Today

- More on memory bugs
- Java vs C, the battle.

Memory-Related Perils and Pitfalls

- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks
- Failing to free blocks

Dereferencing Bad Pointers

- The classic `scanf` bug

```
int val;

...

scanf("%d", val);
```

Reading Uninitialized Memory

- **Assuming that heap data is initialized to zero**

```
/* return y = Ax */
int *matvec(int **A, int *x) {
   int *y = malloc( N * sizeof(int) );
   int i, j;

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         y[i] += A[i][j] * x[j];
   return y;
}
```

Overwriting Memory

- Allocating the (possibly) wrong sized object

```
int **p;

p = malloc( N * sizeof(int) );

for (i=0; i<N; i++) {
   p[i] = malloc( M * sizeof(int) );
}
```

Overwriting Memory

- **Off-by-one error**

```
int **p;

p = malloc( N * sizeof(int *) );

for (i=0; i<=N; i++) {
   p[i] = malloc( M * sizeof(int) );
}
```

Overwriting Memory

- **Not checking the max string size**

```
char s[8];
int i;

gets(s);  /* reads "123456789" from stdin */
```

- **Basis for classic buffer overflow attacks**
  - Your last assignment

Overwriting Memory

- **Misunderstanding pointer arithmetic**

```
int *search(int *p, int val) {

   while (*p && *p != val)
      p += sizeof(int);

   return p;
}
```

Referencing Nonexistent Variables

- **Forgetting that local variables disappear when a function returns**

```
int *foo () {
   int val;

   return &val;
}
```

Freeing Blocks Multiple Times

- Nasty!

```
x = malloc( N * sizeof(int) );
        <manipulate x>
free(x);

y = malloc( M * sizeof(int) );
        <manipulate y>
free(x);
```

- What does the free list look like?

```
x = malloc( N * sizeof(int) );
        <manipulate x>
free(x);
free(x);
```

Referencing Freed Blocks

- **Evil!**

```
x = malloc( N * sizeof(int) );
  <manipulate x>
free(x);
   ...
y = malloc( M * sizeof(int) );
for (i=0; i<M; i++)
   y[i] = x[i]++;
```

Failing to Free Blocks (Memory Leaks)

- **Slow, silent, long-term killer!**

```
foo() {
   int *x = malloc(N*sizeof(int));
   ...
   return;
}
```

Failing to Free Blocks (Memory Leaks)

- **Freeing only part of a data structure**

```
struct list {
   int val;
   struct list *next;
};

foo() {
   struct list *head = malloc( sizeof(struct list) );
   head->val = 0;
   head->next = NULL;
   <create and manipulate the rest of the list>
    ...
   free(head);
   return;
}
```

Overwriting Memory

- Referencing a pointer instead of the object it points to

```
int *getPacket(int **packets, int *size) {
   int *packet;
   packet = packets[0];
   packets[0] = packets[*size - 1];
   *size--;   // what is happening here?
   reorderPackets(packets, *size, 0);
   return(packet);
}
```

Dealing With Memory Bugs?

- Leaks?
- Unitialized reads?
- Double free?

Dealing With Memory Bugs

- **Conventional debugger (`gdb`)**
  - Good for finding bad pointer dereferences
  - Hard to detect the other memory bugs

- **Debugging `malloc` (UToronto CSRI `malloc`)**
  - Wrapper around conventional **`malloc`**
  - Detects memory bugs at **`malloc`** and **`free`** boundaries
    - Memory overwrites that corrupt heap structures
    - Some instances of freeing blocks multiple times
    - Memory leaks
  - Cannot detect all memory bugs
    - Overwrites into the middle of allocated blocks
    - Freeing block twice that has been reallocated in the interim
    - Referencing freed blocks

How would you make memory bugs go away? (puff)

- Does garbage collection solve everything?

- If not, what else do we need?

Java vs C

- **Reconnecting to Java**
  - Back to CSE143!
  - But now you know a lot more about what really happens when we execute programs

- **Java running native (compiled to C/assembly)**
  - Object representations: arrays, strings, etc.
  - Bounds checking
  - Memory allocation, constructors
  - Garbage collection
- **Java on a virtual machine**
  - Virtual processor
  - Another language: byte-codes

Meta-point to this lecture

- None of this data representation we are going to talk about is *guaranteed* by Java
- In fact, the language simply provides an *abstraction*
- We can't easily tell how things are really represented
- But it is important to understand *an* implementation of the lower levels --- it may be useful in thinking about your program

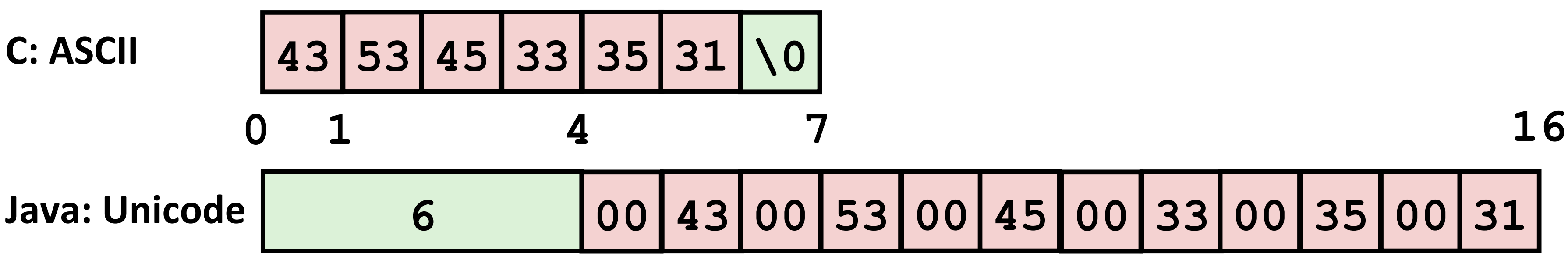
Data in Java

- **Integers, floats, doubles, pointers – same as C**
  - Yes, Java has pointers – they are called 'references' – however, Java references are much more constrained than C's general pointers
- **Null is typically represented as 0**
- **Characters and strings**
- **Arrays**
- **Objects**

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

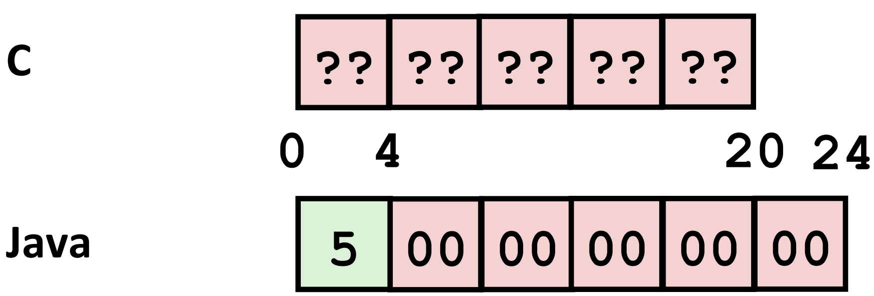
the string ‘CSE351’:



Data in Java

- **Arrays**
  - Bounds specified in hidden fields at start of array (int – 4 bytes)
    - *array.length* returns value of this field
    - *Hmm, since it had this info, what can it do?*
  - Every element initialized to 0

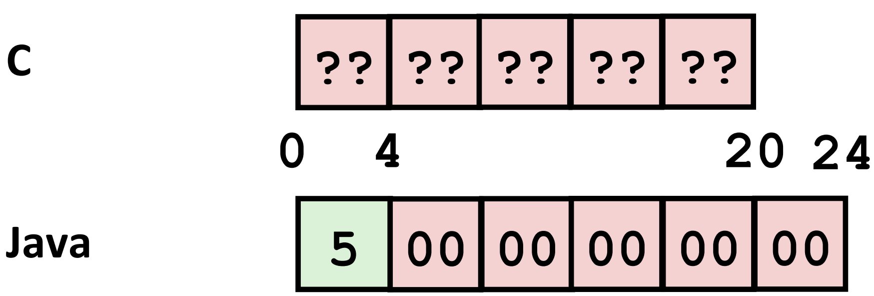
int array[5]:



Data in Java

- **Arrays**
  - Bounds specified in hidden fields 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
    - Trap if out-of-bounds
  - Every element initialized to 0

int array[5]:



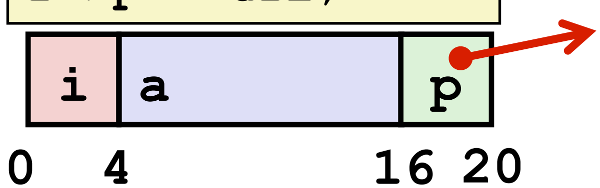
Data structures (objects) in Java

- **Objects (structs) can only include primitive data types**
  - Refer to complex data types (arrays, other objects, etc.) using references

C

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

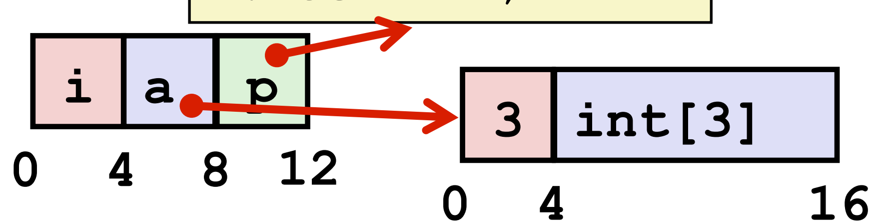
```
struct rec r;
struct rec r2;
r->i = val;
r->a[2] = val;
r->p = &r2;
```



Java

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

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

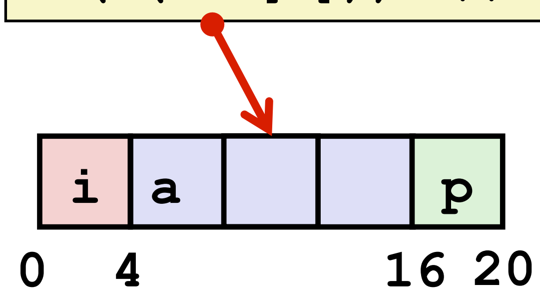


Pointers/References

- **Pointers in C can point to any memory address**
- **References in Java can only point to an object**
  - And only to its first element – not to the middle of it

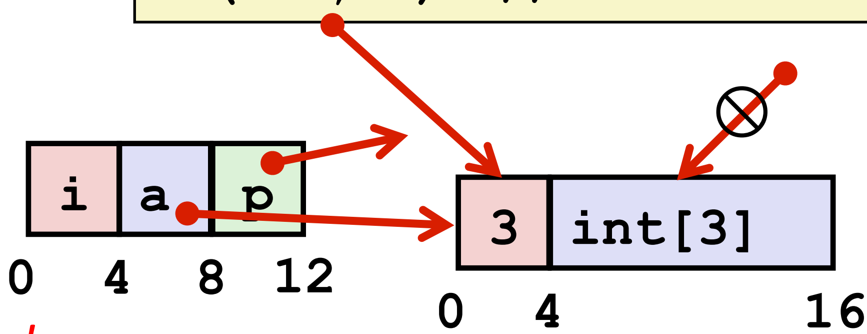
C

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



Java

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



What does this buy us? Wawaweewa!

Pointers to fields

- **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 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

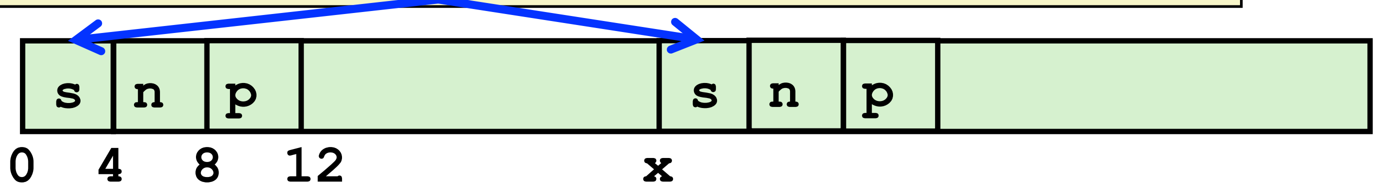
Casting in C

- We can cast any pointer into any other pointer

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

Cast p 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



Casting in Java

- **Can only cast compatible object references**

```
class Object{
  …
};
```

```
class Parent {
  int address;
};
```

```
class Sister extends Parent{
  int hers;
};
```

```
class Brother extends Parent{
  int his;
};
```

```
// Parent is a super class of Brother and Sister, which are siblings
Parent    a = new Parent();
Sister   xx = new Sister();
Brother  xy = new Brother();
Parent   p1 = new Sister();    // ok, everything needed for Parent
                               // is also in Sister
Parent   p2 = p1;              // ok, p1 is already a Parent
Sister  xx2 = new Brother();   // incompatible type – Brother and
                               // Sisters are siblings
Sister  xx3 = new Parent();    // wrong direction; elements in Sister
                               // not in Parent (hers)
Brother xy2 = (Brother) a;     // run-time error; Parent does not contain
                               // all elements in Brother (his)
Sister  xx4 = (Sister) p2;     // ok, p2 started out as Sister
Sister  xx5 = (Sister) xy;     // inconvertible types, xy is Brother
```

How is this implement?

Creating objects in Java

```
class Point {
        double x;
        double y;

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

boolean samePlace(Point p) {
        return (x == p.x) && (y == p.y);
     }

}
…
Point newPoint = new Point();
…
```

fields

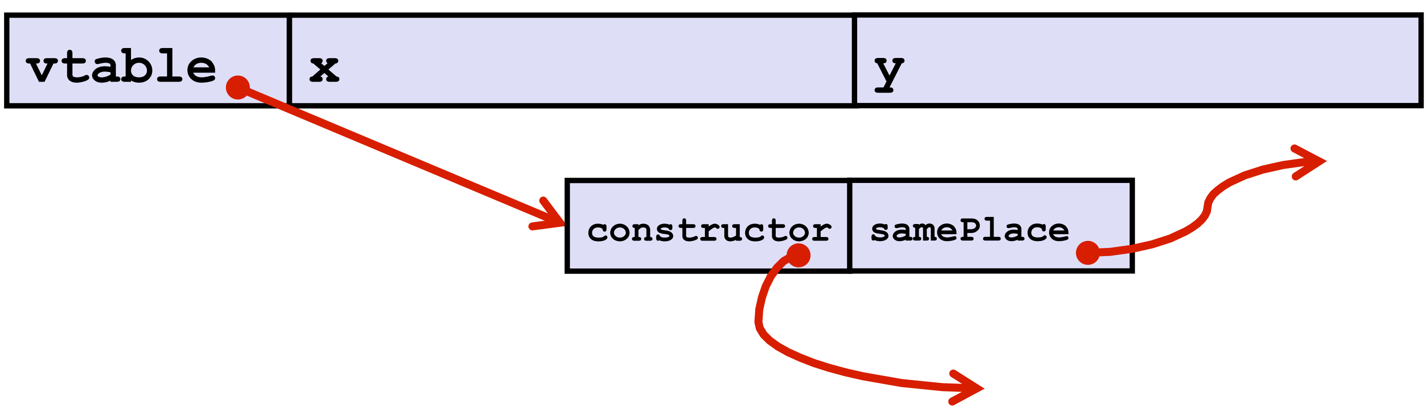
constructor

method

creation

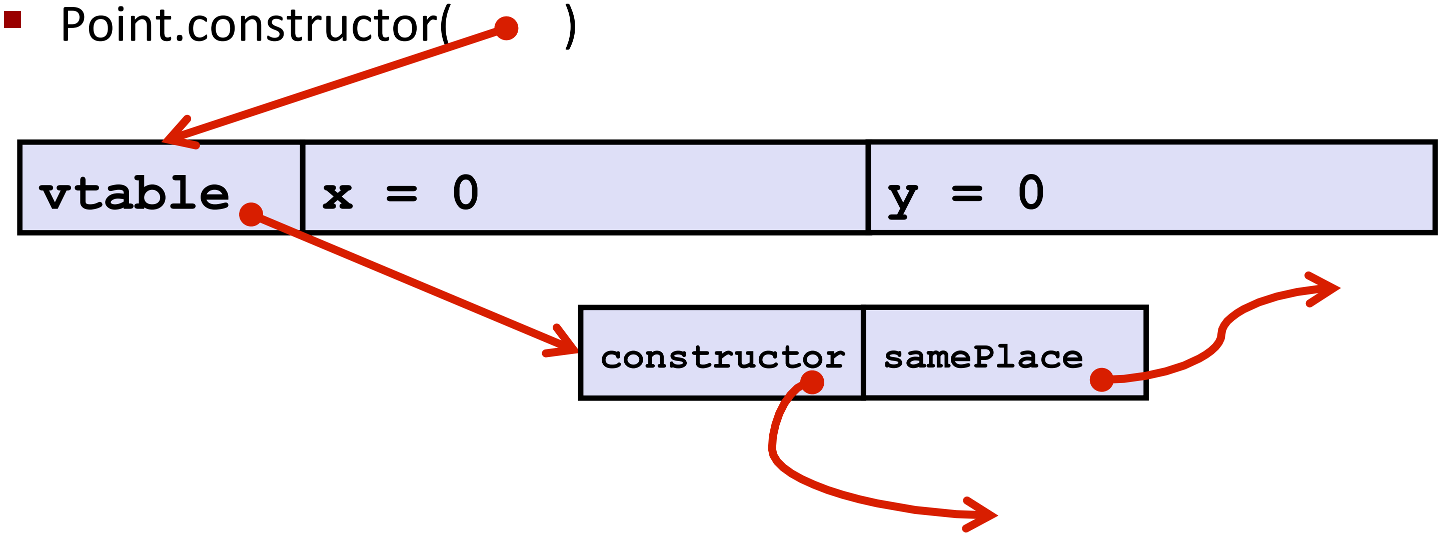
Creating objects in Java

- **"new"**
  - Allocates space for data fields
  - Adds pointer in object to "virtual table" or "vtable" for class (shared)
    - Includes space for "static fields" and pointers to methods' code
  - Returns reference (pointer) to new object in memory
- **Runs "constructor" method**
- **Eventually garbage collected if all references to the object are discarded**



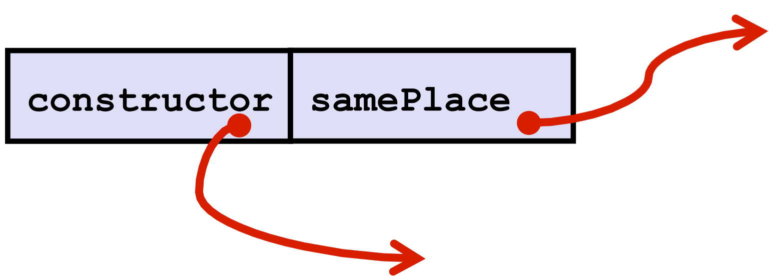
Initialization

- **newPoint's fields are initialized starting with the vtable pointer to the vtable for this class**
- **The next step is to call the 'constructor' for this object type**
- **Constructor code is found using the 'vtable pointer' and passed a pointer to the newly allocated memory area for newPoint so that the constructor can set its x and y to 0**
  - This can be resolved statically, so does't require vtable lookup
  - Point.constructor(   )



What about the vtable itself?

- **Array of pointers to every method defined for the object Point**
- **Compiler decided in which element of the array to put each pointer and keeps track of which it puts where**
- **Methods are just functions (as in C) but with an extra argument – the pointer to the allocated memory for the object whose method is being called**
  - E.g., newPoint.samePlace calls the samePlace method with a pointer to newPoint (called 'this') and a pointer to the argument, p – in this case, both of these are pointers to objects of type Point
  - Method becomes Point.samePlace(Point this, Point p)



Calling a method

- **newPoint.samePlace(p2) is a call to the samePlace method of the object of type Point with the arguments newPoint and p2 which are both pointers to Point objects**
  - Why is newPoint passed as a parameter to samePlace?

Calling a method

- **newPoint.samePlace(p2) is a call to the samePlace method of the object of type Point with the arguments newPoint and p2 which are both pointers to Point objects**
- **In C**
  - CodePtr = (newPoint->vtable)[theRightIndexForSamePlace]
    - Gets address of method's code
  - CodePtr(this, p2)
    - Calls method with references to object and parameter
- **We need 'this' so that we can read the x and y of our object and execute**
  - return x==p.x && y==p.y; which becomes
  - return (this->x==p2->x) && (this->y==p2->y)

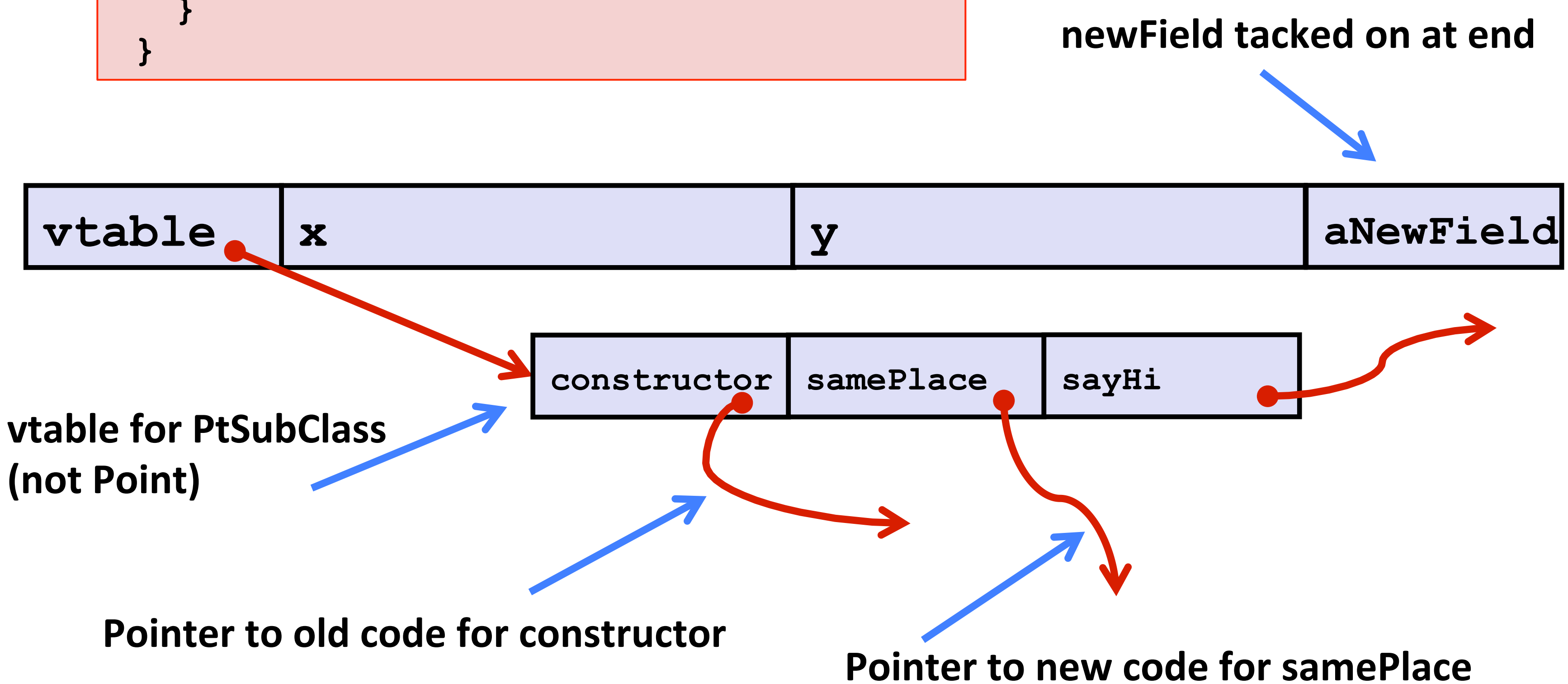
Subclassing

```
class PtSubClass extends Point{
   int aNewField;
   boolean samePlace(Point p2) {
       return false;
   }
   void sayHi() {
      System.out.println("hello");
   }
 }
```

- **Where does “aNewField” go?**
  - At end of fields of Point
- **Where does pointer to code for two new methods go?**
  - To override “samePlace”, write over old pointer
  - Add new pointer at end of table for new method “sayHi”
  - This necessitates “dynamic” vtable

Subclassing

```
class PtSubClass extends Point{
   int aNewField;
   boolean samePlace(Point p2) {
       return false;
   }
   void sayHi() {
      System.out.println("hello");
   }
 }
```

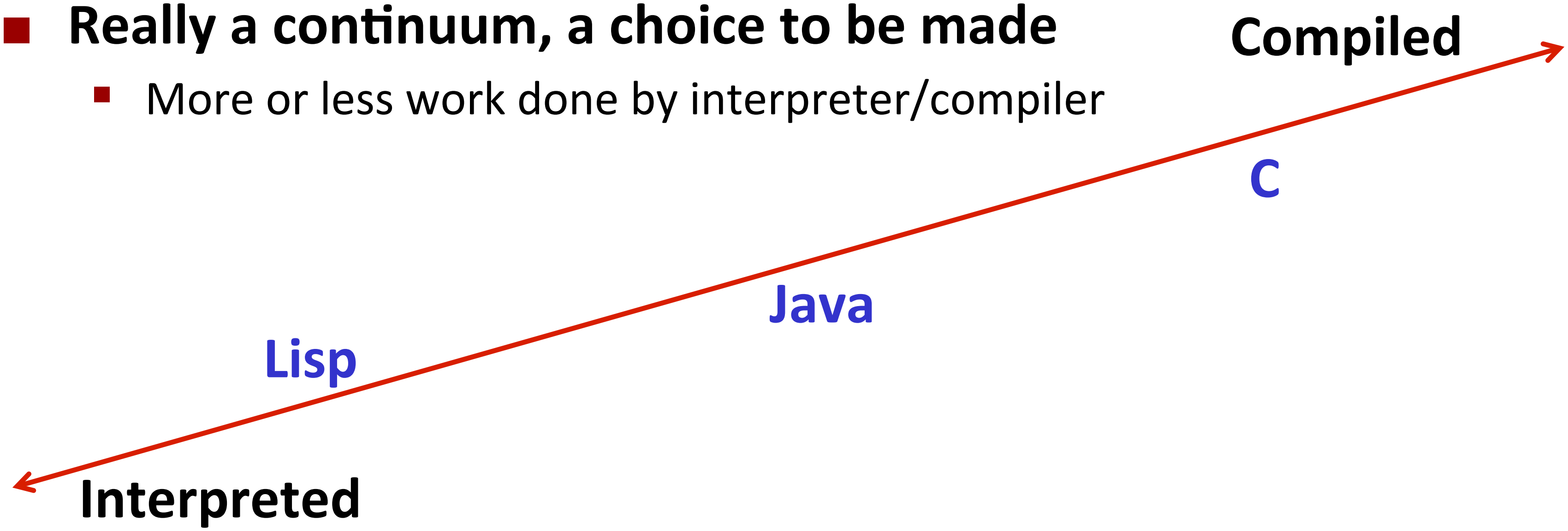


Implementing Programming Languages

- **Many choices in how to implement programming models**
- **We've talked about compilation, can also *interpret***
  - Execute line by line in original source code
  - Less work for compiler – all work done at run-time
  - Easier to debug – less translation
  - Easier to protect other processes – runs in an simulated environment that exists only inside the *interpreter* process
- **Interpreting languages has a long history**
  - Lisp – one of the first programming languages, was interpreted
- **Interpreted implementations are very much with us today**
  - Python, Javascript, Ruby, Matlab, PHP, Perl, ...

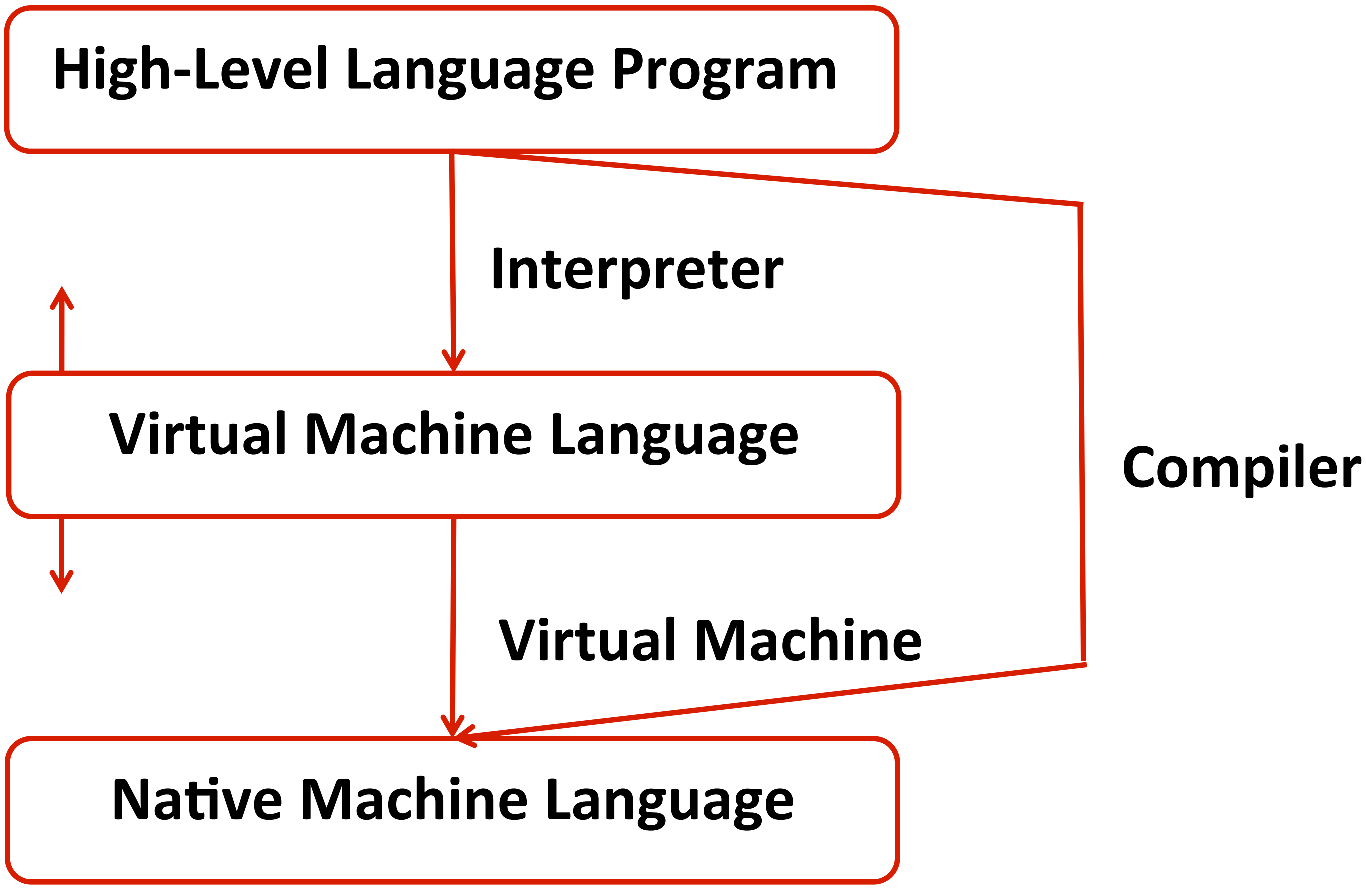
Interpreted vs. Compiled

- **Really a continuum, a choice to be made**
  - More or less work done by interpreter/compiler



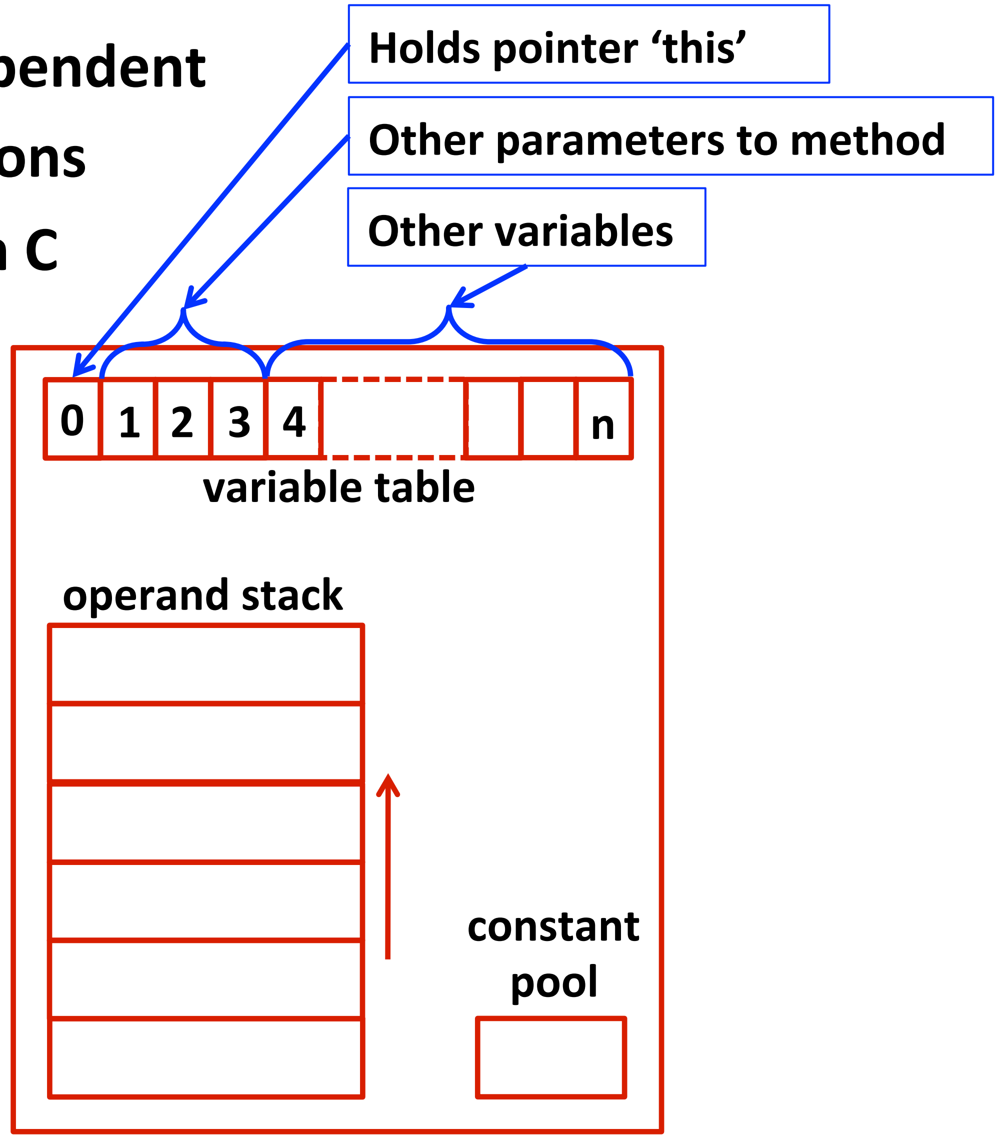
- **Java programs are usually run by a *virtual machine***
  - VMs interpret an intermediate language – partly compiled
- **Java can also be compiled (just as a C program is) or at run-time by a *just-in-time (JIT) compiler* (as opposed to an ahead-of-time (AOT) compiler)**

Virtual Machine Model



Java Virtual Machine

- **Making Java machine-independent**
- **Providing stronger protections**
- **VM usually implemented in C**
- **Stack execution model**
- **There are many JVMs**
  - Some interpret
  - Some compile into assembly



A Basic JVM Stack Example

'i' stands for integer,
'a' for reference,
'b' for byte,
'c' for char,
'd' for double, ...

No knowledge of registers or memory locations (each instruction is 1 byte – byte-code)

```
iload 1     // push 1st argument from table onto stack
iload 2     // push 2nd argument from table onto stack
iadd        // add and pop top 2 element, push result
istore 3    // pop result and put it into third slot in table
```

```
mov 0x8001, %eax
mov 0x8002, %edx
add %edx, %eax
mov %eax, 0x8003
```

A Simple Java Method

```
Method java.lang.String employeeName()

0 aload 0        // "this" object is stored at 0 in the var table

1 getfield #5 <Field java.lang.String name> // takes 3 bytes
                 // pop an element from top of stack, retrieve the
                 // specified field and push the value onto stack
                 // "name" field is the fifth field of the class

4 areturn        // Returns object at top of stack
```

0	1			4
aload_0	getfield	00	05	areturn

In the .class file:

2A	B4	00	05	B0

http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings

Class File Format

- **10 sections to the Java class file structure**
  - Magic number: 0xCAFEBABE (legible hex from James Gosling – Java's inventor)
  - Version of class file format: the minor and major versions of the class file
  - Constant pool: Pool of constants for the class
  - Access flags: for example whether the class is abstract, static, etc
  - This class: The name of the current class
  - Super class: The name of the super class
  - Interfaces: Any interfaces in the class
  - Fields: Any fields in the class
  - Methods: Any methods in the class
  - Attributes: Any attributes of the class (for example the name of the sourcefile, etc)

Example

```
javac Employee.java
javap -c Employee > Employee.bc
```

```
Compiled from Employee.java
class Employee extends java.lang.Object {
public Employee(java.lang.String,int);
public java.lang.String employeeName();
public int employeeNumber();
}

Method Employee(java.lang.String,int)
0 aload_0
1 invokespecial #3 <Method java.lang.Object()>
4 aload_0
5 aload_1
6 putfield #5 <Field java.lang.String name>
9 aload_0
10 iload_2
11 putfield #4 <Field int idNumber>
14 aload_0
15 aload_1
16 iload_2
17 invokespecial #6 <Method void
                    storeData(java.lang.String, int)>
20 return

Method java.lang.String employeeName()
0 aload_0
1 getfield #5 <Field java.lang.String name>
4 areturn

Method int employeeNumber()
0 aload_0
1 getfield #4 <Field int idNumber>
4 ireturn

Method void storeData(java.lang.String, int)
…
```

Other languages for JVMs

- **Apart from the Java language itself, The most common or well-known JVM languages are:**
  - AspectJ, an aspect-oriented extension of Java
  - ColdFusion, a scripting language compiled to Java
  - Clojure, a functional Lisp dialect
  - Groovy, a scripting language
  - JavaFX Script, a scripting language targeting the Rich Internet Application domain
  - JRuby, an implementation of Ruby
  - Jython, an implementation of Python
  - Rhino, an implementation of JavaScript
  - Scala, an object-oriented and functional programming language
  - And many others, even including C

Microsoft's C# and .NET Framework

- C# has similar motivations as Java
- Virtual machine is called the Common Language Runtime (CLR)
- Common Intermediate Language (CLI) is C#'s byte-code