cs2220: Engineering Software

Class 24: Garbage Collection

Fall 2010
UVa
David Evans

Menu

Memory review: Stack and Heap

Garbage Collection

- Mark and Sweep
- Stop and Copy
- Reference Counting

Java's Garbage Collector

Exam 2

Out Thursday, due next Tuesday

Coverage: anything in the class up to last lecture

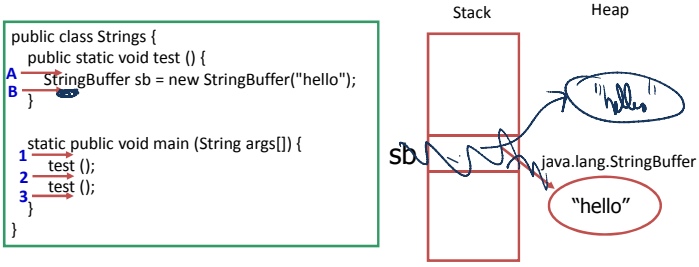
Main Topics

- **Type Hierarchy**: Subtyping, Inheritance, Dynamic Dispatch, behavioral subtyping rules, substitution principle
- **Concurrency abstraction**: multi-threading, race conditions, deadlocks
- **Java Security:** bytecode verification, code safety, policy enforcement

You will have 5 days for Exam 2, but it is designed to be short enough that you should still have plenty time to work on your projects while Exam 2 is out.

Stack and Heap Review

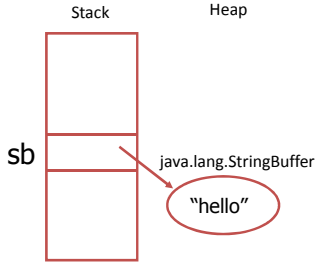
```
public class Strings {
  public static void test () {
A   StringBuffer sb = new StringBuffer("hello");
B }

  static public void main (String args[]) {
1   test ();
2   test ();
3   }
}
```



When do the stack and heap look like this?

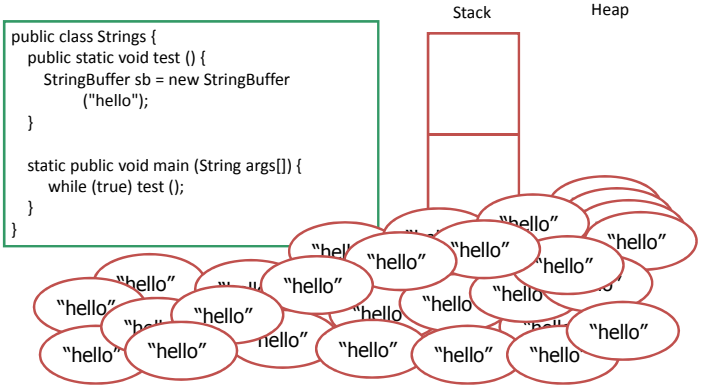
Stack and Heap Review

```
public class Strings {
  public static void test () {
    StringBuffer sb = new StringBuffer
          ("hello");
B }

  static public void main (String args[]) {
2   test ();
    test ();
  }
}
```



Garbage Heap

```
public class Strings {
  public static void test () {
    StringBuffer sb = new StringBuffer
          ("hello");
  }

  static public void main (String args[]) {
    while (true) test ();
  }
}
```



Explicit Memory Management

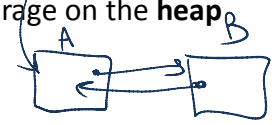
```
public class Strings {
   public static void test () {
      StringBuffer sb =
         new StringBuffer ("hello");
      free (sb);
   }

   static public void main (String args[]) {
      while (true) test ();
   }
}
```

C/C++: programmer uses **free (*pointer*)** to indicate that the storage pointer points to should be reclaimed.

Very painful!
Missing free: memory leak
Dangling references: to free'd objects

Garbage Collection

System needs to **reclaim** storage on the **heap** used by **garbage** objects



How can it identify garbage objects?

Stack + class globals ← non garbage

How come we don't need to garbage collect the stack?

Mark and Sweep

Mark and Sweep

John McCarthy, 1960 (first LISP implementation)

Start with a set of **root references**

Mark every object you can reach from those references

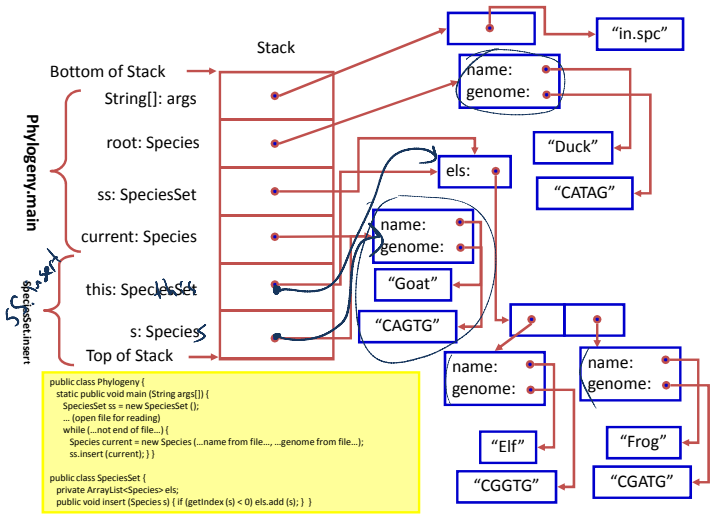
Sweep up the unmarked objects

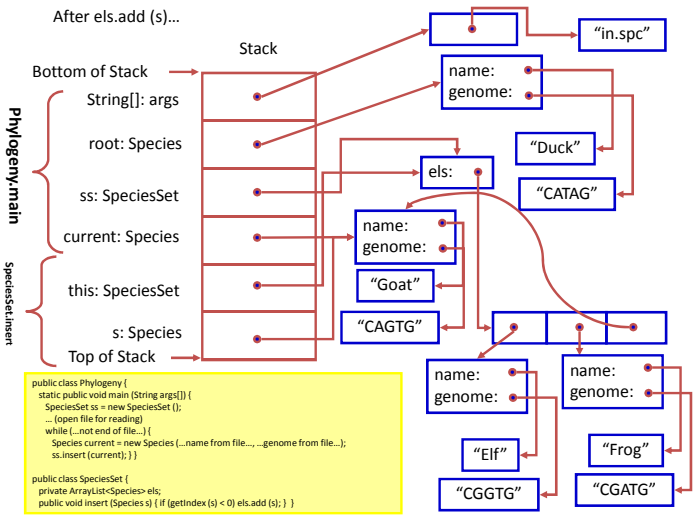
In a Java execution, what are the root references?

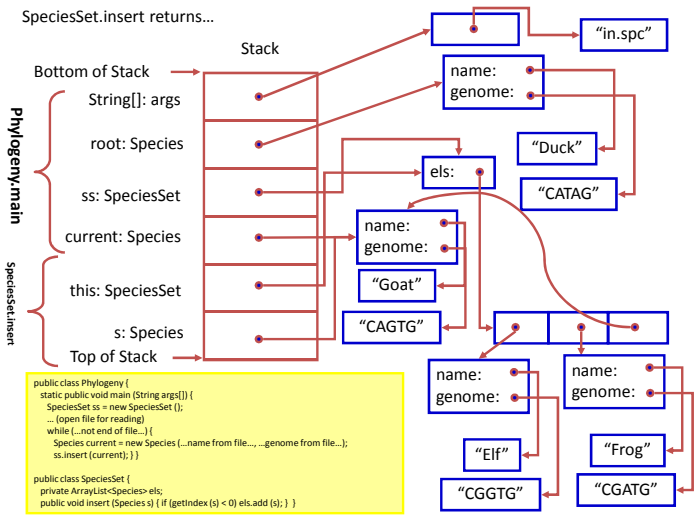
References on the stack.

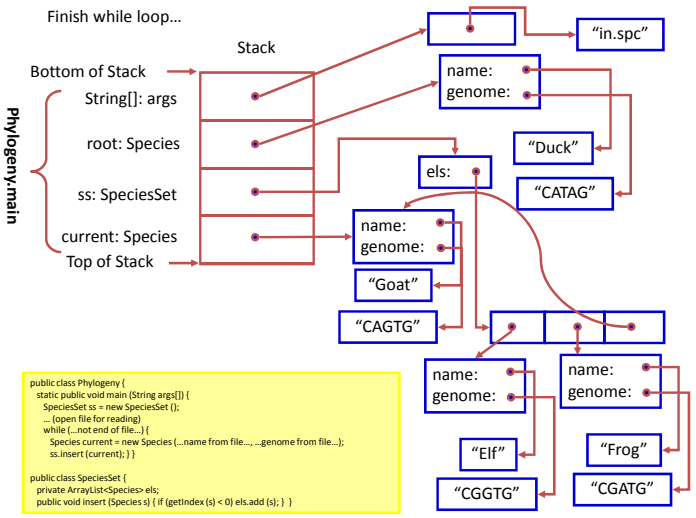
```
public class Phylogeny {
  static public void main (String args[]) {
    SpeciesSet ss = new SpeciesSet ();
    … (open file for reading)
    while (…not end of file…) {
      Species current = new Species (…name from file…,
                                     …genome from file…);
      ss.insert (current);
    }
  }

public class SpeciesSet {
  private ArrayList<Species> els;
  public void insert (Species s) {
    if (getIndex (s) < 0) els.add (s);
  }
}
```

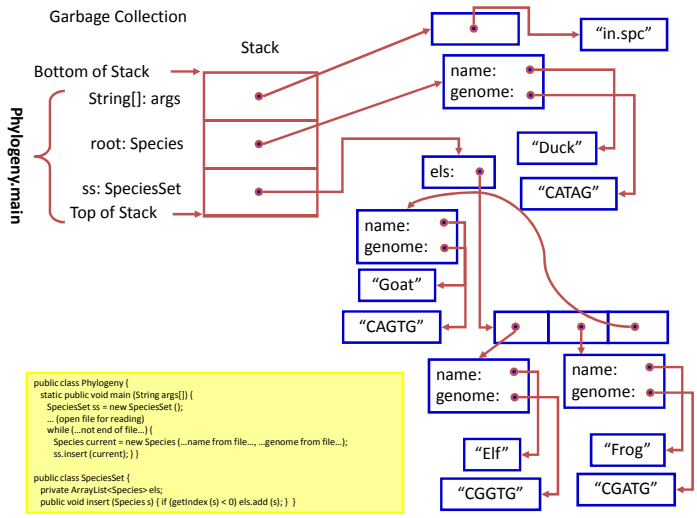
this.











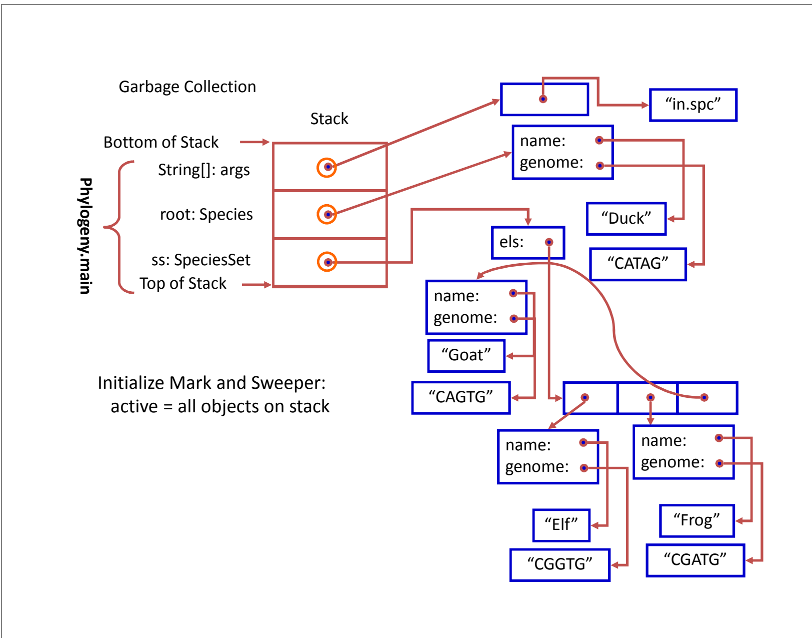
Mark and Sweep Algorithm

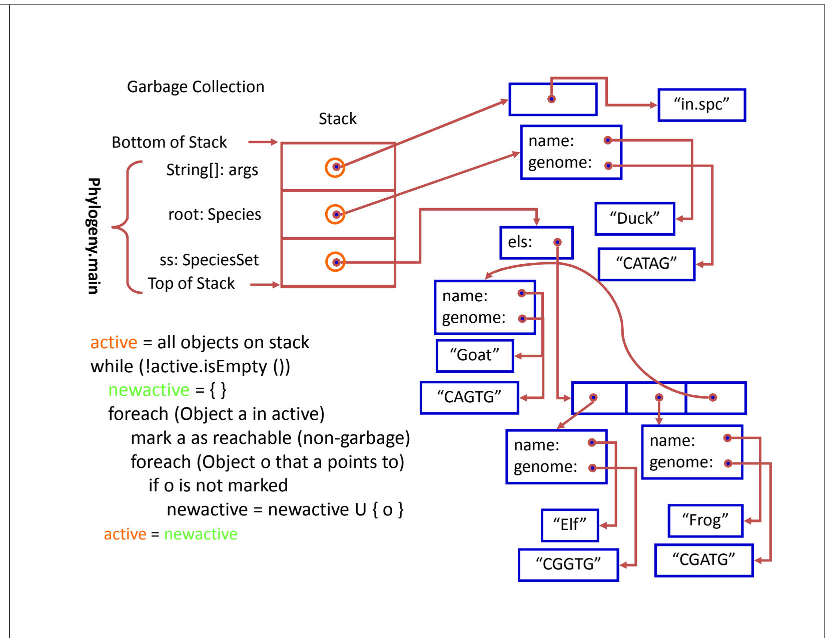
Set(Object) active = root references (collect from stack & globals)

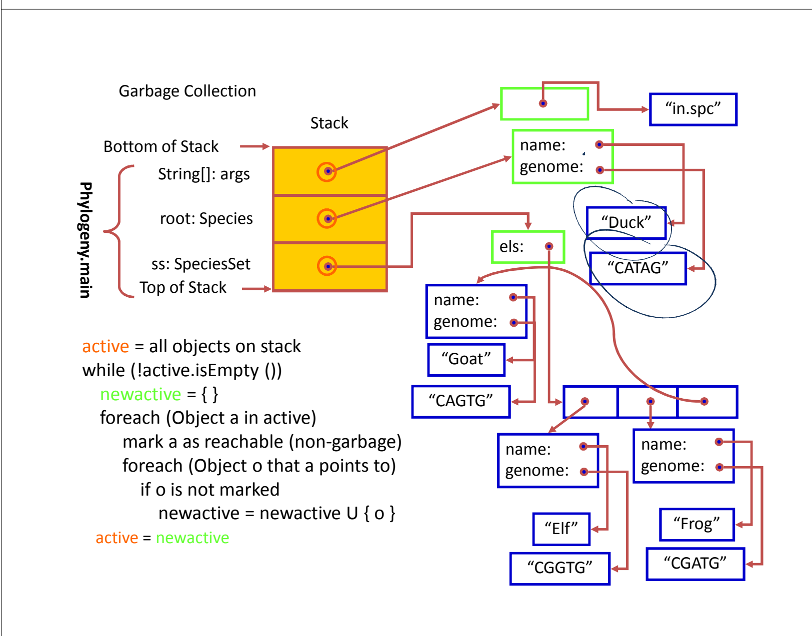
while (!active.isEmpty()) {
  Set(Object) newactive = {};
  foreach (Object a: active) {
    mark a
    foreach (Object o: reachable from a) {
      if o is not marked {
        ~~mark o~~
        newactive.add(o); } }
  active = newactive;

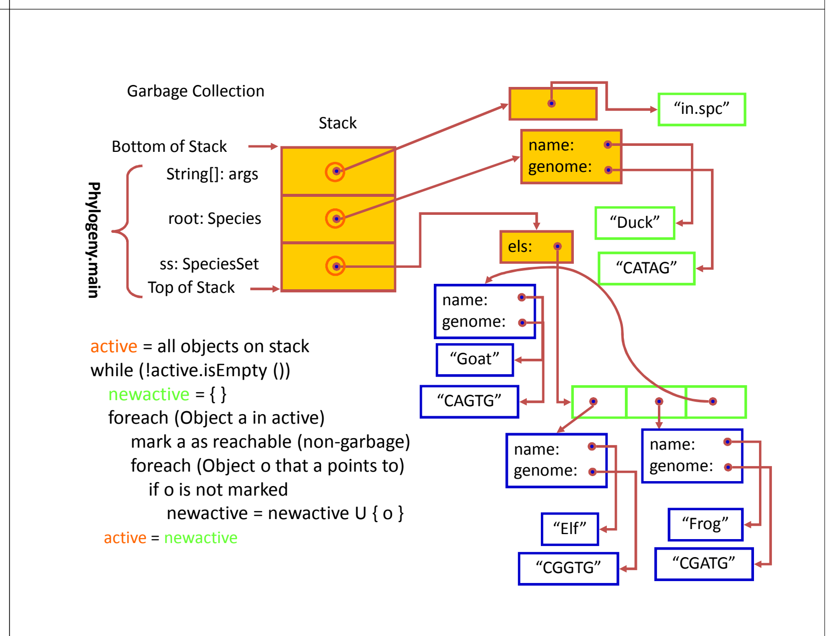
Mark and Sweep Algorithm

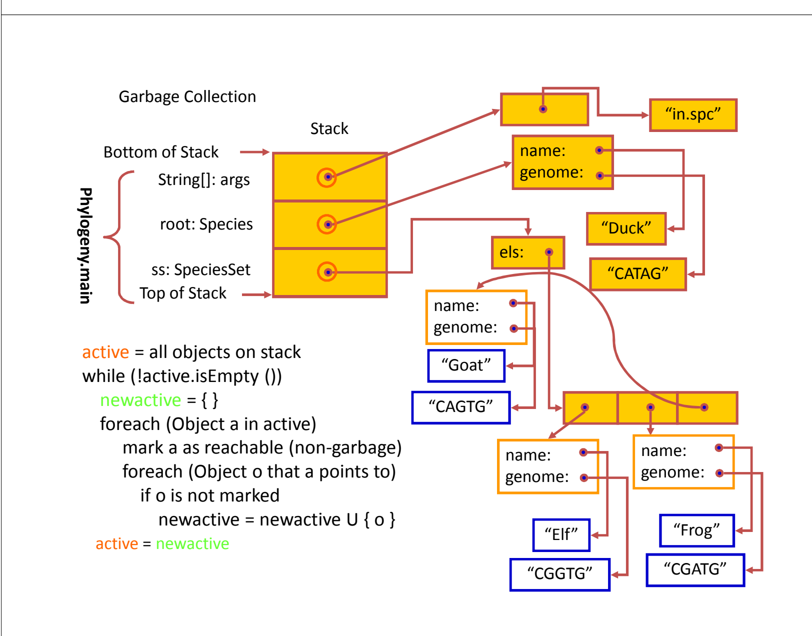
active = all objects on stack
while (!active.isEmpty ())
  newactive = { }
  foreach (Object a in active)
    mark a as reachable (non-garbage)
    foreach (Object o that a points to)
      if o is not marked
        newactive = newactive U { o }
  active = newactive

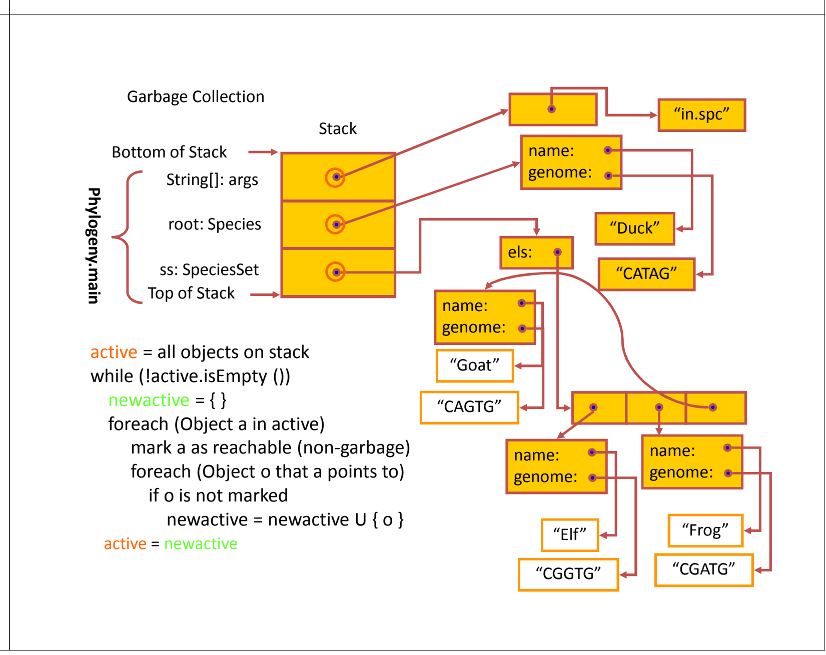

Garbage Collection
Stack
Bottom of Stack
String[]: args
root: Species
ss: SpeciesSet
Top of Stack
Phylogeny.main
"in.spc"
name:
genome:
"Duck"
"CATAG"
els:
name:
genome:
"Goat"
"CAGTG"
name:
genome:
name:
genome:
"Elf"
"CGGTG"
"Frog"
"CGATG"
Initialize Mark and Sweeper:
active = all objects on stack

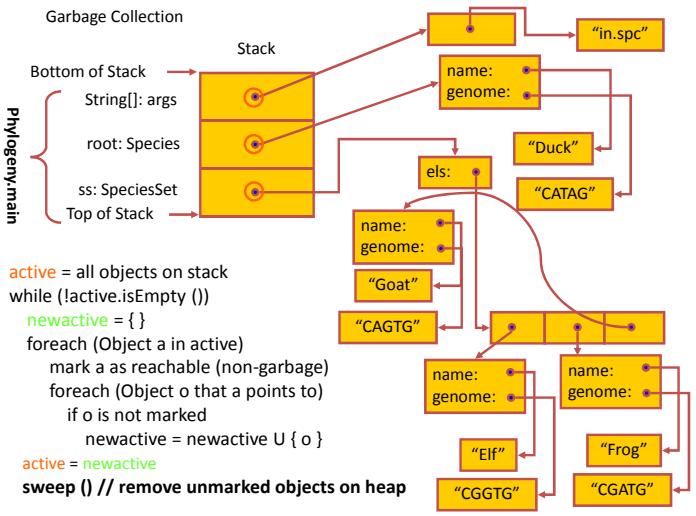

Garbage Collection
Stack
Bottom of Stack
String[]: args
root: Species
ss: SpeciesSet
Top of Stack
Phylogeny.main
"in.spc"
name:
genome:
"Duck"
"CATAG"
els:
name:
genome:
"Goat"
"CAGTG"
name:
genome:
name:
genome:
"Elf"
"CGGTG"
"Frog"
"CGATG"
active = all objects on stack
while (!active.isEmpty ())
newactive = { }
foreach (Object a in active)
mark a as reachable (non-garbage)
foreach (Object o that a points to)
if o is not marked
newactive = newactive U { o }
active = newactive

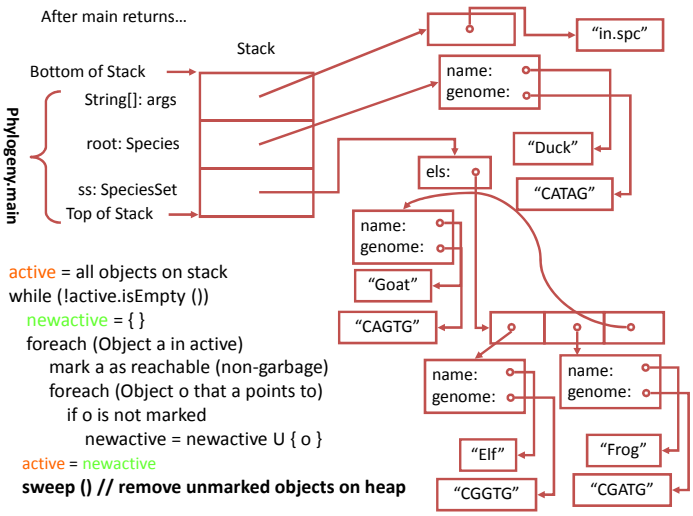

Garbage Collection
Stack
Bottom of Stack
String[]: args
root: Species
ss: SpeciesSet
Top of Stack
Phylogeny.main
"in.spc"
name:
genome:
"Duck"
"CATAG"
els:
name:
genome:
"Goat"
"CAGTG"
name:
genome:
name:
genome:
"Elf"
"CGGTG"
"Frog"
"CGATG"
active = all objects on stack
while (!active.isEmpty ())
newactive = { }
foreach (Object a in active)
mark a as reachable (non-garbage)
foreach (Object o that a points to)
if o is not marked
newactive = newactive U { o }
active = newactive

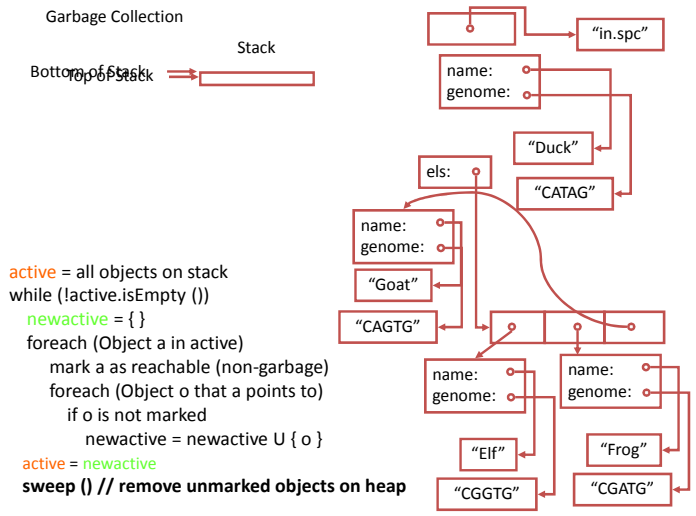

Garbage Collection
Stack
Bottom of Stack
String[]: args
root: Species
ss: SpeciesSet
Top of Stack
Phylogeny.main
"in.spc"
name:
genome:
"Duck"
"CATAG"
els:
name:
genome:
"Goat"
"CAGTG"
name:
genome:
name:
genome:
"Elf"
"CGGTG"
"Frog"
"CGATG"
active = all objects on stack
while (!active.isEmpty ())
newactive = { }
foreach (Object a in active)
mark a as reachable (non-garbage)
foreach (Object o that a points to)
if o is not marked
newactive = newactive U { o }
active = newactive


Garbage Collection
Stack
Bottom of Stack
String[]: args
root: Species
ss: SpeciesSet
Top of Stack
Phylogeny.main
"in.spc"
name:
genome:
"Duck"
"CATAG"
els:
name:
genome:
"Goat"
"CAGTG"
name:
genome:
name:
genome:
"Elf"
"CGGTG"
"Frog"
"CGATG"
active = all objects on stack
while (!active.isEmpty ())
newactive = { }
foreach (Object a in active)
mark a as reachable (non-garbage)
foreach (Object o that a points to)
if o is not marked
newactive = newactive U { o }
active = newactive


Garbage Collection
Stack
Bottom of Stack
String[]: args
root: Species
ss: SpeciesSet
Top of Stack
Phylogeny.main
"in.spc"
name:
genome:
"Duck"
"CATAG"
els:
name:
genome:
"Goat"
"CAGTG"
name:
genome:
name:
genome:
"Elf"
"CGGTG"
"Frog"
"CGATG"
active = all objects on stack
while (!active.isEmpty ())
newactive = { }
foreach (Object a in active)
mark a as reachable (non-garbage)
foreach (Object o that a points to)
if o is not marked
newactive = newactive U { o }
active = newactive







Problems with Mark and Sweep

Fragmentation: free space and alive objects will be mixed

- Harder to allocate space for new objects
- Poor locality means bad memory performance
  - Caches make it quick to load nearby memory

Multiple Threads

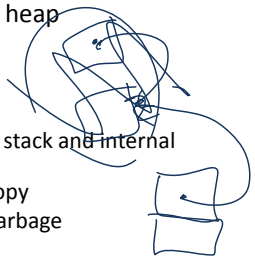
One stack per thread, one heap shared by all threads

All threads must stop for garbage collection

Stop and Copy

Stop execution
Identify all reachable objects (as in Mark and Sweep)
Copy all reachable objects to a new memory area
After copying, **reclaim** the whole old heap

- Solves fragmentation problem
- Disadvantages:
  - More complicated: need to change stack and internal object pointers to new heap
  - Need to save enough memory to copy
  - Expensive if most objects are not garbage



Generational Collectors

Observation:

- Most objects are short-lived
  - Temporary objects that get garbage collected right away
- Other objects are long-lived
  - Data that lives for the duration of execution

Separate storage into **regions**

Short term: collect frequently

Long term: collect infrequently

Stop and copy, but move copies into longer-lived areas

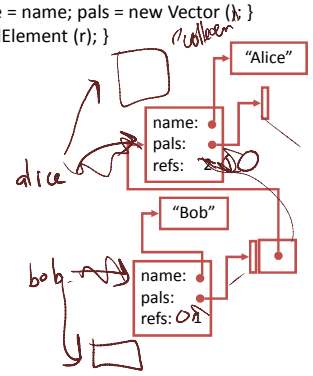
Reference Counting

What if each object kept track of the number of references to it?

If the object has zero references, it is garbage!

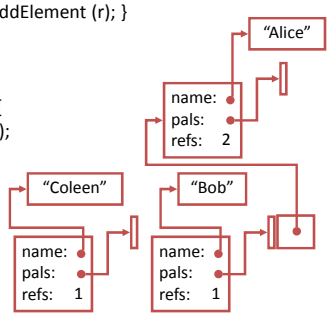
Reference Counting

```
class Recycle {
   private String name; private Vector pals;
   public Recycle (String name) { this.name = name; pals = new Vector (); }
   public void addPal (Recycle r) {  pals.addElement (r); }
}

public class Garbage {
   static public void main (String args[]) {
      Recycle alice = new Recycle ("alice");
      Recycle bob = new Recycle ("bob");
      bob.addPal (alice);
      alice = new Recycle ("coleen");
      bob = new Recycle ("dave");
   }
}
```



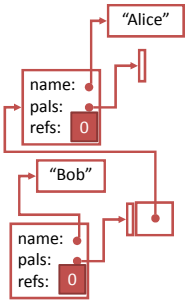
Reference Counting

```
class Recycle {
   private String name; private Vector pals;
   public Recycle (String name) { this.name = name; pals = new Vector (); }
   public void addPal (Recycle r) {  pals.addElement (r); }
}

public class Garbage {
   static public void main (String args[]) {
      Recycle alice = new Recycle ("alice");
      Recycle bob = new Recycle ("bob");
      bob.addPal (alice);
      alice = new Recycle ("coleen");
      bob = new Recycle ("dave");
   }
}
```



Reference Counting

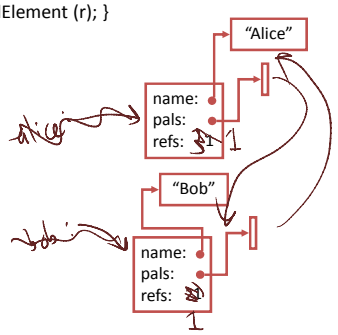
```
class Recycle {
   private String name; private Vector pals;
   public Recycle (String name) { this.name = name; pals = new Vector (); }
   public void addPal (Recycle r) {  pals.addElement (r); }
}

public class Garbage {
   static public void main (String args[]) {
      Recycle alice = new Recycle ("alice");
      Recycle bob = new Recycle ("bob");
      bob.addPal (alice);
      alice = new Recycle ("coleen");
      bob = new Recycle ("dave");
   }
}
```



Can reference counting ever fail to reclaim unreachable storage?

Circular References

```
class Recycle {
   private String name; private Vector pals;
   public Recycle (String name) { this.name = name; pals = new Vector (); }
   public void addPal (Recycle r) {  pals.addElement (r); }
}

public class Garbage {
   static public void main (String args[]) {
      Recycle alice = new Recycle ("alice");
      Recycle bob = new Recycle ("bob");
      bob.addPal (alice);
      alice.addPal (bob);
      alice = null;
      bob = null;
   }
}
```



Reference Counting Summary

Advantages

- Can clean up garbage right away when the last reference is lost
- No need to stop other threads!

Disadvantages

- Need to store and maintain reference count
- Some garbage is left to fester (circular references)
- Memory fragmentation

Java's Garbage Collector

Mark and Sweep collector

Generational

Can call garbage collector directly: **System.gc ()**

but, this should hardly ever be done (except for "fun")

Python's Garbage Collector
Reference counting:
To quickly reclaim most storage
Mark and sweep collector (optional, but on by default):
To collect circular references

java.lang.Object.finalize()

protected void **finalize**() throws Throwable
Called by the garbage collector on an object when garbage collection determines that there are no more references to the object. A subclass overrides the finalize method to dispose of system resources or to perform other cleanup. The general contract of finalize is that it is invoked if and when the Java™ virtual machine has determined that there is no longer any means by which this object can be accessed by any thread that has not yet died, except as a result of an ac
finalized. The finalize method r
usual purpose of finalize, howe
example, the finalize method fo
transactions to break the conne
The finalize method of class Ob
override this definition.
The Java programming languag
object. It is guaranteed, however, that the thread that invokes finalize will not be holding any user-visible synchronization locks when finalize is invoked. **If an un d, the exception is ignored and finalization of that object te**
After the finalize method has been invoked for an obj al machine has again determined that there is no longer any mea hread that has not yet died, including possible actions by other c at which point the object may be discarded.
The finalize method is never invoked more than once b
Any exception thrown by the finalize method causes th erwise ignored.

Summary:
finalize is called when garbage collector reclaims object
no guarantee when it will be called
after finalizer, JVM has to check you didn't do something stupid
its protected because subclasses need to override it (but no one other than the JVM itself should ever call it!)

You should probably never need to override finalize in your code. Only excuse for using it is if you have objects with unknown lifetimes that have associated (non-memory) resources.

```
class Recycle {
   private String name;
   private ArrayList<Recycle> pals;
   public Recycle (String name) {
       this.name = name; pals = new ArrayList<Recycle> (); }
   public void addPal (Recycle r) {  pals.add (r); }
   protected void finalize () { System.err.println (name + " is garbage!"); }
}

public class Garbage {
  static public void main (String args[]) {
      Recycle alice = new Recycle ("alice");
      Recycle bob = new Recycle ("bob");
      bob.addPal (alice);
      alice = new Recycle ("coleen");
      System.out.println("First collection:");
      System.gc ();
      bob = new Recycle ("dave");
      System.out.println("Second collection:");
      System.gc ();
   }
}
```

```
> java Garbage
First collection:
Second collection:
alice is garbage!
bob is garbage!
```

```
class Recycle {
   private String name;
   private ArrayList<Recycle> pals;
   public Recycle (String name) { this.name = name; pals = new ArrayList<Recycle> (); }
   public void addPal (Recycle r) {  pals.add (r); }
   protected void finalize () { System.err.println (name + " is garbage!"); }
}

public class Garbage {
  static public void main (String args[]) {
      System.err.println(Runtime.getRuntime().freeMemory() + " bytes free!");
      Recycle alice = new Recycle ("alice");
      Recycle bob = new Recycle ("bob");
      bob.addPal (alice);
      alice = new Recycle ("coleen");
      System.err.println("First collection:");
      System.err.println(Runtime.getRuntime().freeMemory() + " bytes free!");
      System.gc ();
      System.err.println(Runtime.getRuntime().freeMemory() + " bytes free!");
      bob = new Recycle ("dave");
      System.err.println("Second collection:");
      System.gc ();
      System.err.println(Runtime.getRuntime().freeMemory() + " bytes free!");
   }
}
```

Note running the garbage collector itself uses memory!

```
125431952 bytes free!
First collection:
125431952 bytes free!
125933456 bytes free!
Second collection:
125933216 bytes free!
bob is garbage!
alice is garbage!
```

```
class Recycle {
   private String name;
   private ArrayList<Recycle> pals;
   public Recycle (String name) {
       this.name = name; pals = new ArrayList<Recycle> (); }
   public void addPal (Recycle r) {  pals.add (r); }
   protected void finalize () {
     Garbage.truck = this;
     System.err.println (name + " is garbage!" + this.hashCode()); }
}

public class Garbage {
   static public Recycle truck;
   static public void main (String args[]) {
     printMemory();
     while (true) {
        Recycle alice = new Recycle ("alice");
        printMemory();
        System.gc ();
     }
   }
}
```

Charge

- **In Java:** be happy you have a garbage collector to clean up for you
- **In C/C++:** need to deallocate storage explicitly
  - Why is it hard to write a garbage collector for C?
- **In the real world:** clean up after yourself and others!

Keep working on your projects
Exam 2 out Thursday

Garbage Collectors
(COAX, Seoul, 18 June 2002)