Memory Allocation III, Java & C

CSE 351 Winter 2022

Instructor:
Sam Wolfson

Teaching Assistants:
Angela Xu
Anirudh Kumar
Catherine Guevara
Dara Stotland
Harrison Bay
Ian Hsiao
Kevin Wang
Mara Kirdani-Ryan
Nick Durand
Sanjana Sridhar

My godmother has a new puppy!
Her name is Piper.

Relevant Course Information

- hw23 due tonight, hw25 due Friday (3/11)
- Lab 5 due Friday (3/11)
  - Recommended that you watch the Lab 5 helper videos
  - `lateDays++` (you're now up to 7 total)

- No readings for the remaining lectures!

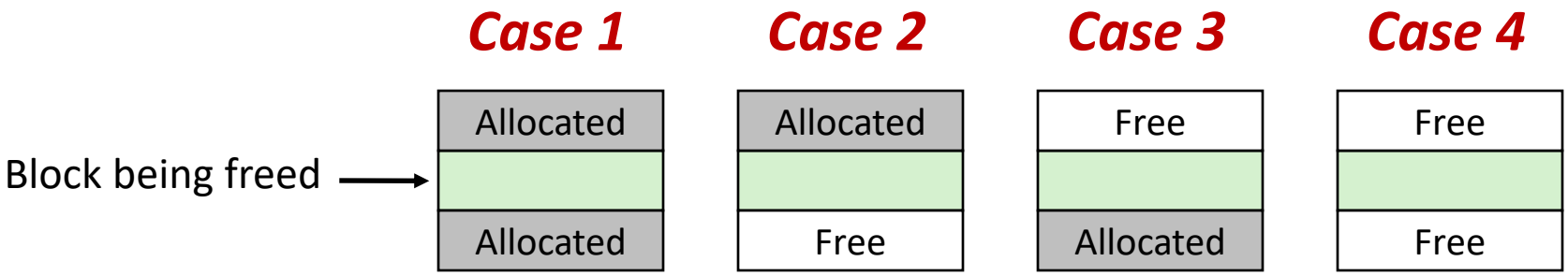
- **Final Exam:** 3/15—3/17
  - Similar to midterm: Gilligan's Island Rule
  - Final review section this week (+ VM)
  - Review Session TBD

Freeing With Explicit Free Lists

- *Insertion policy:* Where in the free list do you put the newly freed block?
  - **LIFO (last-in-first-out) policy**
    - Insert freed block at the beginning (head) of the free list
    - Pro: simple and constant time
    - Con: studies suggest fragmentation is worse than the alternative
  - **Address-ordered policy**
    - Insert freed blocks so that free list blocks are always in address order:

      address(previous) < address(current) < address(next)
    - Con: requires linear-time search
    - Pro: studies suggest fragmentation is better than the alternative (why?)

Coalescing in Explicit Free Lists



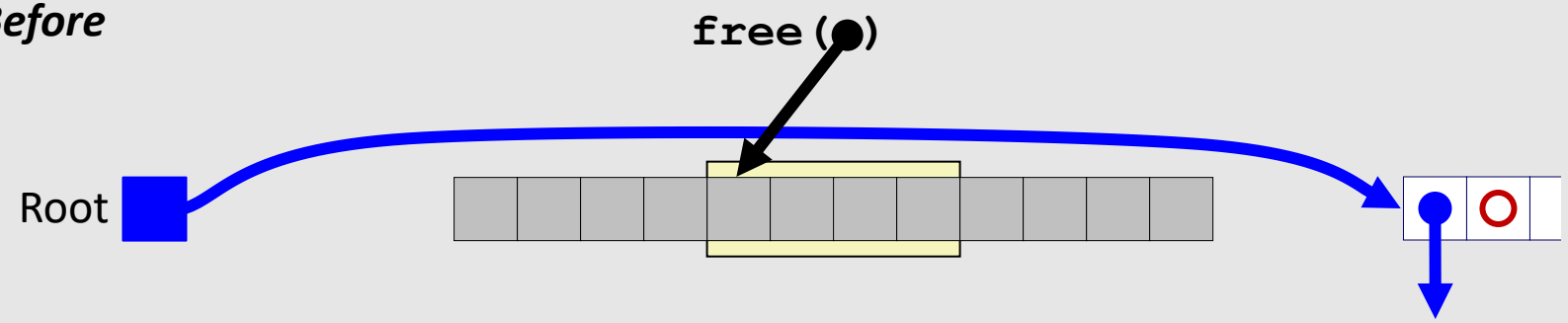
- Neighboring free blocks are *already part of the free list*
  1) Remove old block from free list
  2) Create new, larger coalesced block
  3) Add new block to free list (insertion policy)

- How do we tell if a neighboring block if free?

Freeing with LIFO Policy (Case 1)

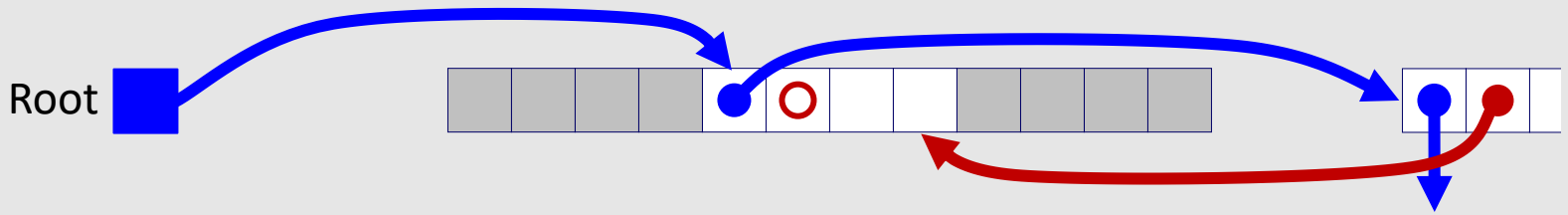
Boundary tags not shown, but don't forget about them!

Before



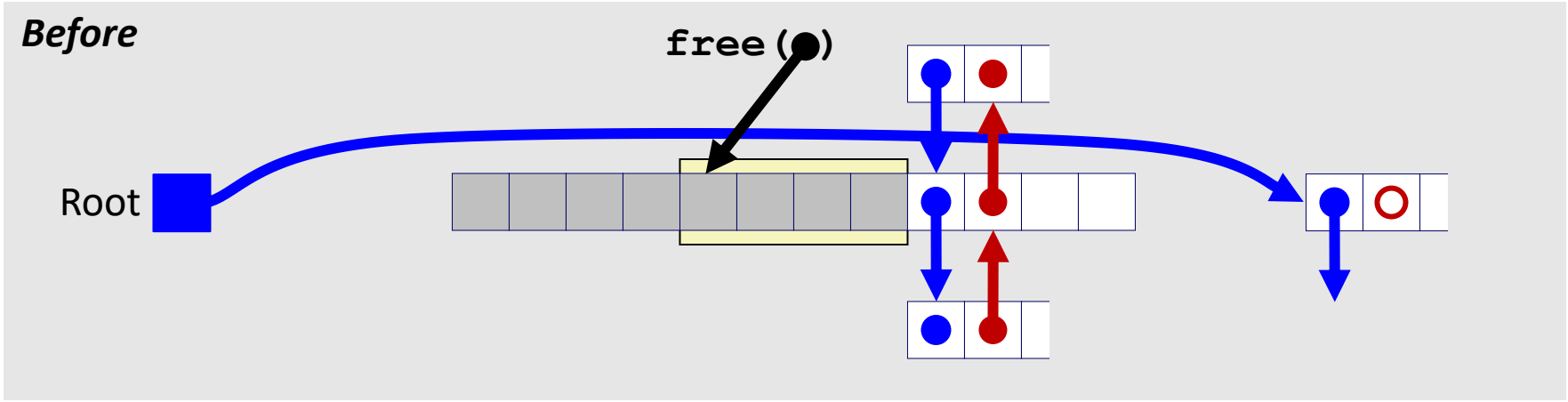
- Insert the freed block at the root of the list

After

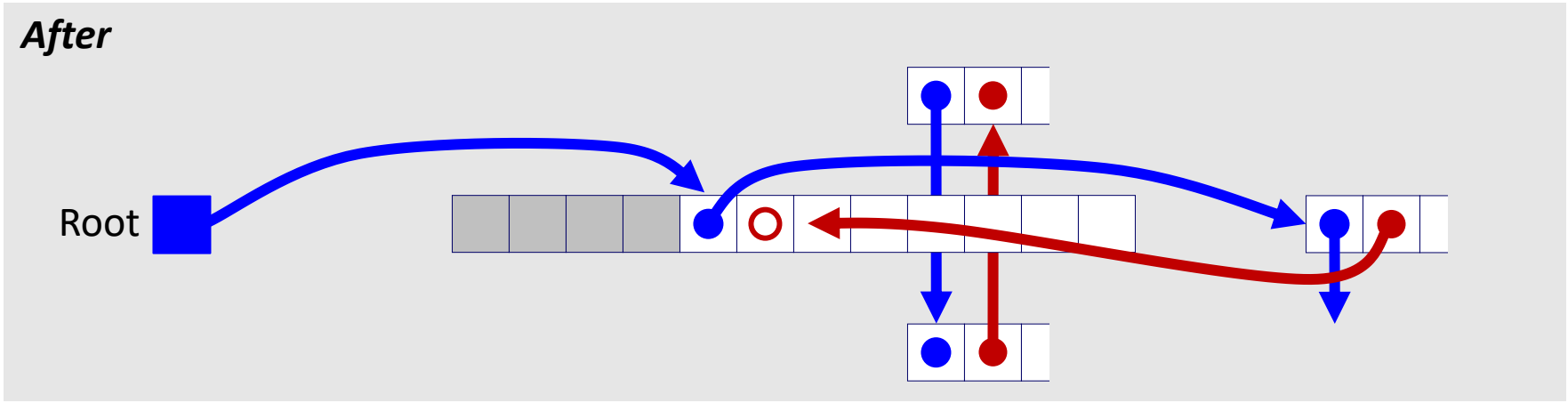


Freeing with LIFO Policy (Case 2)

Boundary tags not shown, but don't forget about them!



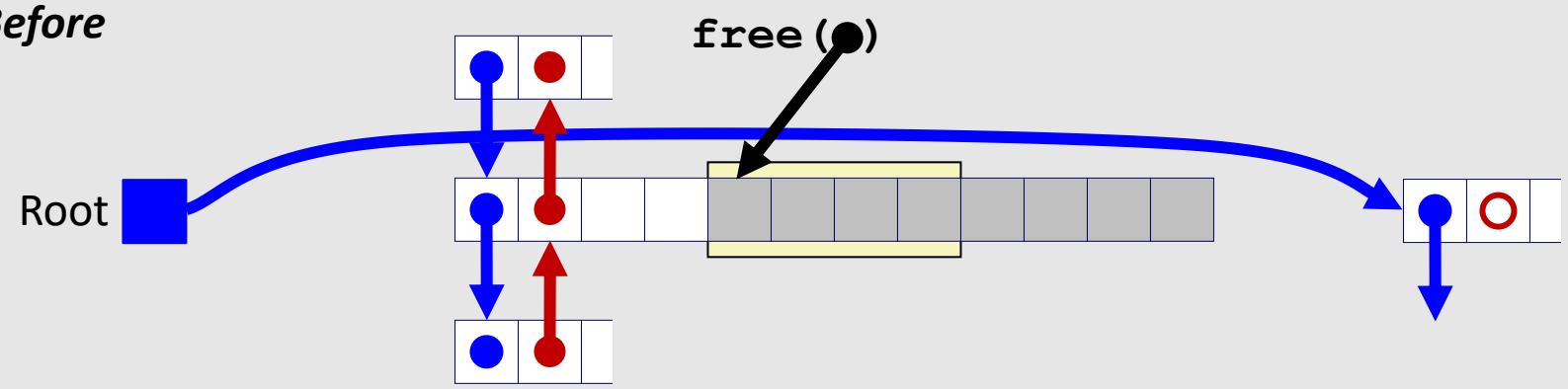
- Splice *following* block out of list, coalesce both memory blocks, and insert the new block at the root of the list



Freeing with LIFO Policy (Case 3)

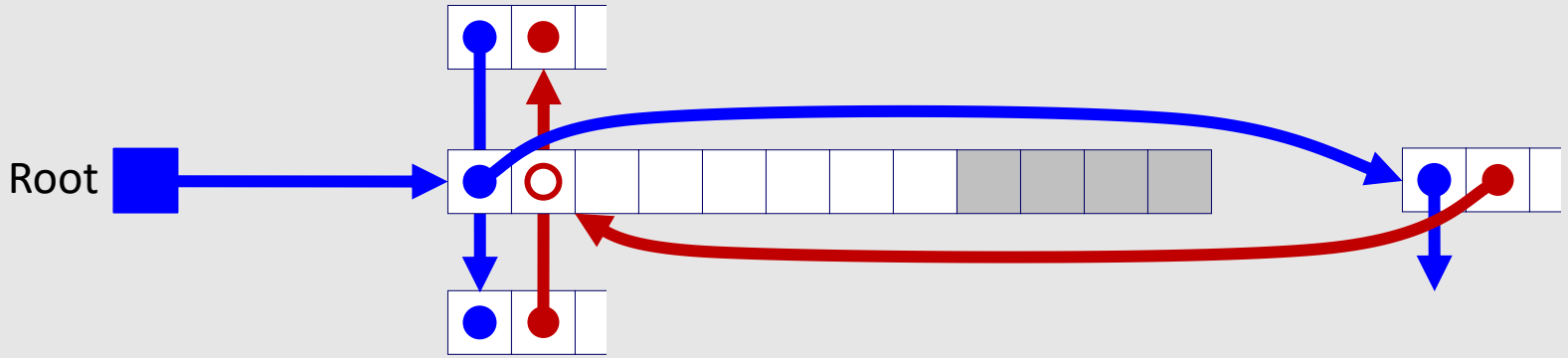
Boundary tags not shown, but don't forget about them!

Before



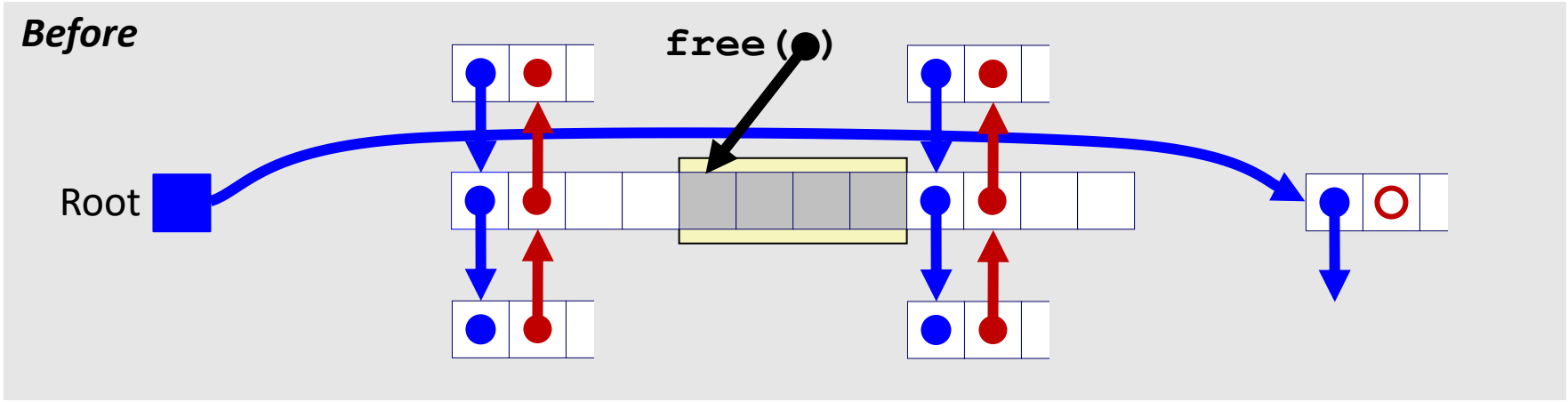
- Splice *preceding* block out of list, coalesce both memory blocks, and insert the new block at the root of the list

After

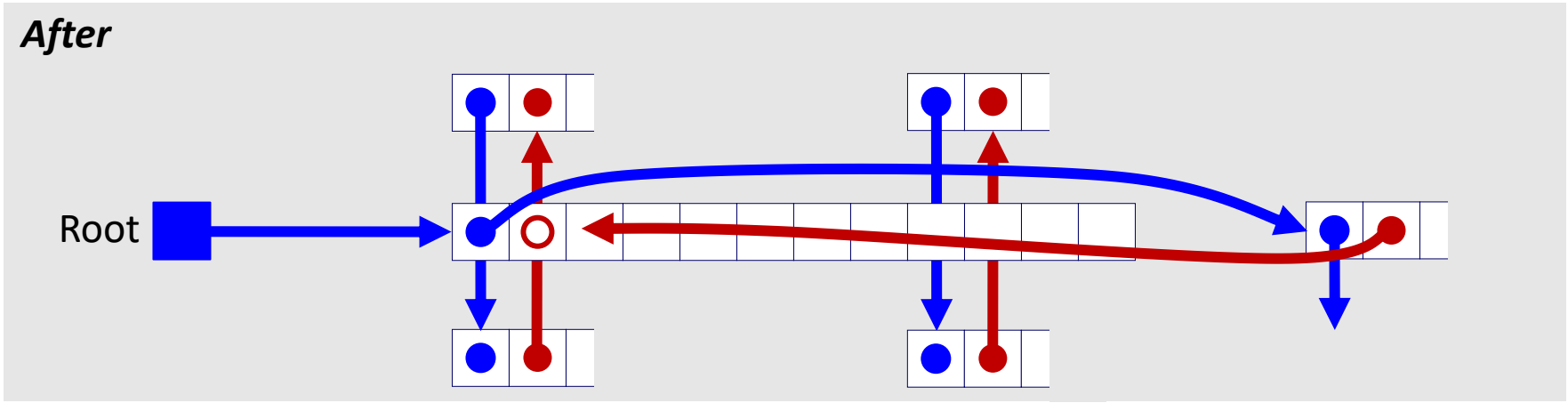


Freeing with LIFO Policy (Case 4)

Boundary tags not shown, but don't forget about them!

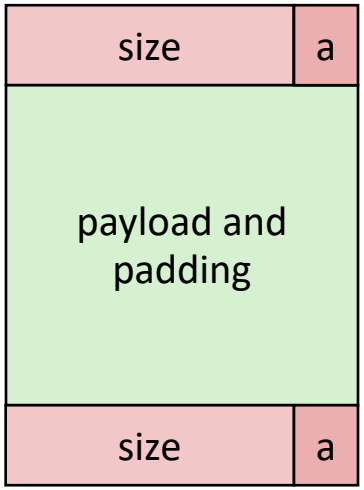


- Splice *preceding* and *following* blocks out of list, coalesce all 3 memory blocks, and insert the new block at the root of the list



Do we always need the boundary tags?

Allocated block:



(same as implicit free list)

Free block:

- Lab 5 suggests no…

Lab 5 Hints

- Struct pointers can be used to access field values, even if no struct instances have been created – just reinterpreting the data in memory
- Pay attention to boundary tag data
  - Size value + 2 tag bits – when do these need to be updated and do they have the correct values?
  - The `examine_heap` function follows the implicit free list searching algorithm – don't take its output as "truth"
- Learn to use and interpret the trace files for testing!!!
- A special heap block marks the end of the heap

Explicit List Summary

- Comparison with implicit list:
  - Block allocation is linear time in number of ***free*** blocks instead of ***all*** blocks
    - ***Much faster*** when most of the memory is full
  - Slightly more complicated allocate and free since we need to splice blocks in and out of the list
  - Some extra space for the links (2 extra pointers needed for each free block)
    - Increases minimum block size, leading to more internal fragmentation

- Most common use of explicit lists is in conjunction with *segregated free lists*
  - Keep multiple linked lists of different size classes, or possibly for different types of objects

Allocation Policy Tradeoffs

- Data structure of blocks on lists
  - Implicit (free/allocated), explicit (free), segregated (many free lists) – others possible!
- Placement policy: first-fit, next-fit, best-fit
  - Throughput vs. amount of fragmentation
- When do we split free blocks?
  - How much internal fragmentation are we willing to tolerate?

More Info on Allocators

- D. Knuth, "*The Art of Computer Programming*", 2nd edition, Addison Wesley, 1973
  - The classic reference on dynamic storage allocation

- Wilson et al, "*Dynamic Storage Allocation: A Survey and Critical Review*", Proc. 1995 Int'l Workshop on Memory Management, Kinross, Scotland, Sept, 1995.
  - Comprehensive survey
  - Available from CS:APP student site (csapp.cs.cmu.edu)

Memory Allocation

- Dynamic memory allocation
  - Introduction and goals
  - Allocation and deallocation (free)
  - Fragmentation
- Explicit allocation implementation
  - Implicit free lists
  - Explicit free lists (Lab 5)
  - Segregated free lists
- **Implicit deallocation: garbage collection**

Reading Review

- Terminology:
  - Garbage collection: mark-and-sweep
  - Memory-related issues in C

- Questions from the Reading?

Imagine a world in which…

- …we never had to free memory!
- Do you free objects in Java?
  - Reminder: *implicit* allocator

Garbage Collection (GC)
(Automatic Memory Management)

- *Garbage collection:* automatic reclamation of heap-allocated storage – application never explicitly frees memory

```
void foo() {
    int* p = (int*) malloc(128);
    return;  /* p block is now garbage! */
}
```

- Common in most all modern languages
  - Lisp, Racket, Erlang, Go, ML, Haskell, Scala, Java, C#, Perl, Ruby, Python, Lua, JavaScript, Dart, Mathematica, MATLAB, many more…
- Variants ("conservative" garbage collectors) exist for C and C++
  - However, cannot necessarily collect all garbage

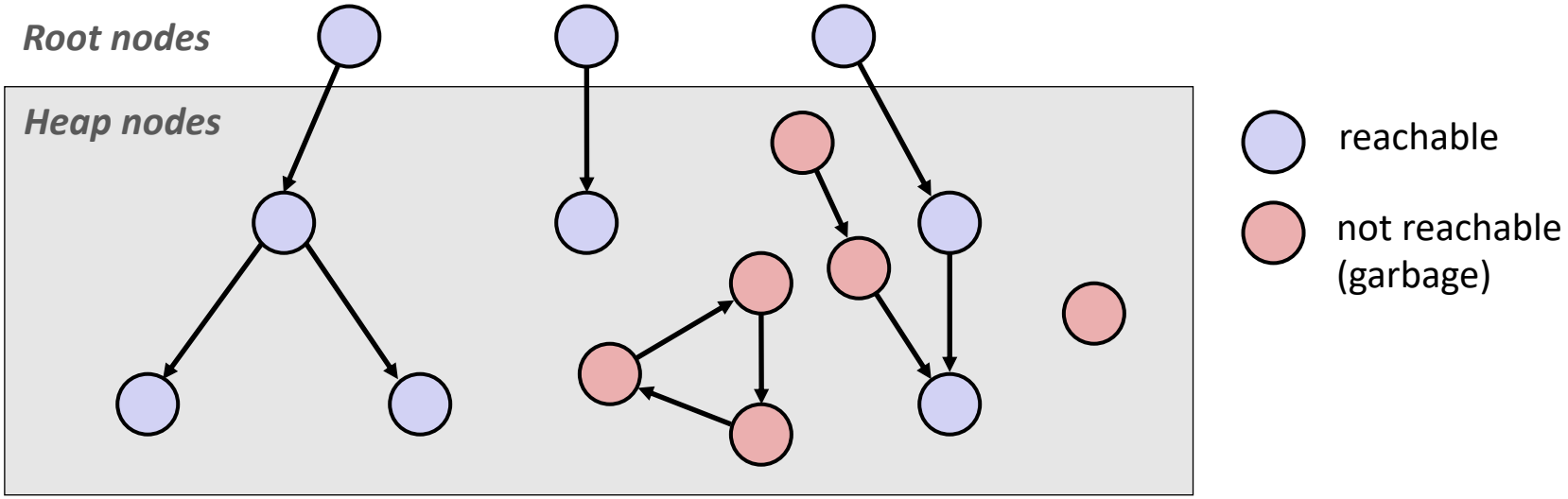
Garbage Collection

- How does the memory allocator know when memory can be freed?
  - In general, we cannot know what is going to be used in the future since it depends on conditionals
  - But we *can* guarantee that certain blocks will not be used if they are *unreachable* (via pointers in registers/stack/globals)

- Memory allocator needs to know what is a pointer and what is not – how can it do this?
  - Sometimes with help from the compiler

Memory as a Graph

- We view memory as a directed graph
  - Each allocated heap block is a node in the graph
  - Each pointer is an edge in the graph
  - Locations not in the heap that contain pointers into the heap are called ***root*** nodes (*e.g.*, registers, stack locations, global variables)



A node (block) is *reachable* if there is a path from any root to that node
Non-reachable nodes are *garbage* (cannot be needed by the application)

Garbage Collection

- Dynamic memory allocator can free blocks if there are no pointers to them

- How can it know what is a pointer and what is not?

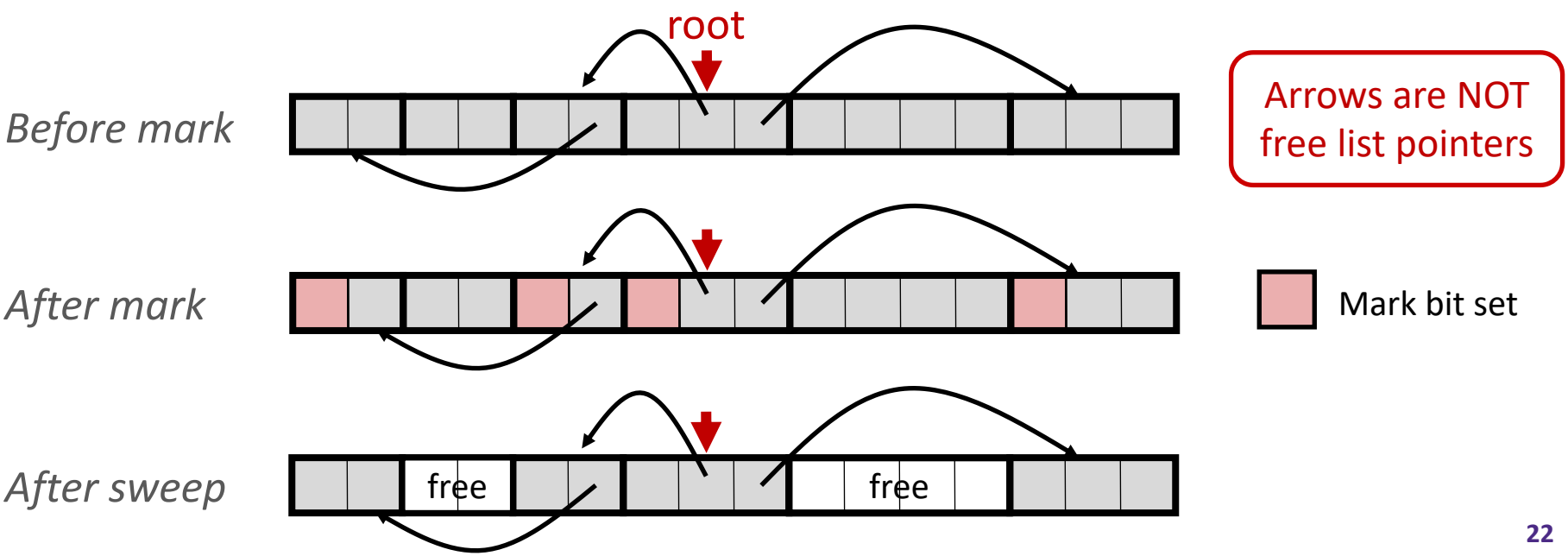
- We'll make some *assumptions* about pointers:
  - Memory allocator can distinguish pointers from non-pointers
  - All pointers point to the start of a block in the heap
  - Application cannot hide pointers
    (*e.g.*, by coercing them to a `long`, and then back again)

Classical GC Algorithms

- **Mark-and-sweep collection** (McCarthy, 1960)
  - Does not move blocks (unless you also "compact")
- Reference counting (Collins, 1960)
  - Does not move blocks (not discussed)
- Copying collection (Minsky, 1963)
  - Moves blocks (not discussed)
- Generational Collectors (Lieberman and Hewitt, 1983)
  - Most allocations become garbage very soon, so focus reclamation work on zones of memory recently allocated.
- For more information:
  - Jones, Hosking, and Moss, *The Garbage Collection Handbook: The Art of Automatic Memory Management*, CRC Press, 2012.
  - Jones and Lin, *Garbage Collection: Algorithms for Automatic Dynamic Memory*, John Wiley & Sons, 1996.

Mark and Sweep Collecting

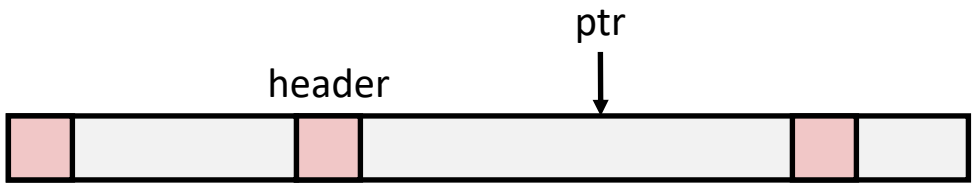
- Can build on top of `malloc`/`free` package
  - Allocate using `malloc` until you "run out of space"
- When out of space:
  - Use extra ***mark bit*** in the header of each block
  - ***Mark:*** Start at roots and set mark bit on each reachable block
  - ***Sweep:*** Scan all blocks and free blocks that are not marked



Can you do this in C?

Non-testable Material

- Ehh, kinda-sorta.
  - For mark-and-sweep, we need to assume that pointers are to the **beginning** of allocated memory blocks (so we can mark them)
  - But in C, pointers can point into the **middle** of allocated blocks (not so in Java)
    - Makes it tricky to find all allocated blocks in mark phase



  - There are ways to work around this problem in C, but the resulting garbage collector is conservative:
    - Every reachable node correctly identified as reachable, but some unreachable nodes might be incorrectly marked as reachable
  - In Java, all pointers (*i.e.*, references) point to the starting address of an object structure – the start of an allocated block

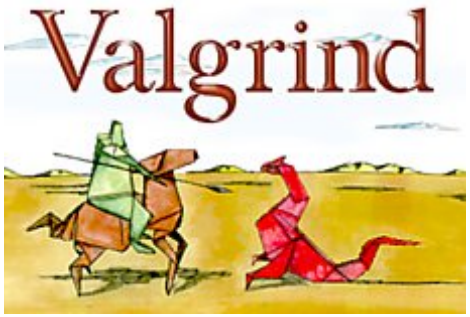
Quick Debugging Note

- Staring at code until you think you spot a bug is generally *not* an effective way to debug!
  - Of course it looks logically correct to you – you wrote it!
  - Language like C doesn't abstract away memory – it's part of your program state that you need to keep track of
    - Your code will only get longer and more complicated in the future: there's too much to try to keep track of mentally

- Instead, start with bad/unexpected behavior to guide your search
  - Memory bugs/"errors" can be especially tricky because they often don't result in explicit errors or program stoppages

Dealing With Memory Bugs

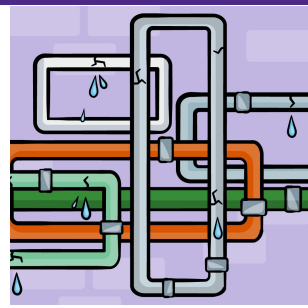
- Make use of all the tools available to you:
  - Pay attention to compiler errors **and warnings**
  - Use debuggers like GDB to track down runtime errors
    - Good for bad pointer dereferences, bad with other memory bugs
  - `valgrind` is a powerful debugging and analysis utility for Linux, especially good for memory bugs
    - Checks each individual memory reference at *runtime* (*i.e.*, only detects issues with parts of code used in a specific execution)
    - Can catch many memory bugs, including bad pointers, reading uninitialized data, double-frees, and memory leaks



What about Java or ML or Python or …?

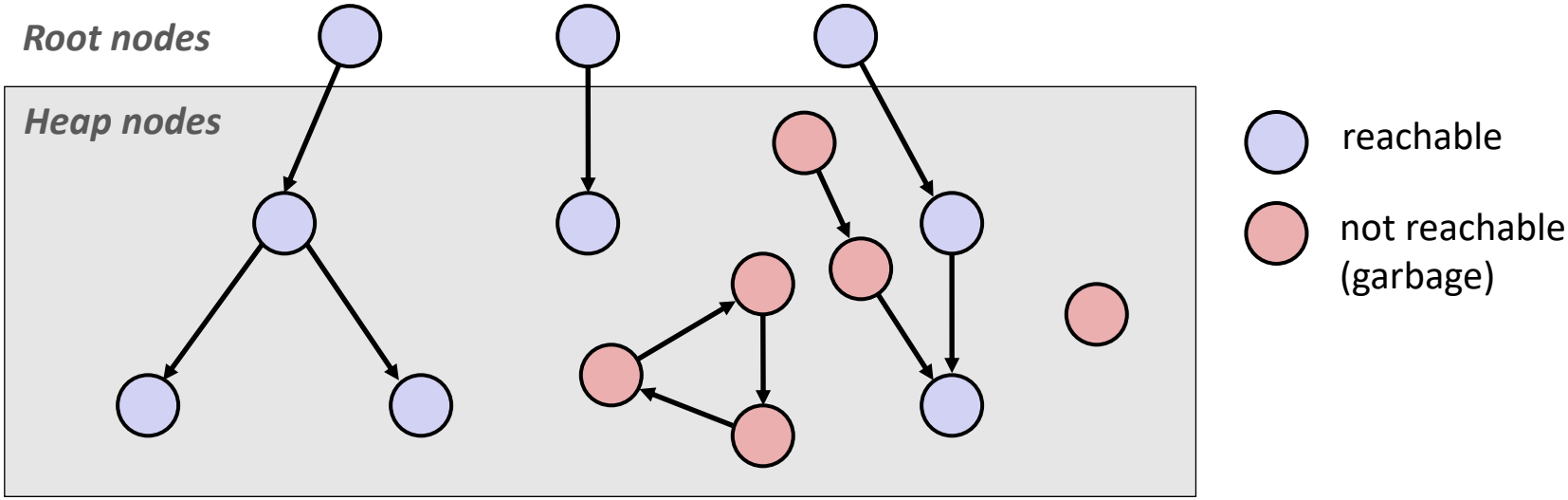
- In *memory-safe languages*, most of these bugs are impossible 🥳
  - Cannot perform arbitrary pointer manipulation
  - Cannot get around the type system
  - Array bounds checking, null pointer checking
  - Automatic memory management

- But one memory-related bug is possible. Which one?



Memory Leaks with GC

- Not because of forgotten `free` — we have GC!
- Unneeded "leftover" roots keep objects reachable
- *Sometimes* nullifying a variable is not needed for correctness but is for performance
- Example: Don't leave big data structures you're done with in a static field



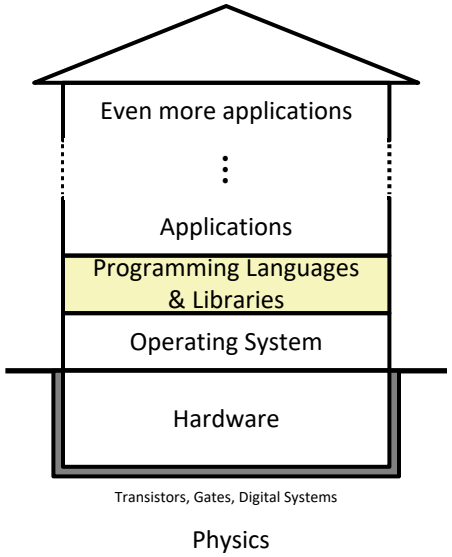
More Java vs C

- Let's make more connections to Java (hello, CSE 14x)!
  - But now you know a lot more about what really happens when we execute programs!

- We've seen how these all work in C – let's see how they differ in Java!
  - Representation of data
  - Pointers / references
  - Casting
  - Function / method calls including dynamic dispatch

The Hardware/Software Interface

- Topic Group 1: **Data**
  - **Memory**, **Data**, Integers, Floating Point, **Arrays**, Structs, **Objects** ← take 341 or 401
- Topic Group 2: **Programs**
  - x86-64 Assembly, **Procedures**, **Stacks**, **Executables**
- Topic Group 3: **Scale & Coherence**
  - Caches, Processes, Virtual Memory, Memory Allocation



These apply to execution regardless of source language

Apply more generally than just C!!!

Worlds Colliding

- CSE 351 has (hopefully ;)) given you a "really different feeling" about what computers do and how programs are executed

- We have occasionally contrasted to Java, but CSE 143 may still feel like "a different world"
  - But it's not! It's just a higher-level of abstraction
    - Residing on a higher floor in the House of Computing
  - How could we implement (parts of) Java in CSE 351 terms?

Meta-point to this lecture

- None of the data representations we are going to talk about are **guaranteed** by Java

- In fact, the language simply provides an **abstraction** (Java language specification)
  - Tells us how code should **behave** for different language constructs, but we can't easily tell how things are really represented
  - But it is important to understand **a possible implementation** of the lower levels – useful in thinking about your program

Data in Java

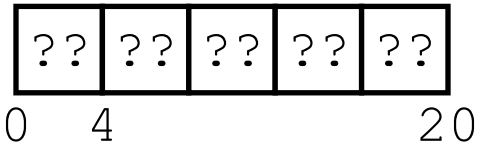
- Integers, floats, doubles, pointers – same as C
  - "Pointers" are called "references" in Java, but are much more constrained than C's general pointers
  - Java's portability-guarantee fixes the sizes of all types
    - Example: `int` is 4 bytes in Java regardless of machine
  - No unsigned types to avoid conversion pitfalls
    - Added some useful methods in Java 8 (also use bigger signed types)
- `null` is typically represented as `0` but "you can't tell"
- Much more interesting:
  - **Arrays**
  - **Characters and strings**
  - **Objects**

Data in Java: Arrays

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

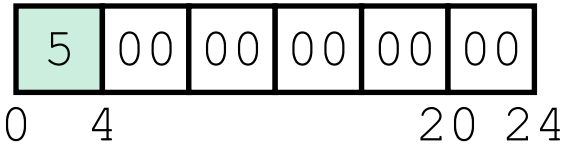
C:

int array[5];



Java:

int[] array = **new** int[5];

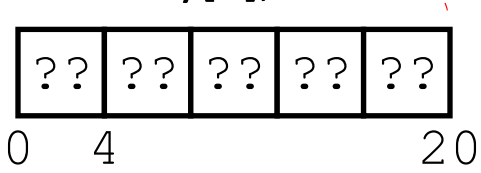


Data in Java: Arrays

- Every element initialized to `0` or `null`
- Length specified in immutable field at start of array (`int`: 4B)
  - `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

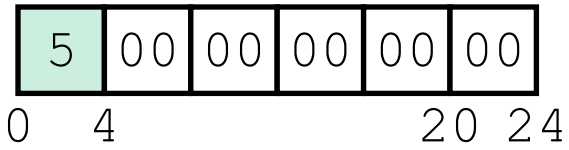
C:

int array[5];



Java:

int[] array = **new** int[5];



To speed up bounds-checking:

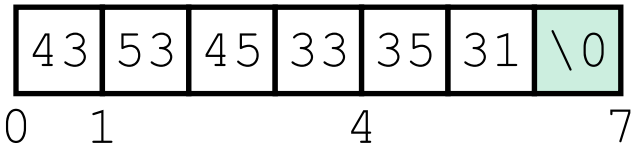
- Length field is likely in cache
- Compiler may store length field in register for loops
- Compiler may prove that some checks are redundant

Data in Java: Characters & Strings

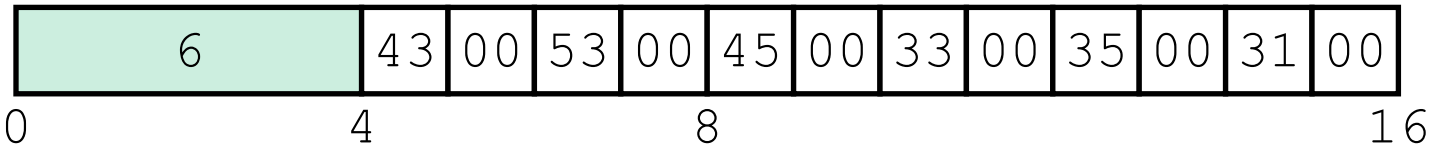
- Two-byte Unicode instead of ASCII
  - Represents most of the world's alphabets (and is being actively updated)
- String not bounded by a `'\0'` (null character)
  - Bounded by hidden length field at beginning of string
- All `String` objects are read-only

Example: the string "CSE351"

C:
(ASCII)



Java:
(Unicode)



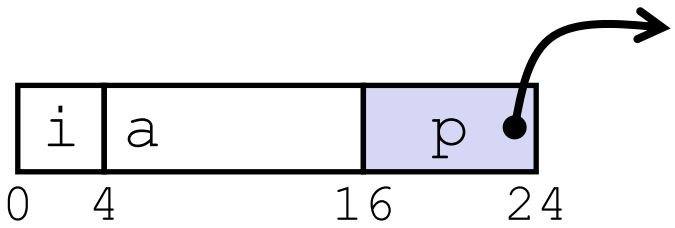
Data in Java: ~~Structs~~ Objects

- Data structures (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;
};
```

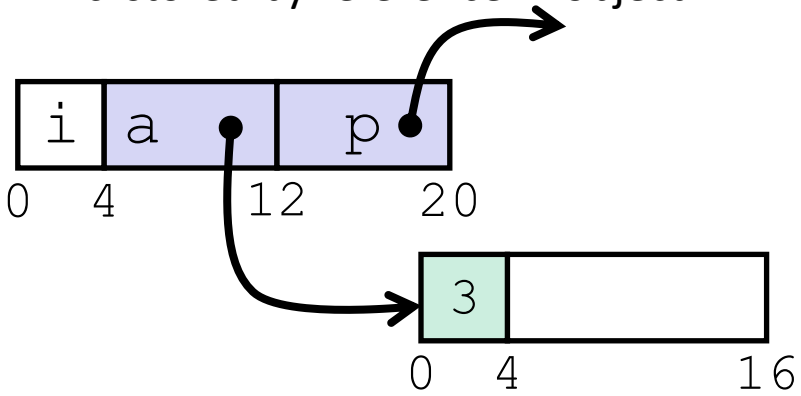
- `a[]` stored "inline" as part of struct



Java:

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

- `a` stored by reference in object



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 `r->a` in C
  - No Java field needs more than 8 bytes!

C:

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

Java:

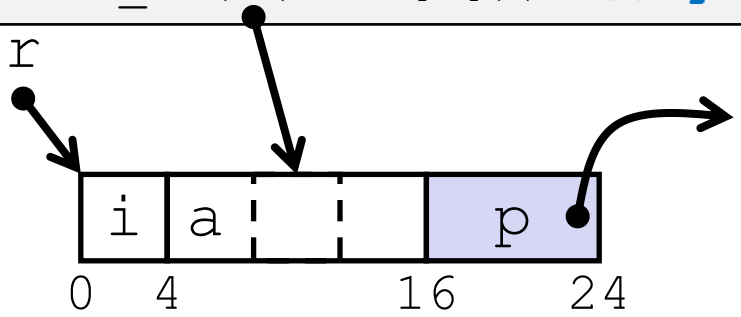
```
r = new Rec();
r2 = new Rec();
r.i = val;
r.a[2] = val;
r.p = r2;
```

Pointers/References

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

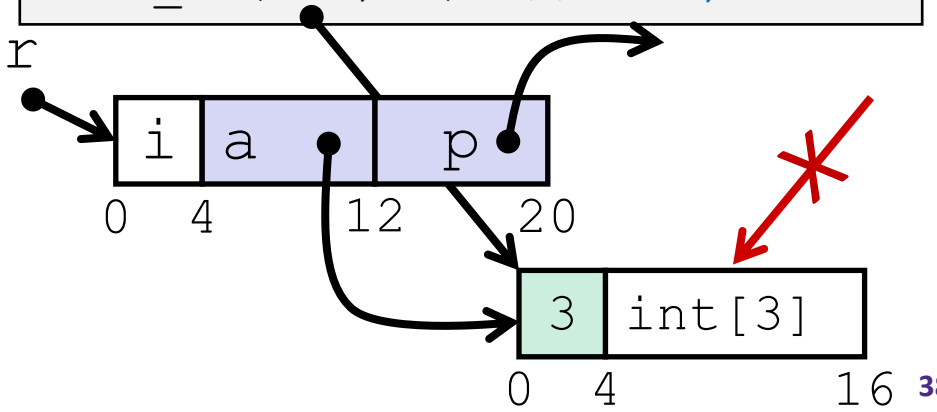
C:

```
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
```



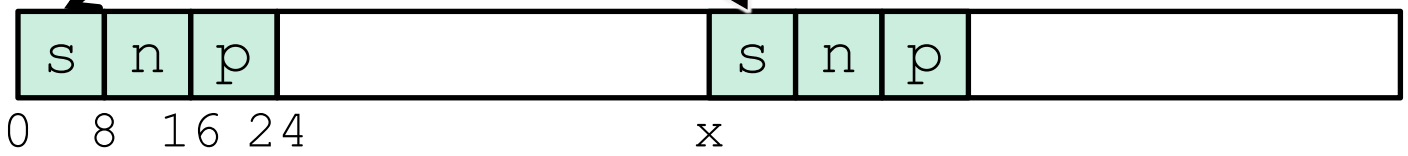
Casting in C (example from Lab 5)

- Can cast any pointer into any other pointer
  - Changes dereference and arithmetic behavior

```
struct block_info {
        size_t size_and_tags;
        struct block_info* next;
        struct block_info* prev;
};
typedef struct block_info block_info;
...
int x;
block_info* b;
block_info* new_block;
...
new_block = (block_info*) ( (char*) b + x );
...
```

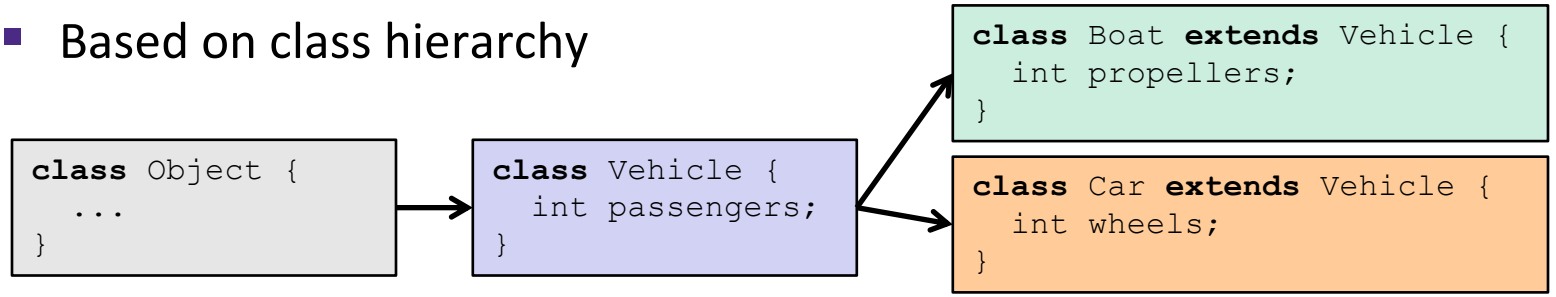
Cast b into char* to do unscaled addition

Cast back into block_info* to use as block_info struct



Type-safe casting in Java

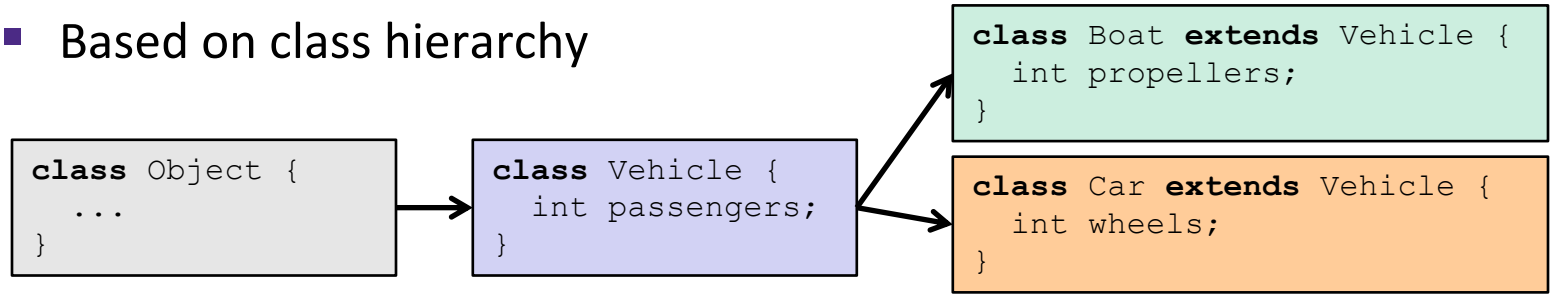
- Can only cast compatible object references
  - Based on class hierarchy




```
Vehicle  v = new Vehicle(); // super class of Boat and Car
Boat    b1 = new Boat();    // |--> sibling
Car     c1 = new Car();     // |--> sibling

Vehicle v1 = new Car();
Vehicle v2 = v1;
Car     c2 = new Boat();

Car     c3 = new Vehicle();

Boat    b2 = (Boat) v;

Car     c4 = (Car) v2;
Car     c5 = (Car) b1;
```

Type-safe casting in Java

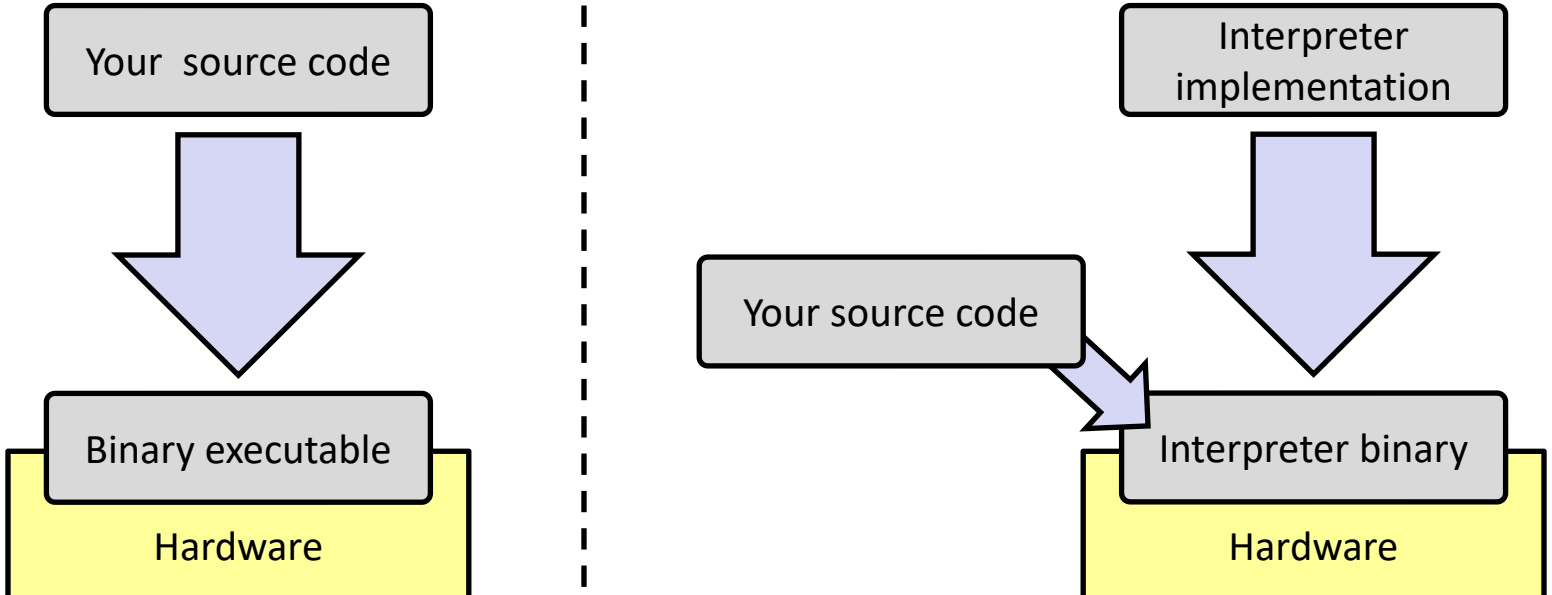
- Can only cast compatible object references
  - Based on class hierarchy




```
Vehicle  v = new Vehicle(); // super class of Boat and Car
Boat    b1 = new Boat();    // |--> sibling
Car     c1 = new Car();     // |--> sibling

Vehicle v1 = new Car();     <-- ✓ Everything needed for Vehicle also in Car
Vehicle v2 = v1;            <-- ✓ v1 is declared as type Vehicle
Car     c2 = new Boat();    <-- ✗ Compiler error: Incompatible type – elements in
                                  Car that are not in Boat (siblings)
Car     c3 = new Vehicle(); <-- ✗ Compiler error: Wrong direction – elements Car
                                  not in Vehicle (wheels)
Boat    b2 = (Boat) v;      <-- ✗ Runtime error: Vehicle does not contain all
                                  elements in Boat (propellers)
Car     c4 = (Car) v2;      <-- ✓ v2 refers to a Car at runtime
Car     c5 = (Car) b1;      <-- ✗ Compiler error: Unconvertable types – b1 is
                                  declared as type Boat
```

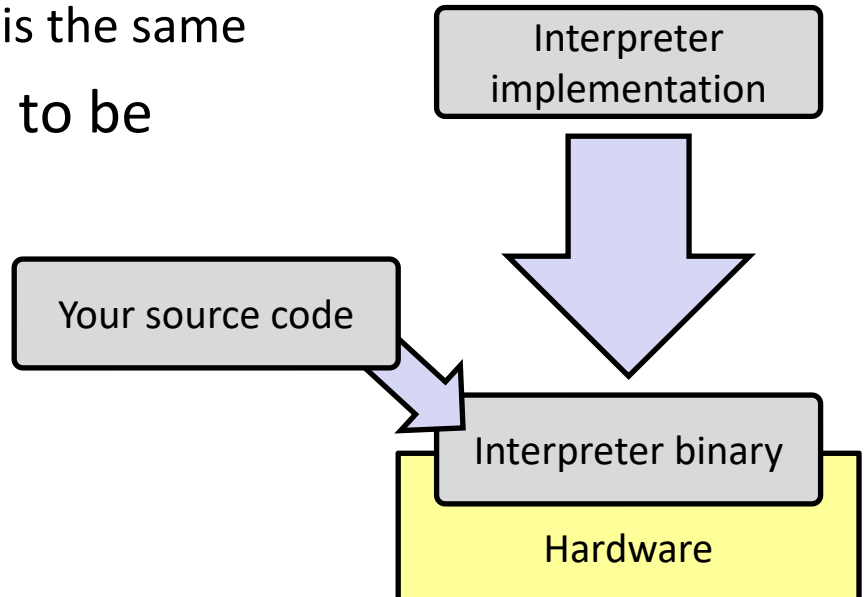
Implementing Programming Languages

- Many choices in programming model implementation
  - We've previously discussed compilation
  - We can also *interpret*
- Interpreters have a long history and are still in use
  - *e.g.*, Lisp, an early programming language, was interpreted
  - *e.g.*, Python, Javascript, Ruby, Matlab, PHP, Perl, …



Interpreters

- Execute (something close to) the *source code* directly, meaning there is less translation required
  - This makes it a simpler program than a compiler and often provides more transparent error messages
- Easier to run on different architectures – runs in a simulated environment that exists only inside the *interpreter* process
  - Just port the interpreter (program), and then interpreting the source code is the same
- Interpreted programs tend to be slower to execute and harder to optimize
- You can think of a CPU as an *interpreter* for x86!



Interpreters vs. Compilers

- You can choose to execute code written in a particular language via either a compiler or an interpreter, if they exist
- "Compiled languages" vs. "interpreted languages" a misuse of terminology
  - But very common to hear this
  - And has *some* validation in the real world (*e.g.*, JavaScript vs. C)
- Some modern language implementations are a mix
  - *e.g.*, Java compiles to bytecode that is then interpreted
  - Doing just-in-time (JIT) compilation of parts to assembly for performance

"The JVM"

Note: The JVM is different than the CSE VM running on VMWare. Yet *another* use of the word "virtual"!

- Java programs are usually run by a Java *virtual machine (JVM)*
  - JVMs interpret an intermediate language called *Java bytecode*
  - Many JVMs compile bytecode to native machine code
    - **Just-in-time (JIT) compilation**
    - http://en.wikipedia.org/wiki/Just-in-time_compilation
  - Java is sometimes compiled ahead of time (AOT) like C

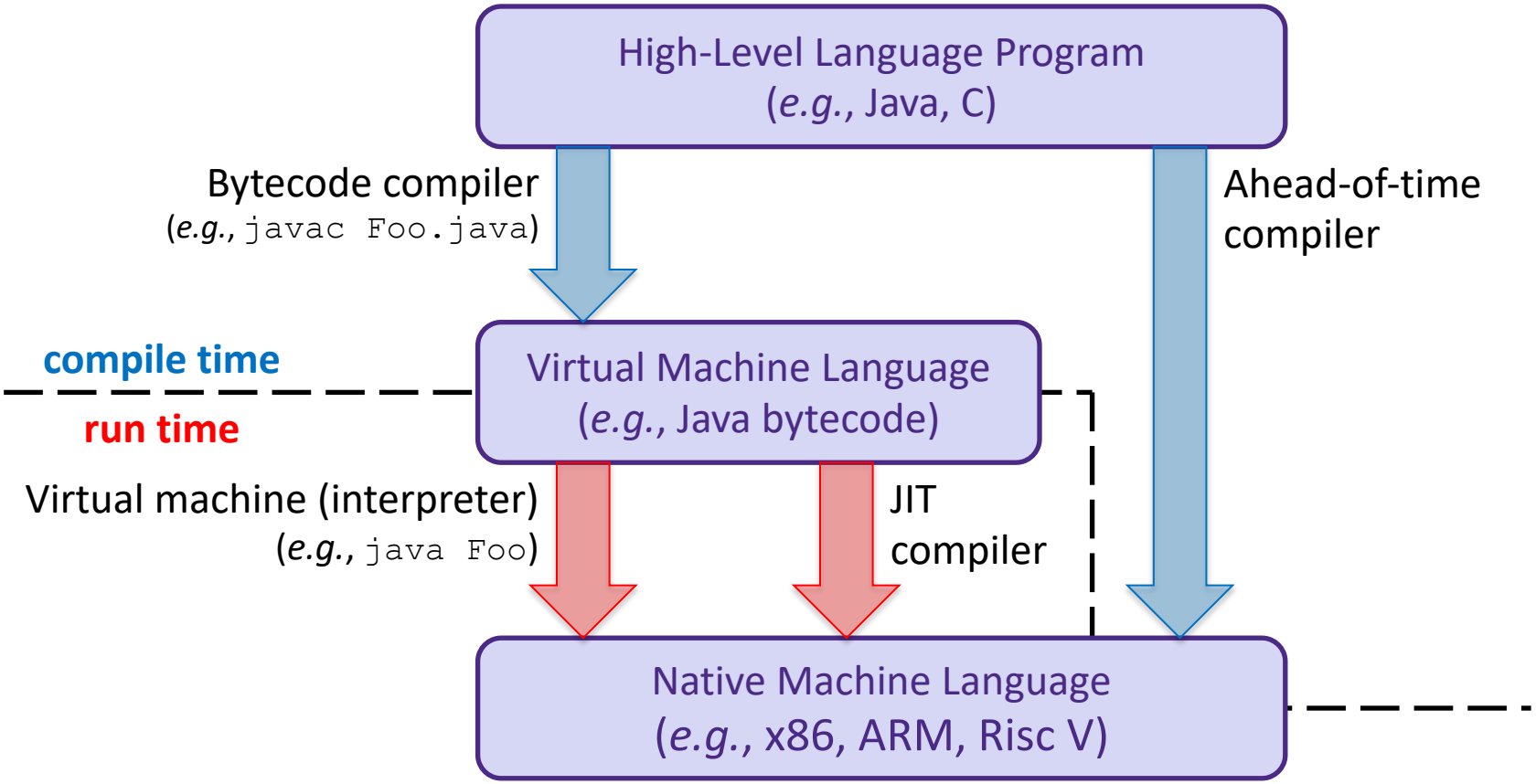
Compiling and Running Java

1. Save your Java code in a `.java` file

2. To run the Java compiler:
   - `javac Foo.java`
   - The Java compiler converts Java into *Java bytecode*
     - Stored in a `.class` file

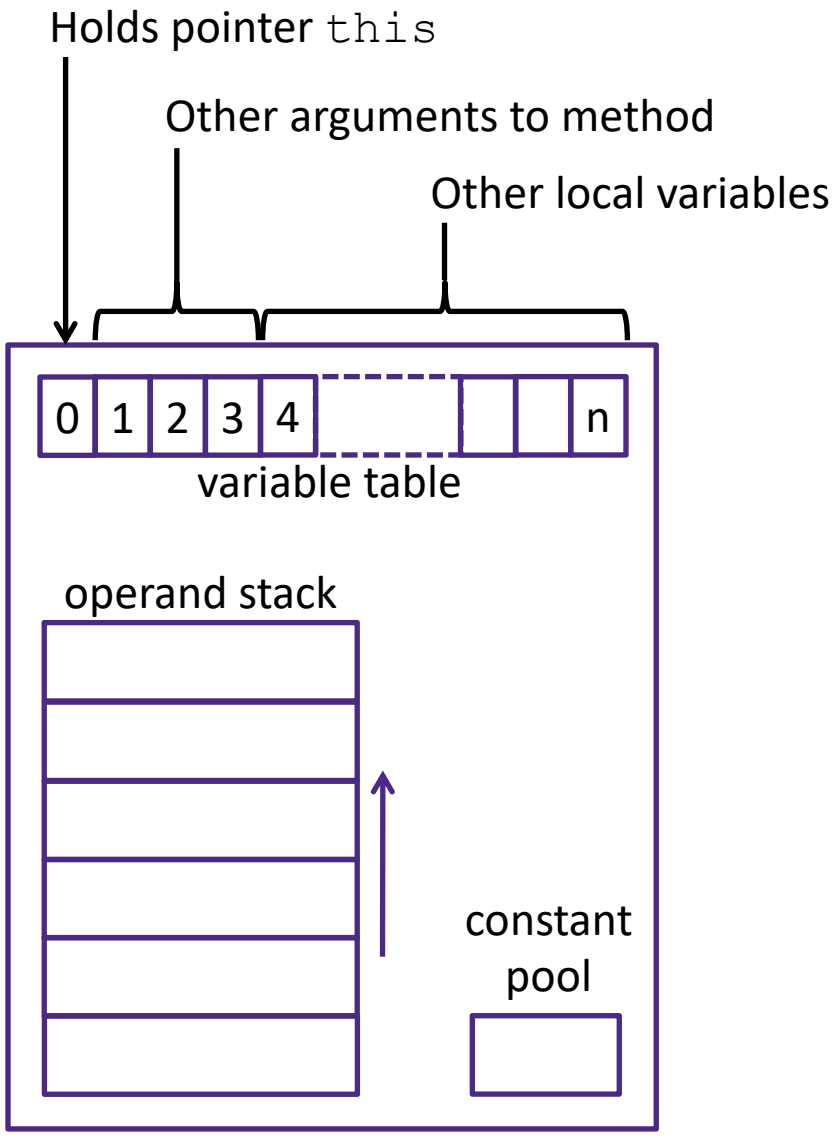
3. To execute the program stored in the bytecode, these can be interpreted by the Java Virtual Machine (JVM)
   - Running the virtual machine: `java Foo`
   - Loads `Foo.class` and interprets the bytecode

Virtual Machine Model

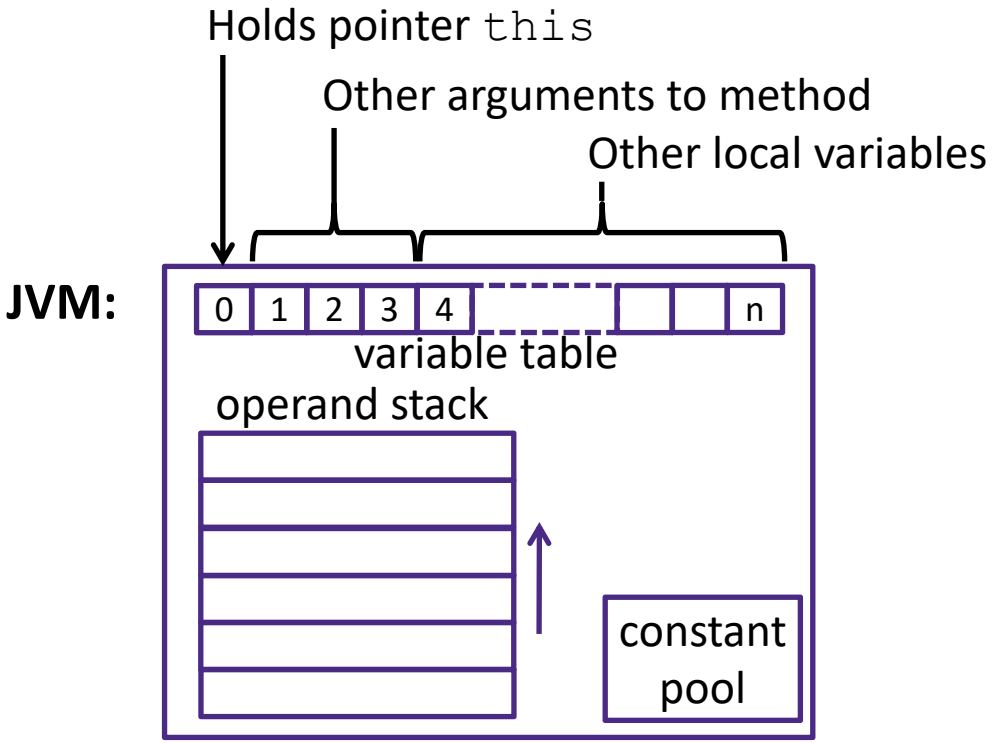


Java Bytecode

- Like assembly code for JVM, but works on *all* JVMs
  - Hardware-independent!
- Typed (unlike x86 assembly)
- Strong JVM protections



JVM Operand Stack



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

Bytecode:

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

No registers or stack locations!
All operations use operand stack

Compiled to (IA32) x86:

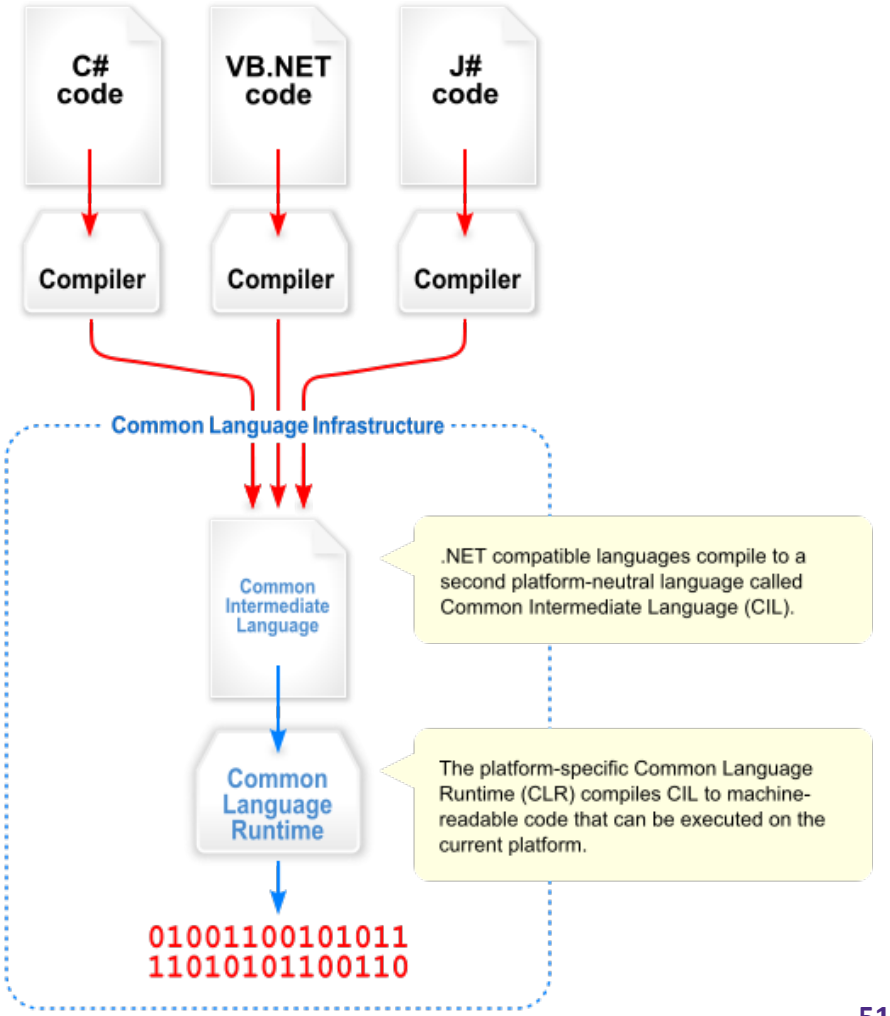
```
mov 8(%ebp),  %eax
mov 12(%ebp), %edx
add %edx, %eax
mov %eax, -8(%ebp)
```

Other languages for JVMs

- JVMs run on so many computers that compilers have been built to translate many other languages to Java bytecode:
  - **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 for web apps
  - **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!
- Originally, JVMs were designed and built for Java (still the major use) but JVMs are also viewed as a safe, GC'ed platform

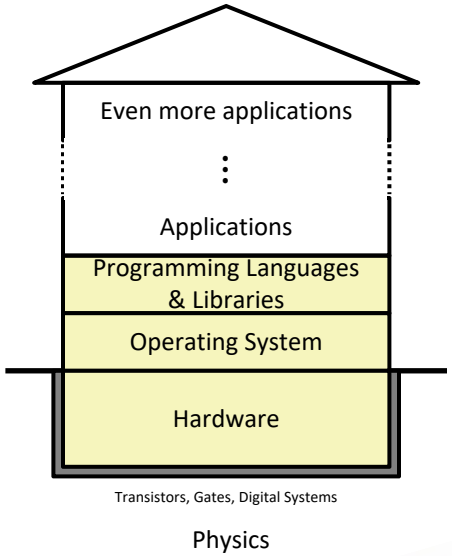
Microsoft's C# and .NET Framework

- C# has similar motivations as Java
  - Virtual machine is called the *Common Language Runtime*
  - *Common Intermediate Language* is the bytecode for C# and other languages in the .NET framework



We made it! 😌 😎

- Topic Group 1: **Data**
  - Memory, Data, Integers, Floating Point, Arrays, Structs
- Topic Group 2: **Programs**
  - x86-64 Assembly, Procedures, Stacks, Executables
- Topic Group 3: **Scale & Coherence**
  - Caches, Processes, Virtual Memory, Memory Allocation



BONUS SLIDES

Additional explanations for the Memory-Related Issues problems in hw25.

Find That Bug! (Slide 61)

```
char s[8];   //small buffer
int i;

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

no bounds checking

buffer overflow!

Error Type: E

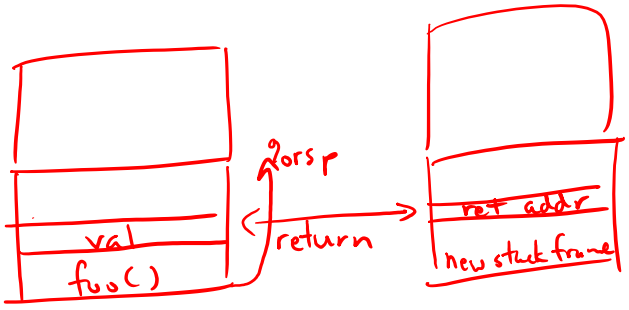
Prog stop Possible? Y

Fix: fgets(s, 8)

Find That Bug! (Slide 62)

```
int* foo() {
    int val = 0;

    return &val;
}
```

can't be in a register



referencing nonexistent variables

valid address on the stack

Error Type: G

Prog stop Possible? N

Fix: pass-by-reference to foo or use malloc instead

Find That Bug! (Slide 63)

```
int** p;

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

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

↑ allocates N ints = 4*N bytes

↑ writes to N int* = 8*N bytes

- N and M defined elsewhere (#define)

wrong allocation size

runs off end of allocated block

Error Type: H

Prog stop Possible? Y

Fix: N * sizeof(int*)

Find That Bug! (Slide 64)

```
/* return y = Ax */
int* matvec(int** A, int* x) {
    int* y = (int*)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;
}
```

y[i] = y[i] + A[i][j] * x[j];
↑ reads garbage!

- A is NxN matrix, x is N-sized vector (so product is vector of size N)
- N defined elsewhere (#define)

reading uninitialized memory

just using garbage values – runs fine but get weird results

Error Type: F

Prog stop Possible? N

Fix: calloc(N, sizeof(int))

Find That Bug! (Slide 65)

- The classic `scanf` bug
  - `int scanf(const char *format)`

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

← reads input, parses int, stores into location val

segfault if val does not contain a valid address

dereferencing a non-pointer

Error Type: A

Prog stop Possible? Y

Fix: `scanf("%d", &val);`

Find That Bug! (Slide 66)

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

    ...

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

free again

undefined behavior (some systems will segfault)

Error Type: C

Prog stop Possible? Y

Fix: free (y) ← probably a typo

Find That Bug! (Slide 67)

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

    ...

y = (int*)malloc( M * sizeof(int) );
for (i=0; i<M; i++)
    y[i] = x[i]++;
```

access freed memory

undefined behavior

Error Type: B

Prog stop Possible? Y

Fix: free(x) later (at bottom)

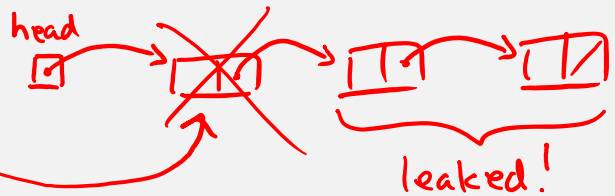
Find That Bug! (Slide 68)

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

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

node:

mallocs here

only frees first node!



how do you detect?

memory leak

Error Type: D

Prog stop Possible? N

Fix: recursive/iterative free over list