Programming Languages and Techniques (CIS120)

Lecture 37

December 4, 2015

Advanced Java Concepts: GC & Threads

FINAL EXAM

- **Wednesday, December 16, noon – 2PM**
  Two locations:
  - **CHEM 102 last names A - R**
  - **LEVH 101 last names S - Z**
- *Comprehensive* exam over course concepts:
  - OCaml material (though we won't worry much about syntax)
  - All Java material (emphasizing material since midterm 2)
  - all course content
  - old exams posted
- Closed book, but:
  - One letter-sized, *handwritten* sheet of notes allowed
- Review Session:
  - TBA

Game project grading

- Final Program Due: (88 points)

  Tuesday December 8th at 11:59pm

  – Submit zipfile online, submission *only* checks if your code compiles

- Grade based on demo with your TA during reading days

  – Make sure that you test your program in Moore 100, especially if you use outside libraries

  – Grading rubric on the assignment website

  – Recommendation: don't be too ambitious.

- *NO LATE SUBMISSIONS PERMITTED*

How is the Game Project going so far?

1. not started
2. got an idea
3. submitted design proposal
4. started coding
5. it's somewhat working
6. it's mostly working
7. debugging / polishing
8. done!

Advanced Java Miscellany

- Threads & Synchronization
- Garbage Collection

We'll touch on these.

- 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 & Reflection
  - Interaction between generics and arrays
  - The `Class` class

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

Garbage Collection

Cleaning up the Heap

Garbage Collection

- 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
- Some languages (C and C++) use *manual memory management*:
  - The programmer explicitly allocates heap objects (using 'new')
  - The programmer explicitly de-allocates the objects (using '`free`')
- Java (and most other 'managed' languages) uses *garbage collection* (GC).

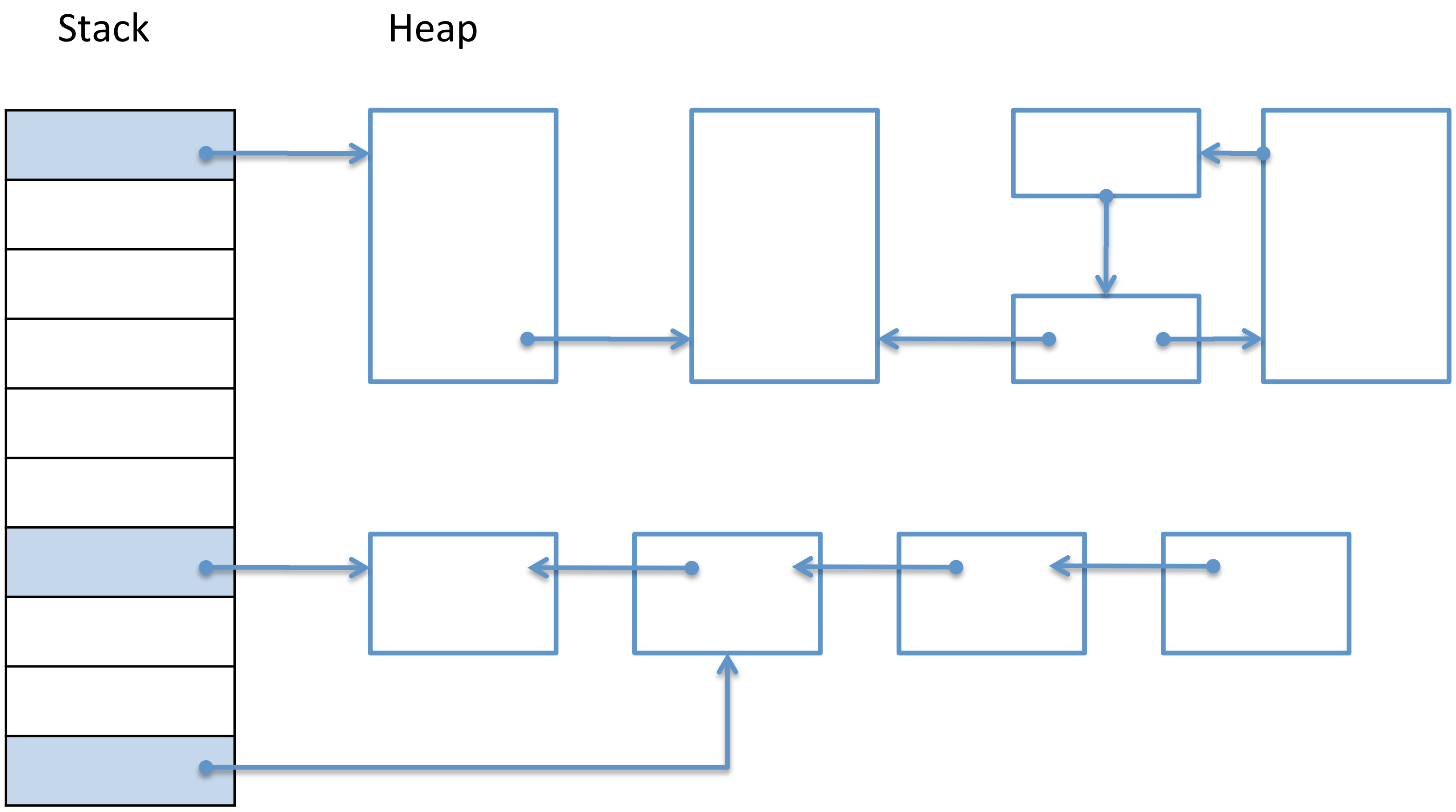
Why Garbage Collection?

- 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*

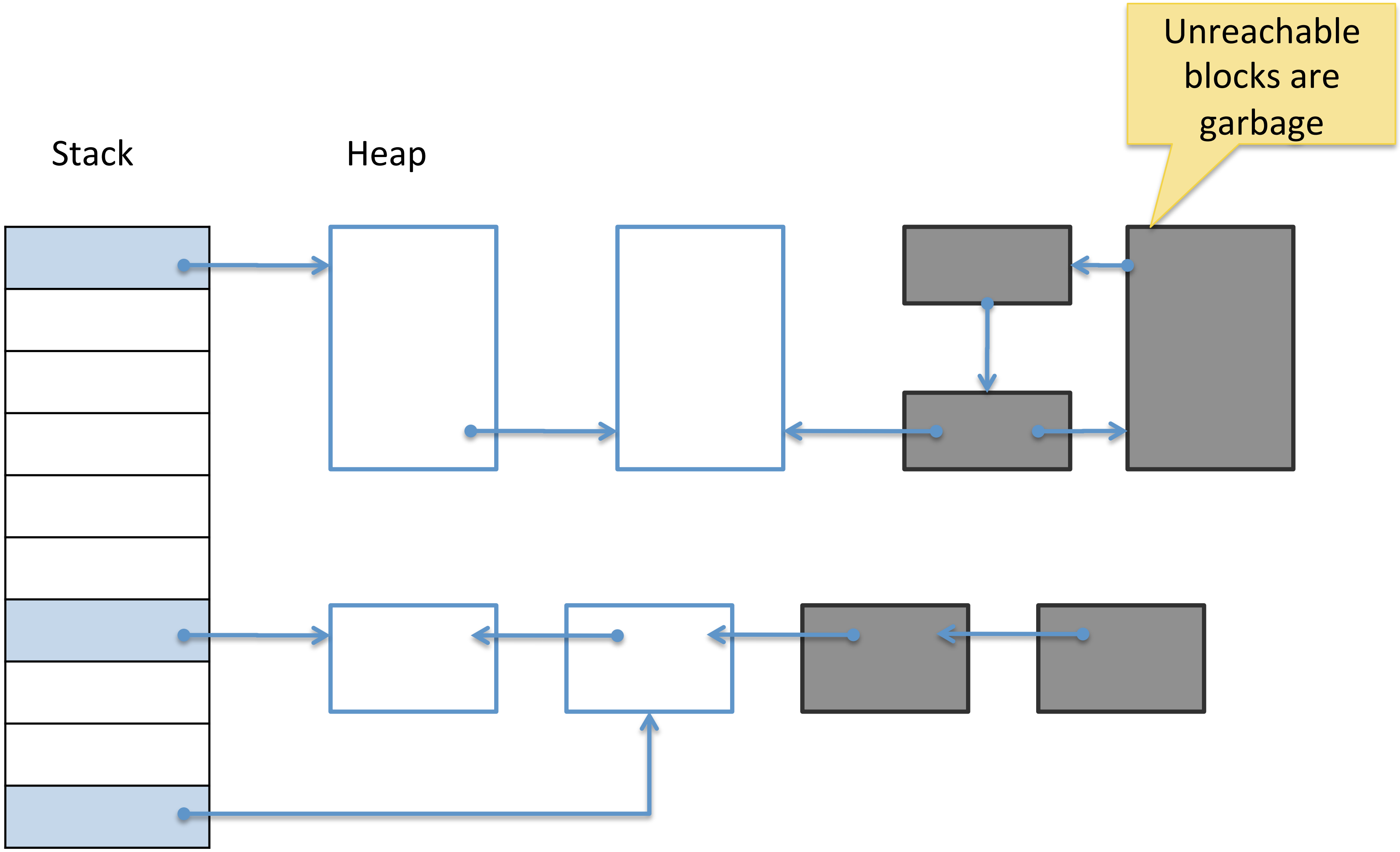


Memory Use & Reachability

- 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).

Results of Marking Graph



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")

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"
  - 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

Threads & Synchronization

Avoid Race Conditions!

(see Multithreaded.java)

Threads

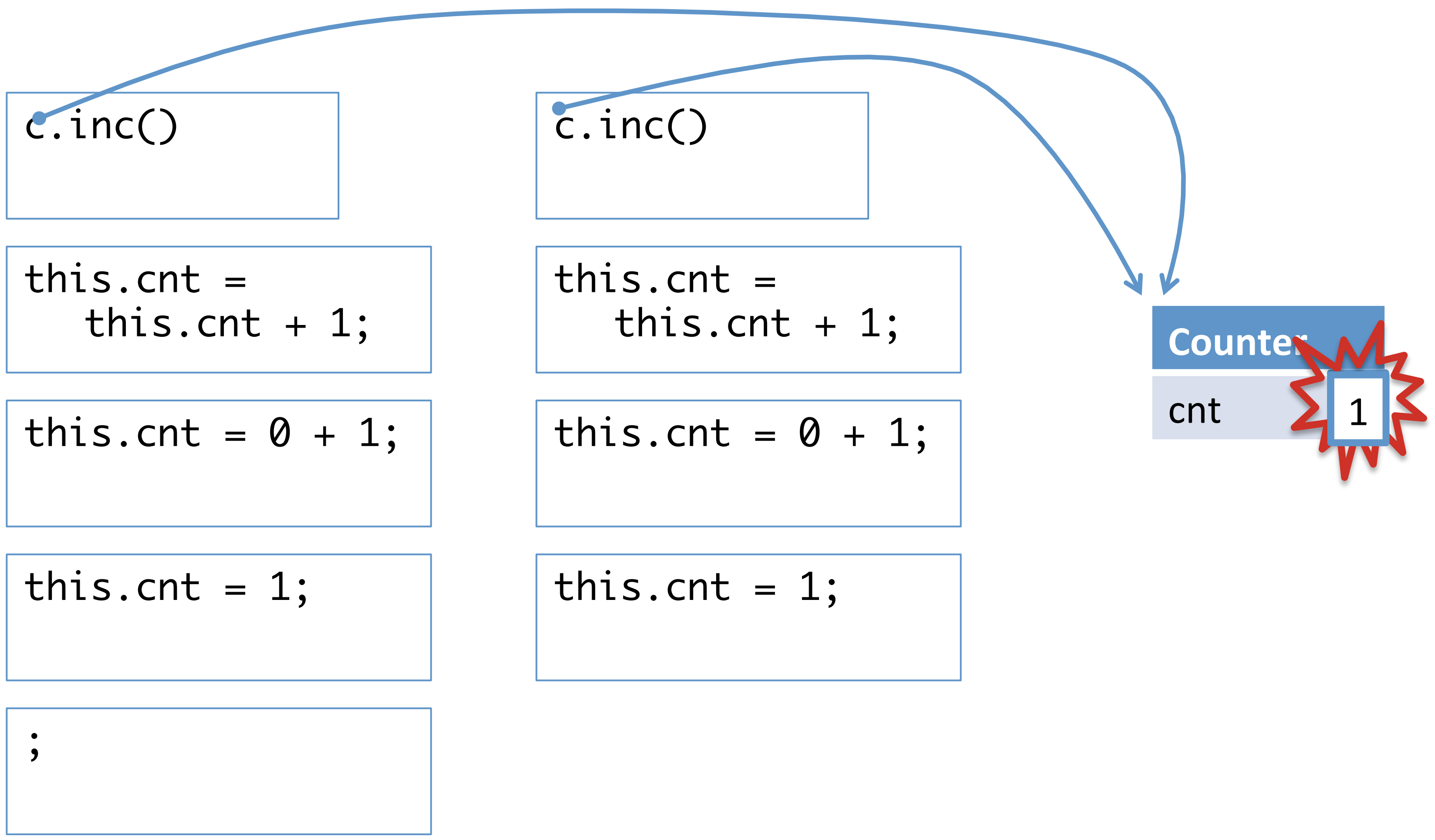
- Java programs can be *multithreaded*
  - more than one "thread" of control operating simultaneously
- A `Thread` object can be created from any class that implements the `Runnable` interface
  - `start`: launch the thread
  - `join`: wait for the thread to finish
- Abstract Stack Machine:
  - Each 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 while another thread is writing?
  - What if more than one thread tries to write different values at the same time?

Data Races



```
c.inc()
```

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

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

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

```
;
```

```
c.inc()
```

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

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

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

Synchronization

- Java provides the `synchronized` keyword
  - only one thread at a time can be 'active' in a synchronized method
  - careful use can rule-out races
  - tradeoff: less concurrency means worse performance
- 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
  - 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