Programming Languages and Techniques (CIS120)

Lecture 38

Advanced Java Miscellany

(Concurrency / Hashing / Garbage Collection / Lambdas & Streams)

Advanced Java Miscellany

- Threads & Synchronization
- Hashing: HashSets & HashMaps
- Java 1.8 Lambdas (& Streams)
- Garbage Collection

The slides touch on these. Lecture will cover only some parts...

- Packages
- JVM (Java Virtual Machine) and compiler details:
  - class loaders, security managers, just-in-time compilation
- Advanced Generics
  - *Bounded Polymorphism*: type parameters with 'extends' constraints
    `class C<A extends Runnable> { … }`
  - `Type Erasure`
  - `Interaction between generics and arrays`
- `Reflection`
  - `The Class class`

For all the nitty-gritty details:
Java Language Specification
http://docs.oracle.com/javase/specs/

Threads & Synchronization

Avoid Race Conditions!

(see Multithreaded.java)

Threads

- Java programs can be *multithreaded*
  - more than one “thread” of control operating simultaneously
- A `Thread` object can be created from any class that implements the `Runnable` interface
  - `start`: launch the thread
  - `join`: wait for the thread to finish
- Abstract Stack Machine:
  - Each thread has its own workspace and stack
  - All threads *share* a common heap
  - Threads can communicate via shared references

Uses + Perils

- Threads are useful when one program needs to do multiple things simultaneously:
  - game animation + user input
  - chat server interacting with multiple chat clients
  - hide latency: do work in one thread while another thread waits (e.g. for disk or network I/O)

- Problem: Race Conditions
  - What happens when one thread tries to read a memory location at the same time another thread is writing it?
  - What if more than one thread tries to write different values at the same time?

(Unsynchronized) Implementation

```
interface Counter {
  public void inc();
  public int get();
}

class UCounter implements Counter {
  private int cnt = 0;

  public void inc() {
    cnt = cnt + 1;
  }

  public int get() {
    return cnt;
  }
}
```

Setting up a Computation Thread

```java
// The computation thread simply increments
// the provided counter 1000 times
class CounterUser implements Runnable {
  private Counter c;
  private int id;

  CounterUser(int id, Counter c) {
    this.id = id;
    this.c = c;
  }

  @Override
  public void run() {
    for (int i = 0; i < 1000; i++) {
      // System.out.println("Thread: " + id);
      c.inc();
    }
  }
}
```

First Try: Two Threads & One Counter

```
public class MultiThreaded {

  public static void main(String[] args) {
    Counter c = new UCounter();

    // set up a race on the shared counter c
    Thread t1 = new Thread(new CounterUser(1, c));
    Thread t2 = new Thread(new CounterUser(2, c));
    t1.start();
    t2.start();
    try {
      t1.join();
      t2.join();
    } catch (InterruptedException e) {
    }
    System.out.println("Counter value = " + c.get());
  }

}
```

Create thread 1

Create thread 2

Start thread 1

Start thread 2

Wait for thread 1 to finish

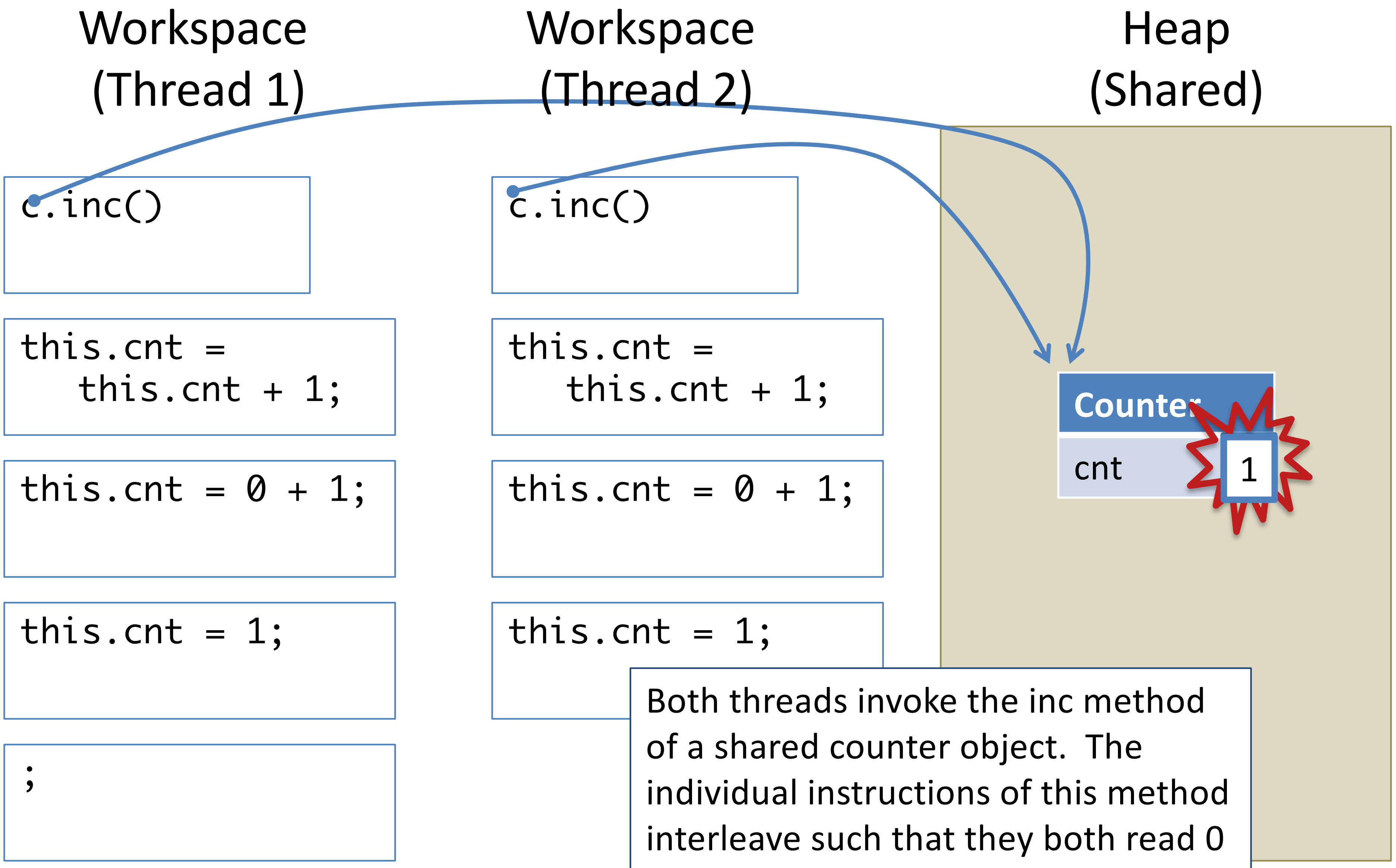
Wait for thread 2 to finish

What behavior do you expect from Multithreaded.java?

1. The program will print "Counter value = 1000"
2. The program will print "Counter value = 2000"
3. The program will print "Counter value = ????" for some other number ????
4. The program will throw an exception.

Answer: The program with print “Counter value = *val*”
for 1000 <= val <= 2000.
The answer will likely be *different* each time the program is run!!!!

Data Races



Thread 1:

```
c.inc()
```

```
this.cnt =
   this.cnt + 1;
```

```
this.cnt = 0 + 1;
```

```
this.cnt = 1;
```

```
;
```

Thread 2:

```
c.inc()
```

```
this.cnt =
   this.cnt + 1;
```

```
this.cnt = 0 + 1;
```

```
this.cnt = 1;
```

Both threads invoke the inc method of a shared counter object. The individual instructions of this method interleave such that they both read 0 and write 1.

The `synchronized` keyword

- Synchronized methods are *atomic*
  - At most one thread can be executing code within an atomic method

- Careful use will eliminate races

- Tradeoff:
  - less concurrency means worse performance

Second Try: use Synchronization

```
//This class uses synchronization
class SynchronizedCounter implements Counter {
  private int cnt = 0;

  public synchronized void inc() {
   cnt = cnt + 1;
  }

  public synchronized int get() {
   return cnt;
  }
}
```

Using The New Counters

```java
public class MultiThreaded {

  public static void main(String[] args) {

    Counter c = new SynchronizedCounter();

    // set up a race on the shared counter c
    Thread t1 = new Thread(new CounterUser(1, c));
    Thread t2 = new Thread(new CounterUser(2, c));
    t1.start();
    t2.start();
    try {
      t1.join();
      t2.join();
    } catch (InterruptedException e) {
    }

    System.out.println("Counter value = " + c.get());
  }

}
```

New!!

Now what behavior do you expect from Multithreaded.java?

1. The program will print "Counter value = 1000"

2. The program will print "Counter value = 2000"

3. The program will print "Counter value = ????" for some other number ????

4. The program will throw an exception.

Answer: The program with print “Counter value = 2000” every time.

Other Synchronization in Java

Need *thread safe* libraries:

- `java.util.concurrent` has `BlockingQueue` and `ConcurrentMap`
- help rule out synchronization errors
- Note: Swing is *not* thread safe!

- Java also provides *locks*
  - objects that act as synchronizers for blocks of code

- *Deadlock*: cyclic dependency in synchronization of locks
  - Thread A waiting for lock held by B,
    Thread B waiting for lock held by A

Immutability!

- Note that read-only datastructures are immune to race conditions
  - It’s OK for multiple threads to read a heap location simultaneously
  - Less need for locking, synchronization

- As always: immutable data structures simplify your code

Real-world example:

FaceBook's Haxl Library

- Library written in Haskell
- Concurrency / Distributed Database
- https://github.com/facebook/Haxl

Hash Sets & Hash Maps

array-based implementation of sets and maps

Hash Sets and Maps: The Big Idea

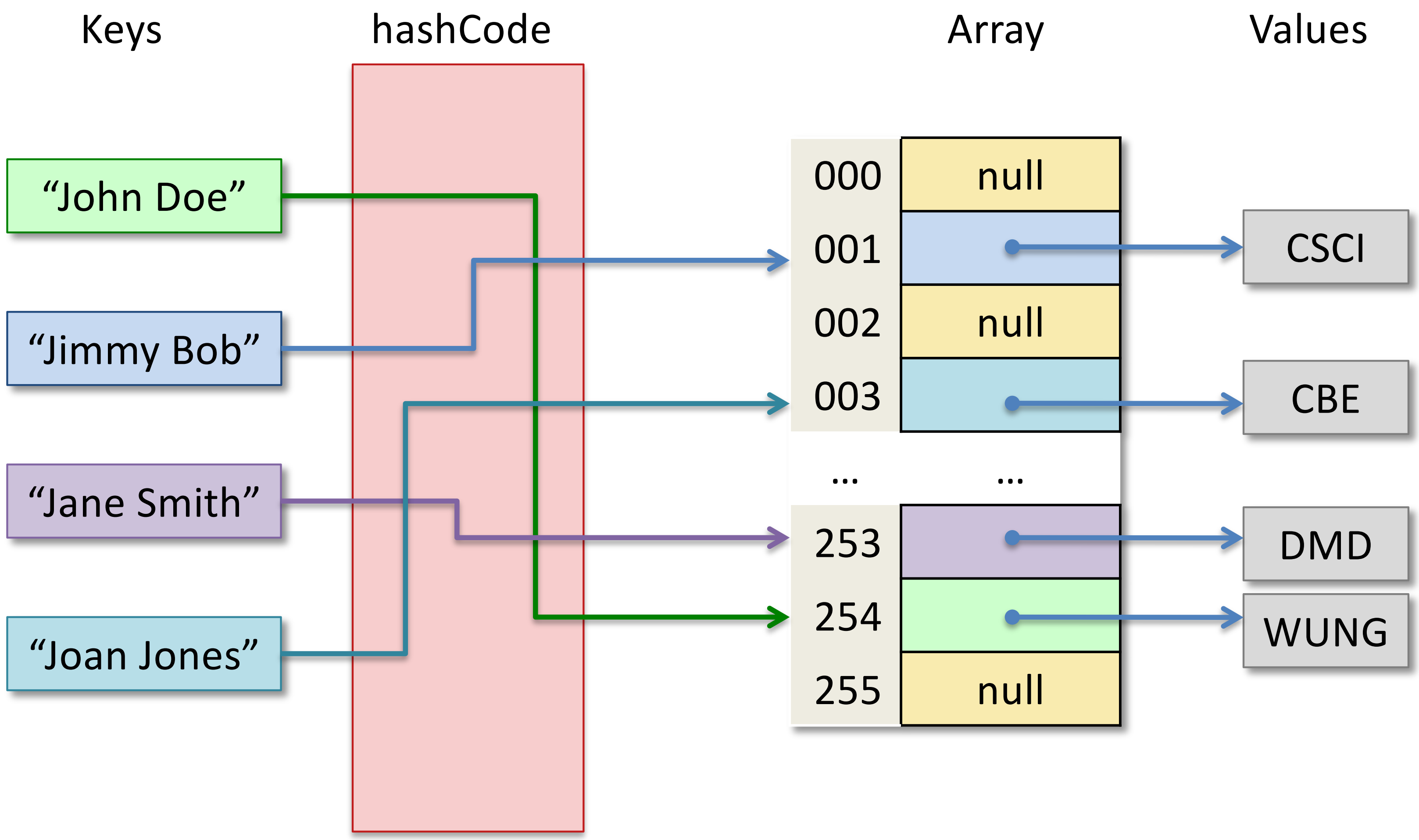
Combine:

- the advantage of arrays:
  - *efficient* random access to its elements
- with the advantage of a map datastructure
  - arbitrary keys (not just integer indices)

How?

- Create an index into an array by *hashing* the data in the key to turn it into an int
  - Java's hashCode method maps key data to ints
  - Generally, the space of keys is much larger than the space of hashes, so, unlike array indices, hashCodes might not be unique

Hash Maps, Pictorially



A schematic HashMap taking Strings (student names) to Undergraduate Majors. The hashCode takes each string name to an integer code, which we then take "mod 256" to get an array index between 0 and 255.
For example, `"John Doe".hashCode()` mod 256 is 254.

Hash Collisions

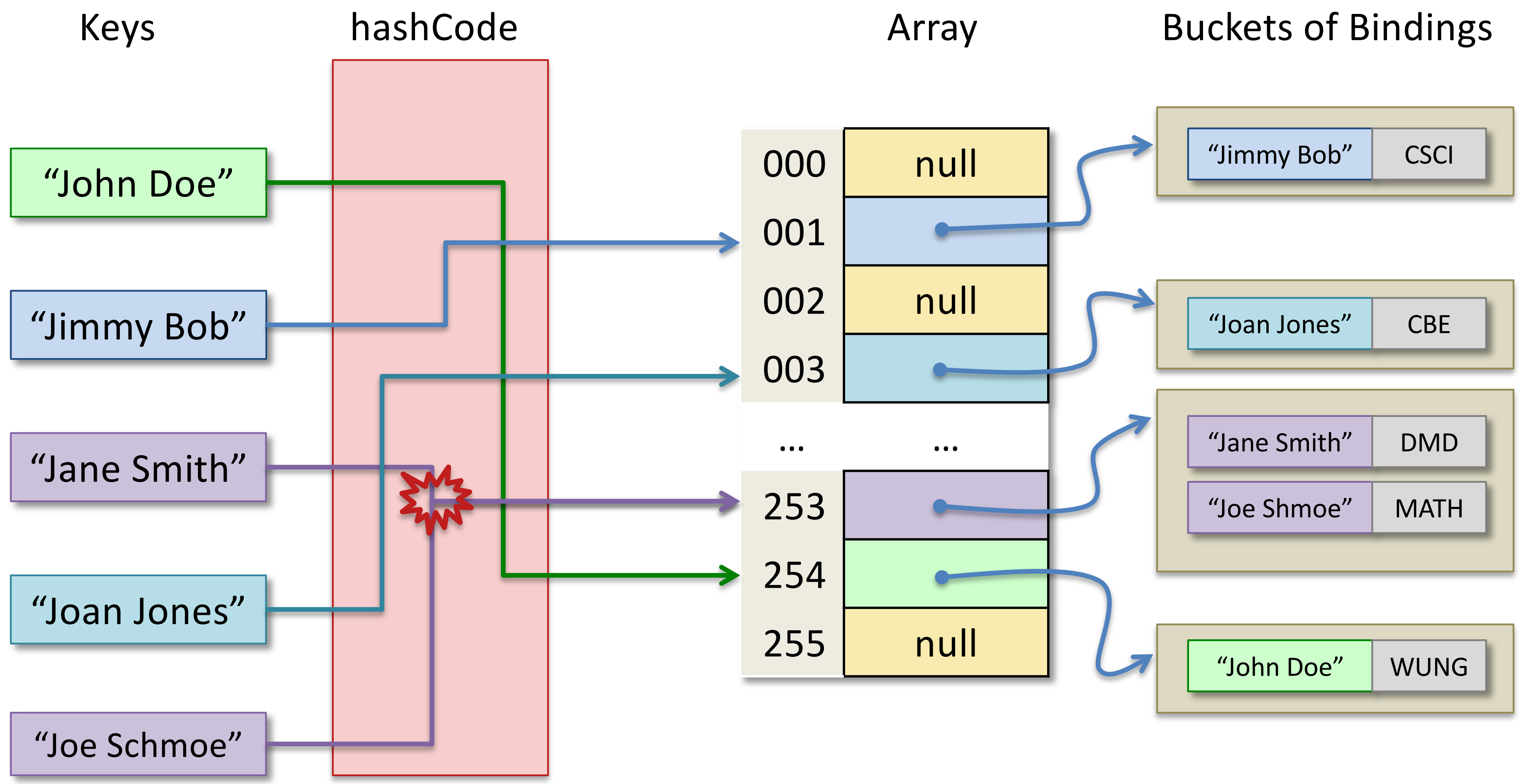
- Uh Oh: Indices derived via hashing may not be unique!

```
"Jane Smith".hashCode() % 256 ➔ 253
"Joe Schmoe".hashCode() % 256 ➔ 253
```

- Good hashCode functions make it *unlikely* that two keys will produce the same hash

- But, it can still sometimes happen that two keys produce the same index – that is, their hashes *collide*

Bucketing and Collisions



Here, "Jane Smith".hashCode() and "Joe Schmoe".hashCode() happen to collide. The *bucket* at the corresponding index of the Hash Map array stores the map data.

Bucketing and Collisions

- Using an array of *buckets*
  - Each bucket stores the mappings for keys that have the same hash.
  - Each bucket is itself a map from keys to values (implemented by a linked list or binary search tree).
  - The buckets can't use hashing to index the values – instead they use key equality (via the key's equals method)

- To look up a key in the Hash Map:
  1. Find the right bucket by indexing the array through the key's hash
  2. Search linearly through the bucket contents to find the value associated with the key

- Not the only solution to the collision problem

Hashing and User-defined Classes

```
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) { this.x = x; this.y = y;
}
    public int getX() { return x; }
    public int getY() { return y; }
}

// somewhere else…
Map<Point,String> m = new HashMap<Point,String>();
m.put(new Point(1,2), "House");
System.out.println(m.containsKey(new Point(1,2)));
```

What gets printed to the console?

1. true
2. false
3. I have no idea

ANSWER: 2 – hashCode not implemented

HashCode Requirements

Whenever you override `equals` you must also override `hashCode` in a consistent way:

- whenever `o1.equals(o2) == true` you must ensure that `o1.hashCode() == o2.hashCode()`

Why? Because comparing hashes is supposed to be a quick approximation for equality.

- Note: the converse does not have to hold:
  - `o1.hashcode() == o2.hashCode()` does *not* necessarily mean that o1.equals(o2)

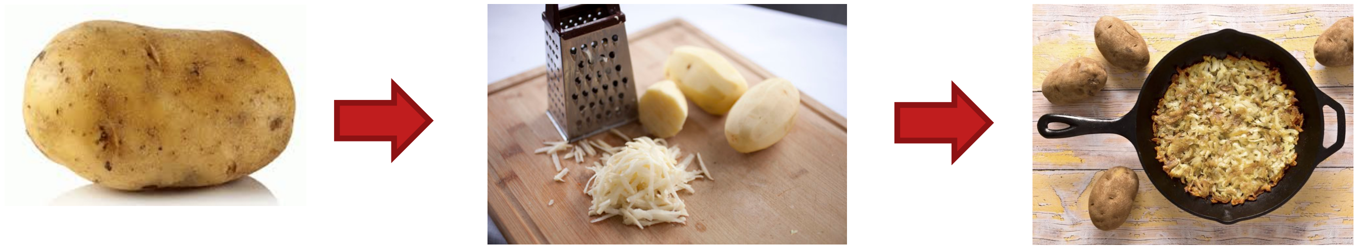
Example for Point

```
public class Point {
   @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + x;
        result = prime * result + y;
        return result;
    }
}
```

- Examples:
  - (new Point(1,2)).hashCode()   yields 994
  - (new Point(2,1)).hashCode()   yields 1024
- Note that equal points (in the sense of `equals`) have the same hashCode
- Why 31? Prime chosen to create more uniform distribution
- Note: Tools (e.g. eclipse) can *generate* this code

Recipe: Computing Hashes

- What is a good recipe for computing hash values for your own classes?
  - intuition: "smear" the data throughout all the bits of the resulting



1. Start with some constant, arbitrary, non-zero int in `result.`
2. For each significant field f of the class (i.e. each field used when computing equals), compute a "sub" hash code `c` for the field:
   - For boolean fields: `(f ? 1 : 0)`
   - For byte, char, int, short: `(int) f`
   - For long: `(int) (f ^ (f >>> 32))`
   - For references: 0 if the reference is null, otherwise use the `hashCode()` of the field.
3. Accumulate those subhashes into the result by doing (for each field's `c`):
   `result = prime * result + c;`
4. return `result`

Hash Map Performance

- Hash Maps can be used to efficiently implement Maps and Sets
  - There are many different strategies for dealing with hash collisions with various time/space tradeoffs
  - Real implementations also dynamically rescale the size of the array (which might require re-computing the bucket contents)
  - See CIS 121 for more info!

- If the hashCode function gives a good (close to uniform) distribution of hashes, the buckets are expected to be small (only one or two elements)
- If the hashCode function gives a bad distribution (e.g. return 0;), the buckets will be large (and performance will be bad)

- Performance depends on workload

NOTE: Terminological Clash

- The word "hash" is also used in *cryptography*
  - SHA-1, SHA-2, SHA-3, MD5, etc.
- All hash functions reduce large objects to short summaries
- Cryptographic hashes have some extra requirements:
  - Are "one way" (i.e. very hard to *invert)*
  - Should only very rarely have collisions
  - Are considerably more expensive to compute than hashCode (so not suitable for hash tables)

- Never use hashCode when you need a cryptographic hash!
  - See CIS 331 for more details

Hashing: take away lessons

equals

hashCode

compareTo

Collections Requirements

- All collections invoke `equals` method on elements
  - Defaults to == (reference equality)
  - Override `equals` to create structural equality
  - Should always be an equivalence relation: reflexive, symmetric, transitive

- HashSets/HashMaps also invoke `hashCode` method on elements
  - Override when equals is overridden
  - Should be "compatible" with `equals`
  - Should try to distribute hash codes uniformly
  - Iterators are not guaranteed to follow order of hashCodes

- Ordered collections (`TreeSet`, `TreeMap`) require element type to implement `Comparable` interface
  - Provide `compareTo` method
  - Should implement a *total order*
  - Should be compatible with equals
    - (i.e. `o1.equals(o2)` exactly when `o1.compareTo(o2) == 0`)

Java 1.8 Functional Programming and Lambdas

or: How I Learned to Stop Worrying and Love Functional Programming in Java

(See PaintF.java)

Problem – Boilerplate Code in Java

- Using anonymous inner classes (Not great, but better than named classes)

```
quit.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
       System.exit(0);
   }
 });
```

- Using Lambdas (Much better!)

```
 quit.addActionListener(e ->
       System.exit(0)
 );
```

Lambdas – Why and What?

- Often implementation of anonymous classes is simple
  - e.g., an interface that contains only one method
- *Lambda* expressions:*
  - treat functionality as method argument, or code as data.
  - Java's version of first-class functions
- Pass functionality as an argument to another method,
  - e.g., what action should be taken when someone clicks a button.
- *Any* interface that has exactly one method can be implemented via a "lambda" (anonymous function).
  - method "name" implicitly determined by the type at which the lambda is used
  - https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

*The term "lambda" comes from the *lambda calculus,* which was introduced by Alonzo Church in the 1930s. The lambda calculus forms the theoretical basis of all functional programming languages.

Lambdas – Why and What?

- Helps create instances of single-method classes more easily
- Think of them as anonymous methods

```
thick.addItemListener(e -> {
   if (e.getStateChange() == ItemEvent.SELECTED) {
      stroke = thickStroke;
   } else {
      stroke = thinStroke;
});
```

Method Argument(s)

```
SwingUtilities.invokeLater(() -> new PaintF());
```

Java Lambda In A Nutshell

Lambda Notation	"Ordinary" Java Notation

`x -> x + x`	`int method1(int x) {`
	`  return x + x;`
	`}`

`(x,y) -> x.m(y)`	`int method2(A x, B y) {`
	`  return x.m(y);`
	`}`

`(x,y) -> {`
`  System.out.println(x);`
`  System.out.println(y);`
`}`
	`void method3(String x,`
	`             String y) {`
	`  System.out.println(x);`
	`  System.out.println(y);`
	`}`

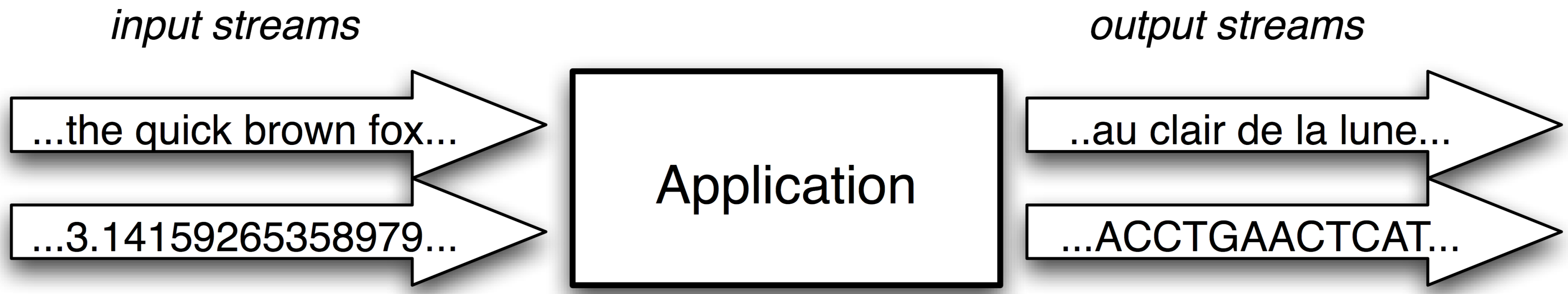
Method names and types are inferred from the context.

Functional Programming + Streams

(See Streams.java)

I/O Streams

- The *stream* abstraction represents a communication channel with the outside world.
  - can be used to read or write a potentially unbounded number of data items (unlike a list)
  - data items are read from or written to a stream one at a time
- The Java I/O library uses subtyping to provide a unified view of disparate data sources and sinks.



Streams redux

- Use *streams* of elements to support functional-style operations on collections
- Key differences between streams and collections:
  - No storage (i.e., not a data structure)
  - Functional in nature (i.e., do not modify the source)
  - Possibly unbounded (i.e., computations on infinite streams can complete in finite time)
  - Consumable (i.e., similar to Iterator)
  - Laziness-seeking
    - "Find the first input String that begins with a vowel" doesn't need to look at all Strings from the input

Creating Streams (1)

- From a `Collection` via the `stream()` and `parallelStream()` methods
- From an array via `Arrays.stream()`
- The lines of a file can be obtained from `BufferedReader.lines()`
- Streams of random numbers can be obtained from `Random.ints();`
- Numerous other stream-bearing methods in the JDK

Creating Streams (2)

- Can create your own Low-Level Stream
- Similar to having a custom class like `WordScanner` that implements `Iterator`

- `Spliterator` – parallel analogue to `Iterator`
  - (Possibly infinite) Collection of elements
  - Support for:
    - Sequentially advancing elements (similar to `next()`)
    - Bulk Traversal (performs the given action for each remaining element, *sequentially* in the current thread)
    - Splitting off some portion of the input into another spliterator, which can be processed in *parallel* (much easier than doing threads manually!)

Stream Pipeline Operations

- Intermediate (Stream-producing) operations
  - E.g., `filter`, `map`, `sorted`
  - Similar to transform in Ocaml
  - Return a new stream
  - Always lazy (produce elements as needed, not ahead of time)
  - Traversal of the source does not begin until the terminal operation of the pipeline is executed
- Terminal (value- or side-effect-producing) operations
  - E.g. `forEach`, `reduce`, `findFirst`, `allMatch`, `max`, `min`
  - Similar to fold in Ocaml
  - Produce a result or side-effect

- Combined to create Stream pipelines

Lambdas, Streams, Pipelines

- Beauty and Joy of functional programming, now in Java!

```
roster.stream()
      .filter(p ->
              p.getGender() == Person.Sex.MALE
              && p.getAge() >= 18
              && p.getAge() <= 25)
      .map(p -> p.getEmailAddress())
      .forEach(email -> System.out.println(email));
```

```
int sum = widgets.stream()
                 .filter(b -> b.getColor() == RED)
                 .mapToInt(b -> b.getWeight())
                 .sum();
```

Functional Programming + Parallelism

(See Streams.java)

Functional Programming + Parallelism

- Parallelism by design in Java 1.8
  - Streams are functional in nature (i.e., do not modify the source)
  - `Spliterator`
- Much easier than doing it manually
  - No need for `synchronized`
  - No need for locks
  - Don’t have to worry about race conditions!
- Use `parallelStream()` (instead of `stream()`)!
  - Java will automatically create the necessary threads and scale based on your computer’s hardware

Sample Problem

- Given a list of numbers, find the sum of the squares of the numbers

- **Iterative Approach**

```
int sum = 0;
for (int i = 0; i < list.size(); i++) {
   sum += list.get(i);
}
```

- Works, more likely to have bugs (off-by-one), harder to parallelize

Sample Problem

- Given a list of numbers, find the sum of the squares of the numbers

- **Functional Approach**
- Use `transform` and `fold` (aka `map` and `reduce` in Java)

```
list.parallelStream()
    .map(x -> x * x)
    .reduce(0, Integer::sum);
```

- Less likely to have bugs, much easier to parallelize

Garbage Collection & Memory Management

Cleaning up the Heap

Memory Management

- The Java Abstract Machine stores all objects in the heap.
  - We imagine that the heap has limitless space…
    … but: real machines have limited amounts of memory
- *Manual memory management*
  - C and C++
  - The programmer explicitly allocates heap objects (`malloc` / `new`)
  - The programmer explicitly de-allocates the objects (`free` / `delete`)
- *Automatic memory management (garbage collection)*
  - Reference Counting: Objective C, Swift, Python, many scripting languages
  - Mark & sweep/Copying GC: Java, OCaml, C#, Haskell (and most other ‘managed’ languages)

Manual Memory Management

See manmem.c

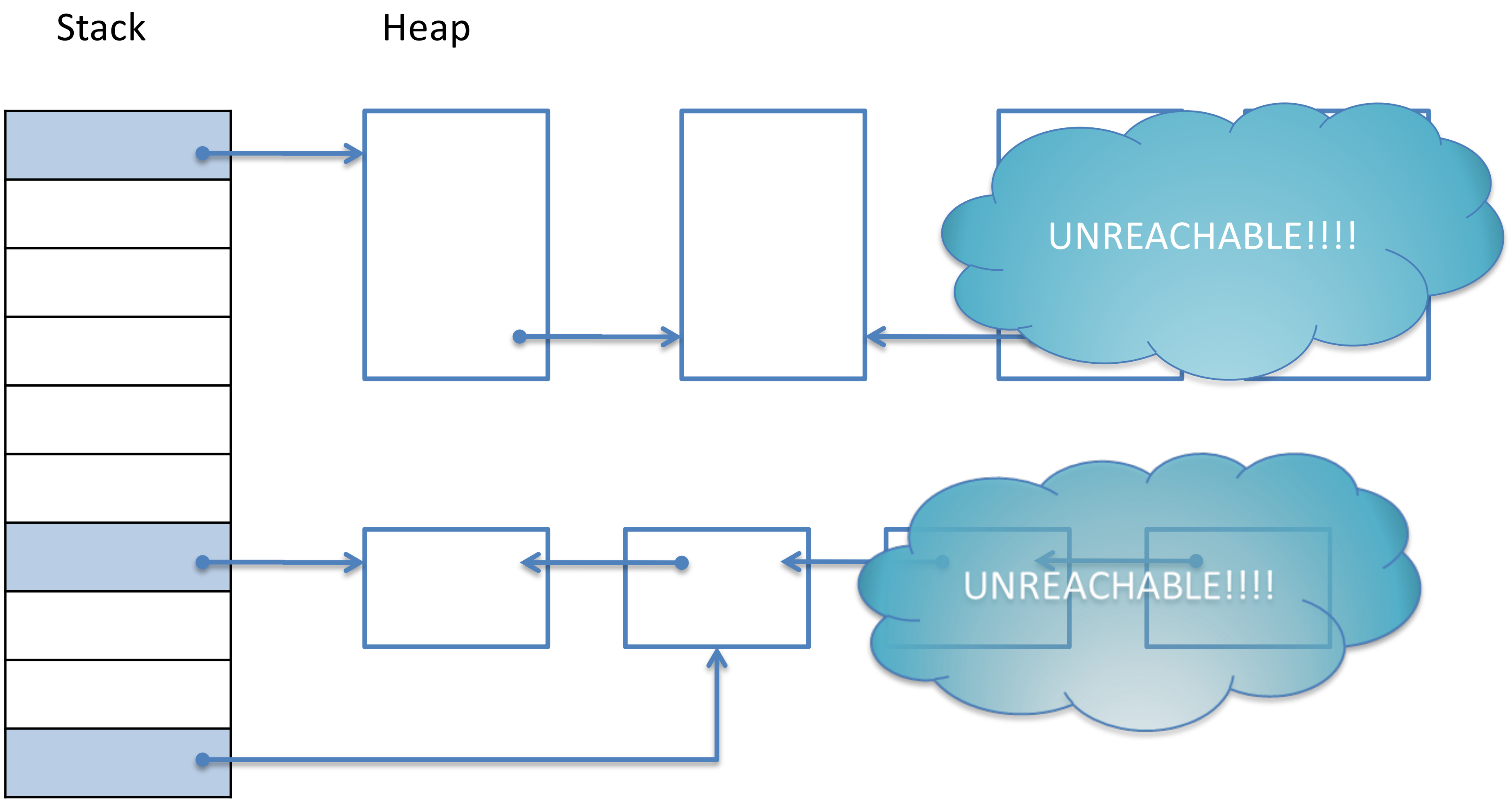
Why Garbage Collection?

- Manual memory management is cumbersome & error prone:
  - Freeing the same reference twice is ill defined (crashes or other bugs)
  - Explicit `free` isn't modular: To properly free all allocated memory, the programmer has to know what code "owns" each object. Owner code must ensure `free` is called just once.
  - Not calling `free` leads to *space leaks*: memory never reclaimed
    - Many examples of space leaks in long-running programs

- Garbage collection:
  - Have the language runtime system determine when an allocated chunk of memory will no longer be used and free it automatically.
  - Extremely convenient and safe
  - Garbage collection does impose costs (performance, predictability)

Graph of Objects in the Heap

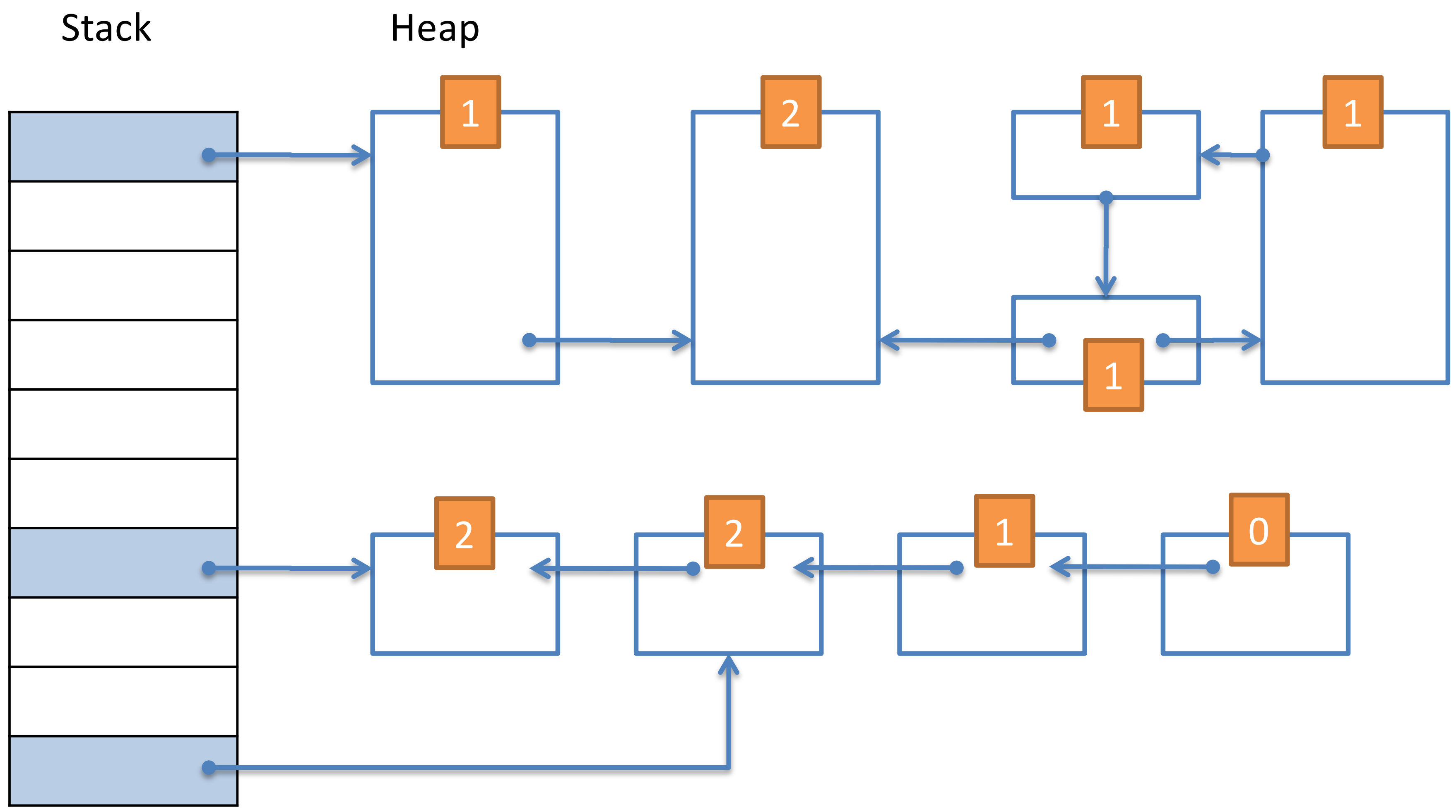
- References in the stack and global static fields are *roots*



Reference Counting

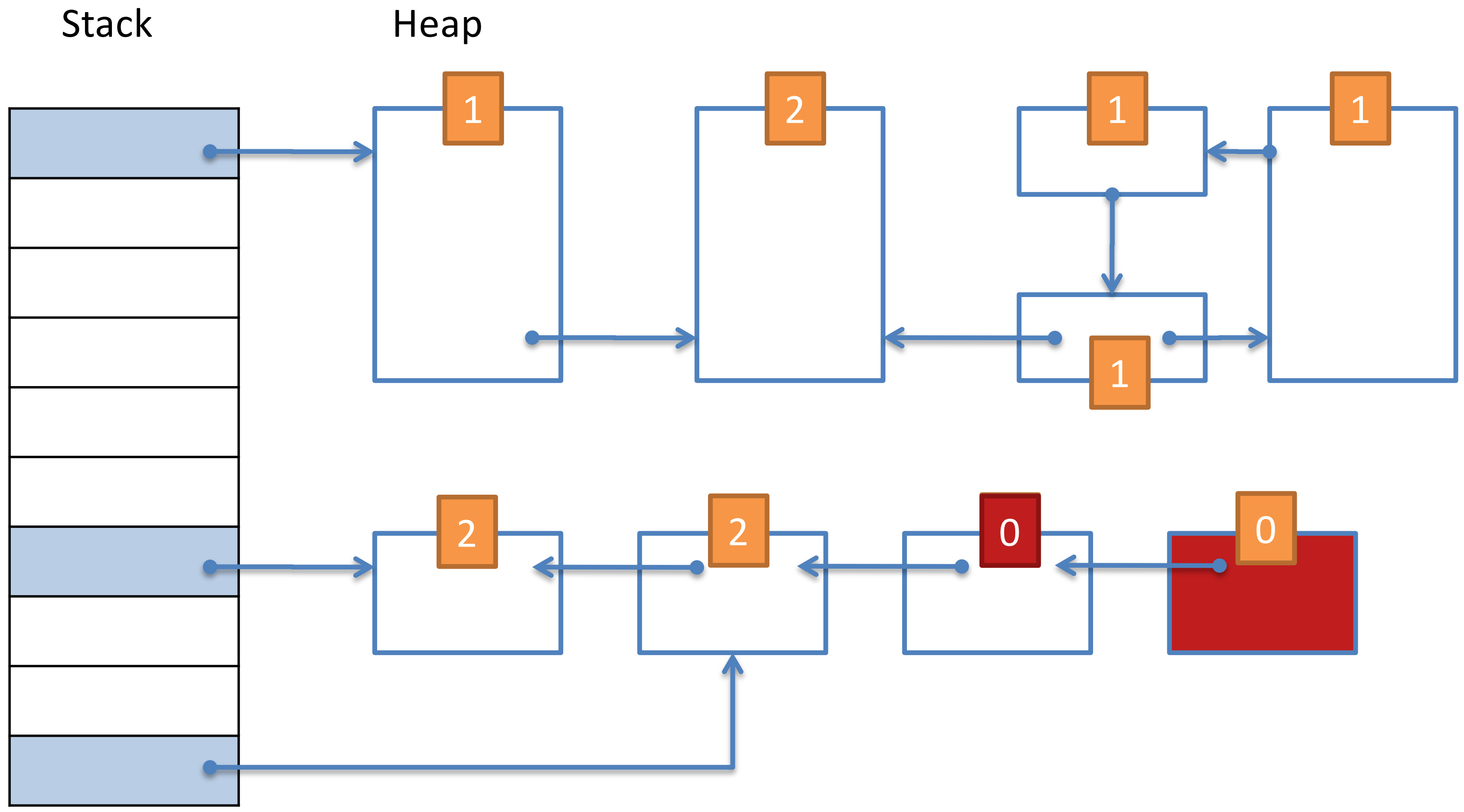
Reference Counting

- Each heap object tracks how many references point to it:



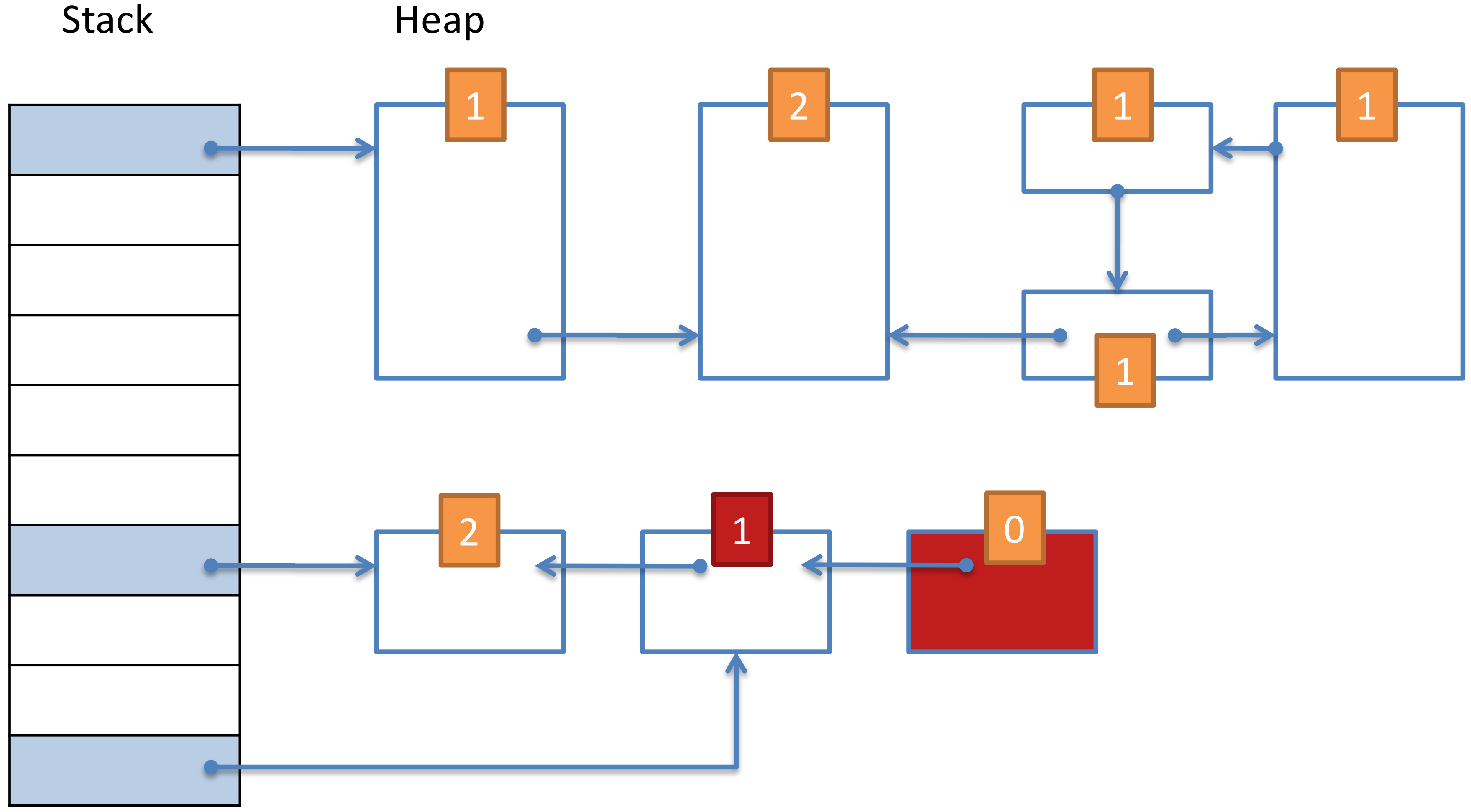
Reference Counting

- When reference count goes to 0, reclaim that space
  - and decrement counts for objects pointed to by that object



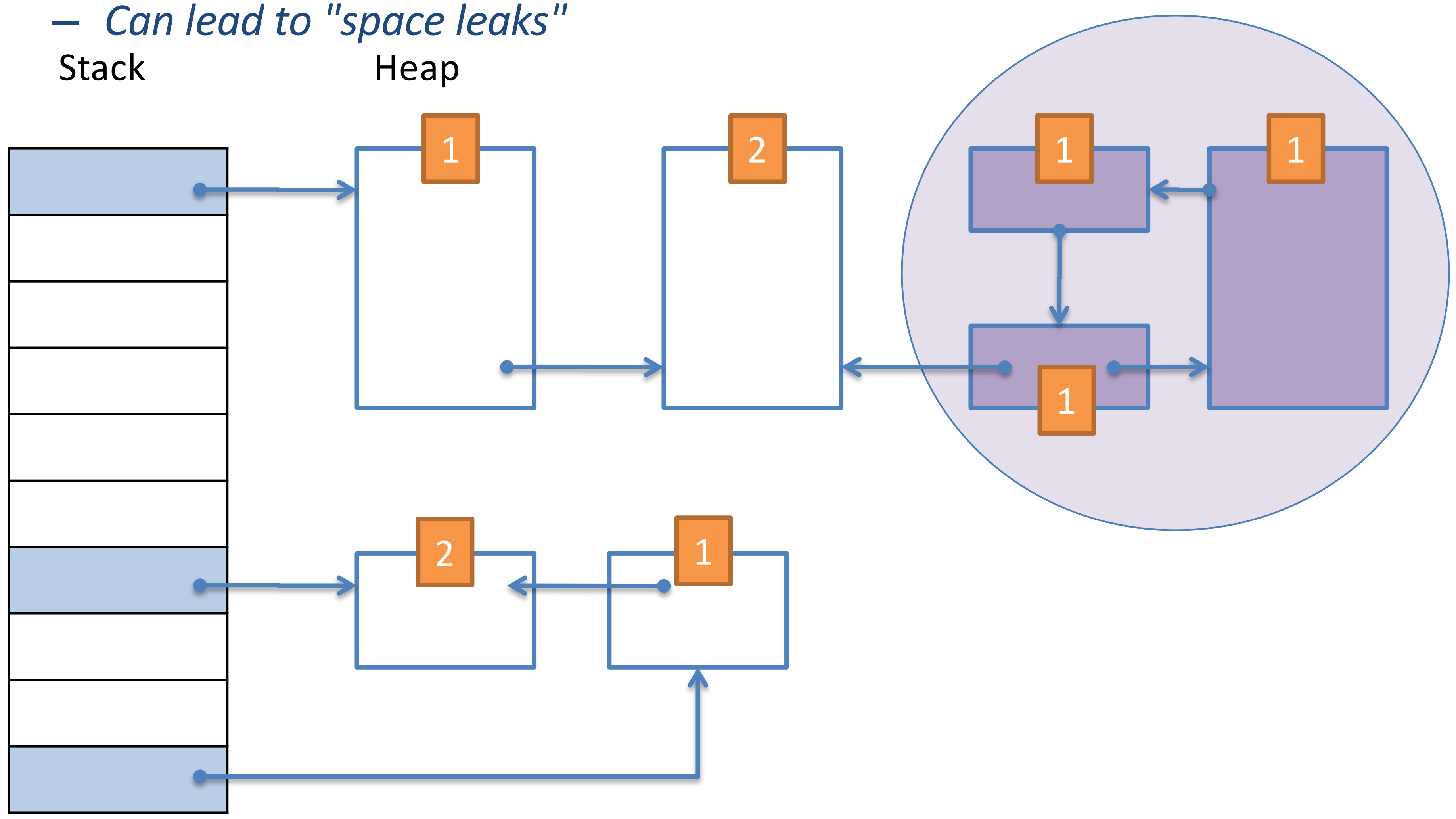
Reference Counting

- When reference count goes to 0, reclaim that space
  - and decrement counts for objects pointed to by that object



Problem: Cyclic Data

- Cycles of data will never decrement to 0!
  - *Can lead to "space leaks"*



Dealing with Cycles

- Option 1: Require programmers to explicitly null-out references to break cycles.
- Option 2: Periodically run mark & sweep GC to collect cycles
- Option 3: Require programmers to distinguish “weak pointers” from “strong pointers”
  - *weak pointers*: if all references to an object are “weak” then the object can be freed even with non-zero reference count.
  - “Back edges” in the object graph should be designated as weak
  - (Aside: weak pointers useful in GC settings too.)

Mark & Sweep / Copying

Traverse the Heap

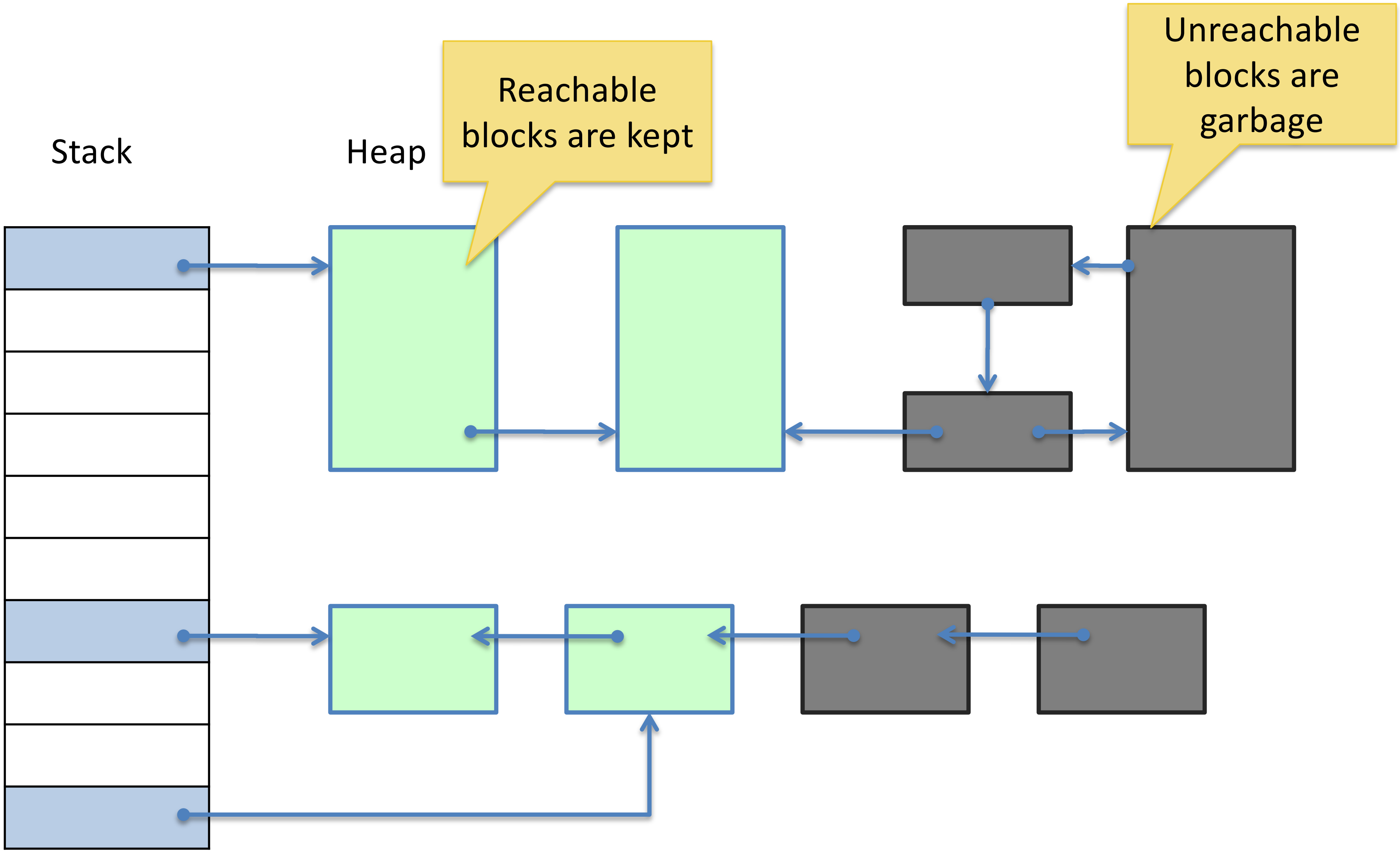
Memory Use & Reachability

- When is a chunk of memory no longer needed?
  - In general, this problem is undecidable.
- We can approximate this information by freeing memory that can't be reached from any *root* references.
  - A *root reference* is one that might be accessible directly from the program (i.e. they're not in the heap).
  - Root references include (global) static fields and references in the stack.
- If an object can be reached by traversing pointers from a root, it is *live*.
- It is safe to reclaim all heap allocations not reachable from a root (such objects are *garbage* or *dead* objects).

Mark and Sweep Garbage Collection

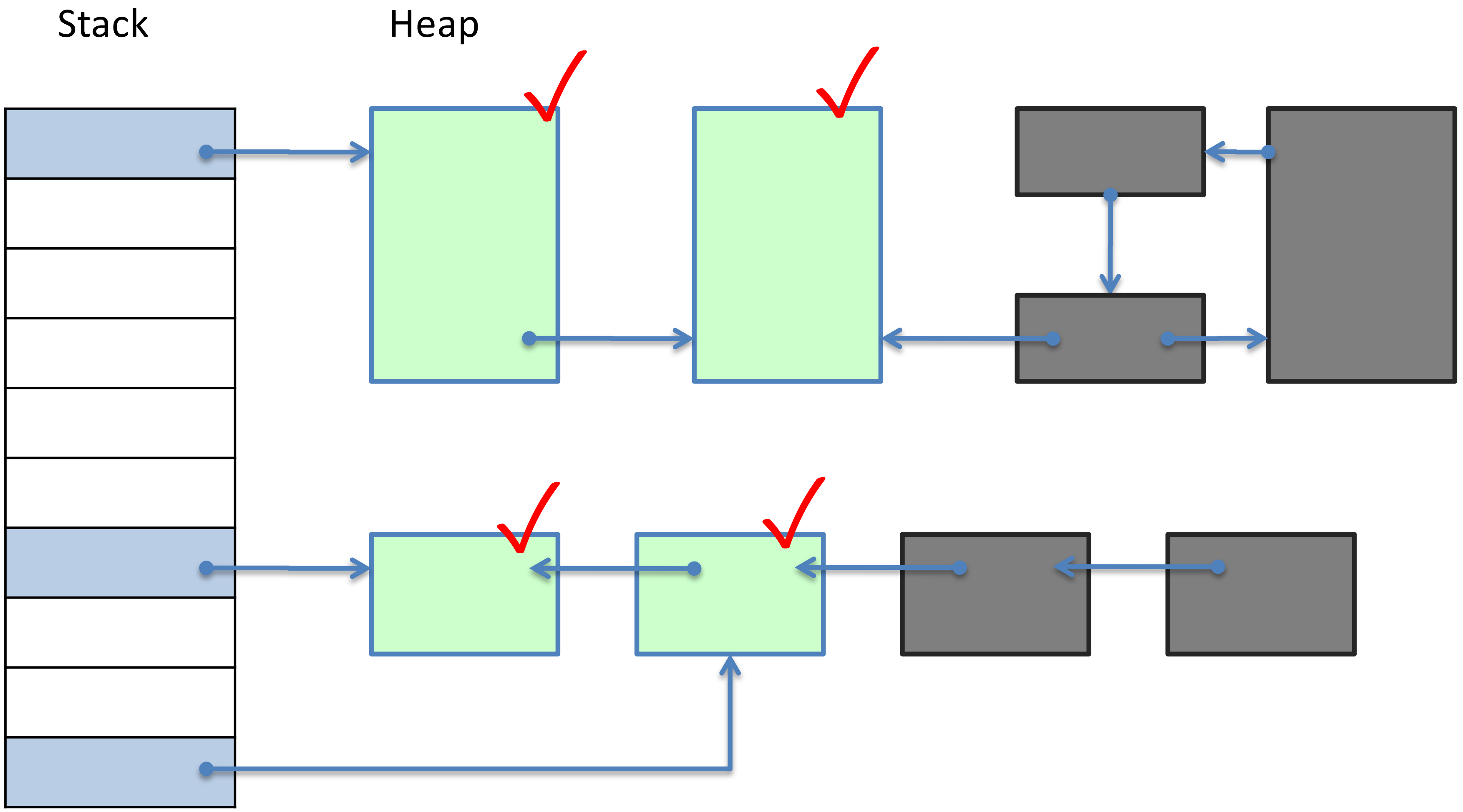
- Classic algorithm with two phases:
- Phase 1: Mark
  - Start from the roots
  - Do depth-first traversal, marking every object reached.
- Phase 2: Sweep
  - Walk over *all* allocated objects and check for marks.
  - Unmarked objects are reclaimed.
  - Marked objects have their marks cleared.
  - Optional: compact all live objects in heap by moving them adjacent to one another. (Needs extra work & indirection to "patch up" references)
- (In practice much more complex: "generational GC")

Results of Marking Graph



Second Phase: Drop "Unreachable"

- Sweep over all objects, dropping the ones marked as unreachable and keeping the ones marked reachable.



Costs & Implications

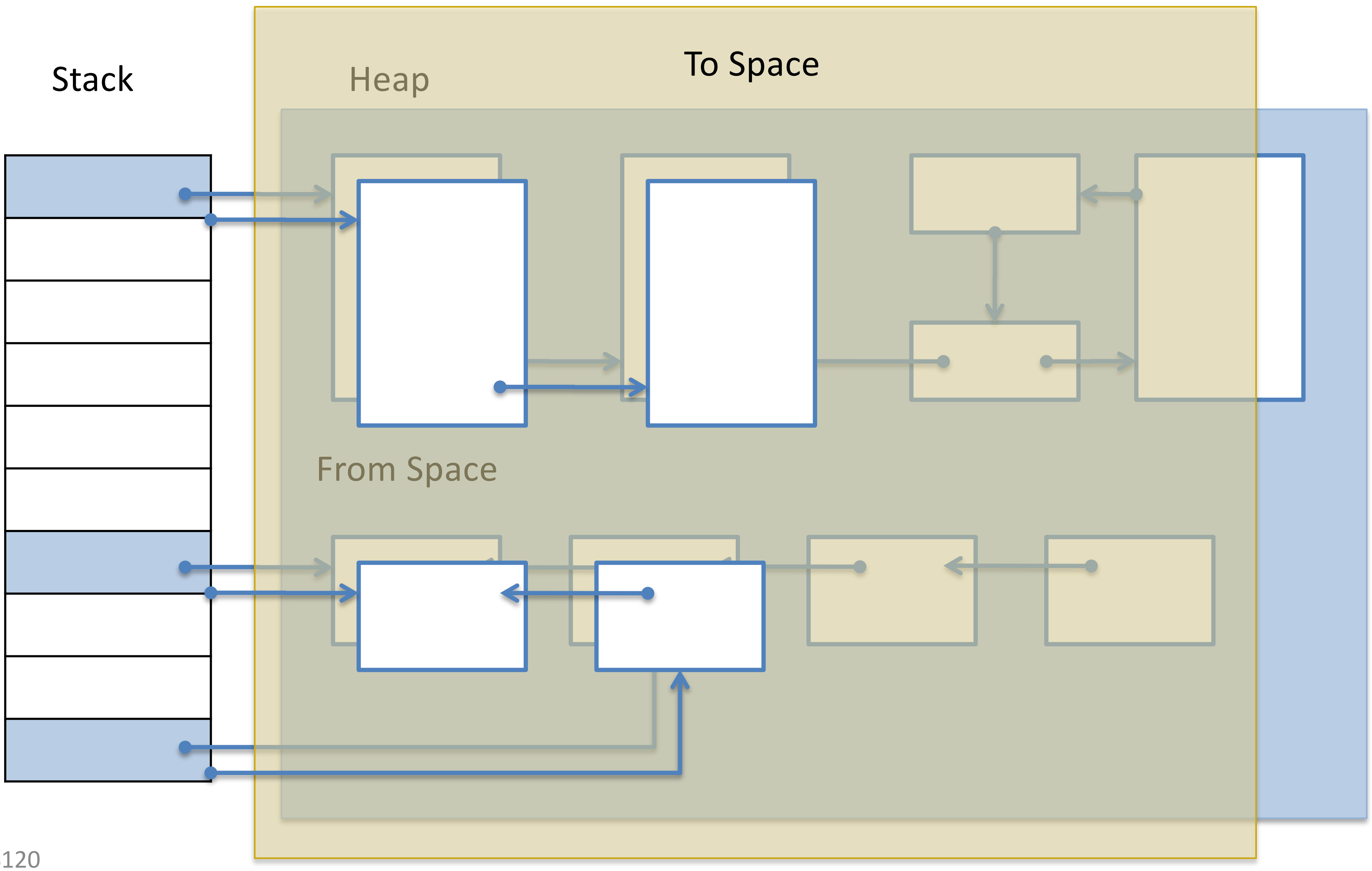
- Need to generalize to account for objects that have multiple outgoing pointers.
- Mark & Sweep algorithm reads all memory in use by the program (even if it's garbage!)
  - Running time is proportional to the total amount of allocated memory (both live and garbage).
  - Can pause the programs for long times during garbage collection.

Copying Garbage Collection

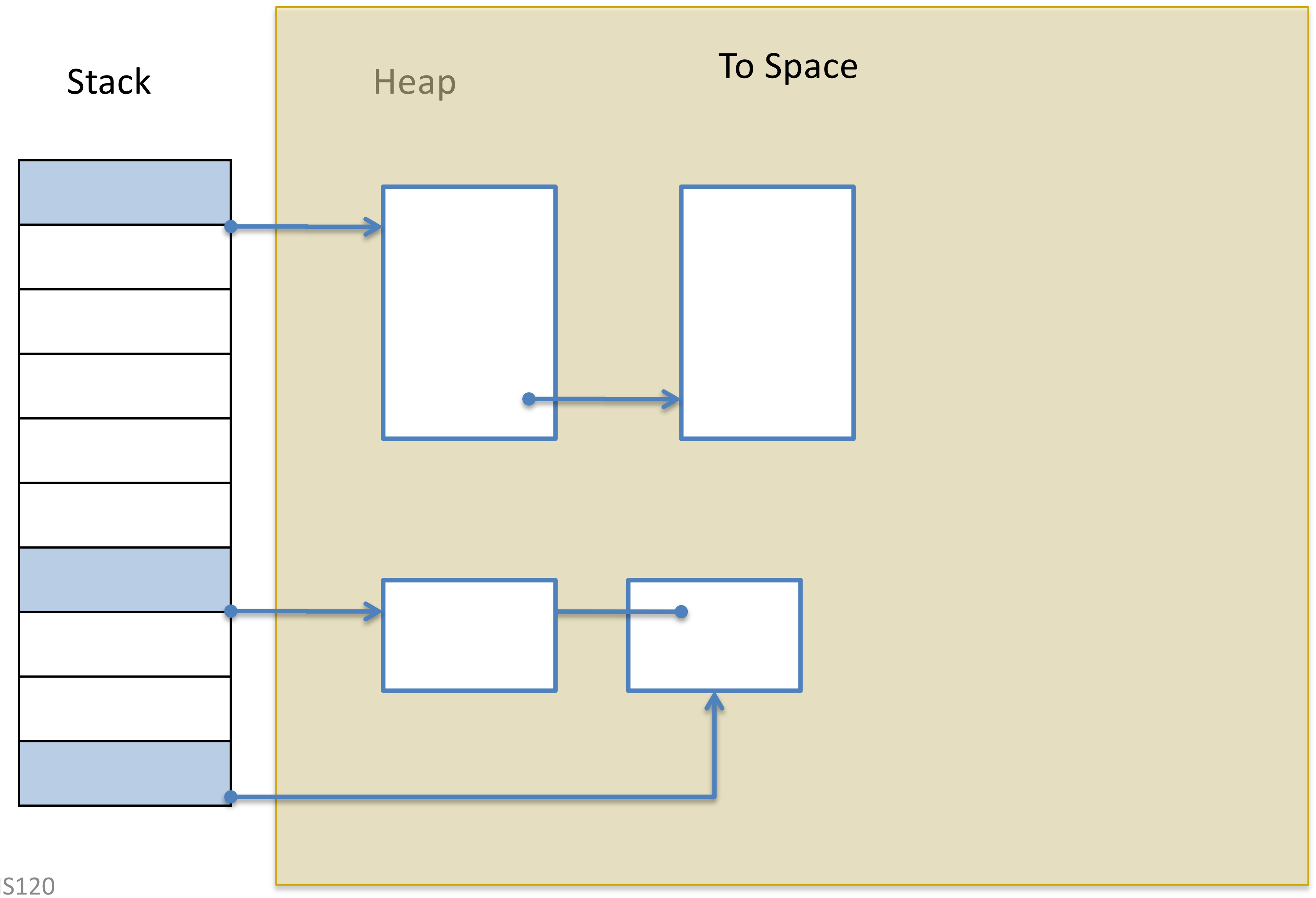
- Like mark & sweep: collects all garbage.
- Basic idea: use *two* regions of memory
  - One region is the memory in use by the program. New allocation happens in this region.
  - Other region is idle until the GC requires it.

- Garbage collection algorithm:
  - Traverse over live objects in the active region (called the "*from-space*"), copying them to the idle region (called the "*to-space*").
  - After copying all reachable data, switch the roles of the from-space and to-space.
  - All dead objects in the (old) from-space are discarded en masse.
  - A side effect of copying is that all live objects are compacted together.

Copy from "From" to "To"



Discard the "From Space"



GCDemo

See GCTest.java

Garbage Collection Take Aways

- Big idea: the Java runtime system tries to free-up as much memory as it can automatically.
  - Almost always a big win, in terms of convenience and reliability

- Sometimes can affect performance:
  - Lots of dead objects might take a long time to collect
  - When garbage collection will be triggered can be hard to predict, so there can be “pauses” (modern GC implementations try to avoid this!)
  - Global data structures can have references to “zombie” objects that won’t be used, but are still reachable ⇒ “space leak”.

- There are many advanced programming techniques to address these issues:
  - Configuring the GC parameters
  - Explicitly triggering a GC phase
  - “Weak” references