The Java Virtual Machine

Managing Object Creation, Execution and Garbage Collection

Why a Virtual Machine?

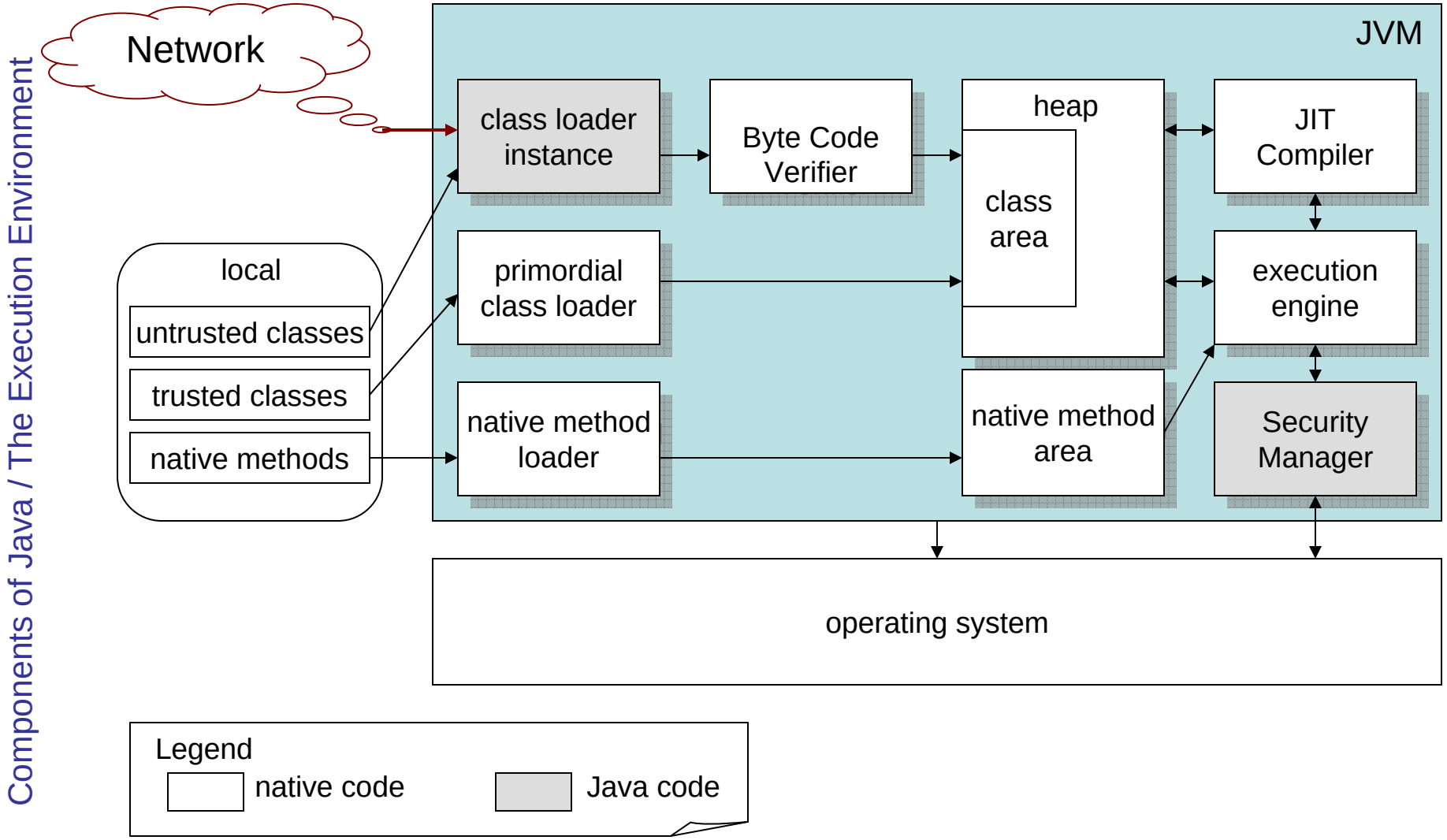
- ✓ **Portability**
  - ❑ Hardware and OS independence
  - ❑ “Write once, run anywhere" - the core value proposition of the Java platform
    - ▪ J.D. says - write once, test everywhere
- ✓ **Security**
  - ❑ Protect users from malicious programs
  - ❑ Isolated from the host operating system and its viruses
- ✓ **Small size**
  - ❑ Mobile code
  - ❑ Distributed Embedded Systems application
- ✓ **Network-Centric Programming**
  - ❑ Sun's motto has always been "The network is the computer.“

JVM Specification ("What")

- ✓ **Bytecodes**
  - A well-defined set of instructions for the virtual processor
- ✓ **Class file format**
  - A binary format to describe a class in a platform independent way.
- ✓ **Verification Algorithm**
  - To identify programs that might be attacks
- ✓ **JVM spec leaves to the implementation**
  - Memory layout of runtime data areas - heap
  - Garbage collection algorithm used
    - Doesn't even actually require GC
  - Optimizations

http://java.sun.com/docs/books/jvms/

The Java Virtual Machine (JVM)



JVM (2)

- ✓ **class loaders**
  - locate and load classes into the JVM
  - primordial class loader
    - loads trusted classes (system classes found on the boot class path)
      - Directories and JAR files listed in the system property **`sun.boot.class.path`**, which by default includes the core runtime classes in **`rt.jar`** and a few other standard JARs
  - Extensions class loader
    - Directories listed in the system property to the lib/ext directory of the JRE
  - Application class loader instances
    - Instances that extend **`java.lang.ClassLoader`**
    - Directories and JARs listed in the system property **`java.class.path`**
    - load untrusted classes from the local file system or from the network and passes them to the class file verifier
    - application developers can implement their own class loaders

```
public class CustomClassLoader extends ClassLoader {

   public CustomClassLoader()
   {
      super(CustomClassLoader.class.getClassLoader());
   }
//not normally needed
//e.g. support run-time class replacement,
}
```

Extending the ClassLoader – an example

The `java.lang.ClassLoader` class is an abstract interface for the loading of classes into the runtime environment
The `java.io.InputStream` class is the basis for loading data into the runtime environment from different sources and in different formats.
Put them together, and form the basis for loading classes from all of the sources accessible from subclasses of `InputStream`.

```
package dcj.util;
import java.lang.*;
import java.net.*;
import java.io.*;
import java.util.Hashtable;

public abstract class StreamClassLoader extends ClassLoader
{
  // Instance variables and default initializations
  Hashtable classCache = new Hashtable();
  InputStream source = null;

  // Constructor
  public StreamClassLoader()
  { }
. . .
}
```

from
Java Distributed Computing
by Jim Farley, O'Reilly

Remaining code elided for the slide

JVM (3)

- **Byte Code Verifier**
  - syntactic analysis
    - all arguments to flow control instructions must cause branches to the start of a valid instruction
    - all references to local variables must be legal
    - all references to the constant pool must be to an entry of appropriate type
    - all opcodes must have the correct number of arguments
    - exception handlers must start at the beginning of a valid instruction
  - data flow analysis
    - attempts to reconstruct the behavior of the code at run time without actually running the code
    - keeps track only of types not the actual values in the stack and in local variables
  - it is theoretically impossible to identify all problems that may occur at run time with static analysis

- **What is true when the Byte Code Verification is complete**
  - There are no operand stack overflows or underflows
  - The types of the parameters of all bytecode instructions are known to always be correct
    - Within the limits of static checking
  - Object field accesses are known to be legal--private, public, or protected

JVM (4)

Code from Component Development for the Java™ Platform, by Stuart Dabbs Halloway, Addison Wesley

✓ native method loader

- native methods are needed to access some of the underlying operating system functions (e.g., graphics and networking features)
- once loaded, native code is stored in the native method area for easy access
- Load native code using the **`System.loadLibrary`** method.
- Put the call to **`loadLibrary`** in a static initializer for a class that declares native methods

```
public class UltimateQuestion {
  static { System.loadLibrary("UltimateQuestion"); }
}
```

JVM (5)

- **the heap**
  - memory used to store objects during execution
  - how objects are stored is implementation specific
- **execution engine**
  - a virtual processor that executes bytecode
  - has virtual registers, stack, etc.
  - performs memory management, thread management, calls to native methods, etc.
- **JIT – Just In Time compiler**
  - translates bytecode into native code on-the-fly
    - works on a method-by-method basis
    - the first time a method is called, it is translated into native code and stored in the class area
    - future calls to the same method run the native code
  - all this happens after the class has been loaded and verified

JVM (6)

✓ Security Manager

- enforces access control at run-time (e.g., prevents applets from reading or writing to the file system, accessing the network, printing, ...)
- application developers can implement their own Security Manager
- or use the policy based SM implementation provided by the JDK

- More on security management in Session 9

Classloading Rules

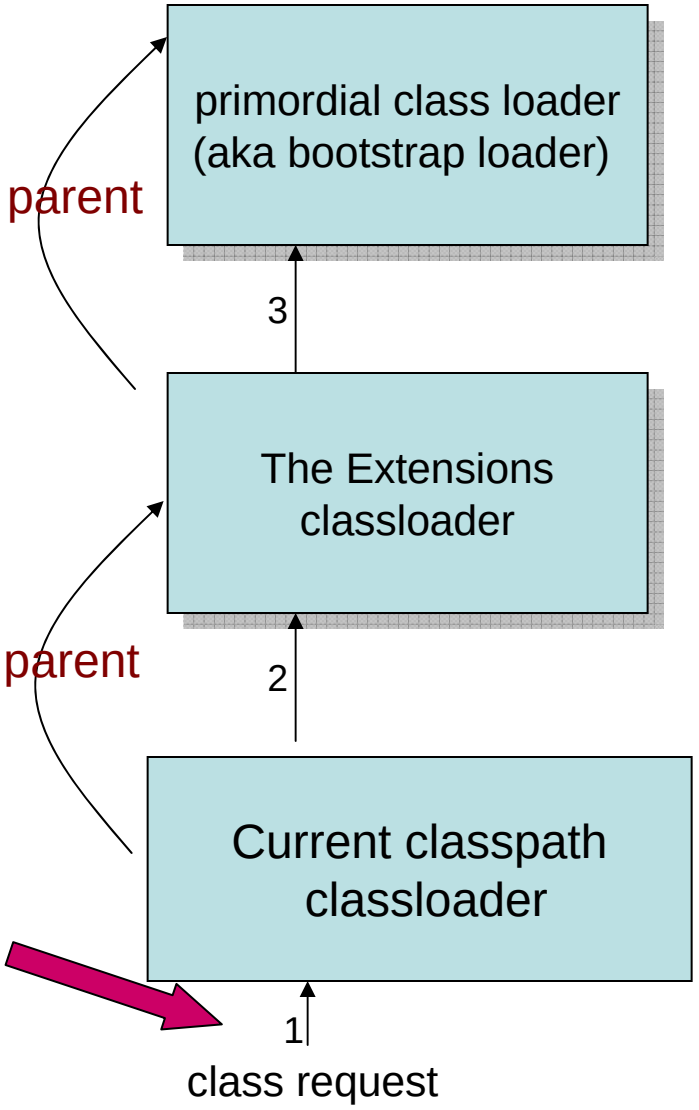
- ✓ **The consistency rule:**
  - ❑ Class loaders never load the same class more than once.
- ✓ **The delegation rule:**
  - ❑ Class loaders always consult a parent class loader before loading a class.
  - ❑ A classloader parent class refers to the delegation hierarchy, not the superclass hierarchy
- ✓ **The visibility rule:**
  - ❑ Classes can only "see" other classes loaded by their class loader's delegation , the recursive set of a class's loader and all its parent loaders.

Rules from
Component Development for the Java™ Platform, by
Stuart Dabbs Halloway, Addison Wesley

Class loading task delegation

1. Explicit class loading – call loadclass()
   - not very flexible and not usually done
2. Implicit class loading – the norm

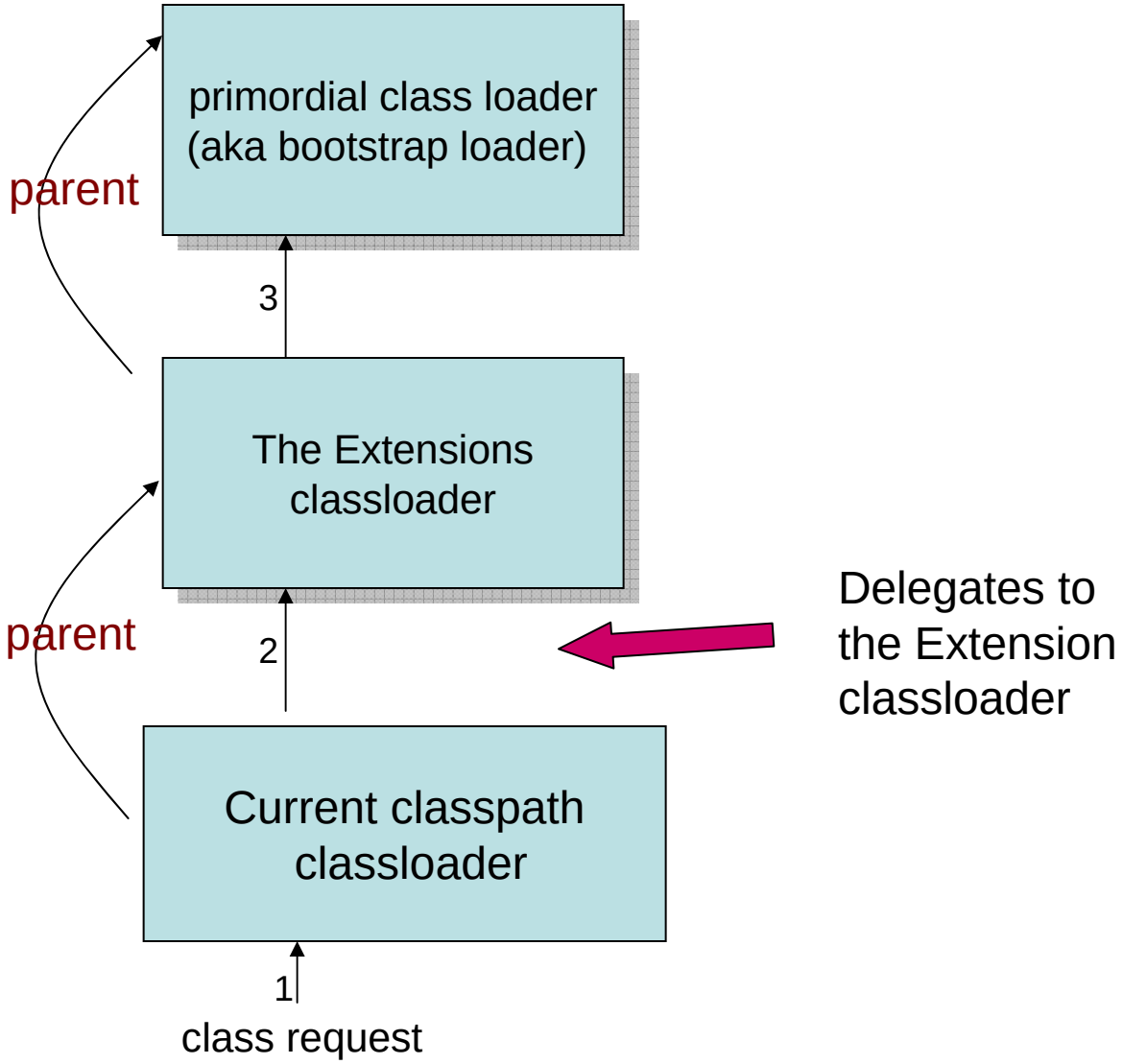
When a class refers to another class (e.g. as the type of a field or parameter), the referent is loaded implicitly, by the same class loader. Classes loaded by two different classloader hierarchies can not communicate.



Class loading task delegation

1. Explicit class loading – call loadclass()
   - not very flexible and not usually done
2. Implicit class loading – the norm

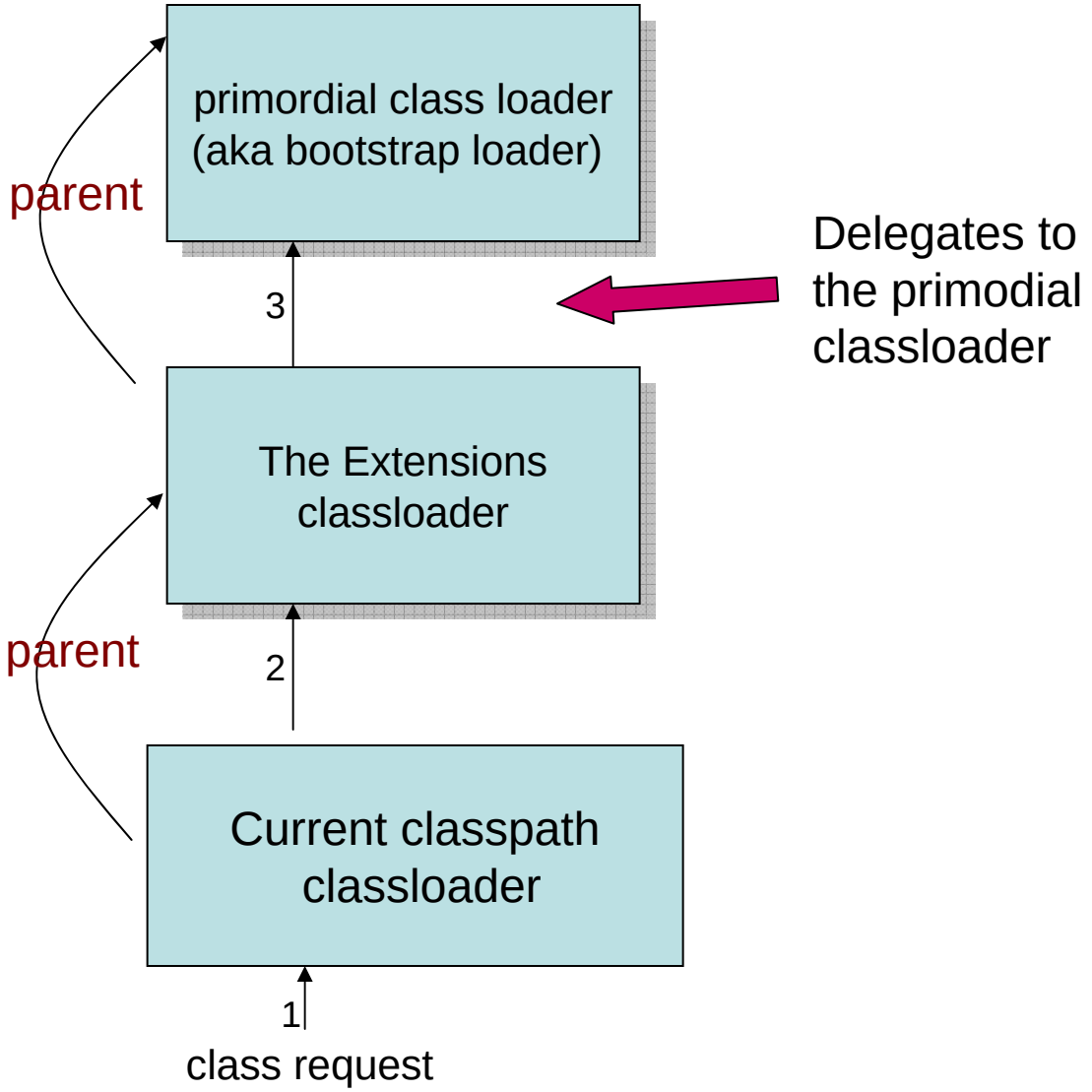
When a class refers to another class (e.g. as the type of a field or parameter), the referent is loaded implicitly, by the same class loader. Classes loaded by two different classloader hierarchies can not communicate.



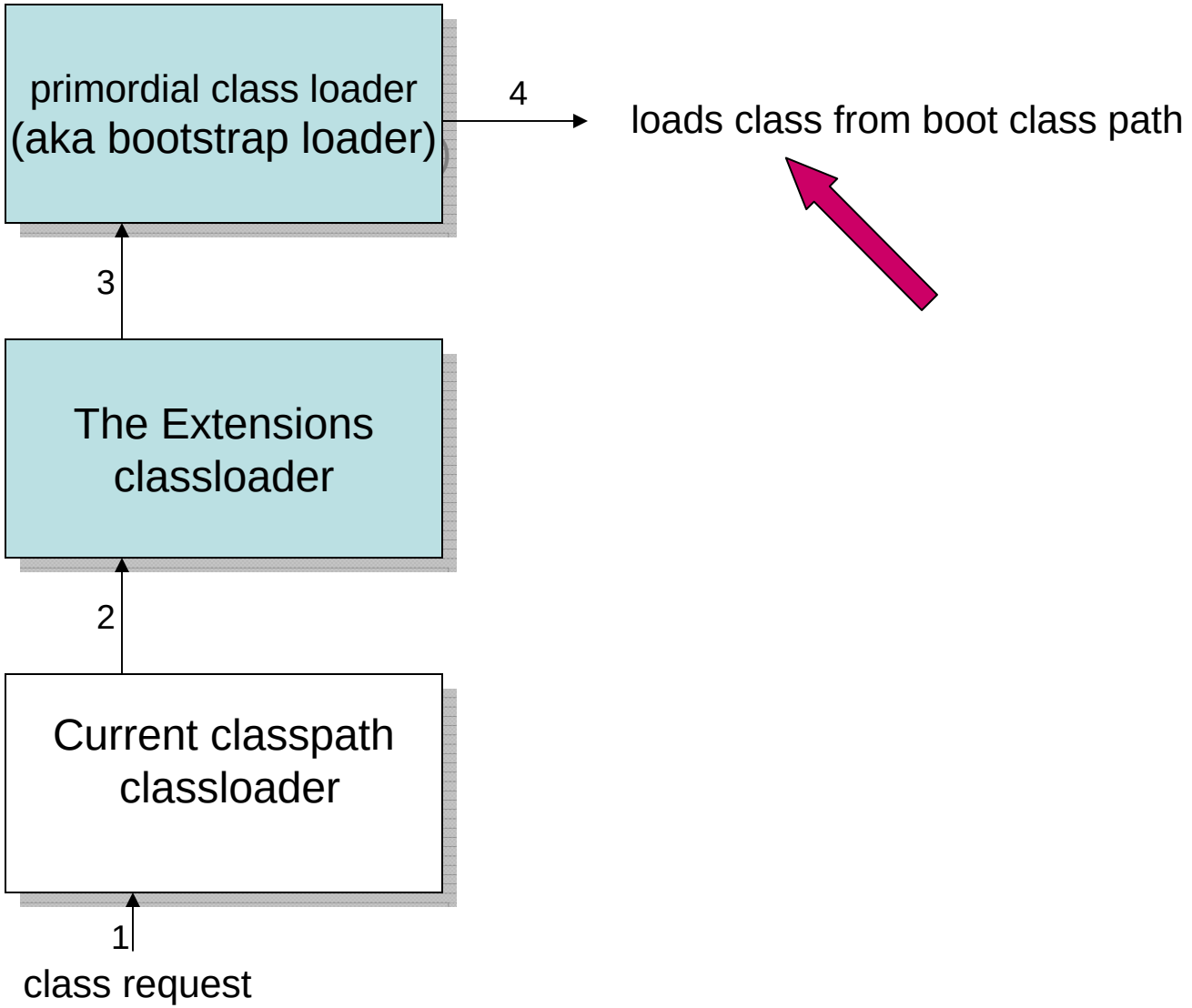
Class loading task delegation

1. Explicit class loading – call loadclass()
   - not very flexible and not usually done
2. Implicit class loading – the norm

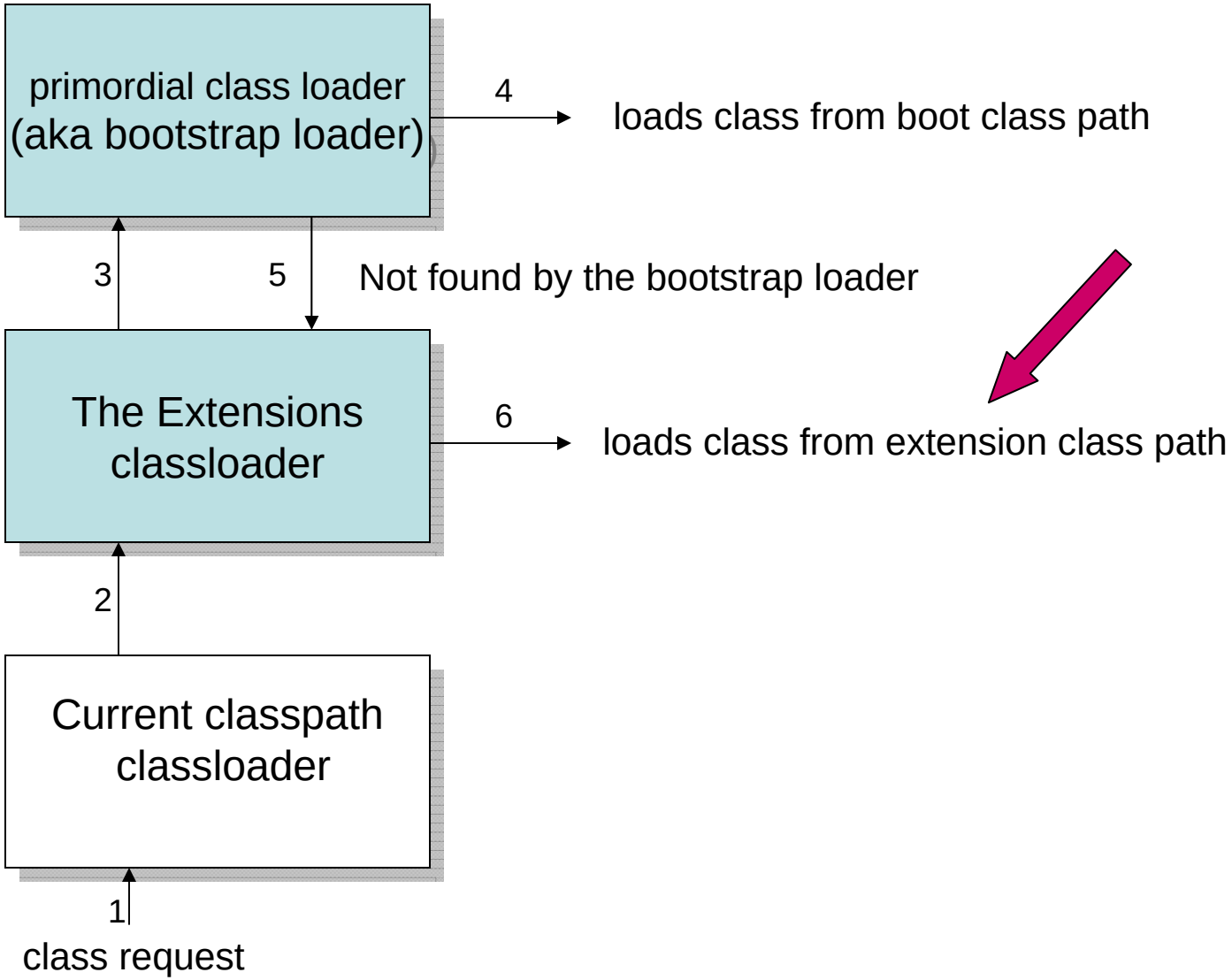
When a class refers to another class (e.g. as the type of a field or parameter), the referent is loaded implicitly, by the same class loader. Classes loaded by two different classloader hierarchies can not communicate.



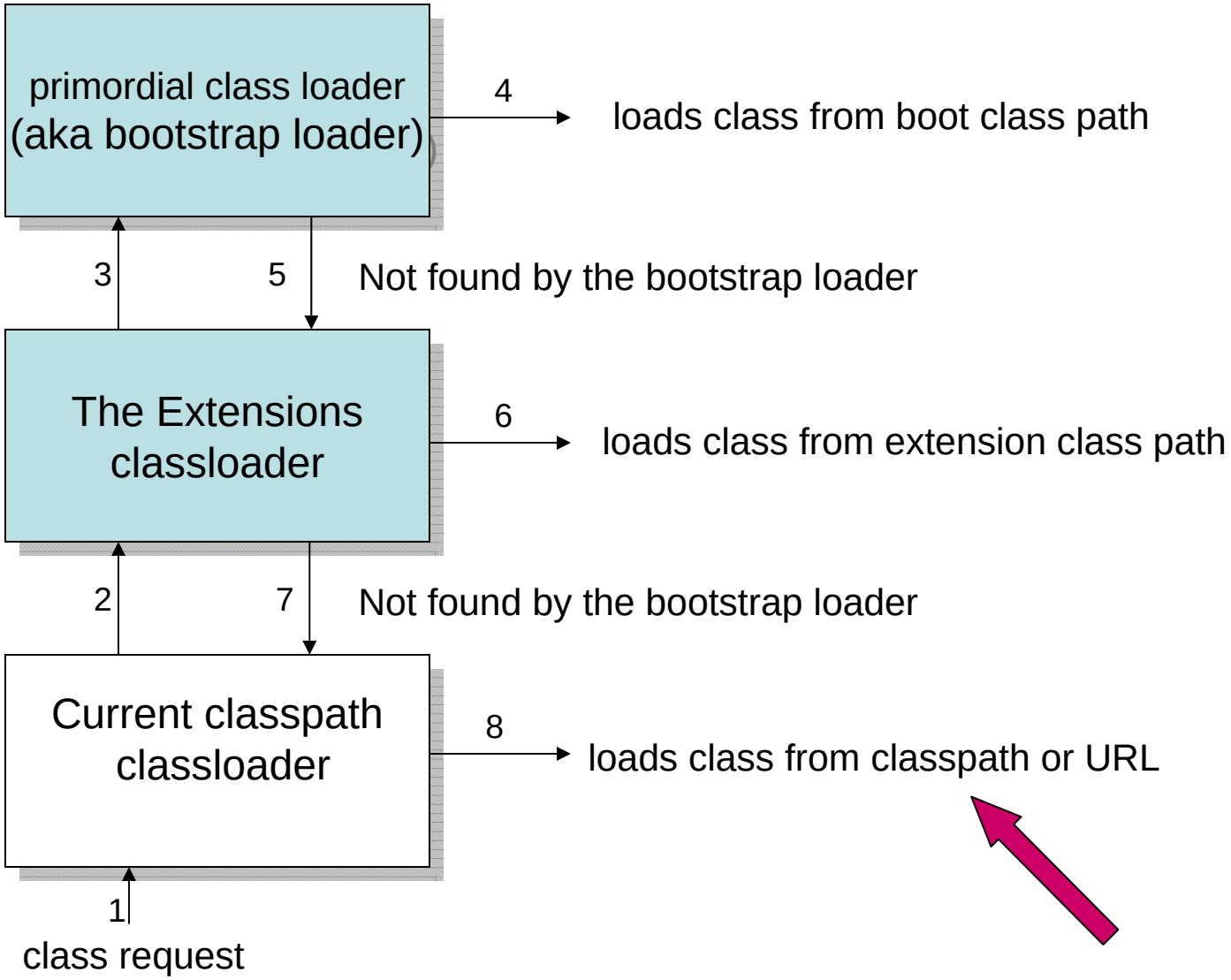
Class loading task delegation



Class loading task delegation



Class loading task delegation



Examining the Byte Code

```
V:\...\Lesson 1\code samples>javap TestGC
Compiled from "TestGC.java"
public class TestGC extends java.lang.Object{
    public TestGC();
    public static void main(java.lang.String[]);
}
```

Basic Information

```
V:\...\Lesson 1\code samples>javap -s -verbose TestGC
Compiled from "TestGC.java"
public class TestGC extends java.lang.Object
  SourceFile: "TestGC.java"
  minor version: 0
  major version: 50
  Constant pool:
const #1 = Method       #25.#39;        //  java/lang/Object."<init>":()V
const #2 = Field        #40.#41;        //  java/lang/System.out:Ljava/io/PrintS
tream;
const #3 = String       #42;    //  Begin timing...
const #4 = Method       #43.#44;        //  java/io/PrintStream.println:(Ljava/lang/String;)V
const #5 = Method       #40.#45;        //  java/lang/System.currentTimeMillis:()J
const #6 = Field        #24.#46;        //  TestGC.start:J
const #7 = int  50000;
const #8 = class        #47;    //  edu/ucsd/examples/Rock
```

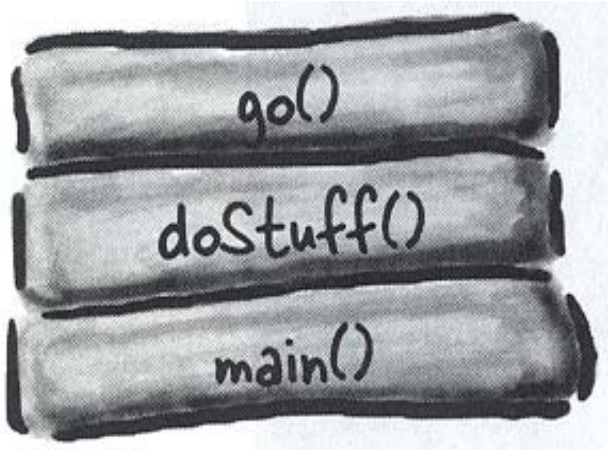
Detailed information

A total of 286 lines of information from this simple class

Executing Java Byte Code

“The Stack” and “The Heap”

- ✓ **What is the program Stack? What is stored there?**
- ✓ **What is the program Heap? What is stored there?**
- ✓ **Why are they important?**



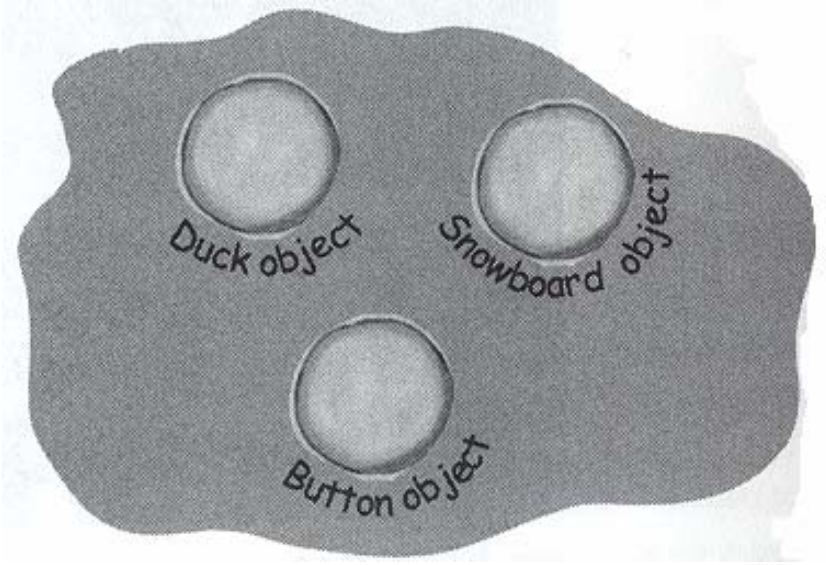


Figure source: *Head First Java*. Sierra & Bates. © 2003 O’Reilly and Associates

Terminology

- **Stack:**
  - A memory area where stack frame data is
    - pushed when a procedure is called
    - and popped when it returns
  - Contains local variables (primitives and object references)
  - The stack frame is pushed onto the stack when another method is called
    - The return address
    - The arguments passed to the method
    - The local variables, including actual parameaters

- **Heap:**
  - A memory area where objects can be created and deleted in any order
  - ‘new’ operators allocate space in the heap for objects
  - The garbage collector deletes objects from the heap

Terminology (cont.)

- **Root Set:**
  - A set of objects that a program always has direct access to
  - E.g. global variables, or variables in the main program (stored on the program stack)

- **A Heap Object (also called a cell or simply object):**
  - An individually allocated piece of data in the heap

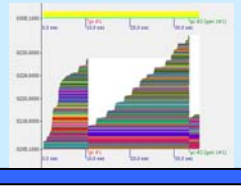
- **Reachable Objects:**
  - Objects that can be reached transitively from the root set objects

Terminology (cont.)

- **Garbage:**
  - Objects that are unreachable from root set objects but are not free either
- **Dangling references:**
  - A reference to an object that was deleted
  - May cause the system to crash (if we are lucky!)
  - May cause more subtle bugs
- **Mutator**
  - The user's program, which mutates or changes the heap and stack data
  - Compare to the 'collector'

Execution Organization

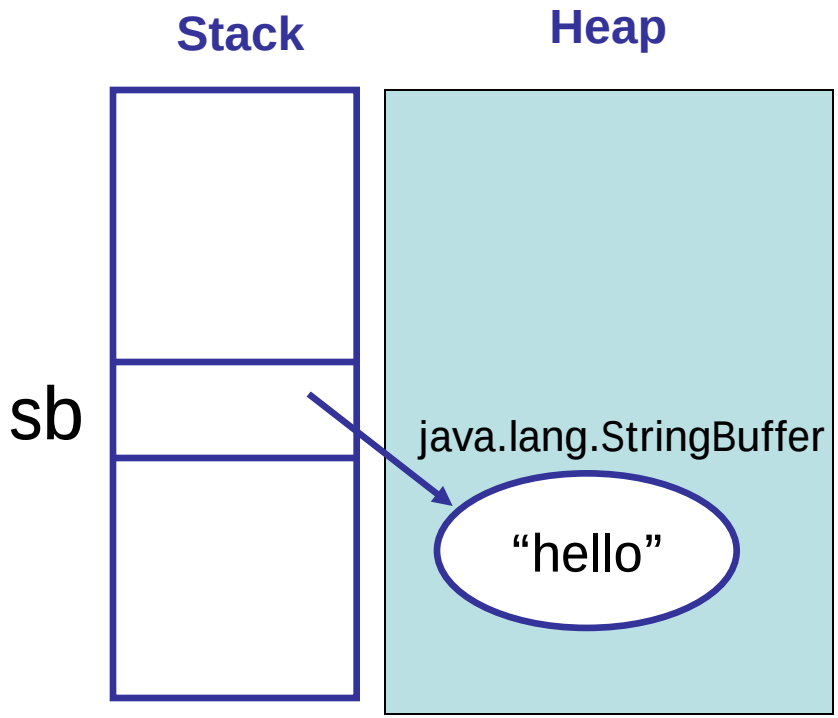
- ✓ **The pc register**
  - One per thread
- ✓ **Stack**
  - One per thread - checked for overflow
- ✓ **Native method stacks**
  - for supporting native methods.
- ✓ **Heap**
  - One per application



Garbage Collection

References on the Stack and Objects in the Heap

```
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   }
}
```

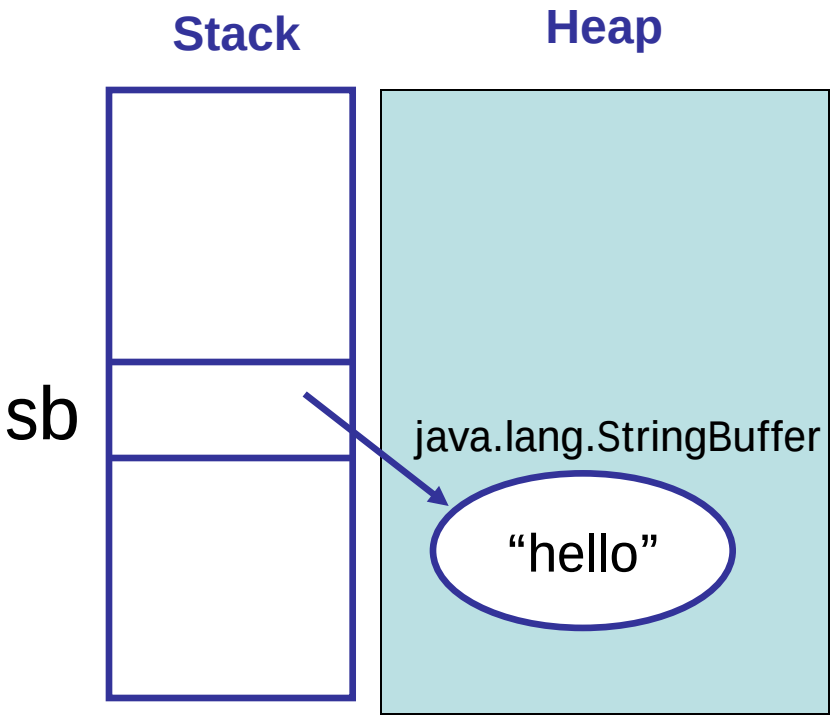


When do the stack and heap look like this?

GC from slides by David Evans http://www.cs.virginia.edu/evans. Used with permission

References on the Stack and Objects in the Heap

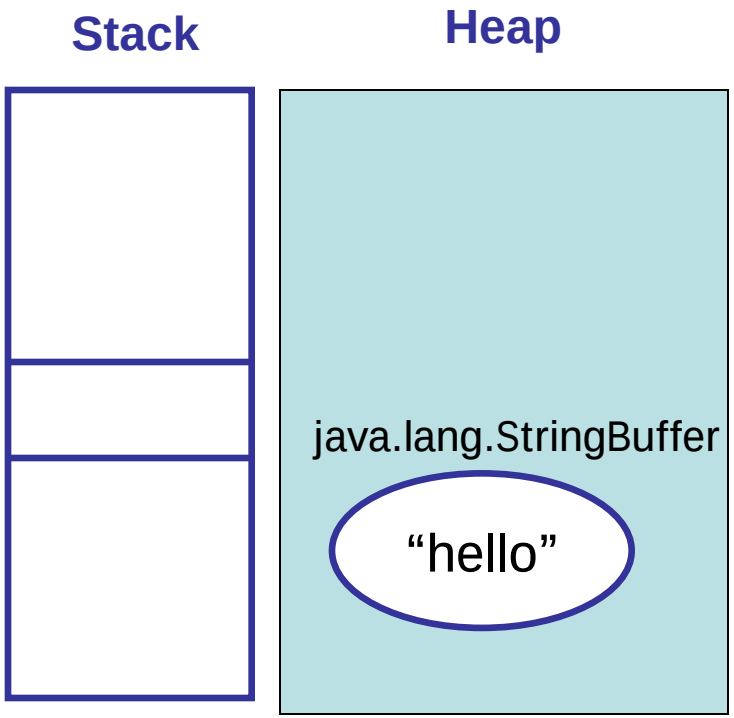
```
public class Strings {
    public static void test () {
        StringBuffer sb = new StringBuffer
            ("hello");
B   }

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



sb is in scope only during the execution of test()
The StringBuilder object only exists after the constructor is executed

References on the Stack and Objects in the Heap

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

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

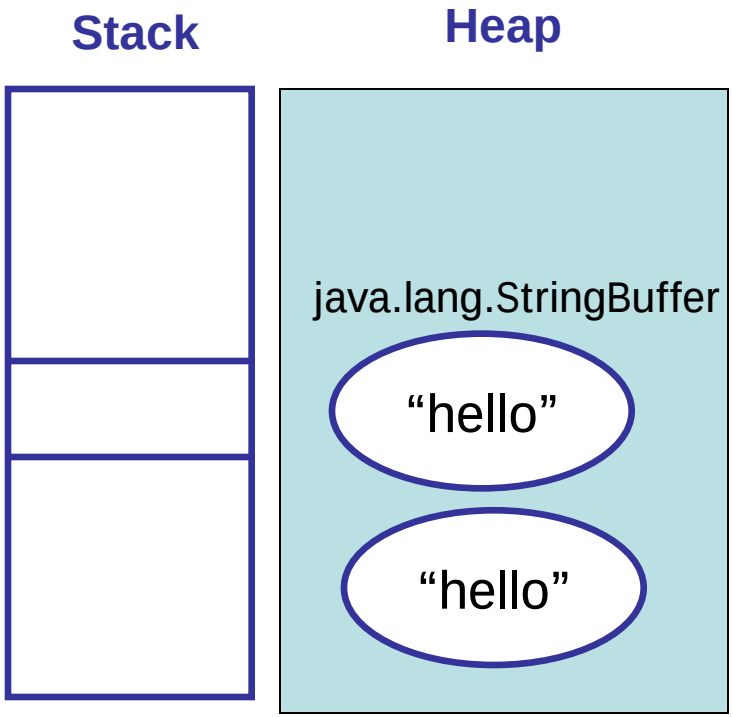


sb is no longer in scope
The StringBuilder object is no longer reachable but it may or may not have been garbage collected

References on the Stack and Objects in the Heap

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

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

3



Eligible for garbage collection does not mean collected

Identifying Garbage

- **How does the System identify garbage? How is that memory reclaimed ?**

- **Direct Algorithms**
  - Each object has an additional field recording the number of objects that point to it
  - An object is considered garbage when zero objects point to it
  - Example: the reference counting algorithm (described shortly)

- **Tracing Algorithms**
  - Used more widely than reference counting
  - They visit the heap objects and determine which ones are not longer used
  - Tracing algorithms differ according to:
    - Whether all objects are visited or not
    - Whether they use the heap in an optimal way or not
    - Whether the collector is executed in parallel with the mutator or not
    - The duration of the pauses that the mutator undergoes when the algorithm is executed

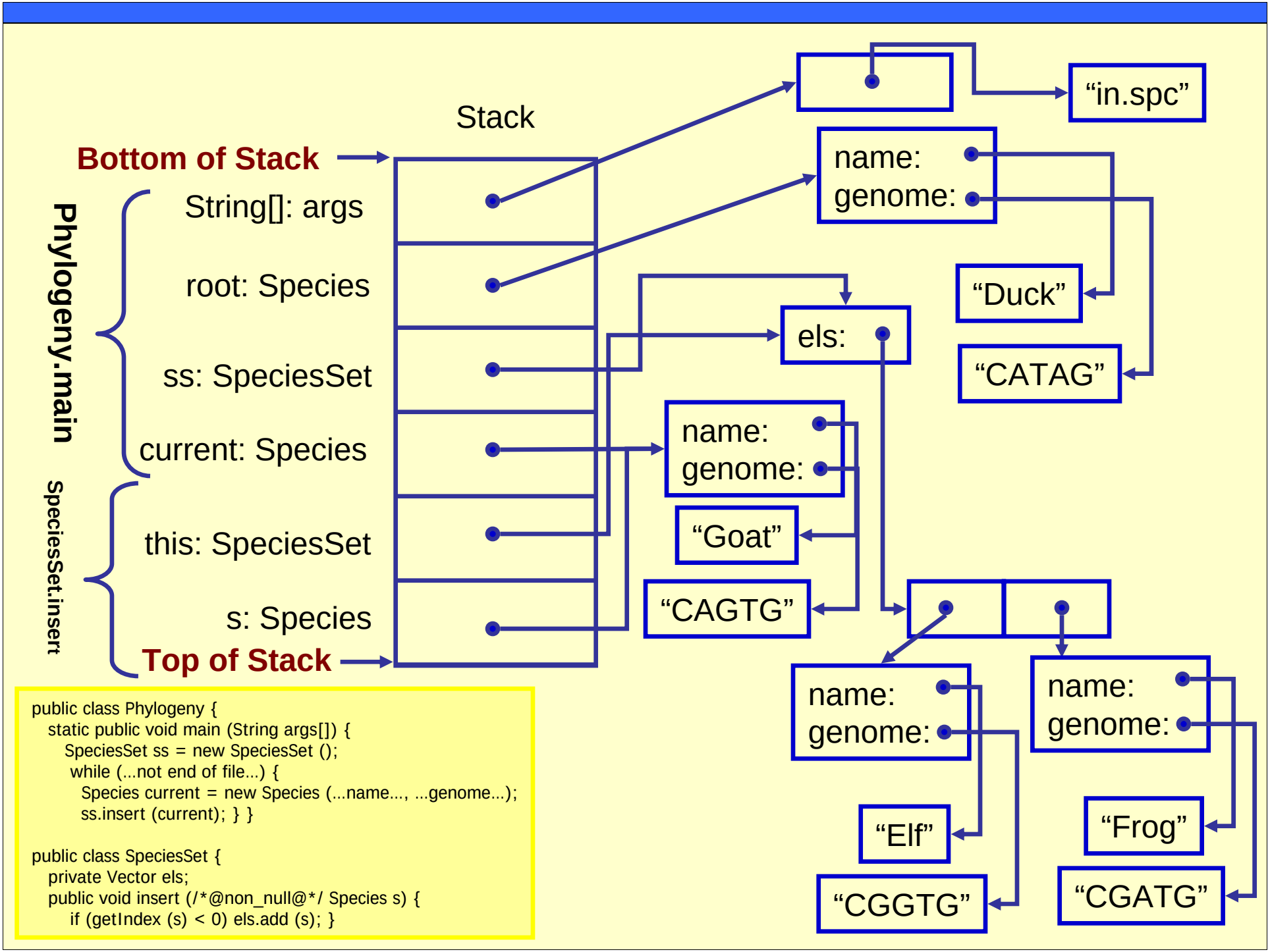
Mark and Sweep

- ✓ **Not a new concept**
  - John McCarthy, 1960 (first LISP implementation)
- ✓ **Start with a set of root references**
  - Typically on the stack
- ✓ **Mark every object you can reach from those references**
- ✓ **Sweep up the unmarked objects**
  - The garbage collector must stop normal processing before performing its duties and it takes a nontrivial amount of time to do a complete memory cleanup
  - Reduces processing predictability since you don't know when it will run nor for how long
  - Causes fragmentation of the heap, a significant problem in systems that execute for long periods of time
    - Poor performance due to lack of locality of data
    - May not be enough space to allocate to large object graphs

Java Classes

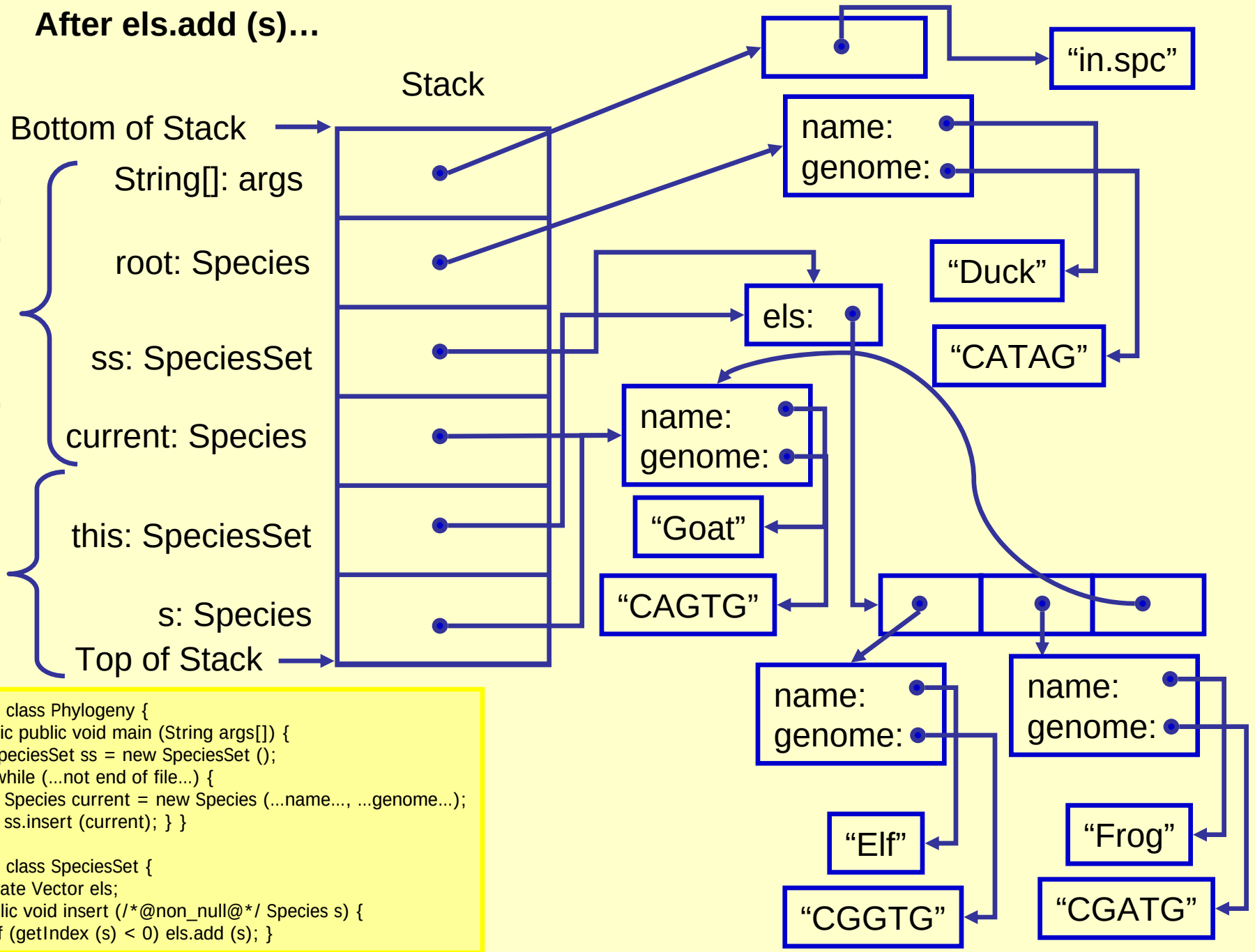
```
public class Phylogeny {
   static public void main (String args[]) {
      SpeciesSet ss = new SpeciesSet ();
      ... (open file for reading)
      while (...not end of file...) {
         Species current = new Species (...name from file...,
                                        ...genome from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

public class SpeciesSet {
   private Vector els;
   public void insert (/*@non_null@*/ Species s) {
       if (getIndex (s) < 0) els.add (s); }
```

From slides by David Evans
http://www.cs.virginia.edu/evans
used with permission



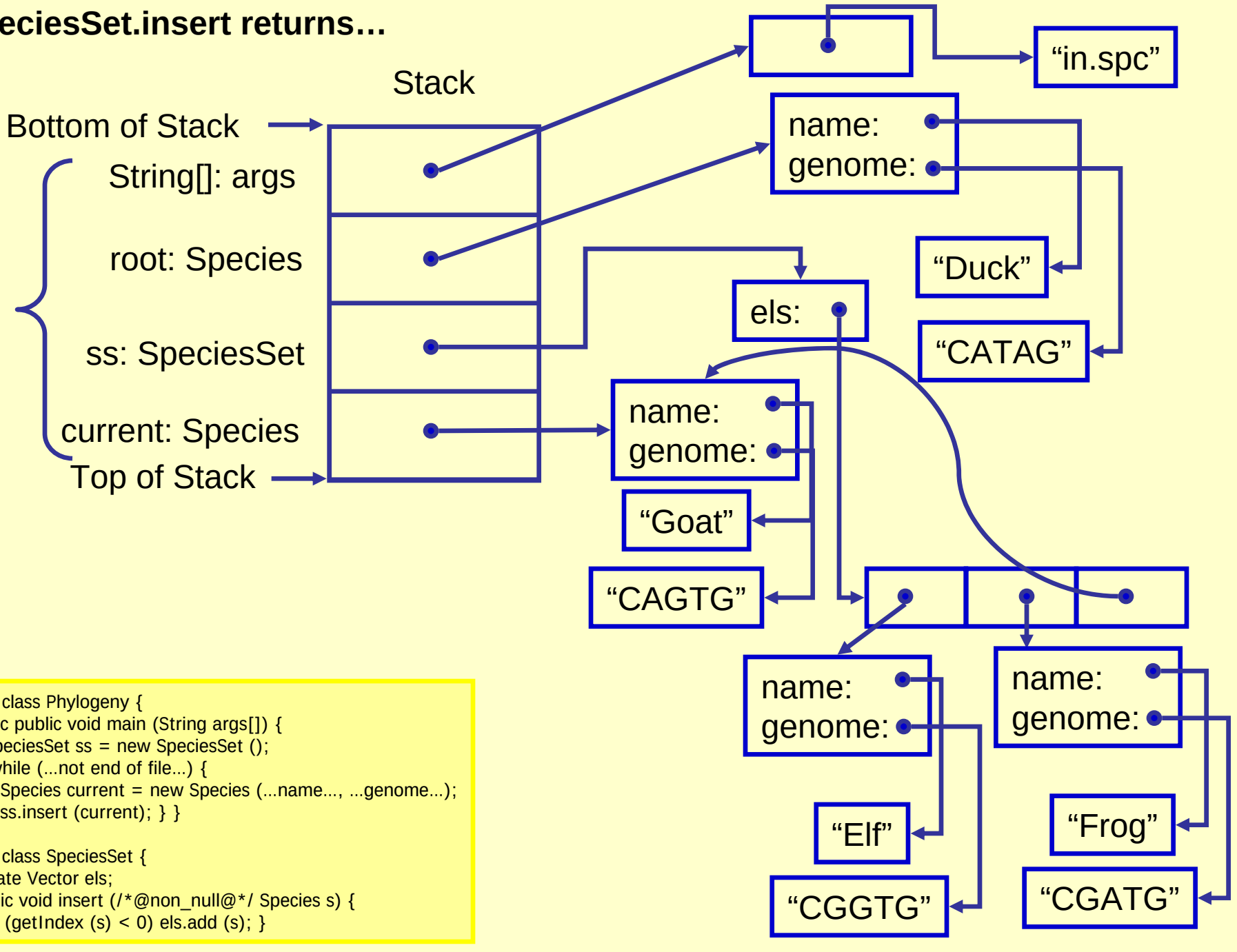

```
public class Phylogeny {
   static public void main (String args[]) {
      SpeciesSet ss = new SpeciesSet ();
       while (...not end of file...) {
         Species current = new Species (...name..., ...genome...);
         ss.insert (current); } }

public class SpeciesSet {
   private Vector els;
   public void insert (/*@non_null@*/ Species s) {
      if (getIndex (s) < 0) els.add (s); }
```

After els.add (s)…



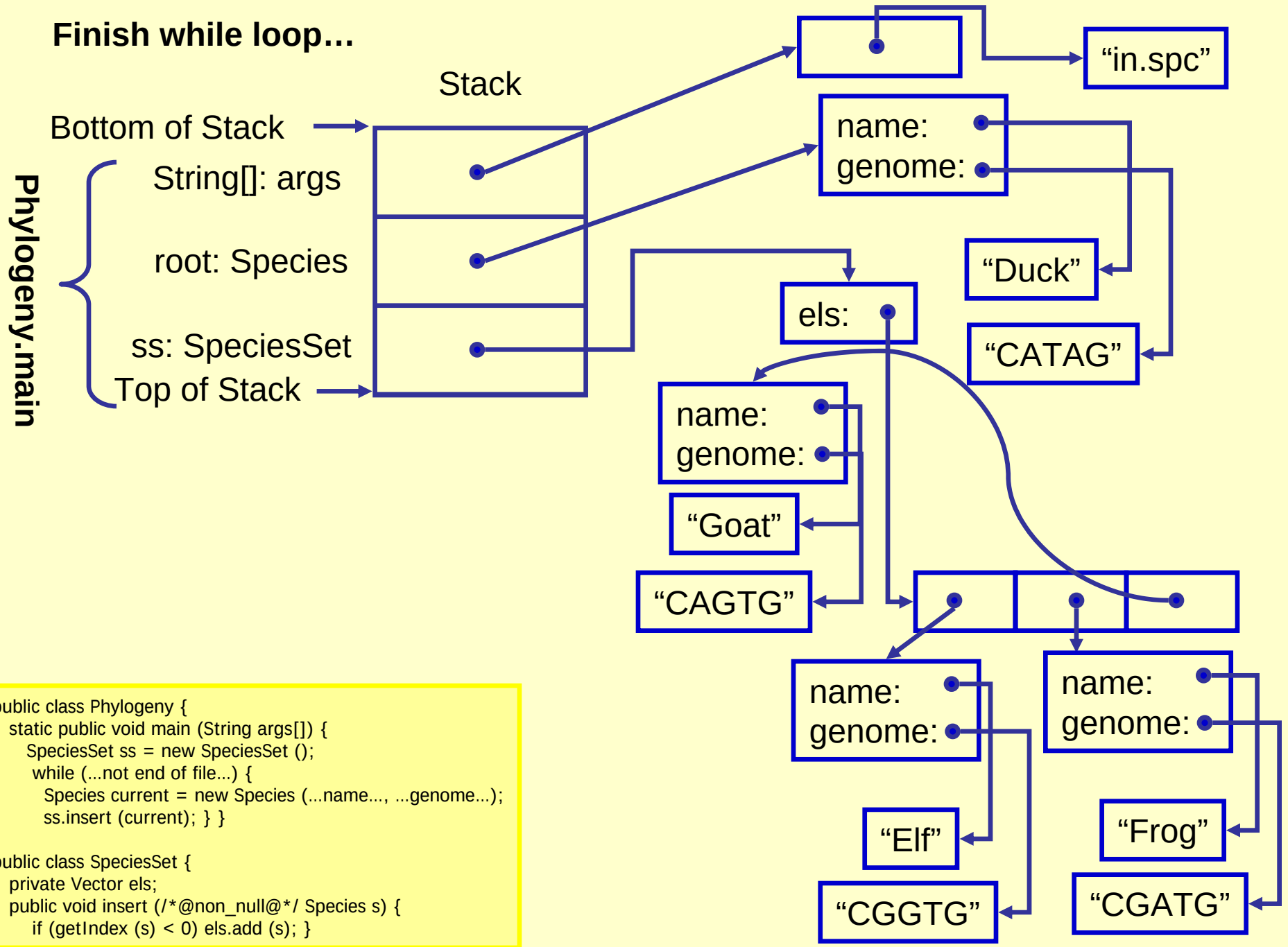

```
public class Phylogeny {
   static public void main (String args[]) {
      SpeciesSet ss = new SpeciesSet ();
       while (...not end of file...) {
         Species current = new Species (...name..., ...genome...);
         ss.insert (current); } }

public class SpeciesSet {
   private Vector els;
   public void insert (/*@non_null@*/ Species s) {
      if (getIndex (s) < 0) els.add (s); }
```

SpeciesSet.insert returns…



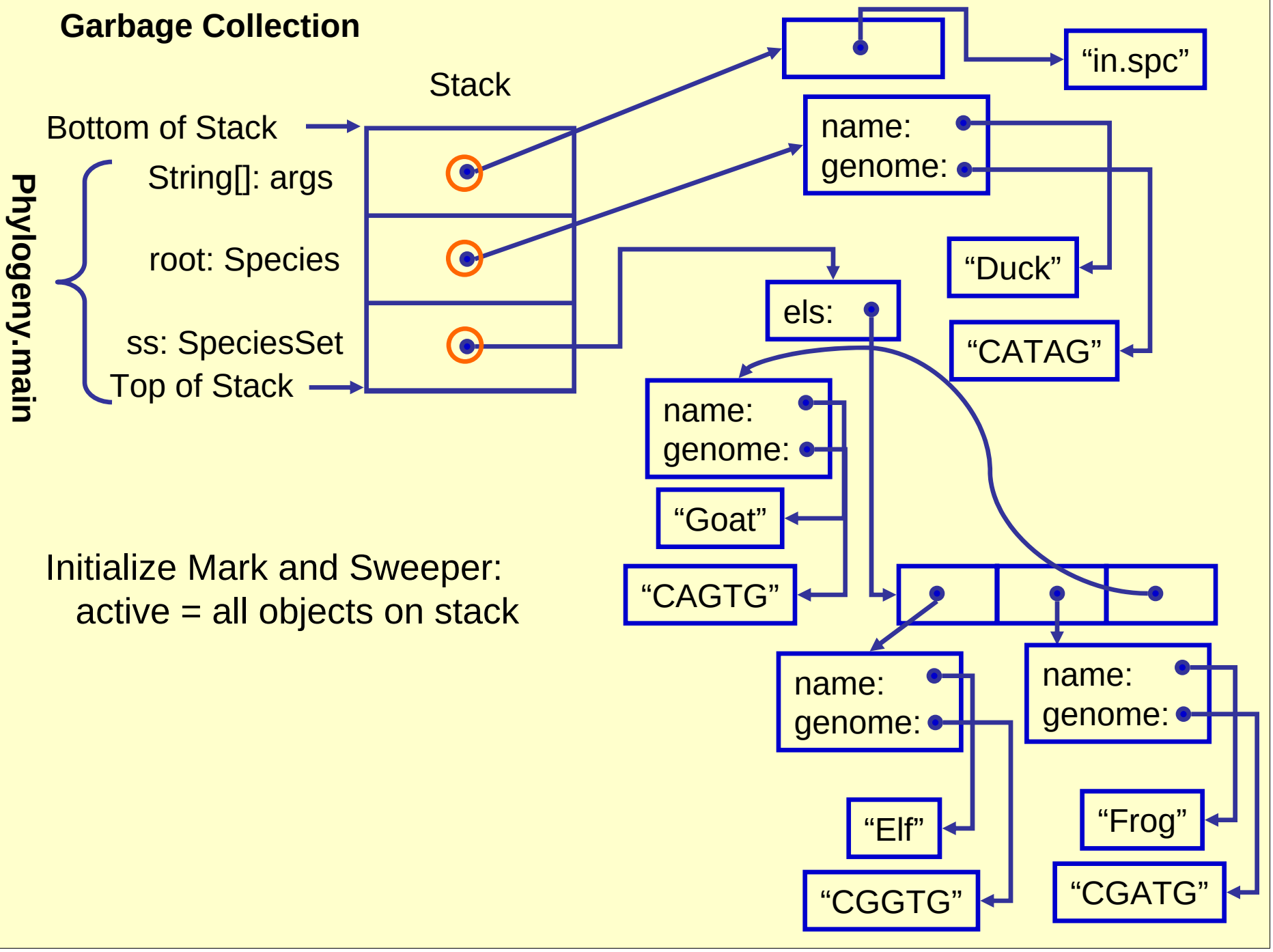

```
public class Phylogeny {
   static public void main (String args[]) {
      SpeciesSet ss = new SpeciesSet ();
       while (...not end of file...) {
         Species current = new Species (...name..., ...genome...);
         ss.insert (current); } }

public class SpeciesSet {
   private Vector els;
   public void insert (/*@non_null@*/ Species s) {
      if (getIndex (s) < 0) els.add (s); }
```

Finish while loop…



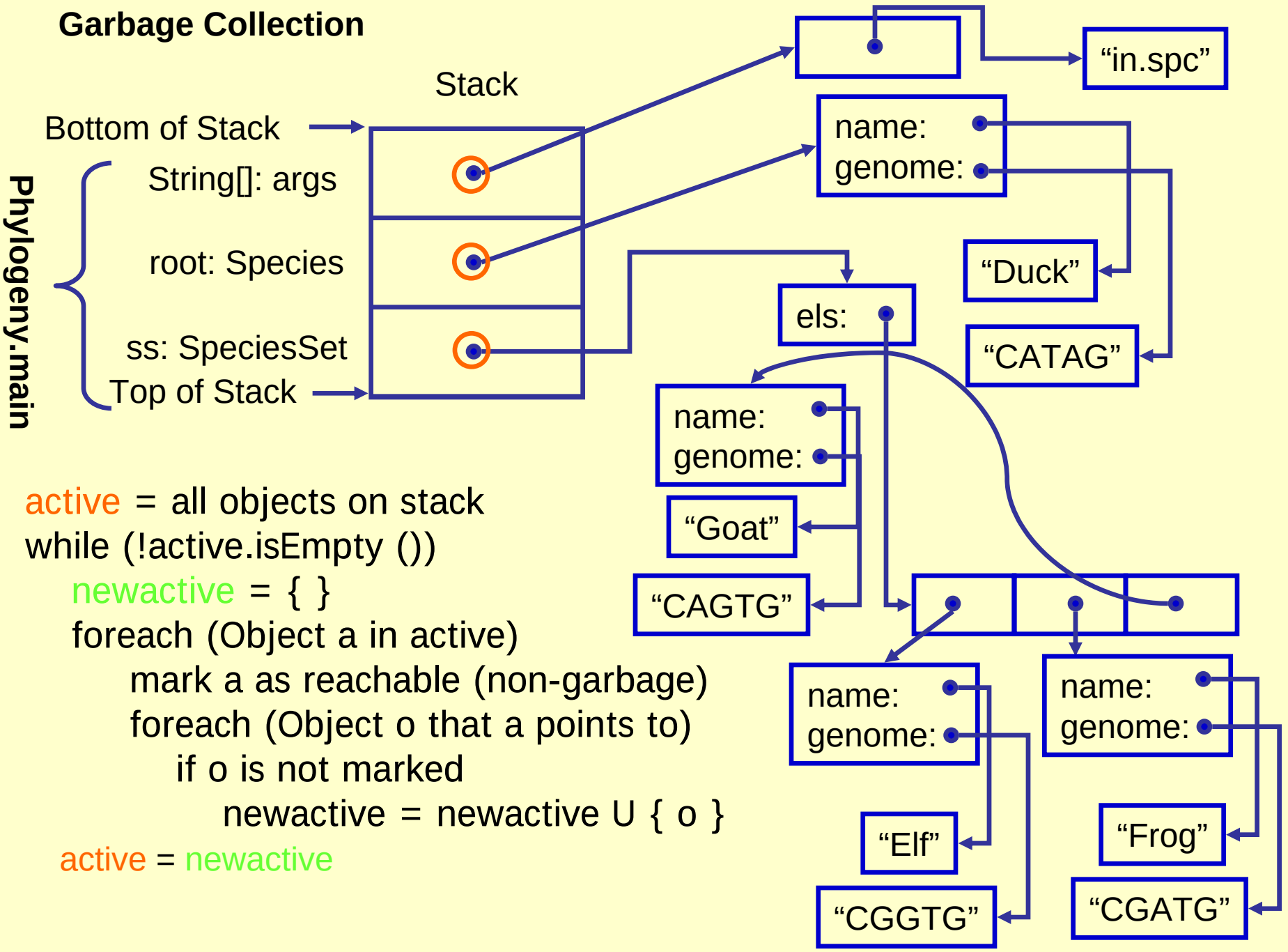

```
public class Phylogeny {
   static public void main (String args[]) {
      SpeciesSet ss = new SpeciesSet ();
       while (...not end of file...) {
         Species current = new Species (...name..., ...genome...);
         ss.insert (current); } }

public class SpeciesSet {
   private Vector els;
   public void insert (/*@non_null@*/ Species s) {
      if (getIndex (s) < 0) els.add (s); }
```

Garbage Collection



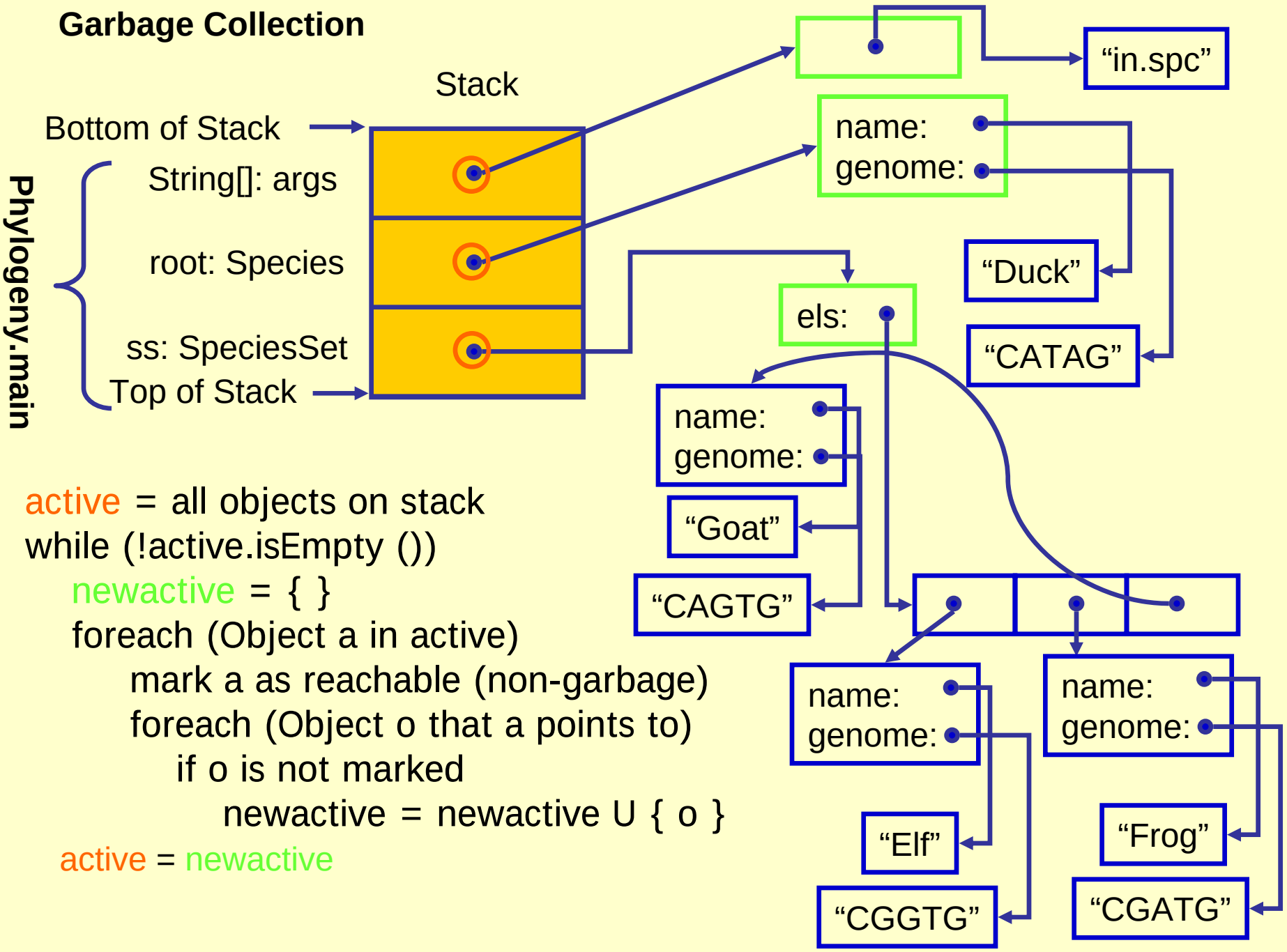
Initialize Mark and Sweeper:
active = all objects on stack

Garbage Collection



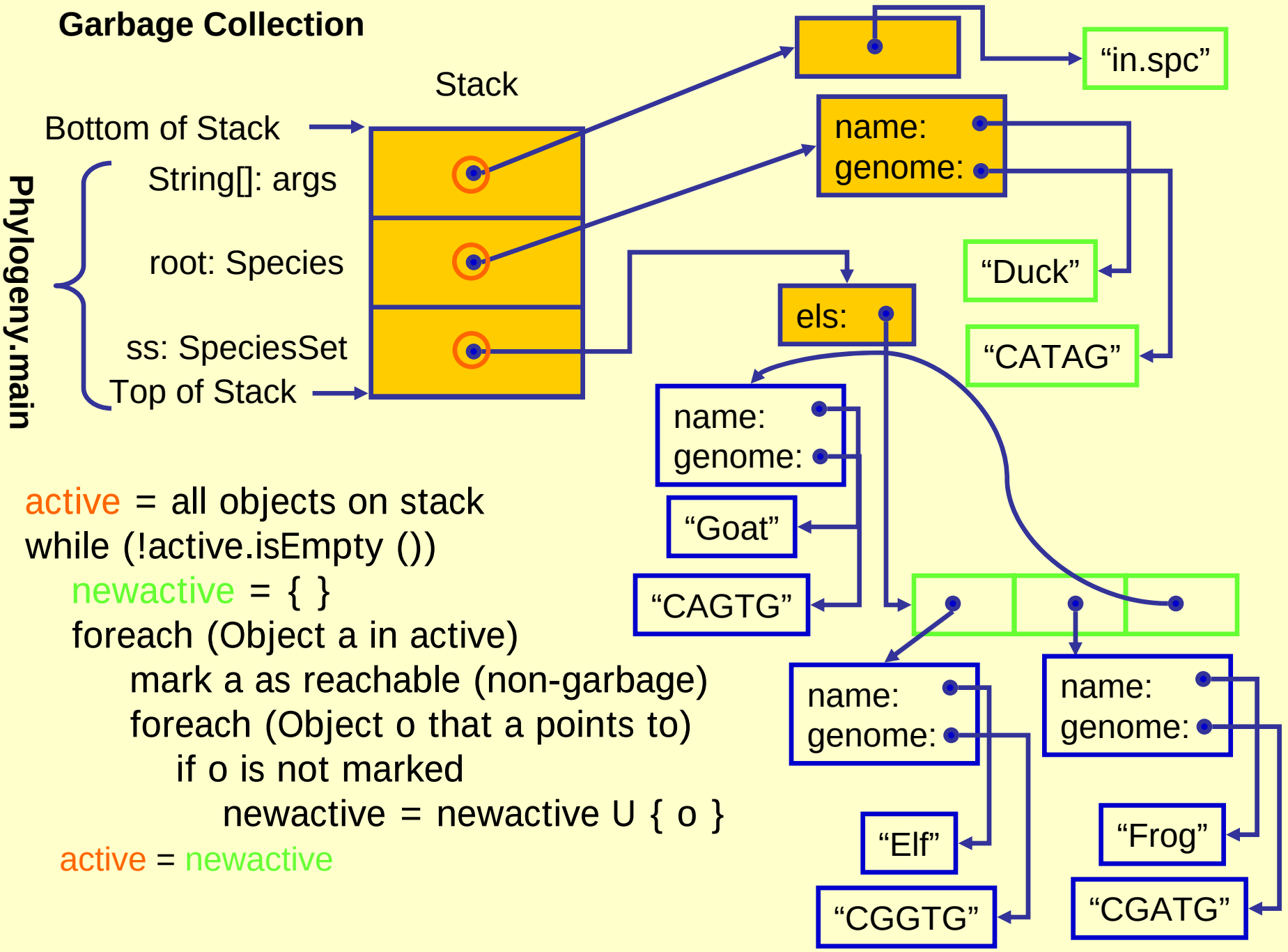

```
active = all objects on stack
while (!active.isEmpty ())
    newactive = { }
    foreach (Object a in active)
        mark a as reachable (non-garbage)
        foreach (Object o that a points to)
            if o is not marked
                newactive = newactive U { o }
    active = newactive
```

Garbage Collection



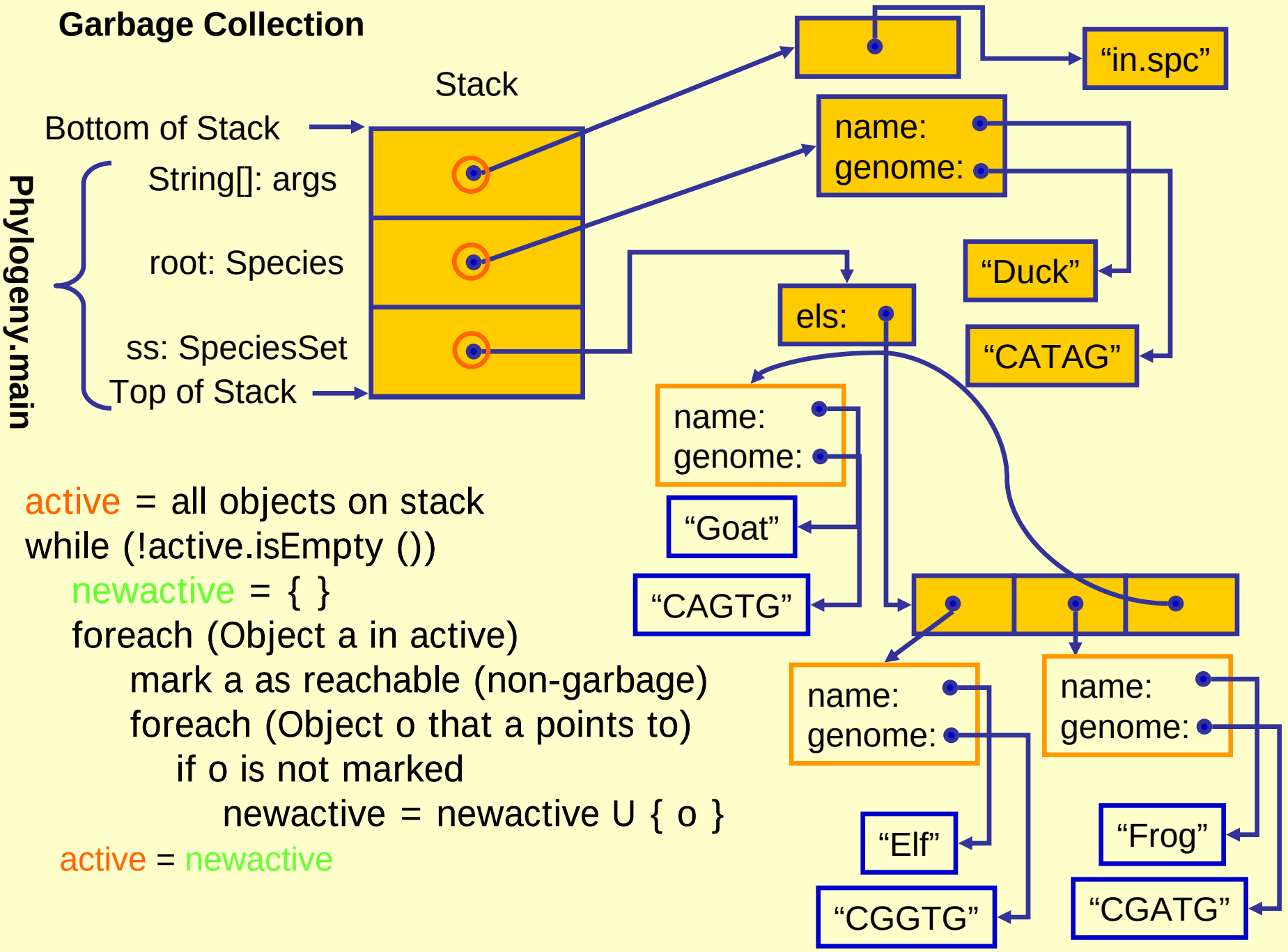

```
active = all objects on stack
while (!active.isEmpty ())
    newactive = { }
    foreach (Object a in active)
        mark a as reachable (non-garbage)
        foreach (Object o that a points to)
            if o is not marked
                newactive = newactive U { o }
    active = newactive
```

Garbage Collection



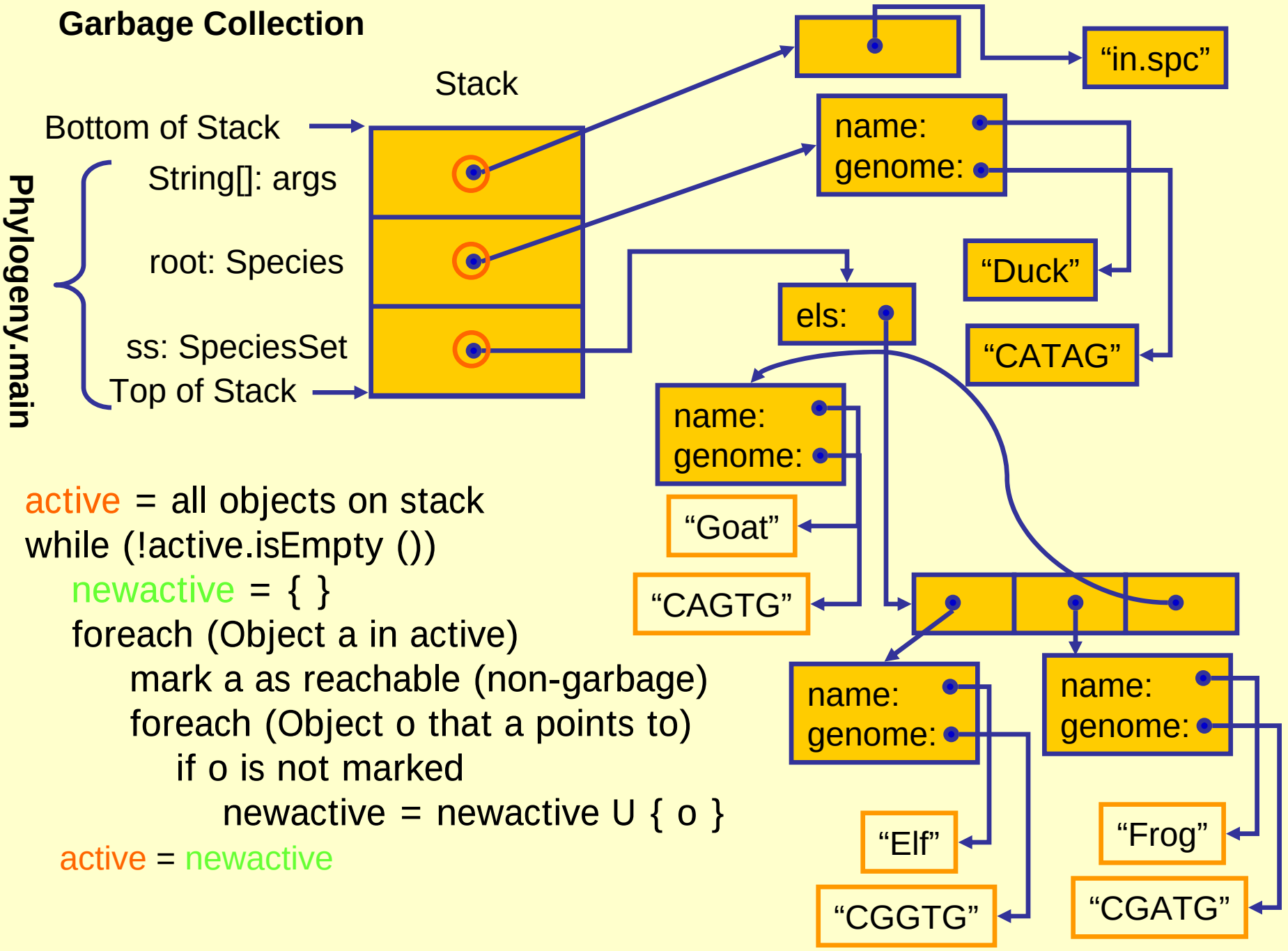

```
active = all objects on stack
while (!active.isEmpty ())
    newactive = { }
    foreach (Object a in active)
        mark a as reachable (non-garbage)
        foreach (Object o that a points to)
            if o is not marked
                newactive = newactive U { o }
    active = newactive
```

Garbage Collection



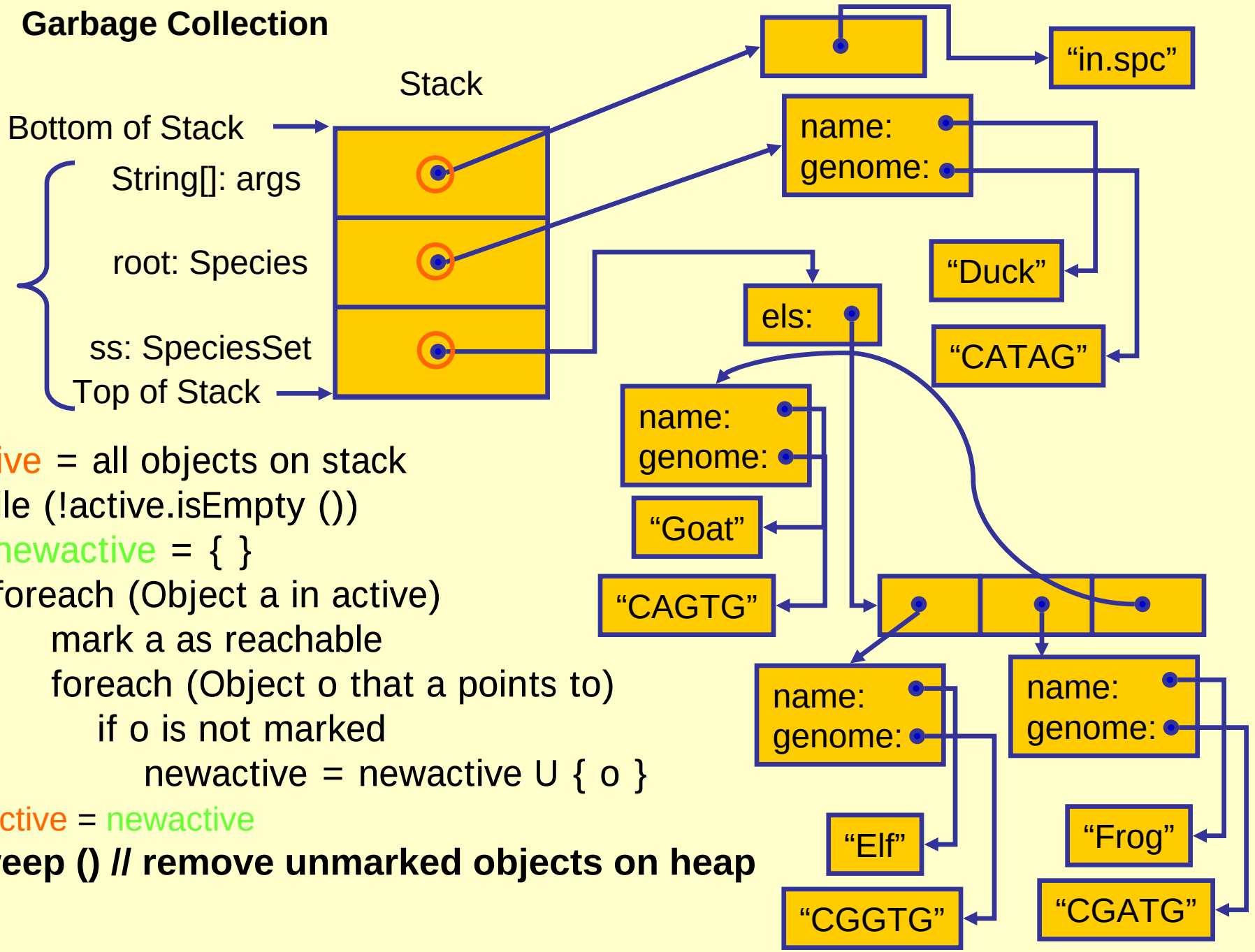

```
active = all objects on stack
while (!active.isEmpty ())
    newactive = { }
    foreach (Object a in active)
        mark a as reachable (non-garbage)
        foreach (Object o that a points to)
            if o is not marked
                newactive = newactive U { o }
    active = newactive
```

Garbage Collection



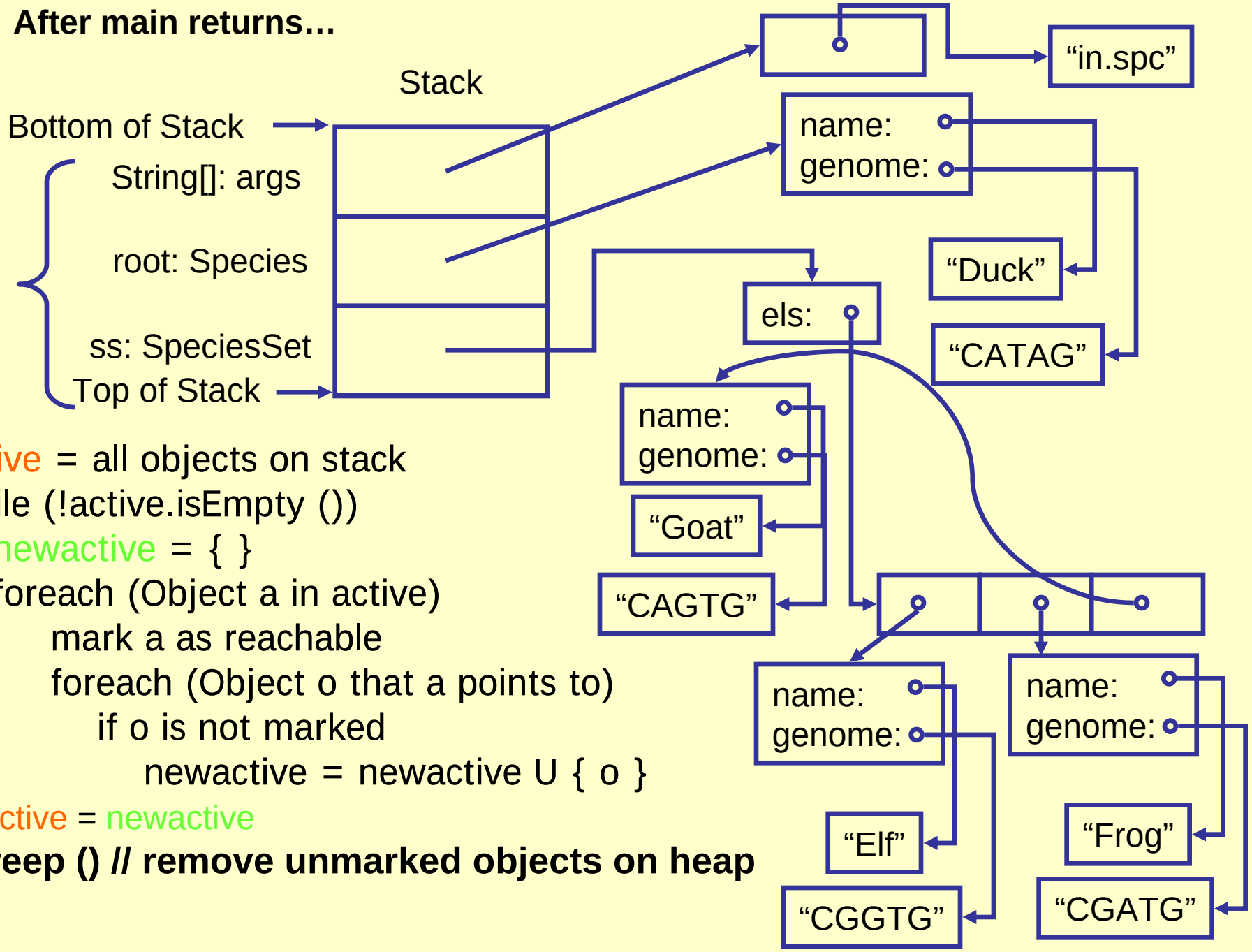

```
active = all objects on stack
while (!active.isEmpty ())
    newactive = { }
    foreach (Object a in active)
        mark a as reachable (non-garbage)
        foreach (Object o that a points to)
            if o is not marked
                newactive = newactive U { o }
    active = newactive
```

Garbage Collection



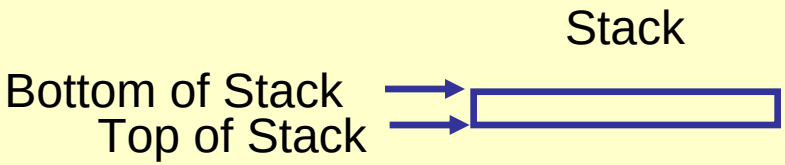

```
active = all objects on stack
while (!active.isEmpty ())
    newactive = { }
    foreach (Object a in active)
        mark a as reachable
        foreach (Object o that a points to)
            if o is not marked
                newactive = newactive U { o }
    active = newactive
sweep () // remove unmarked objects on heap
```

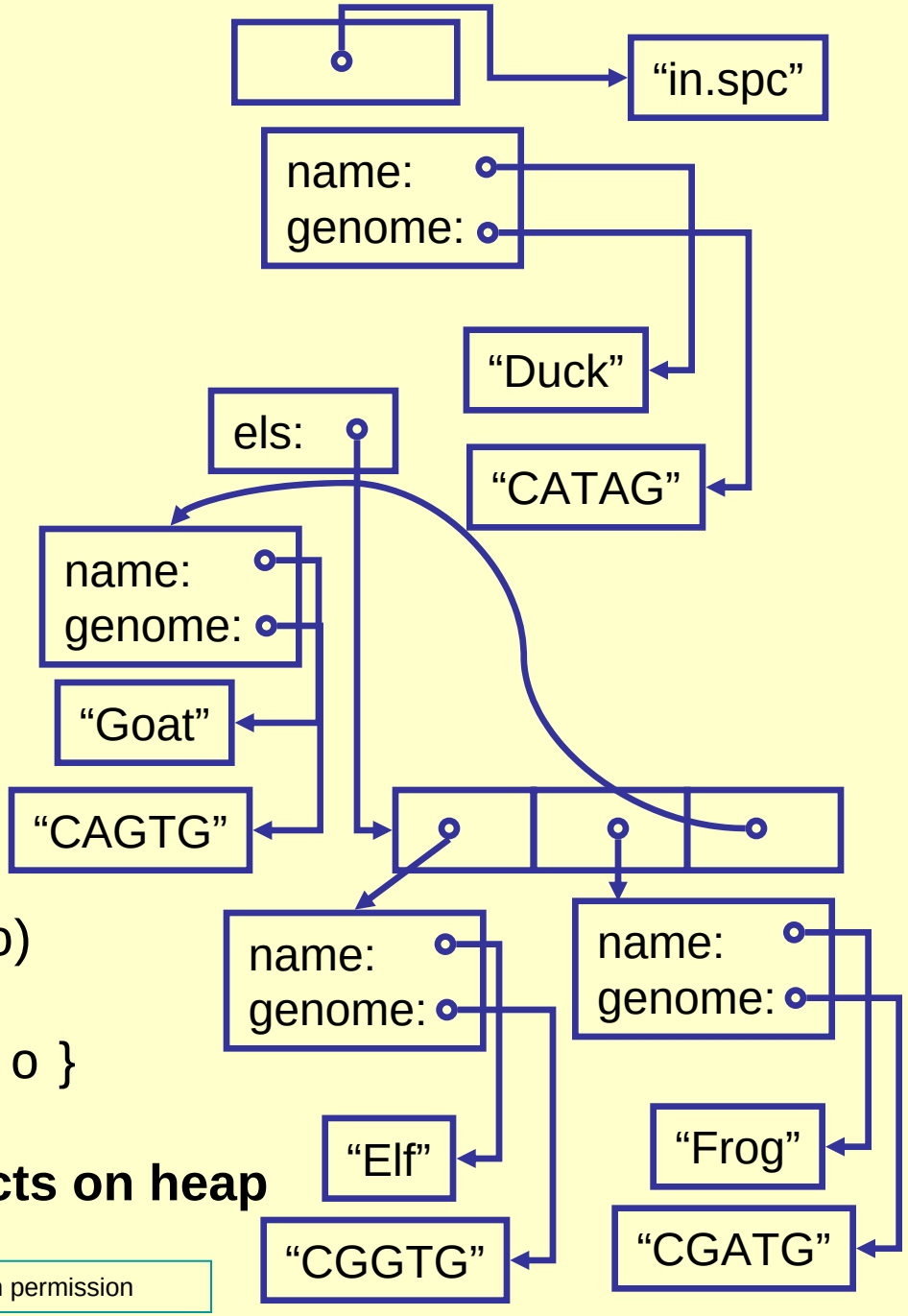
After main returns…




```
active = all objects on stack
while (!active.isEmpty ())
    newactive = { }
    foreach (Object a in active)
        mark a as reachable
        foreach (Object o that a points to)
            if o is not marked
                newactive = newactive U { o }
    active = newactive
sweep () // remove unmarked objects on heap
```

Garbage Collection






```
active = all objects on stack
while (!active.isEmpty ())
    newactive = { }
    foreach (Object a in active)
        mark a as reachable
        foreach (Object o that a points to)
            if o is not marked
                newactive = newactive U { o }
    active = newactive
    sweep () // remove unmarked objects on heap
```

From slides by David Evans http://www.cs.virginia.edu/evans used with permission

Stop and Copy

- ✓ **Solves fragmentation problem**
- ✓ **Copy all reachable objects to a new memory area**
- ✓ **After copying, reclaim the whole old heap**

- ✓ **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:**
  - Many 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**

- ✓ **Sun HotSpot**
  - Sun improved memory management in the Java 2 VMs (JDK 1.2 and on) by switching to a generational garbage collection scheme
  - The heap is separated into two regions:
    - New Objects
    - Old Objects

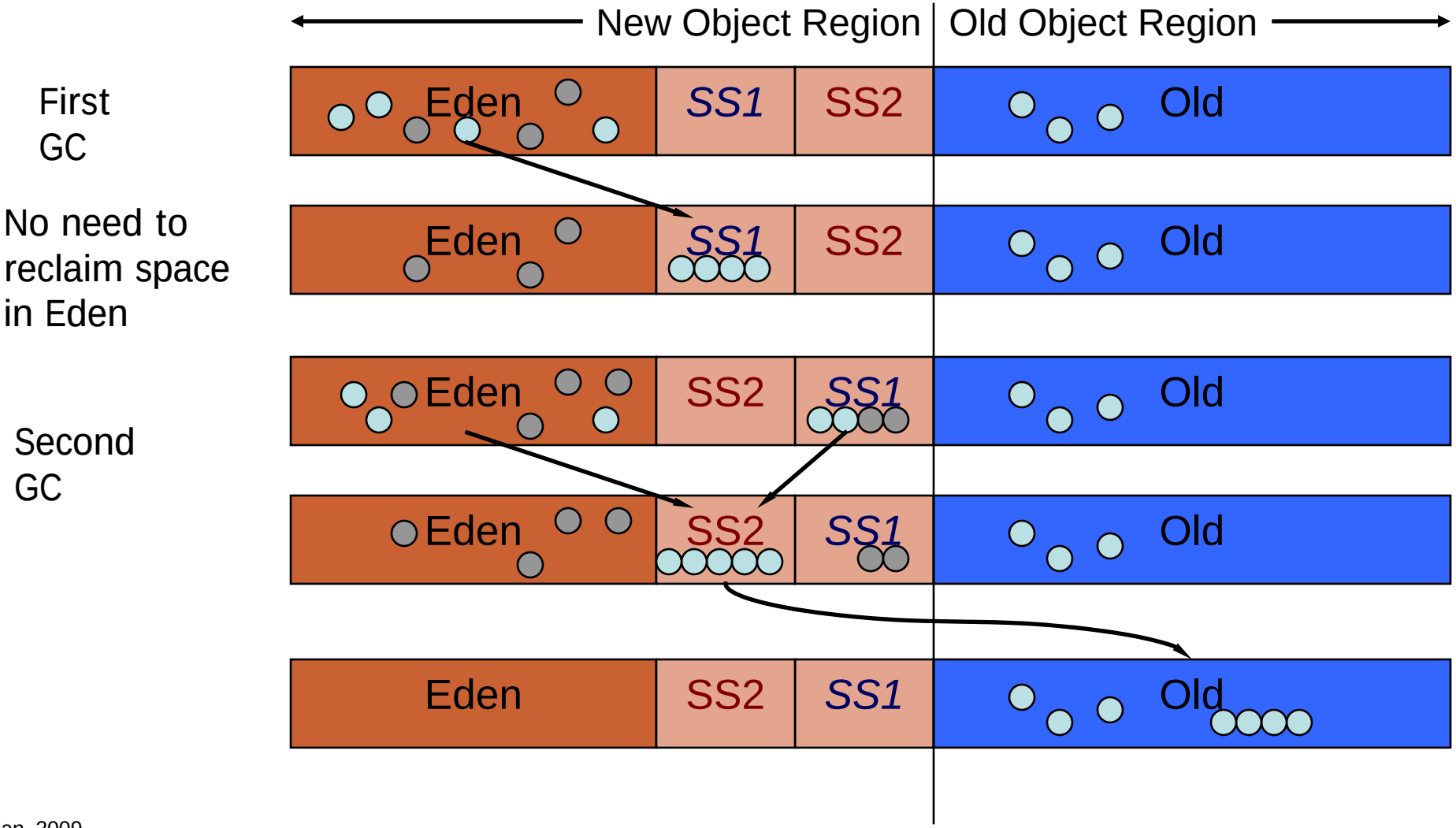
New Object Region in Hotspot

- ✓ **The idea is to use a very fast allocation mechanism and hope that objects all become garbage before you have to garbage collect**
  - The New Object Regions is subdivided into three smaller regions:
    - Eden, where objects are allocated
    - 2 “Survivor” semi-spaces: “From” and “To”
  - The Eden area is set up like a stack - an object allocation is implemented as a pointer increment
  - When the Eden area is full, the GC does a reachability test and then copies all the live objects from Eden to the “To” region
  - The labels on the regions are swapped
  - “To” becomes “From” - now the “From” area has objects

New Object Region in Hotspot

- ✓ **The next time Eden fills objects are copied from both the "From" region and Eden to the "To" area**
  - There's a "Tenuring Threshold" that determines how many times an object can be copied between survivor spaces before it's moved to the Old Object region
  - Note that one side-effect is that one survivor space is always empty
  - That space is smaller than the space necessary for a typical Stop and Copy solution
- ✓ **Old Object Region**
  - The old object region is for objects that will have a long lifetime
  - The hope is that because most garbage is generated by short-lived objects that you won't need to GC the old object region very often

Generational Garbage Collection



Finalizers

- ✓ **All Java objects have a finalize() method inherited from the Object class**
- ✓ **In general, DO NOT override this method in your classes**
  - ❑ See Effective Java
- ✓ **The Garbage Collector calls this method when the object is removed from the heap**
  - ❑ Programmers should never call it directly
- ✓ **It might be tempting to put clean up code (e.g. release database connections, write data to file, etc.) in a finalize() method – DO NOT DO THIS**
  - ❑ Garbage collection might not happen (e.g. the program may simply end first and when the JVM quits, all memory is returned to the operating system)
- ✓ **A finalizer might be used to help with debugging to check the final state of an object (e.g. was the object saved?) [from Thinking in Java]**

```
public void finalize () {
   if (!saved)
      System.out.println("ERROR: Forgot to …");
}
```

  - ❑ Then, to request the system to garbage collect call System.gc right before the program exits (e.g. last line of main)

Optimization Quotes

- ✓ **Rules of Optimization - M.A. Jackson**
  - ❑ Rule 1: Don't do it.
  - ❑ Rule 2 (for experts only): Don't do it yet.

- ✓ **"More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other reason – including blind stupidity." – W.A. Wulf**

- ✓ **We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil." – Donald Knuth**

Optimization 101

✓ Reality

- Hard to predict where the bottlenecks are
  - Write the code you want to be correct and finished first, then worry about optimization
- May already be fast enough!
  - If not, measure the bottleneck
  - Focus optimization on bottleneck using Algorithms and Language optimizations

How To Tune GC

- ✓ **Not a simple topic**
- ✓ **There are no universal magic values - every app is different**
- ✓ **Things to tune – but only when the need is demonstrated during testing**
  - Memory Size
    - overall size, individual region sizes
  - GC parameters
    - Minimum/maximum % of free heap,
  - Type of GC
    - single heap, generational, incremental, concurrent, parallel

Tuning Parameters – Both Sun and Windows VMs

-ms, -Xms

- **sets the initial heap size**

-mx, -Xmx

- **sets the maximum heap size**

-Xss

- **sets the size of the per-thread stacks**

-Xminf [0-1], -XX:MinHeapFreeRatio [0-100]

- **sets the percentage of minimum free heap space - controls heap expansion rate**

-Xmaxf [0-1], -XX:MaxHeapFreeRatio [0-100]

- **sets the percentage of maximum free heap space - controls when the VM will return unused heap memory to the OS**

Tuning Parameters - Sun

-XX:NewRatio

- **sets the ratio of the old and new generations in the heap. A NewRatio of 5 sets the ratio of new to old at 1:5, making the new generation occupy 1/6th of the overall heap**
- **defaults: client 8, server 2**

-XX:NewSize, -XX:MaxNewSize [1.3]

-Xmn [1.4]

- **sets the minimum and maximum sizes of the new object area, overriding the default calculated by the NewRatio**

-XX:SurvivorRatio

- **sets the ratio of the survivor space to the eden in the new object area. A SurvivorRatio of 6 sets the ratio of the three spaces to 1:1:6, making each survivor space 1/8th of the new object region**
- **default: 25**

Conclusions

- ✓ **Garbage collection has a direct impact on your application performance**
- ✓ **Use tools to discover those impacts**
- ✓ **jvmstat - http://java.sun.com/performance/jvmstat/**
  - ❑ “The jvmstat technology adds light weight performance and configuration instrumentation to the HotSpot JVM and provides a set of monitoring APIs and tools for monitoring the performance of the HotSpot JVM in production environments. “
  - ❑ See assignment #1 for more information

Reference Objects

Safely Interacting with the Garbage Collector

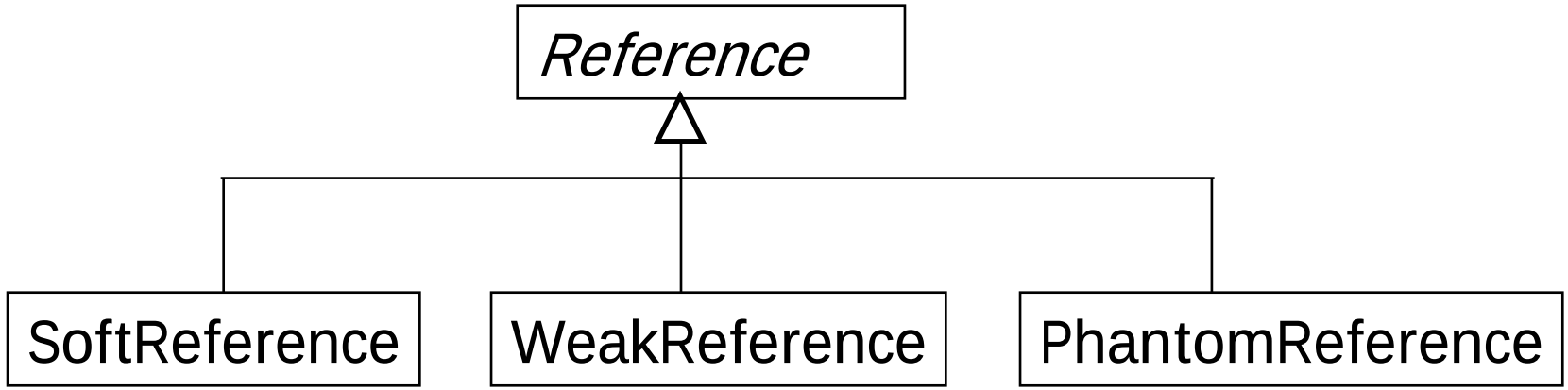
J.D. Baker

Reference Object Definitions

- ✓ **Definitions derived from a JavaOne '99 presentation**
  - ❑ http://industry.java.sun.com/javaone/99/tracks/ - Does Java Technology Have Memory Leaks? (no longer available at this URL)
- ✓ **referent**
  - ❑ object contained by an instance of one of the Reference classes
- ✓ **loiterer**
  - ❑ an object that persists past its usefulness
- ✓ **lapsed listener - an object added to but never removed from a collection of listeners**
  - ❑ common type of loiterer
  - ❑ an argument against anonymous inner classes as listeners

Reference classes

- ✓ **Interact with the garbage collector**
- ✓ **Gives programmer some control over the gc process**
- ✓ **Class hierarchy**
  - ❑ package java.lang.ref



Softly Reachable

- ✓ **Immutable, except for clearing**
  - aReference.clear()
- ✓ **GC CAN reclaim an object that is softly reachable, but does not HAVE to**
- ✓ **GC will clear all soft references before throwing OutOfMemoryError**
- ✓ **Good choice for implementing a cache**
- ✓ **Overrides one method from the abstract class *Reference***
  - public Object get() - Returns referent or null
- ✓ **May be constructed with a ReferenceQueue**

Weakly Reachable

- ✓ **Defined as reachable only through a WeakReference object**
- ✓ **Objects are always eligible for garbage collection if the only referent is a weak reference**
  - ❑ gc clears all weak references
  - ❑ object is finalized then or later
- ✓ **Useful for implementing some listeners**
  - ❑ eliminates anonymous listener problem
  - ❑ WeakHashMap is a good example
- ✓ **May be constructed with a ReferenceQueue**

ReferenceQueue

- ✓ **Associated with a Reference object by the constructor call**
- ✓ **objects are placed in the ReferenceQueue when the reference field is cleared (set to null)**

```
public SoftReference(Object referent,
                     ReferenceQueue q)
```

```
public abstract class Reference
{
   . . .
   private Object referent /* Treated specially by GC */
   . . .
}
```

Conclusion 2

- ✓ **We're now ready to tackle assignment #1**
- ✓ **Remember to do the assigned reading and participate in the weekly discussion.**