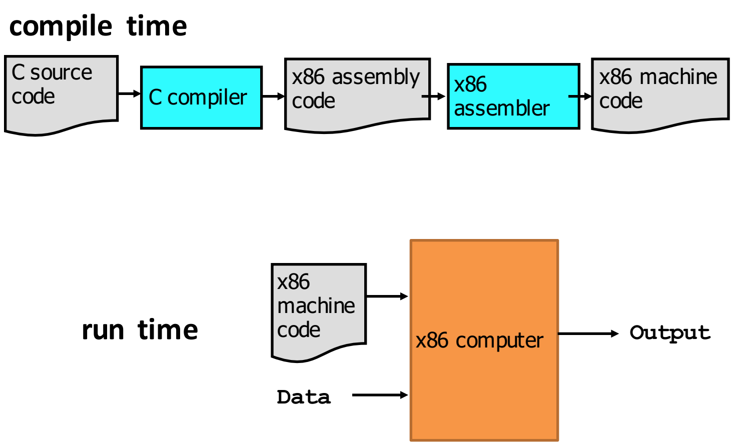
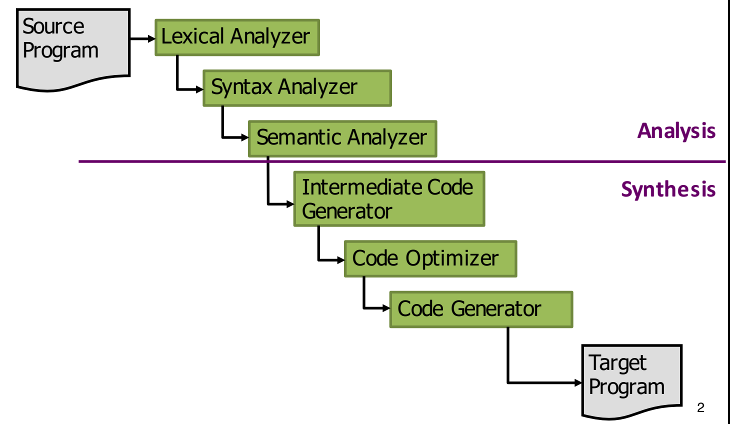
Ahead-of-time compiler



Figures for compilers/runtime systems adapted from slides by Steve Freund.

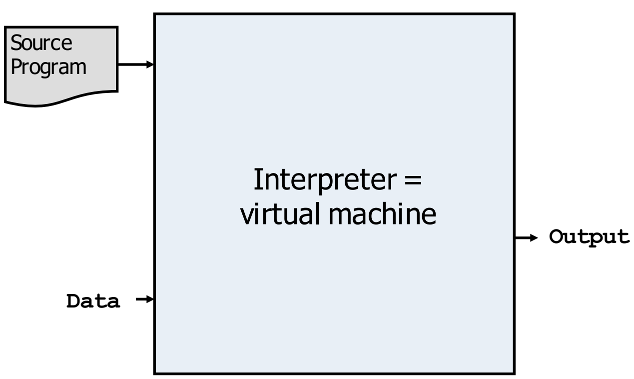
1

Typical Compiler



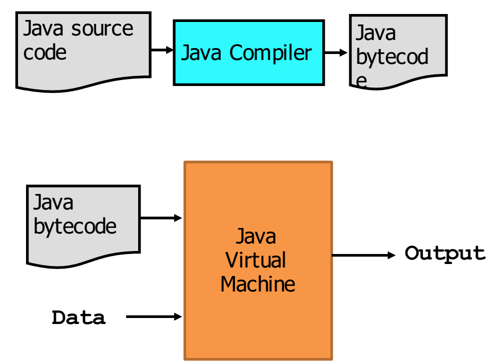
2

Interpreter



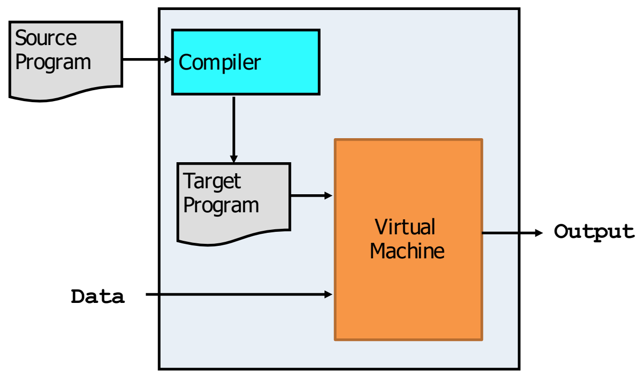
3

Compilers… that target interpreters

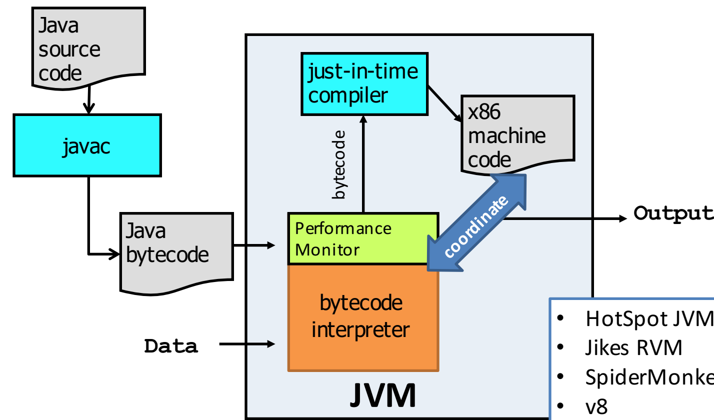


5

Interpreters… that use compilers.

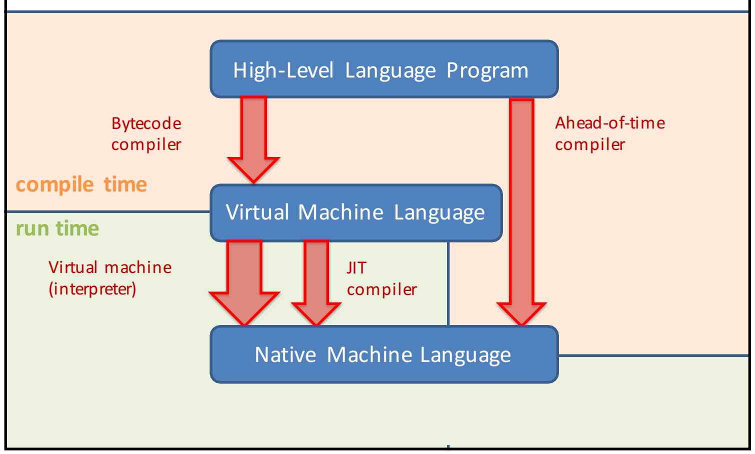


JIT Compilers and Optimization



- HotSpot JVM
- Jikes RVM
- SpiderMonkey
- v8
- Transmeta
- …

Virtual Machine Model



On translation, layout, and implementation

We show natural, common, or conventional translations.

Java: No guarantee of this implementation/layout.
Language is (mostly clean) abstraction.

C: Much of implementation/layout guaranteed.
Language exposes many machine details.

Data in Java

Integers, floats, doubles, pointers – same as C

Null is typically represented as 0

Characters and strings

Arrays

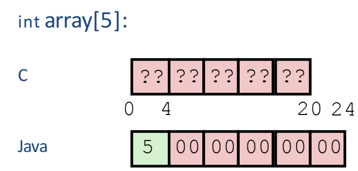
Objects

- pointers? called 'references' – much more constrained

Data in Java

Arrays

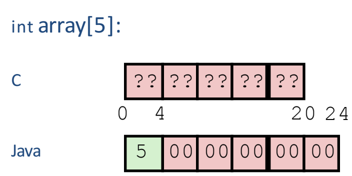
- Every element initialized to 0 or null
- Length specified in immutable field at start of array (int – 4 bytes)
  - *array.length* returns value of this field
  - *Since it has this info, what can it do?*



Data in Java

Arrays

- Every element initialized to 0 or null
- Length specified in immutable field at start of array (int – 4 bytes)
  - *array.length* returns value of this field
- Every access triggers a bounds-check
  - Code is added to ensure the index is within bounds
  - Exception if out-of-bounds



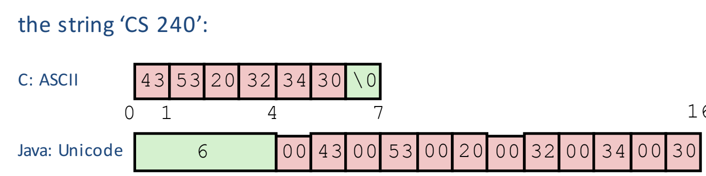
Bounds-checking sounds slow, but:

1. Length is likely in cache.
2. Compiler may store length in register for loops.
3. Compiler may prove that some checks are redundant.

Data in Java

Characters and strings

- Two-byte Unicode instead of ASCII
  - Represents most of the world's alphabets
- String not bounded by a '\0' (null character)
  - Bounded by hidden length field at beginning of string



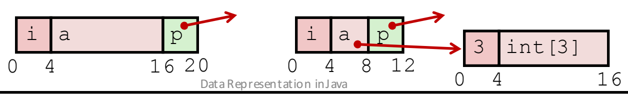
Data structures (objects) in Java

Objects are always stored by reference, never stored inline.

Include complex data types (arrays, other objects, etc.) using *references*

C
```
struct rec {
  int i;
  int a[3];
  struct rec *p;
};
```

Java
```
class Rec {
  int i;
  int[] a = new int[3];
  Rec p;
…
}
```



Pointer/reference fields and variables

In C, we have "->" and "." for field selection depending on whether we have a pointer to a struct or a struct

(*r).a is so common it becomes r->a

In Java, *all non-primitive variables are references to objects*

We always use r.a notation

But really follow reference to r with offset to a, just like C's r->a

```
struct rec *r = malloc(...);
struct rec r2;
r->i = val;
r->a[2] = val;
r->p = &r2;
```

```
r = new Rec();
r2 = new Rec();
r.i = val;
r.a[2] = val;
r.p = r2;
```


Pointers/References

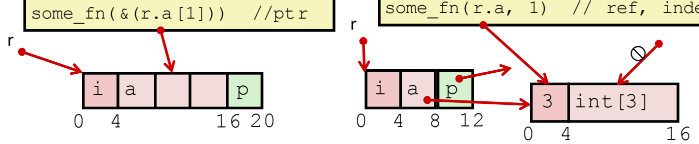
Pointers in C can point to any memory address

References in Java can only point to [the starts of] objects

And can only be dereferenced to access a field or element of that object

C
```
struct rec {
  int i;
  int a[3];
  struct rec *p;
};
struct rec* r = malloc(…);
some_fn(&(r.a[1]))  //ptr
```

Java
```
class Rec {
  int i;
  int[] a = new int[3];
  Rec p;
}
Rec r = new Rec();
some_fn(r.a, 1)  // ref, index
```



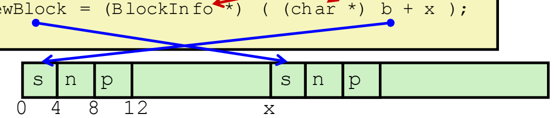
Casting in C

We can cast any pointer into any other pointer; just look at the same bits differently

```
struct BlockInfo {
    int sizeAndTags;
    struct BlockInfo* next;
    struct BlockInfo* prev;
};
typedef struct BlockInfo BlockInfo;
…
int x;
BlockInfo *b;
BlockInfo *newBlock;
…
newBlock = (BlockInfo *) ( (char *) b + x );
…
```

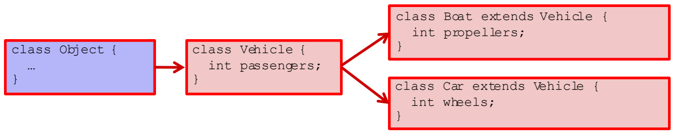
Cast b into char pointer so that you can add byte offset without scaling

Cast back into BlockInfo pointer so you can use it as BlockInfo struct



Type-safe casting in Java

Can only cast compatible object references




```
// Vehicle is a super class of Boat and Car, which are siblings
Vehicle  v = new Vehicle();
Car      c1 = new Car();
Boat     b1 = new Boat();
Vehicle v1 = new Car();  // ok, everything needed for Vehicle
                // is also in Car
Vehicle v2 = v1;         // ok, v1 is already a Vehicle
Car      c2 = new Boat(); // incompatible type – Boat and
                          // Car are siblings
Car      c3 = new Vehicle();  // wrong direction; elements in Car
                              // not in Vehicle (wheels)
Boat     b2 = (Boat) v;   // run-time error; Vehicle does not contain
                          // all elements in Boat (propellers)
Car      c4 = (Car) v2;   // ok, v2 started out as Car
Car      c5 = (Car) b1;   // incompatible types, b1 is Boat
```

How is this implemented / enforced?

Java objects

```
class Point {
  int x;
  int y;

  Point() {
    x = 0;
    y = 0;
  }

  boolean samePlace(Point p) {
    return (x == p.x) && (y == p.y);
  }
  String toString() {
    return "(" + x + "," + y + ")";
  }
}
```

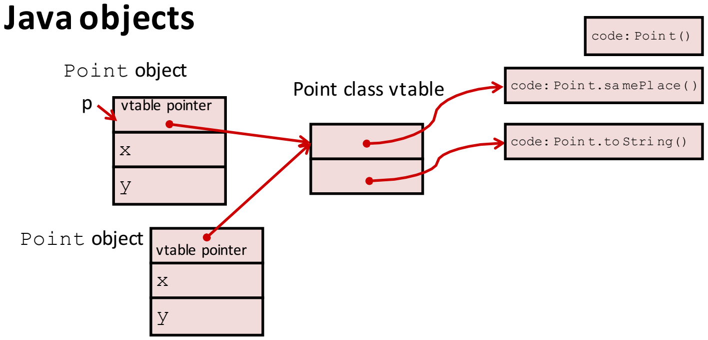
fields

constructor

methods

20

Java objects



For each class, compiler maps: field signature → offset (index)

vtable* pointer : points to per-class *virtual method table (vtable)

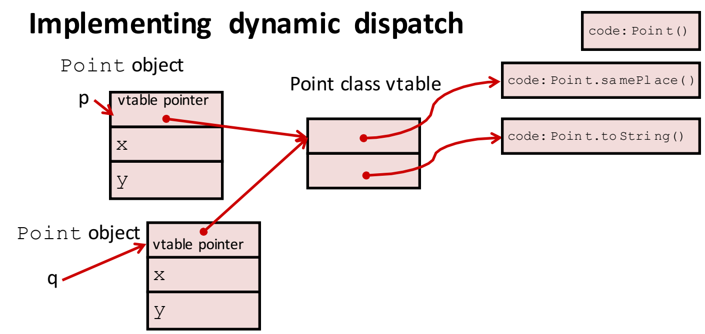
For each class, compiler maps: method signature → index

samePlace: 0

toString: 1

22

Implementing dynamic dispatch



Java:

```
Point p = new Point();
```

what happens (pseudo code):

```
Point* p = calloc(1,sizeof(Point))
p->header = ...;
p->vtable = &Point_vtable;
Point_constructor(p);
```

```
return p.samePlace(q);
```

```
return p.vtable[0](this=p, q);
```

Subclassing

```
class ColorPoint extends Point{
   String color;
   boolean getColor() {
       return color;
   }
   String toString() {
      return super.toString() + "[" + color + "]";
   }
 }
```

How do we access superclass pieces?

fields

inherited methods

Where do we put extensions?

new field

new method

overriding method

dynamic (method) dispatch

Java:

```
Point p = ???;
return p.toString();
```

what happens (pseudo code):

```
return p.vtable[1](p);
```