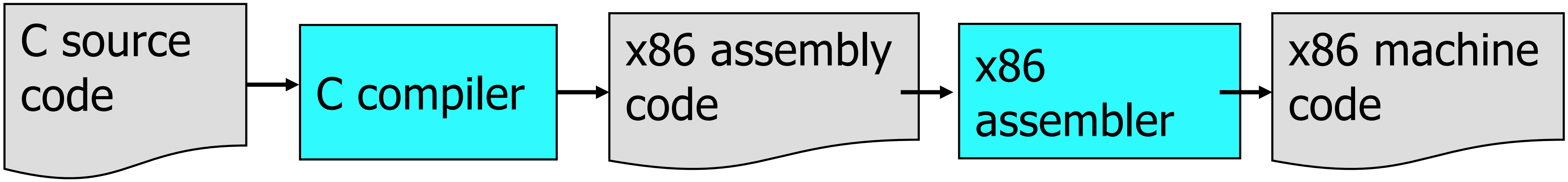
Implementing Higher-Level Languages

Quick tour of programming language implementation techniques.

From the Java level to the C level.

Ahead-of-time compiler

compile time



C source code

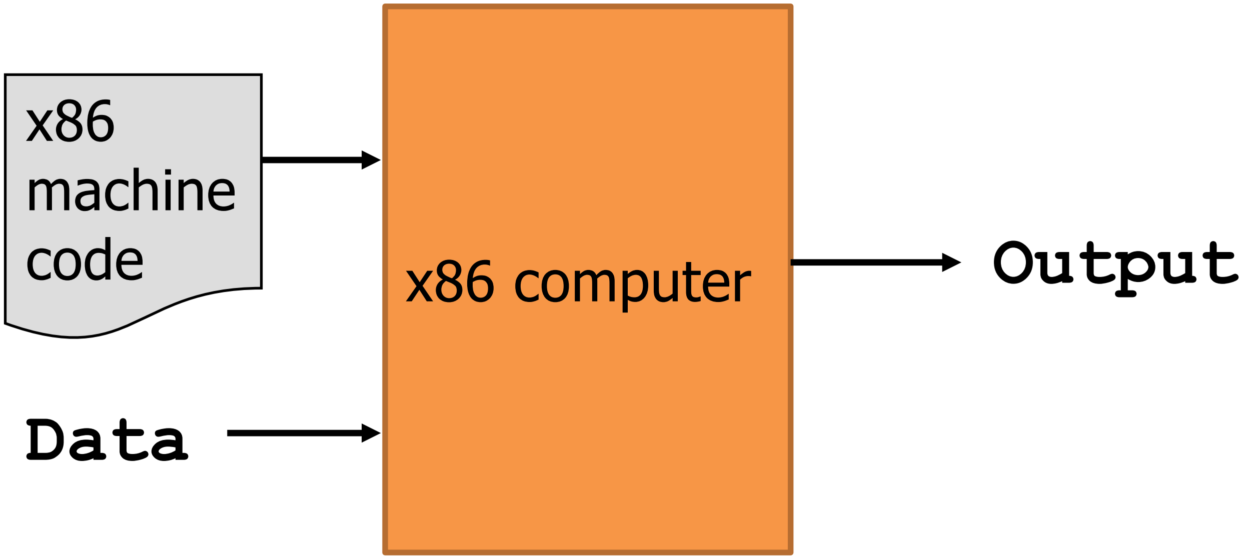
C compiler

x86 assembly code

x86 assembler

x86 machine code

run time



x86 machine code

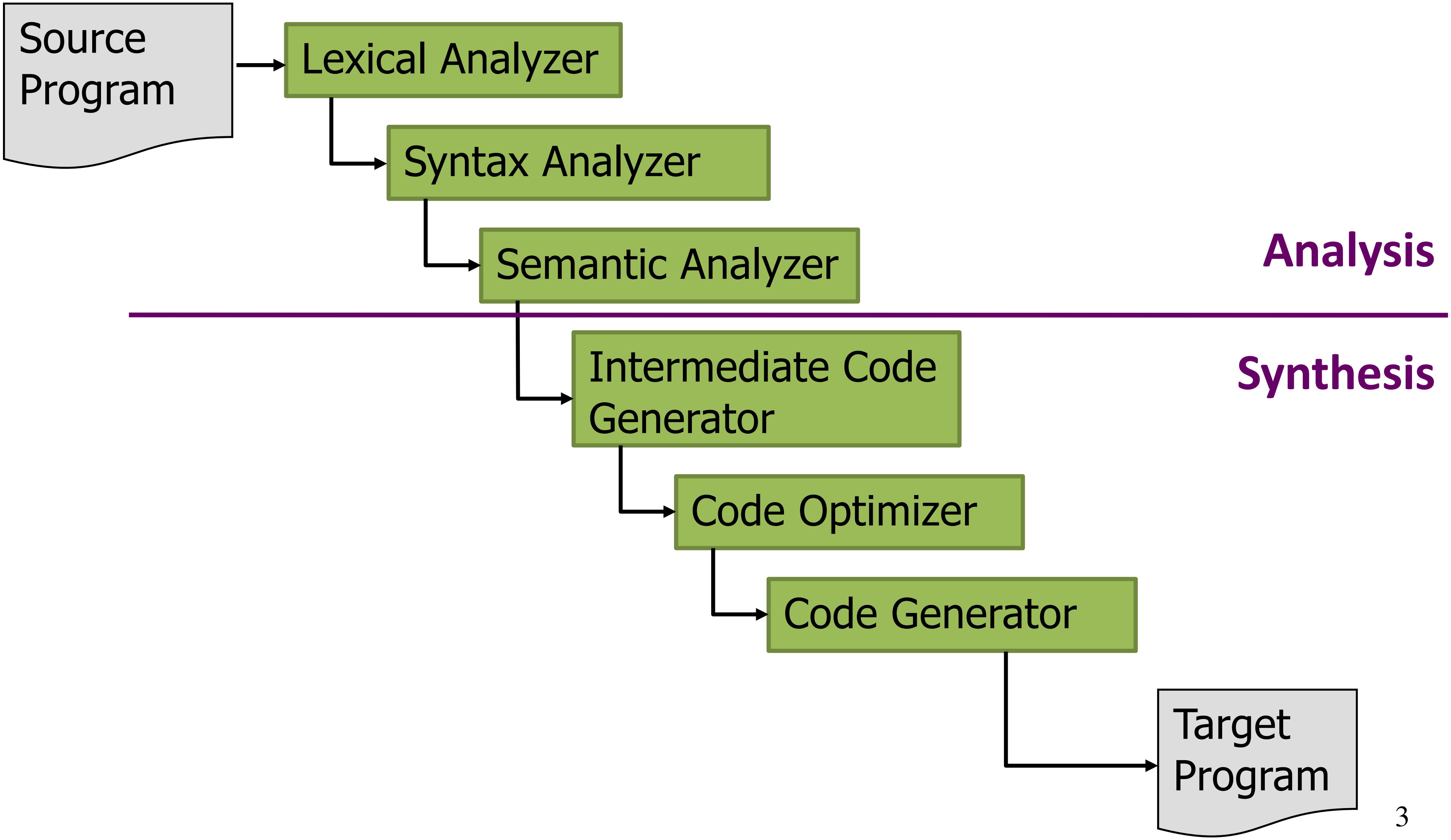
Data

x86 computer

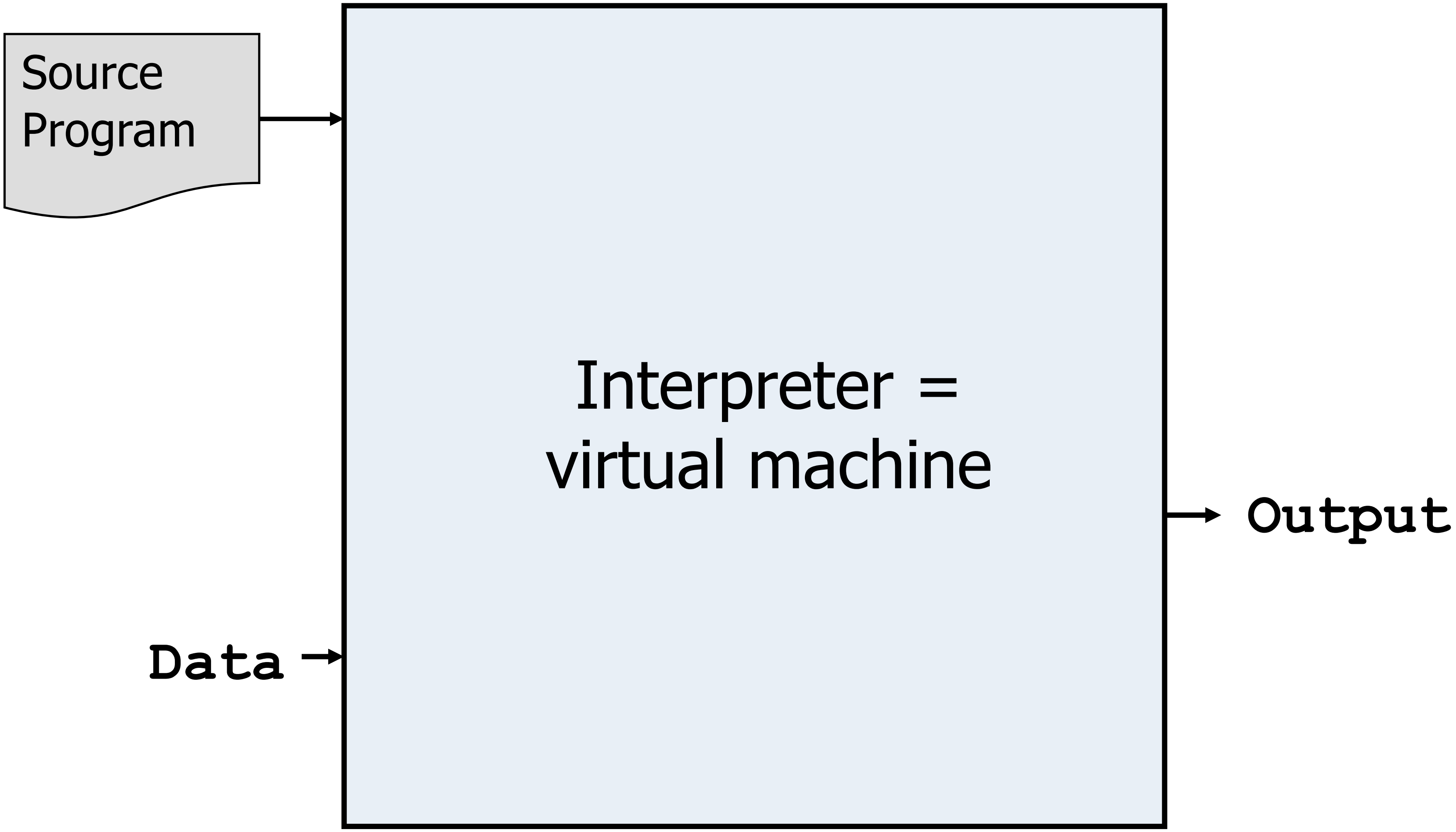
Output

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

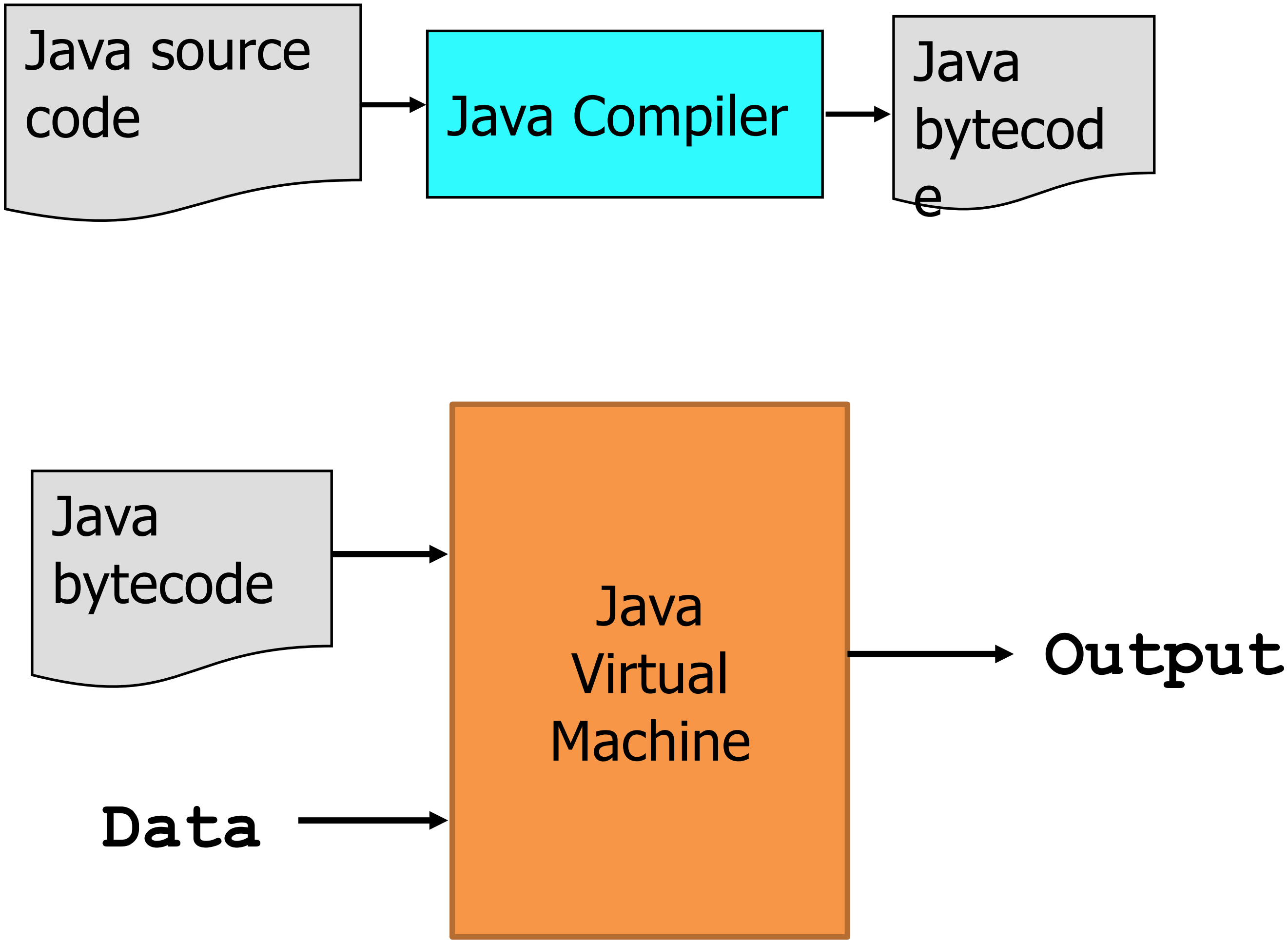
Typical Compiler



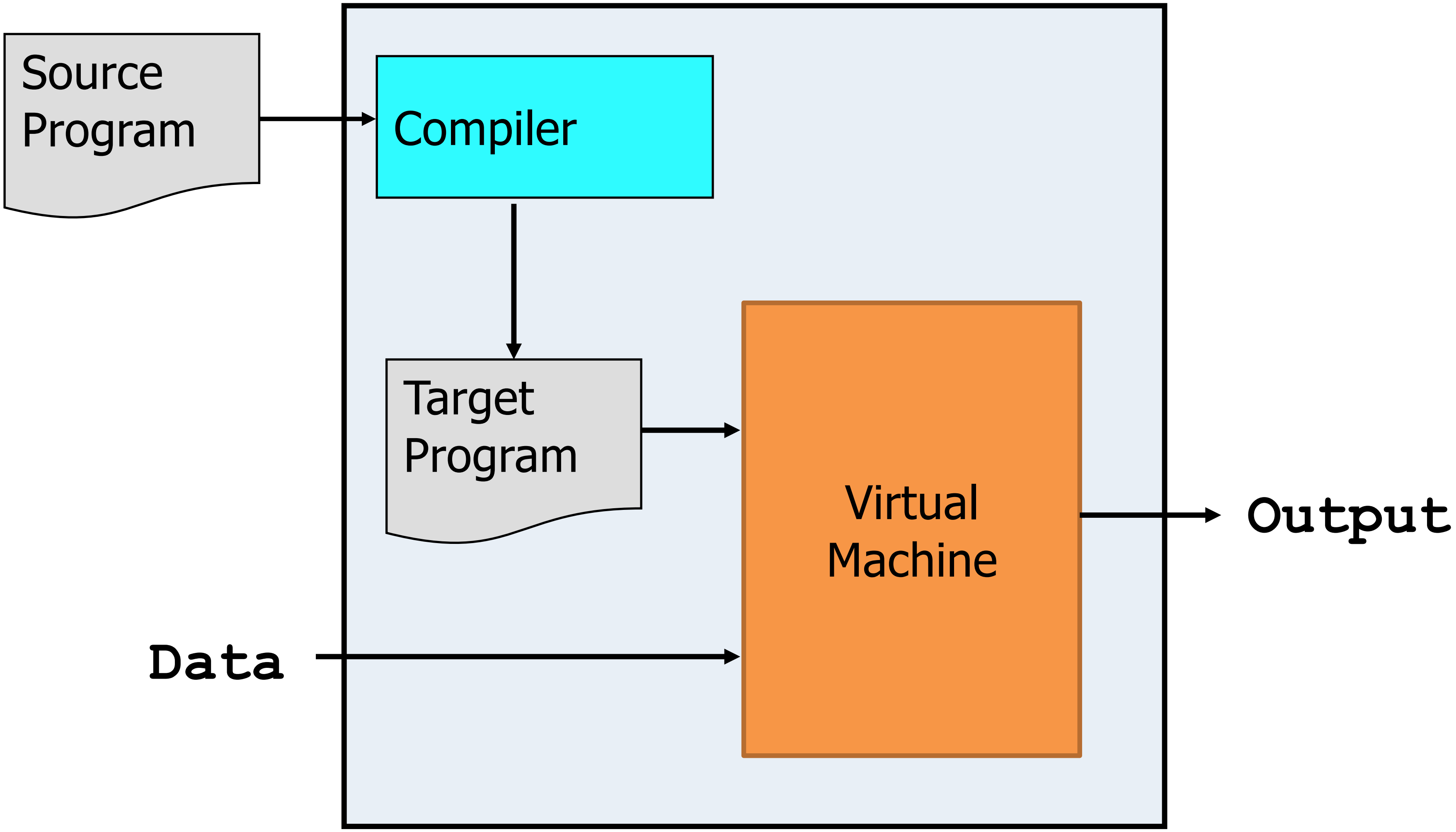
Interpreter



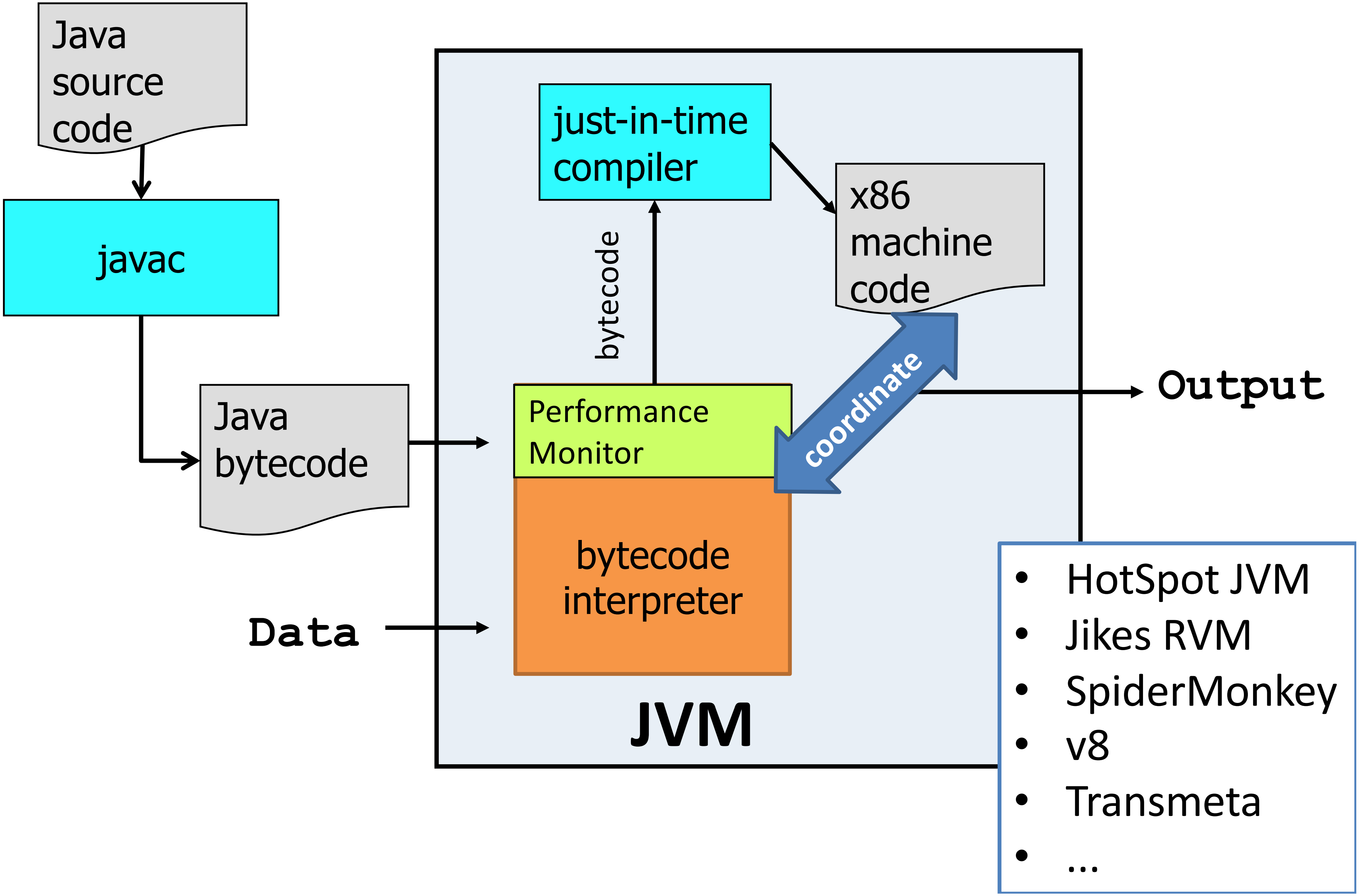
Compilers... that target interpreters



Interpreters... that use compilers.



JIT Compilers and Optimization



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

Data in Java

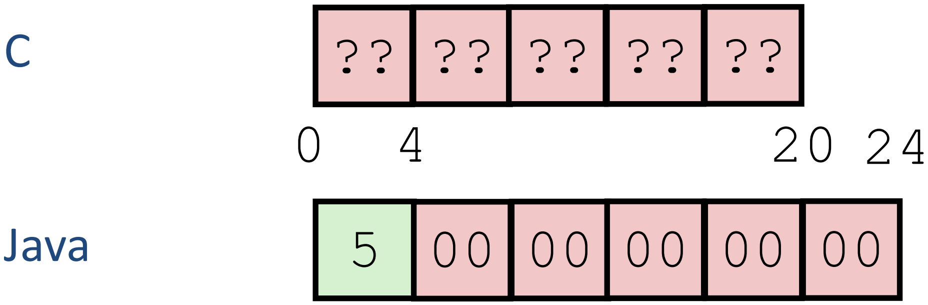
Arrays

Every element initialized to 0 or null

Immutable length field

Since it has this info, what can it do?

int array[5]:



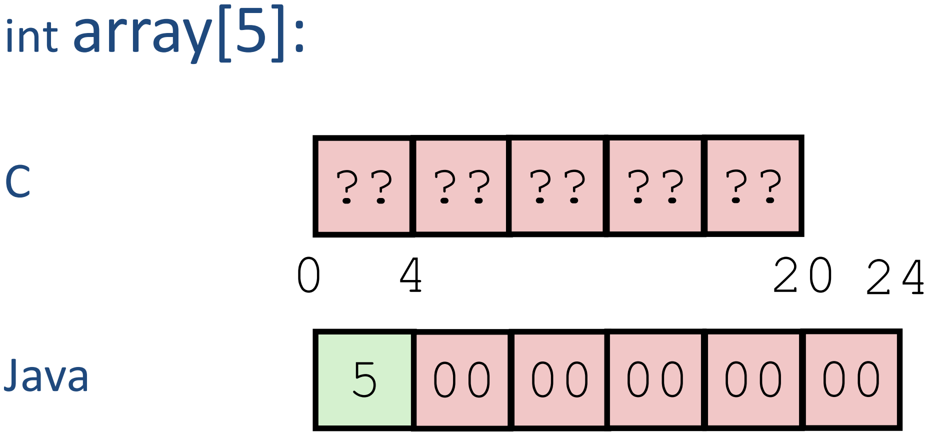
Data in Java

Arrays

Every element initialized to 0 or null

Immutable length field

Bounds-check every access.



int array[5]:

C

?? ?? ?? ?? ??

0 4 20 24

Java

5 00 00 00 00 00

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.

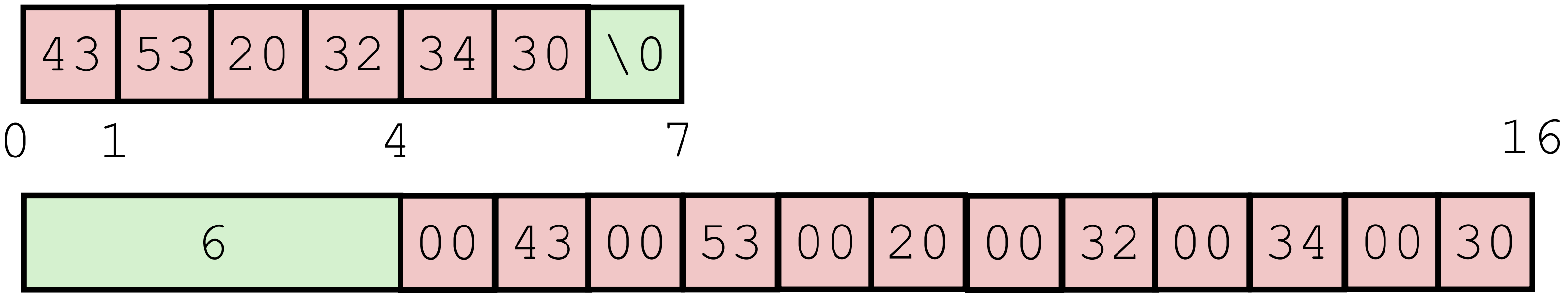
Data in Java

Characters and strings

16-bit Unicode

Explicit length, no null terminator

the string ‘CS 240’:



C: ASCII 43 53 20 32 34 30 \0

0 1 4 7 16

Java: Unicode 6 00 43 00 53 00 20 00 32 00 34 00 30

Data structures (objects) in Java

C: programmer controls layout, inline vs. pointer.

Java: objects always stored by reference, never stored inline.

C

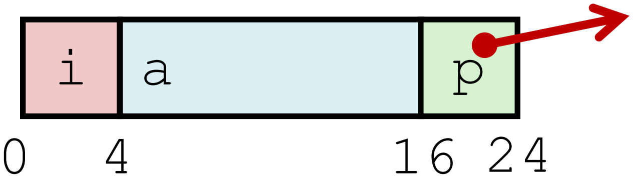
```
struct rec {
  int i;
  int a[3];
  struct rec *p;
};
```

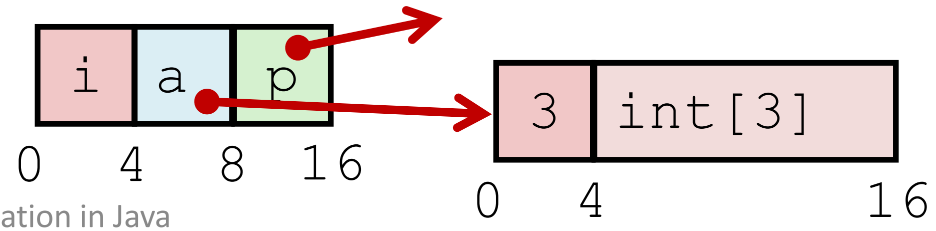
Java

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

```
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

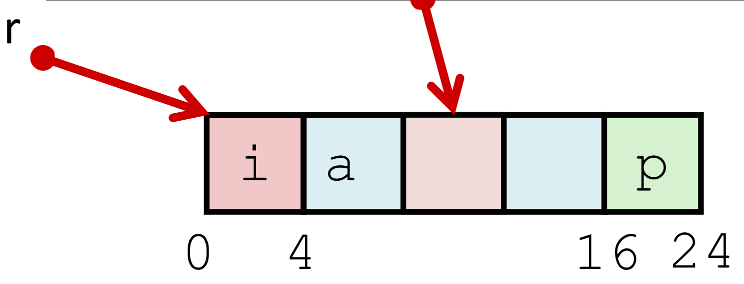
- **Pointers in C can point to any memory address**
- **References in Java can only point to** [the starts of] **objects**
  - And can only be dereferenced to access a field or element of that object

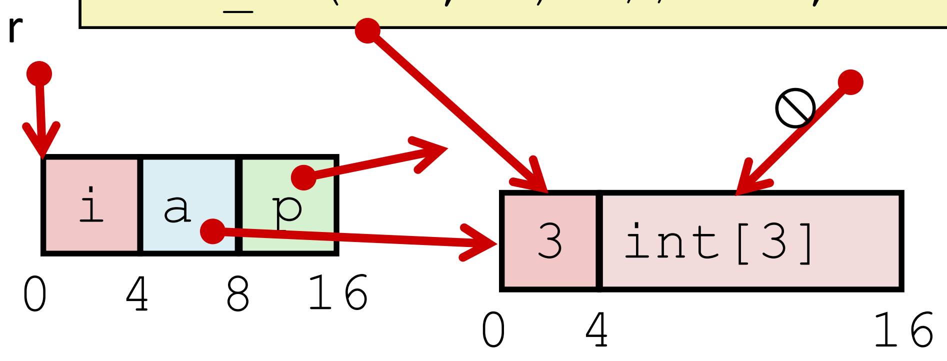
C

```
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
```





Java objects

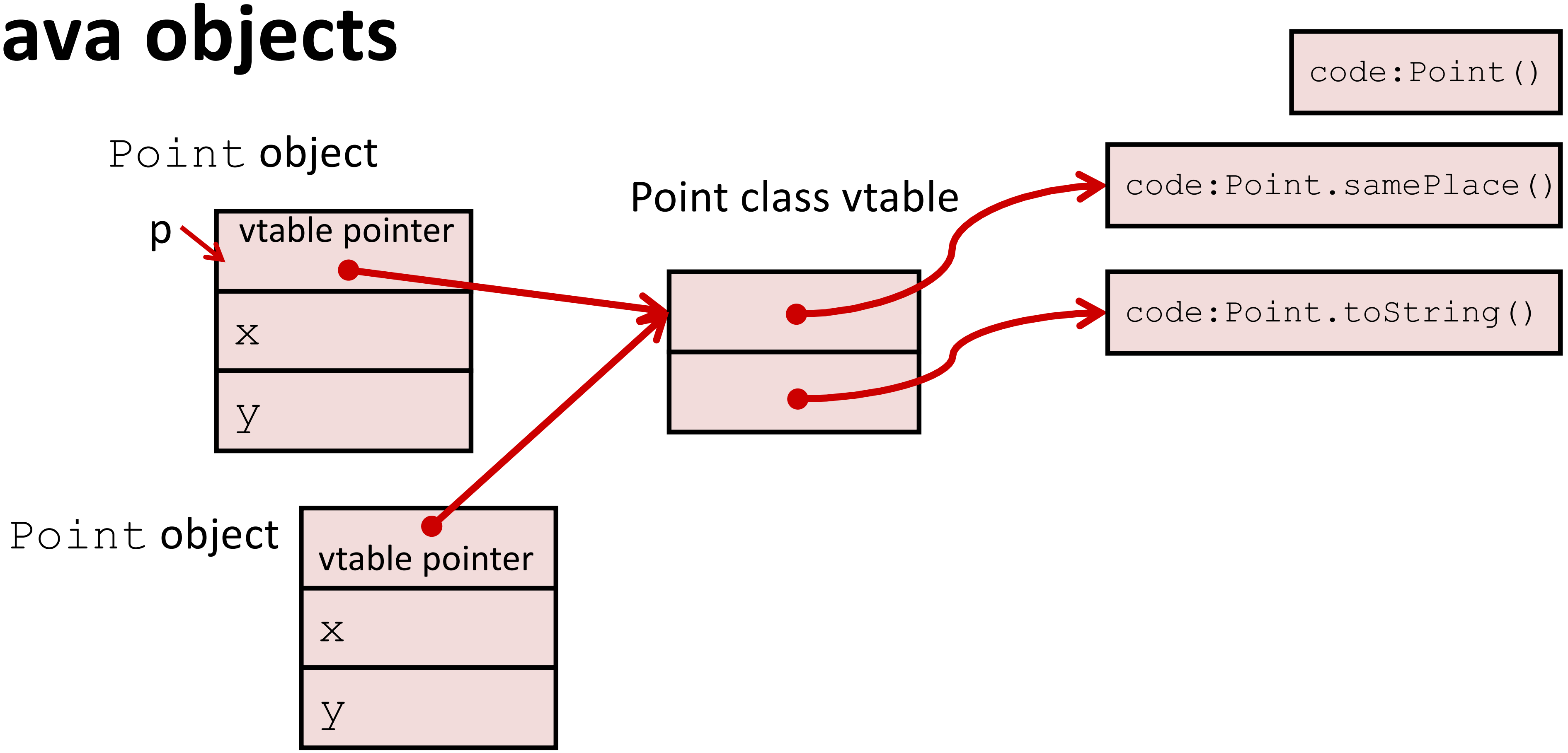
```
class Point {
  int x;
  int y;

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

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

fields

constructor

methods

Java objects



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

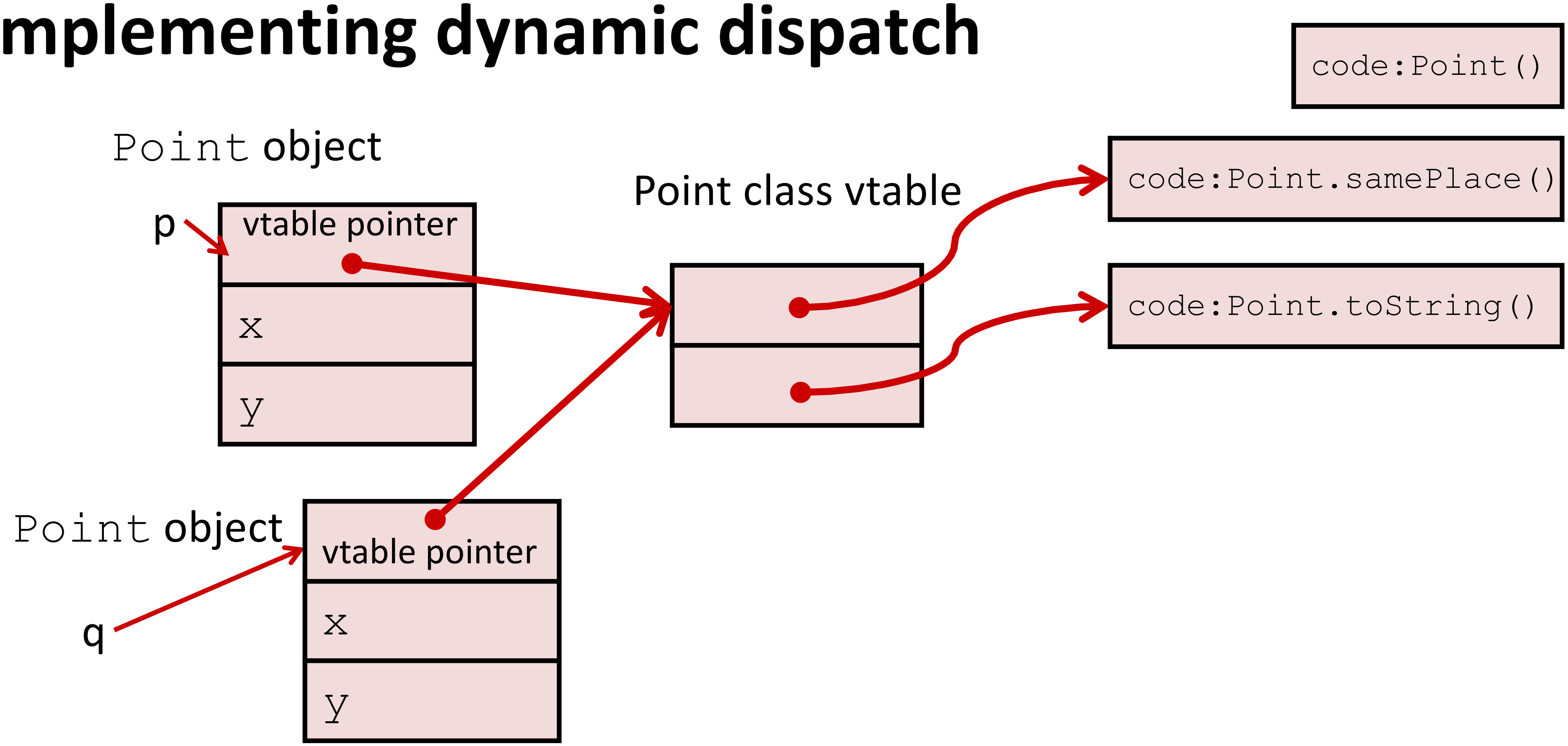
vtable* pointer : points to per-class *virtual method table (vtable)

For each class, compiler maps: method signature → index

samePlace: 0

toString: 1

Implementing dynamic dispatch



what happens (pseudo code):

Java:

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

```
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

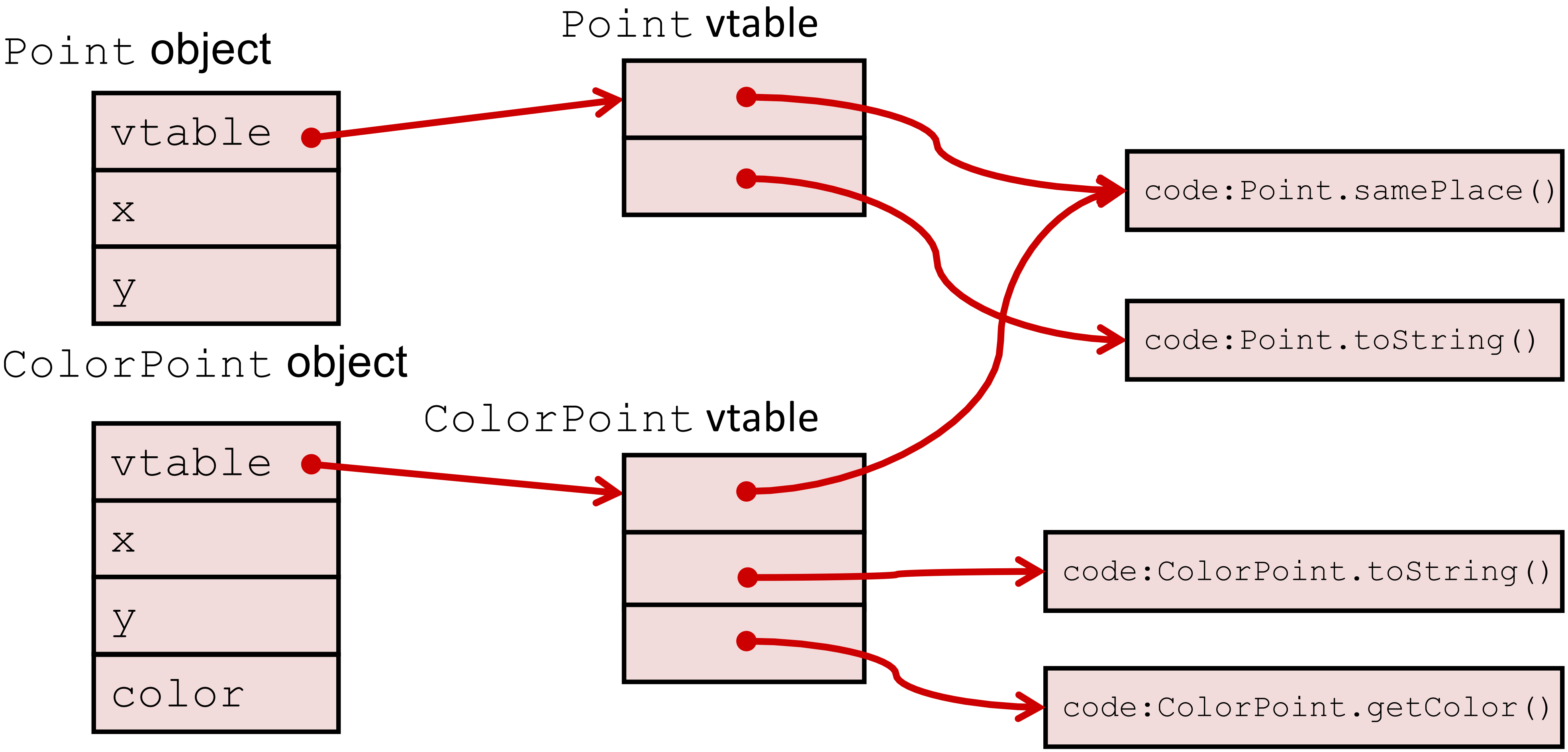
dynamic (method) dispatch

Java:

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

what happens (pseudo code):

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


Point object
vtable
x
y
ColorPoint object
vtable
x
y
color
Point vtable
ColorPoint vtable
code:Point.samePlace()
code:Point.toString()
code:ColorPoint.toString()
code:ColorPoint.getColor()