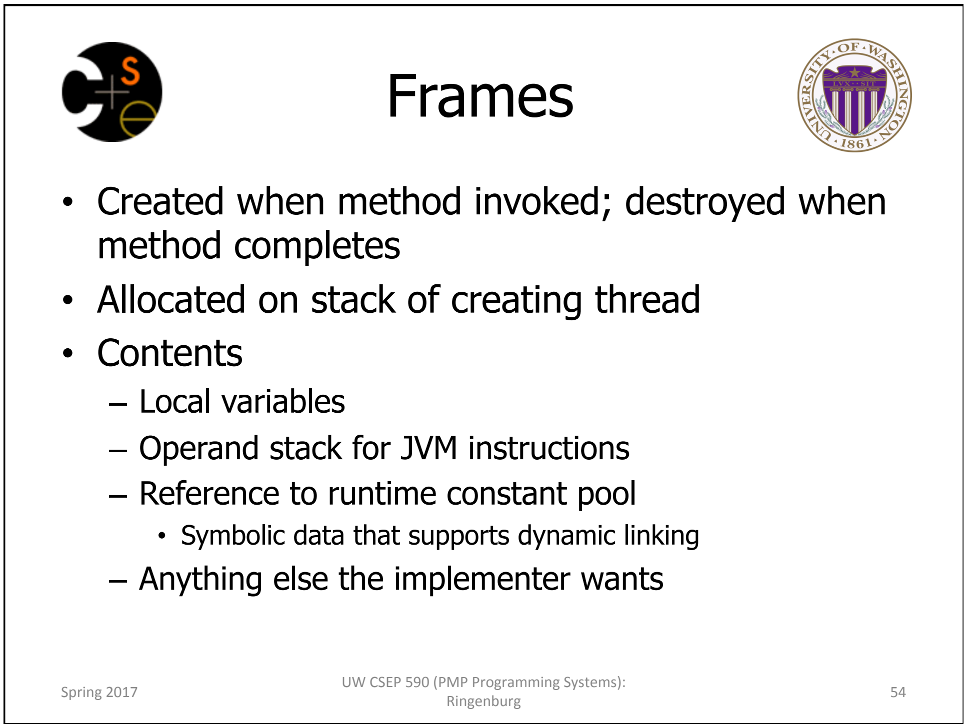
- Primitive types
  - byte, short, int, long, char, float, double, boolean
- Reference types
  - Non-generic only (more on this later)

JVM Runtime Data Areas

- Semantics defined by the JVM Specification
  - Implementer may do anything that preserves these semantics
- Per-thread data
  - pc register
  - Stack
    - Holds frames (details below)
    - May be a real stack or may be heap allocated

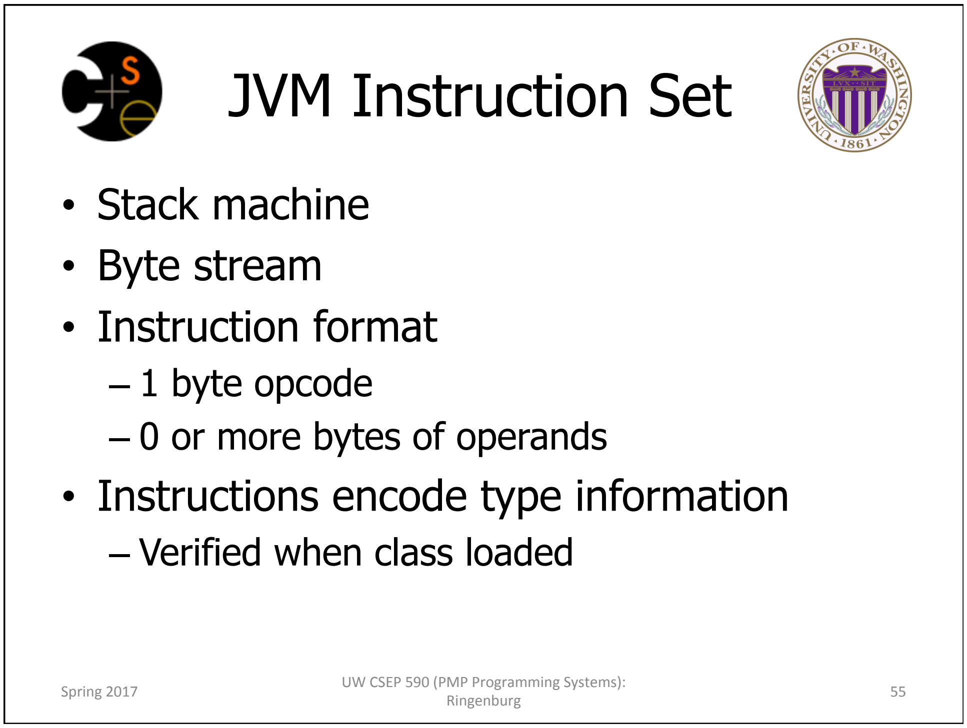
JVM Runtime Data Areas

- Per-VM data – shared by all threads
  - Heap – objects allocated here
  - Method area – per-class data
    - Runtime constant pool
    - Field and method data
    - Code for methods and constructors



Frames

- Created when method invoked; destroyed when method completes
- Allocated on stack of creating thread
- Contents
  - Local variables
  - Operand stack for JVM instructions
  - Reference to runtime constant pool
    - Symbolic data that supports dynamic linking
  - Anything else the implementer wants



JVM Instruction Set

- Stack machine
- Byte stream
- Instruction format
  - 1 byte opcode
  - 0 or more bytes of operands
- Instructions encode type information
  - Verified when class loaded

Instruction Sampler

- Load/store
  - Transfer values between local variables and operand stack
  - Different opcodes for int, float, double, addresses
  - Load, store, load immediate
    - Special encodings for load0, load1, load2, load3 to get compact code for first few local vars

---

Instruction Sampler

- Arithmetic
  - Again, different opcodes for different types
    - byte, short, char & boolean use int instructions
  - Pop operands from operand stack, push result onto operand stack
  - Add, subtract, multiply, divide, remainder, negate, shift, and, or, increment, compare
- Stack management
  - Pop, dup, swap

Instruction Sampler

- Type conversion
  - Widening – int to long, float, double; long to float, double, float to double
  - Narrowing – int to byte, short, char; double to int, long, float, etc.

Instruction Sampler

- Object creation & manipulation
  - New class instance
  - New array
  - Static field access
  - Array element access
  - Array length
  - Instanceof, checkcast

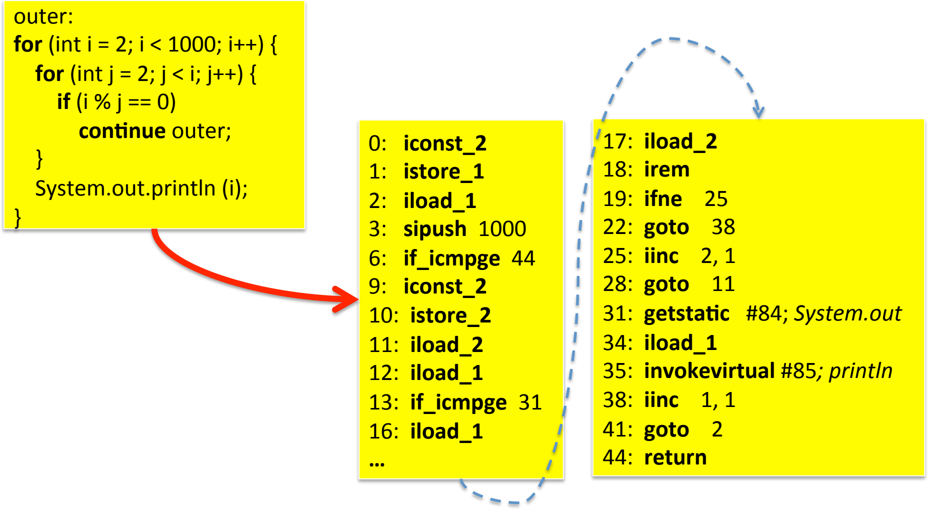
Instruction Sampler

- Control transfer
  - Unconditional branch – goto
  - Conditional branch – ifeq, iflt, ifnull, etc.
  - Compound conditional branches - switch

Instruction Sampler

- Method invocation
  - invokevirtual
  - invokeinterface
  - invokespecial (constructors, superclass, private)
  - invokestatic
- Method return
  - Typed value-returning instructions
  - Return for void methods

Bytecode Example



```
outer:
for (int i = 2; i < 1000; i++) {
   for (int j = 2; j < i; j++) {
      if (i % j == 0)
         continue outer;
   }
   System.out.println (i);
}
```

```
0:  iconst_2
1:  istore_1
2:  iload_1
3:  sipush  1000
6:  if_icmpge  44
9:  iconst_2
10:  istore_2
11:  iload_2
12:  iload_1
13:  if_icmpge  31
16:  iload_1
…
```

```
17:  iload_2
18:  irem
19:  ifne    25
22:  goto    38
25:  iinc    2, 1
28:  goto    11
31:  getstatic   #84; System.out
34:  iload_1
35:  invokevirtual #85; println
38:  iinc    1, 1
41:  goto    2
44:  return
```

Execution Engines

- Basic Choices
  - Interpret JVM bytecodes directly
  - Compile bytecodes to native code, which then executes on the native processor
    - Just-In-Time compiler (JIT)

JIT Levels

- C1: Fast, simple light-weight optimizations
  - Often used for shorter codes (“client” mode)
- C2: More aggressive optimization, significantly slower
  - Often used for long running codes (“server” mode)
- But both cause overhead when invoked

JIT Profiling

- JIT compilation typically profile-driven
  - Compilation (even C1) has a cost
  - Count executions of methods, identify hot loop nests
  - JIT only hot code
- Tiered JIT
  - Multiple hot-ness thresholds
  - Use light-weight JIT (C1) once hit lower threshold
  - Pay cost of heavy-weight JIT (C2) if hit higher threshold

Speculative JIT

- Profiling can do more than just identify code blocks executed
  - Certain branches always/never taken
  - Actual types used/method implementations called
  - Detect if NULL pointers never passed
  - Etc…
- Can speculatively optimize based on profile
  - Remove unused branches
  - Inline particular implementations of virtual methods
  - Remove NULL checks
- Must detect and back off if speculation incorrect
  - Detect via "guards", e.g., checking type or condition, handling SIGSEGV
  - "Deoptimization" … switch back to interpreter
  - Switching back can be expensive

Speculation Example

```
o.foo(a, b, b);
```

↓

```
o = receiver object ;
x = receiver class (o) ;
if ( x == expected-class ) {     // virtual guard
  x.foo(a, b, c);  // direct call can be inlined
} else {
  o.foo(a, b, c);  // guard failed, virtual call
}
```

From IBM Just-In-Time Compiler (JIT) for Java: Best practices and coding guidelines for improving performance

Escape Analysis

- Another optimization based on observation that many methods allocate local objects as temporaries
- Idea: Compiler tries to prove that no reference to a locally allocated object can "escape"
  - Not stored in a global variable or object
  - Not passed as a parameter

Using Escape Analysis

- If all references to an object are local, it doesn't need to be allocated on the heap in the usual manner
  - Can allocate storage for it in local stack frame
    - Essentially zero cost
  - Still need to preserve the semantics of new, constructor, etc.

Other Techniques

- Save profile information from previous executions
  - Can also save JIT'ed results
  - Eliminate "warm-up"
  - Possibly better profile info than fake warm-ups sometimes employed
- Azul Falcon: recently announced JIT using LLVM
  - Take advantage of powerful open source compiler
  - More optimizations potentially available
  - More processor specific optimizations – e.g., vectorization instructions

---

SQL Query Optimization

(Excerpts from Spark Summit Catalyst Optimizer Deep Dive, Spark Summit 2016)