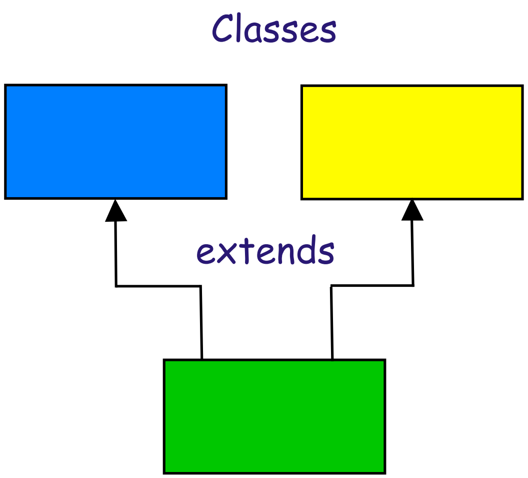
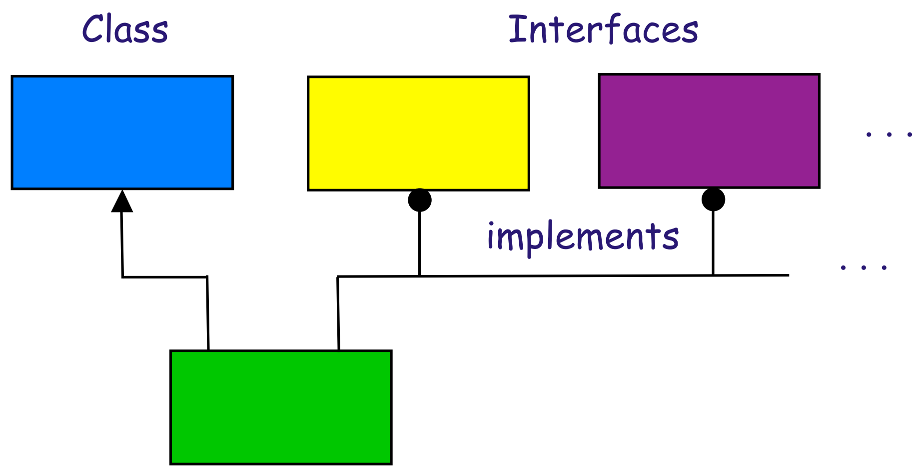
Java Topics

Interface vs. Inheritance



Not legal in Java.

Legal in C++.



Legal in Java.

C++ has no concept of Interface.
Interface is an idiomatic use of inheritance.

Abstract Class vs. Interface

Interface	Abstract Class
Used to **declare** methods common to all implementations.	Used to "factor out" methods common to all derived classes.
Declare as **interface**	Declare as **abstract class**
Cannot be instantiated	Cannot be instantiated
Cannot have instance variables	**Can** have instance variables
Cannot have method implementations	**Can** have method implementations
	abstract method is like method in an interface; must be over-ridden.

Java Collections Framework

Java Collections Framework

- **Coherent** set of interfaces and classes for implementing collections
- Various types of access (random, sequential)

Vector class

- Vector is a growable array
  [O(1) to access, O(n) to insert/remove, O(1) to add.]

- It was early Java, which pre-dated the framework, and less complex than some of the newer classes.

- However, it is also has fewer requirements than some of the newer classes and is simpler to implement (or emulate in another language, an advantage for porting an app).

- As of Java 1.2, it became part of the framework (by implementing interface **List**).

- Also, Vector is *synchronized* for multiple threads.

Enumeration interface

- Enumeration provides a simple, elegant abstraction for enumerating elements of something:
  - **Object nextElement()**

    returns the next element in the enumeration
  - **boolean hasMoreElements()**

    true iff there are more elements in the enumeration
  - Note: an Enumeration can be infinite in extent.

Getting Enumeration from a Vector using elements() method

```
Vector myVector = new Vector();
. . . populate myVector . . .
Enumeration en = myVector.elements();

while( en.hasMoreElements() )
    {
    Object current = en.nextElement();
    . . . use current . . .
    }
```

Enumerating using a for loop

```
Vector myVector = new Vector();
. . . populate myVector . . .

for(Enumeration en = myVector.elements(); en.hasMoreElements();  )
      {
      Object current = en.nextElement();
      . . . use current . . .
      }
```

Note: empty incrementation part in the **for** header.

nextElement() automatically "increments"

ArrayList

- Closest to Vector in the framework.
- Instead of Enumeration, use Iterator

Iterator interface compared with Enumeration

Enumeration	Iterator
nextElement()	next()
hasMoreElements()	hasNext()
(*no counterpart*)	remove() [“optional”: may throw exception. Removes most recent element returned by next().]

LinkedList

- Doubly-linked list.
- O(1) insert, remove; O(n) access
- Not an “open” list
  [no shared list structure]
- Most of the “expected” methods

ListIterator interface

- extends Iterator
- Adds:
  - nextIndex()
  - previous()
  - hasPrevious()
  - previousIndex()
  - set(Object ob) [sets most recent element]
  - add(Object ob) [optional]

Java Iterator Example

```
LinkedList<T> L = new LinkedList<T>();
      . . .
for( Iterator<T> j = L.iterator(); j.hasNext(); )
  {
      . . . use j.next() . . .
}
```

Java For-Each Loop
"syntactic sugar"

Hides the explicit creation of the underlying Iterator, resulting in tidier code.

```
LinkedList<Type> L = new LinkedList<Type>();
     . . .
 for( Type j : L)            // read ':' as "in"
     {
       . . . use j . . . as a variable of type Type
     }
```

Why use Iterator or Enumeration?

- Why use Iterator or Enumeration when explicit access to i^{th} element would do?
  - Access by index may be inefficient.
  - Using an Iterator allows a method to accept **any** Iterator, rather than just from an explicit structure.

Comparative Examples

- void foo(List<T> L) { . . . }

  foo applies to any List

- void foo(Iterator<T> I) { . . . }

  foo applies to any iterator
  (more general, because Iterator can come from List and a variety of other things)

Example Iterator

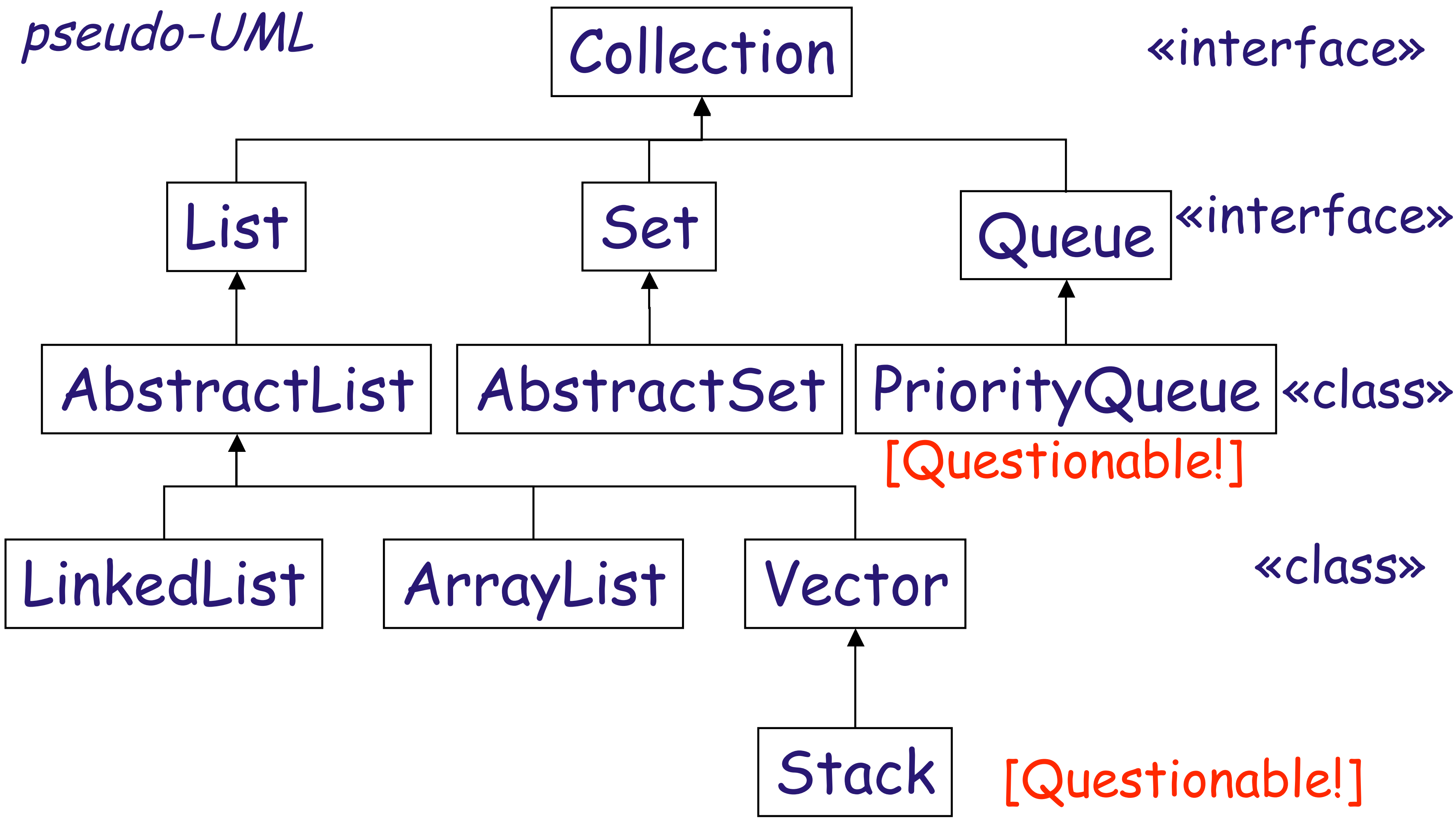
Iterator<BigInteger> p = new PrimesIterator();

. . . p.next() is the next prime

p can be passed to any method having an Iterator argument

. . . myMethod(**Iterator**<BigInteger> i) . . .

Collection Hierarchy (part of)



Applets and Graphics

Applet Drawing

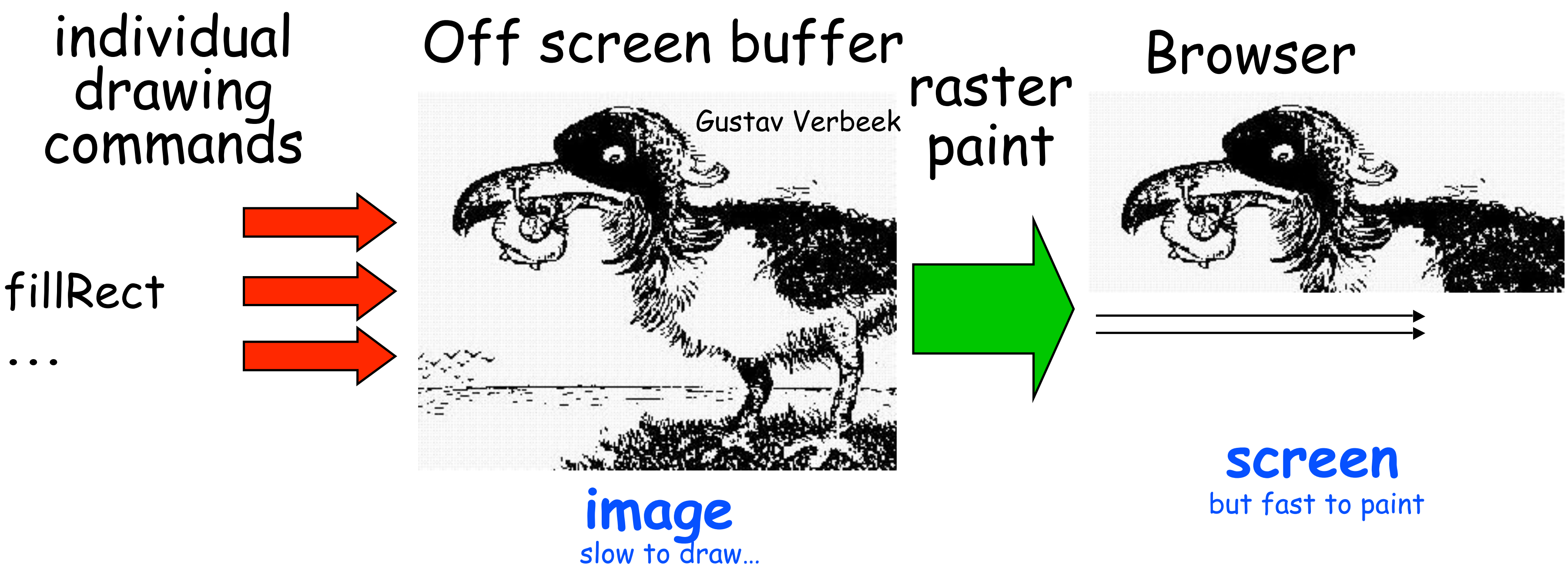
some event occurs, such as calling **repaint**() or making the window visible

update() is called on the window's graphics, which then calls **paint**()

These methods can be overridden.

whatever happens in **paint**() is displayed

Double Buffering
essential for avoiding flicker

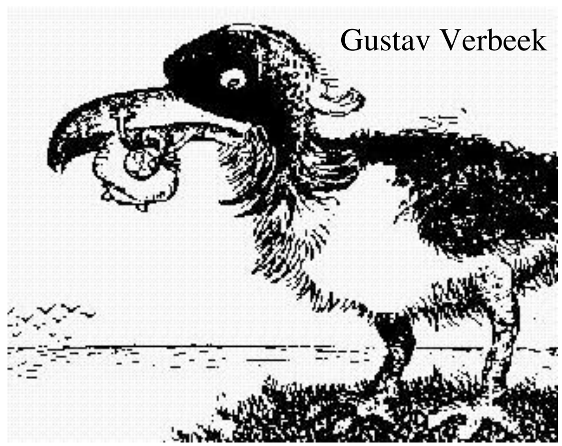


Double Buffering
essential for avoiding flicker

individual drawing commands

fillRect ...

Off screen buffer

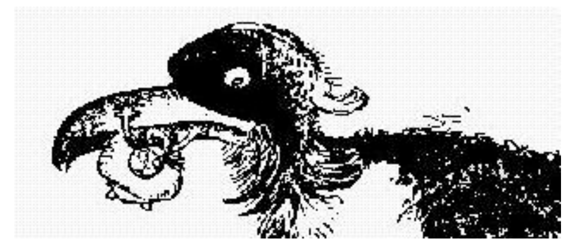


image

possibly slow to draw…

raster copy

Browser window



screen

but fast to copy

repaint()

calls update()

```
public void update(Graphics g)
{
  paint(g);
}
```

```
public void paint(Graphics g)
{
  g.drawImage(image, 0, 0, null);
}
```

Code from DESC

```
public class DESCapplet extends Applet implements Runnable
  {
  Thread myThread;              // Thread for this applet.

  Image image;                  // Image for painting graphics buffer
  Graphics graphics;            // graphics buffer for drawing off-screen

   public void init()           // required for Applet
    {
    setSize(appletWidth, appletHeight); // Set the size for this applet.
    setBackground(backgroundColor);     // Set the background color.
    createGraphicsBuffer();             // Create off-screen buffer.
          . . .
    }

private void createGraphicsBuffer()
    {
    image = createImage(getWidth(), getHeight());
    graphics = image.getGraphics();
    }
```

Code from DESC

```
void drawContents()
   {
   // Paint the background clean first.

   graphics.setColor(backgroundColor);
   graphics.fillRect(0, 0, getWidth(), getHeight());
   graphics.setColor(foregroundColor);

   // Draw the score.

  drawScore();

   // Draw the grid.

   if( grid != null )
     {
     grid.draw(graphics, xGridOffset, yGridOffset);
   }

   repaint();
   }
```

```
@Override
  public void update(Graphics g)
    {
    paint(g);
    }

 @Override
  public void paint(Graphics g)
    {
    g.drawImage(image, 0, 0, null);
    }
```

Code from DESC

```
public void run()                    // required for Applet
   {
   drawContents();

   setSample(initialSample);

   totalSteps = 0;

   while( true )
     {
     sleepAbit();

     oneStep();
    }
   }


 void oneStep()
   {
     totalSteps++;
     simulateHeadMotion();
     drawContents();
   }
```

Threads

Java Threads

- A "thread" means computer code being executed.
- More than one thread can be executed virtually simultaneously (interleaved).
  - The code for the threads can be the **same**, or different.
  - Each thread has its own state, **sort of**.
  - Threads can **share variables**, and modify the variables they share.
- Programs with > 1 thread are called "concurrent programs".
- With multiple "cores" (processors), threads can run physically simultaneously, in principle.

Multi-Processing

- e.g. Jaguar supercomputer at Oak Ridge National Lab:
  - 1,000,000,000,000,000 (1 quadrillion) floating-point operations per second (= 1 petaflop)
  - 182,000 AMD quad-core Opterons, running at 2.3 gigahertz
  - 362 terabytes of memory (with 578 terabytes per second of memory bandwidth)

Contrasts

	Jaguar SC	Human Brain
Capacity	1.69×10^{14} transistors among 728,000 processors + 362×10^{12} bits memory	100×10^{12} connections over 100×10^{9} neurons
Speed	2.3 GHz	1 kz

Timing of Threads

- Threads don't progress in lock-step fashion.
- One may be started and another stopped in an **unpredictable** fashion by the operating system.
- This behavior is called **asynchronous**.

Similar Idea: Processes

- A process is also code in execution.
- Typically processes don't share variables, although limited types of sharing are possible.
- Using multiple processes is common in, e.g. UNIX:

  generate | filter | test | display (here | means "pipe")

- Processes are "heavy weight", threads are "light weight".
- "Weight" refers to the amount of information that needs to be changed in switching the processor from one context to another.

Why are Threads Useful?

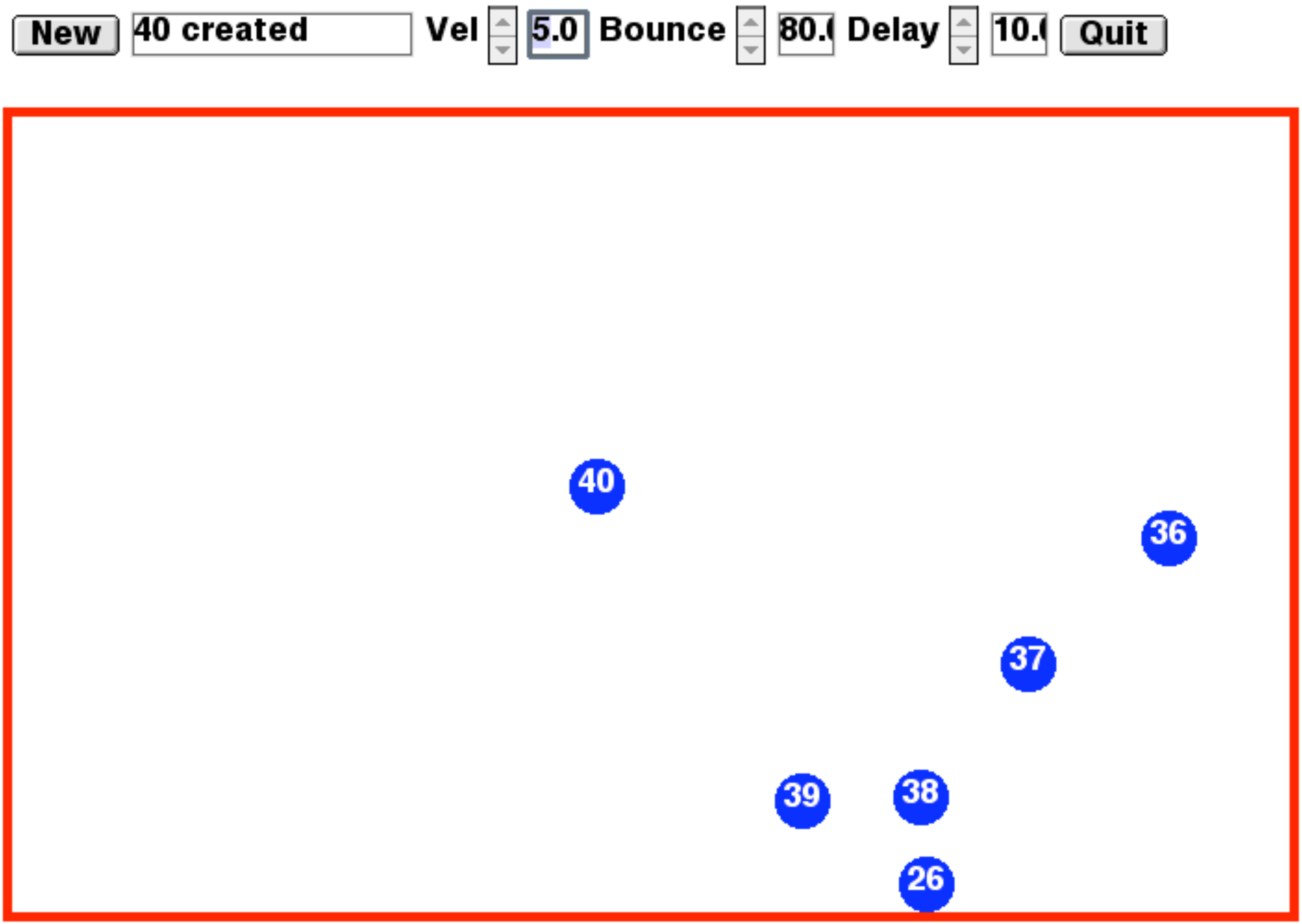
- May wish to have multiple activities going on at once.
- Don't want one activity's waiting (e.g. for an event) to stop the other activities.
- Example: On a user's desktop, there appear to be running simultaneously:
  - Several application programs:
    - A text editor
    - A browser, with several things going on
      - A couple of searches
      - An applet
      - A YouTube video
- This is **only** doable on a 1-processor system with threads (or processes).

Thread Example

- On thread might be a **computational** one, that occasionally needs to wait for input from the outside, say from an input stream of characters.
- Another thread might be a **graphical user interface**, responding to mouse events.
- We don't want **waiting** for input to hold up the graphics, or waiting for a click to hold up the computational thread.
- In fact, the click might tell the computational thread to alter its behavior.

Bouncing Balls Example

- Each ball is run by a separate thread.
- (This is for illustration. It is likely not the way you'd do a video game, because you want more precise control over timing and interactions.)
- Each thread can, in principal, be **interrupted** and re-started independently of the others.
- If a ball is "clicked" in mid-air, it will suspend, and resume if clicked a second time.


New
40 created
Vel
5.0
Bounce
80.(
Delay
10.(
Quit
40
36
37
39
38
26

Two Ways to Have Threads in Java

- extends Thread
  - Thread is a *base class* with threading capability.
- implements Runnable
  - Runnable is an *interface* that requires method
    - void run()
- **The latter is preferred**, because it does not take away the ability to inherit from another class (multiple inheritance is not allowed in Java).

Using “implements Runnable”

- The class that *implements* Runnable *still* needs to ***contain*** a Thread as an instance variable.
- This Thread is what controls starting and stopping.

Ball "extends Thread" Code

```
/**
 * Ball class represents ball's state information
 */

class Ball extends Thread // vs. Thread implements Runnable
  {
  double x, y;                                 // this ball's coordinates
  double deltaX, deltaY;                       // this ball' velocities
  String myNumber;                             // ball's number as a string

  public Ball(...) // constructor {}

  /**
   * over-ride run() method in parent class (Thread)
   */

  public void run()
    {
    while( true )
      {
      move();                                  // move the ball
      sleep(app.delay);                        // sleep (defined in Thread)
      }
    }
```

Ball "implements Runnable" Code

```
class Ball implements Runnable
{
Thread myThread;            // this ball's thread
double x, y;                // this ball's coordinates
double deltaX, deltaY;      // this ball' velocities
String myNumber;            // ball's number as a string

Ball( . . .)                // constructor
  {
  ...
  myThread = new Thread(this);     // make thread for Ball
  }

public void run()                  // run method for this Runnable
    {
    while( true )
      {
      move();                      // move the ball
      myThread.sleep(app.delay);   // sleep
      }
    }
...
}
```

Cautions about Threads

- Reasoning about concurrent programs is inherently more difficult than reasoning about sequential ones.
- They can exhibit **non-deterministic** behavior, when variables are shared among threads.

Non-Determinism

Suppose x == 1 initially.

Thread 1

x = x+2;

What is x now?

Thread 2

x = x*5;

Language Aspects

- Prior to Java, many languages did not have threads as part of the language.
- Those that did were mostly research vehicles.
- Some had add-on libraries for threads (such as pthreads or Posix-threads).
- Java is the most widely-used example where threads are integral to the language.
- The JVM (Java Virtual Machine) is the interpreter for Java's byte-code. It runs the threads.

Interesting Methods of Thread

```
public void start()
```

Causes this thread to begin execution;
The JVM calls the run method of this thread.

The result is that *two threads are running concurrently*:

> the initiating thread (which returns from the call to the start method) and
>
> the initiated thread (which executes its run method).

Throws:
`IllegalThreadStateException` - if the thread was already started.

Methods of Thread

```
public static Thread currentThread()
```

Returns a reference to the currently executing thread object.

Note: "executing" is more specific than "running":

"executing" means "has the processor"

"running" means "able to execute",
but not necessarily executing

Methods of Thread

```
public static void yield()
```

Causes the currently executing thread object to pause temporarily and allow other threads to execute.

Methods of Thread

```
public static void sleep(long millisec)
                    throws InterruptedException
```

Causes the currently executing thread to sleep
(temporarily stop execution)
for the specified number of milliseconds.

Methods of Thread

```
public void interrupt()
```

Interrupts this thread.

Called by another thread having a reference to this one.

First the checkAccess method of this thread is invoked, which may cause a SecurityException to be thrown.

Methods of Thread

```
public final void setPriority(int newPriority)
```

Changes the priority of this thread.

First the checkAccess method of this thread is called with no arguments. This may result in throwing a SecurityException.

Otherwise, the priority of this thread is set to the smaller of the specified newPriority and the maximum permitted priority of the thread's thread group.

Methods of Thread

```
public final void join(long millisrv)
                throws InterruptedException
```

Waits at most millisec milliseconds for this thread to die.
A timeout of 0 means to wait forever.

Runnable

java.lang

Interface Runnable

Known Implementing Classes:
Thread, TimerTask

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called run.

This interface is designed to provide a common protocol for objects that wish to execute code while they are active. For example, Runnable is implemented by class Thread.
Being active simply means that a thread has been started and has not yet been stopped.

In addition, Runnable provides the means for a class to be active while not subclassing Thread.
A class that implements Runnable can run without subclassing Thread by instantiating a Thread instance and passing itself in as the target.

In most cases, the Runnable interface should be used if you are only planning to override the run() method and no other Thread methods.
This is important because classes should not be subclassed unless the programmer intends on modifying or enhancing the fundamental behavior of the class.

re. Applets

- As you have seen, applets implement Runnable.
- This is in part so the applet can carry out two activities **concurrently**:
  - The main activity or activities of the applet
  - The event-listening activities that deal with user events such as pushing a button, etc.
  - The latter call user-supplied methods, known as **listeners** or **call-backs**, enabling communication with the main activity through common variables.

Memory Allocation and Recycling

O(1) Addressing

- (Assume no "paging" nor "caching" for now).
- Linear Address Space:
  Memory is effectively like a big array.
- Each word is accessible in the same amount of time.
  - A **decoder tree** in logic permits this.
  - The time bound for decoding is actually O(log n).
  - However, the clock interval is designed to be long enough so that it **practically** is O(1).

How Memory is Used

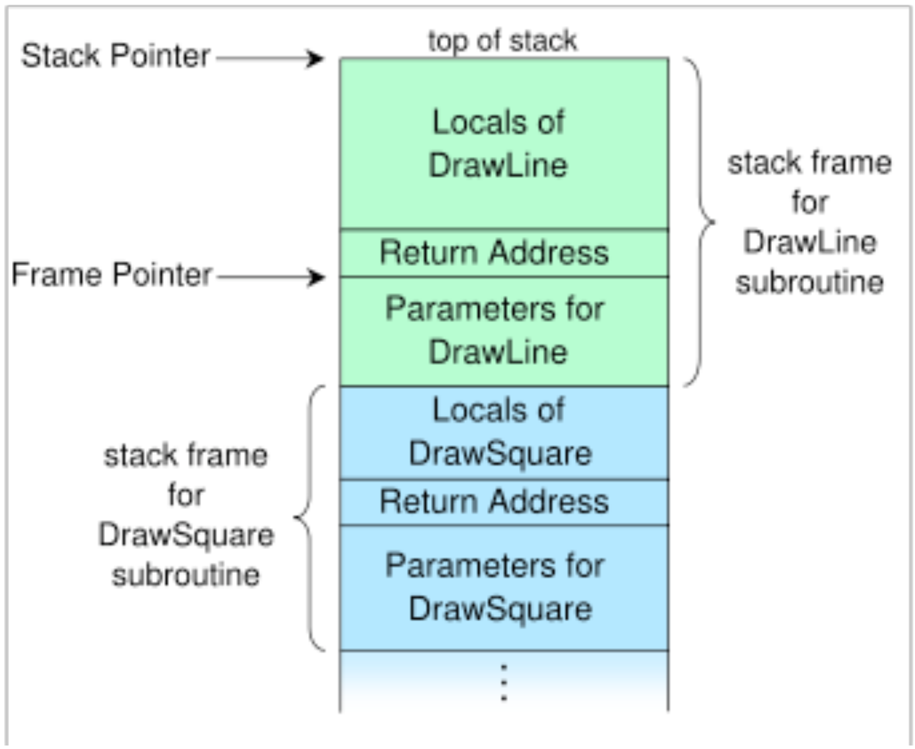
- Code
- Static variables:
  remain allocated throughout execution
- “Automatic” variables:
  e.g. arguments and local variables
  of nested functions
  These **cease to exist** after the
  function returns.
- Dynamic variables:
  instance variables of objects created
  during execution

Why “Automatic”?

- Automatic is not strictly necessary; dynamic could be used for it.
- However, due to **nested** calling discipline, reclamation of automatic is cheaper than dynamic in general.

Stack-Based Allocation

Low-overhead, but confining

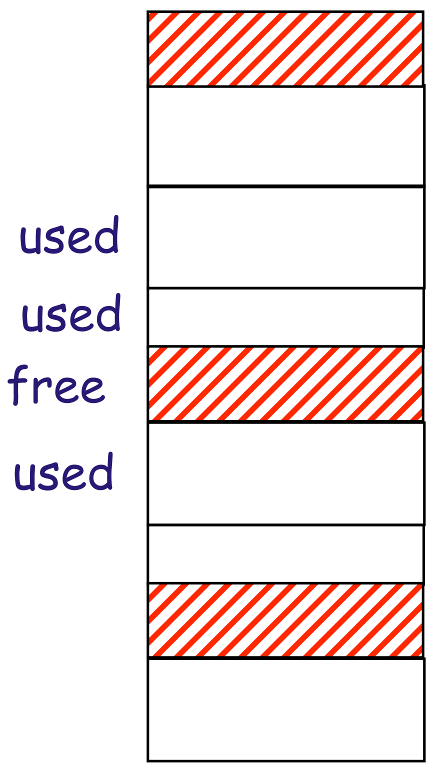


From http://en.wikipedia.org/wiki/Call_stack

Heap-Based Allocation

More flexibility, but more overhead

Heap:



Space overheads:

Each block stores a size.

Free blocks also store a pointer to the “next” free block.

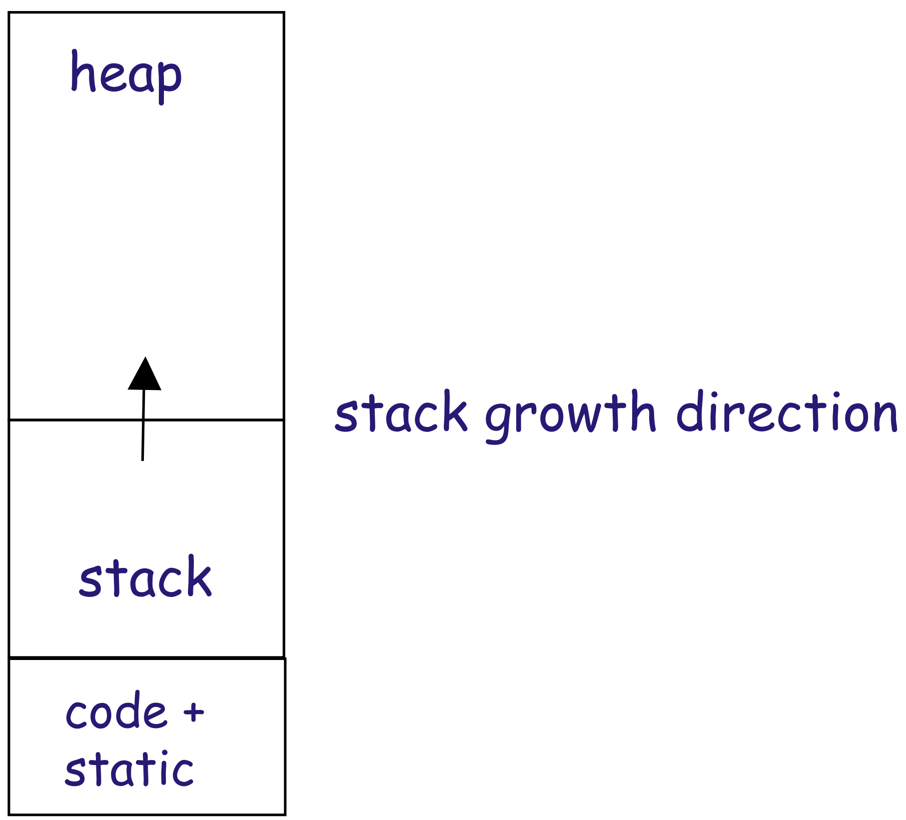
Time overheads:

Must *search* for an adequate free block.

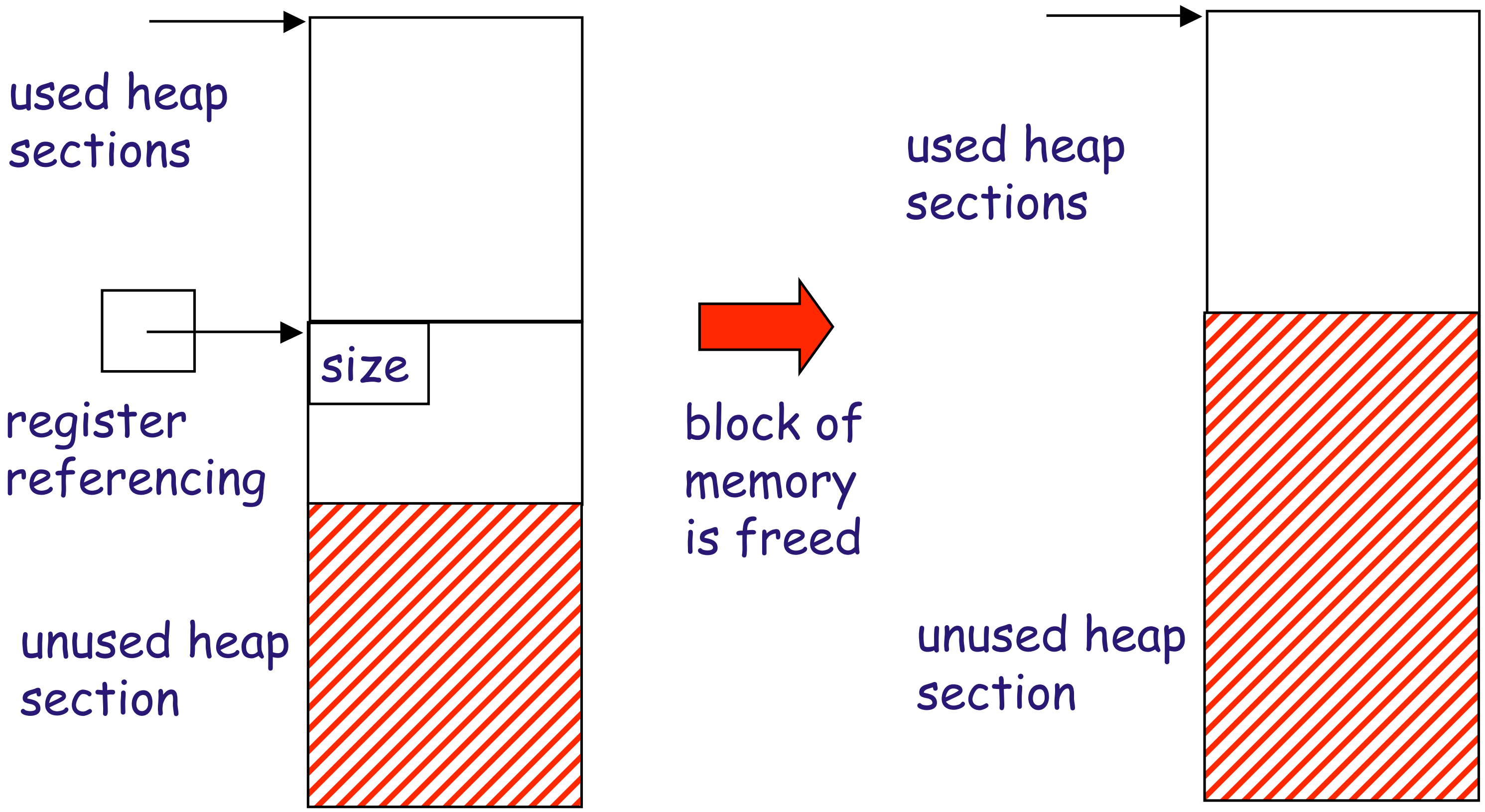
Must *sub-divide* free blocks that are bigger than requirement.

Must coalesce blocks as they become freed.

Memory is Pre-Divided

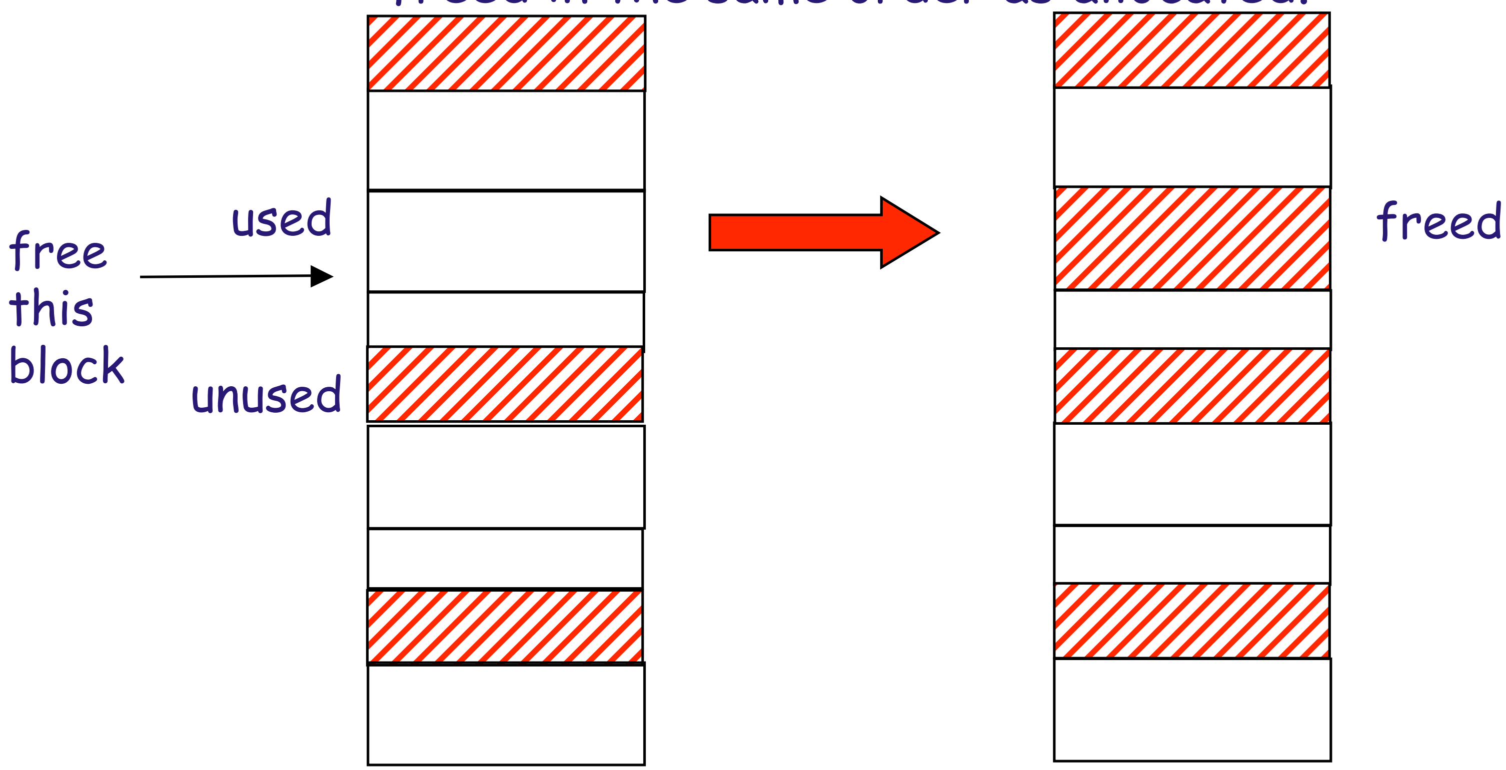


The Recycling Aspect



The Bigger Picture

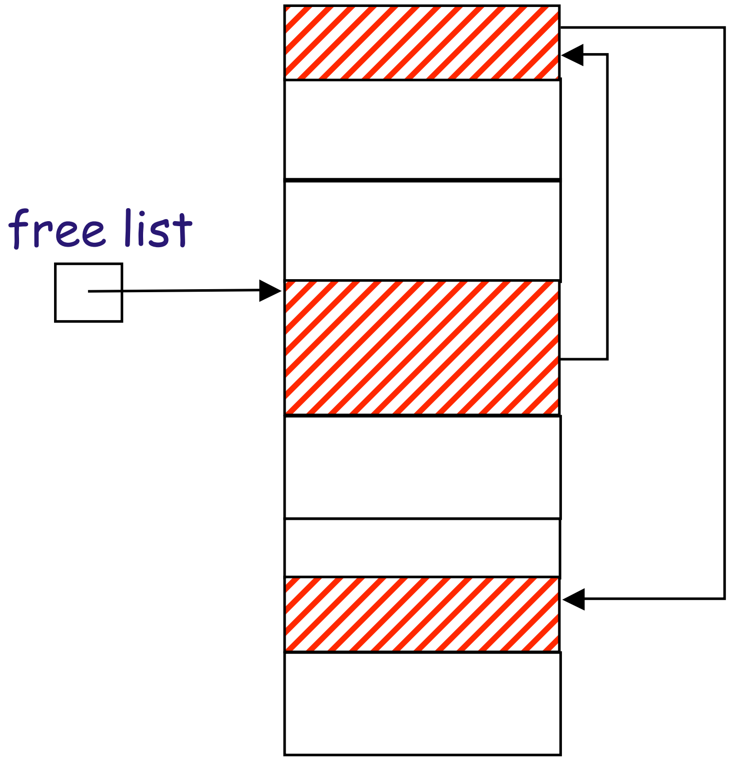
Memory becomes “fragmented” because blocks are not necessarily freed in the same order as allocated.



There are never **adjacent** unused blocks. They are always coalesced into one. (Why?)

Maintaining the "Free List"

Each unused block also has a **size** field.



Heap Issues

- Fragmentation (“checkerboarding”)
  - Why is this an issue?
- Allocation Policy:
  - First-fit
  - Best-fit
  - . . .

Approaches to Recycling Heap Memory

- Don't-do-it approach
- Programmer-burden approach
- Automatic approaches
  - Reference-Counting
  - Garbage collection
    - Mark-Sweep
    - Copying
    - Generational
    - others

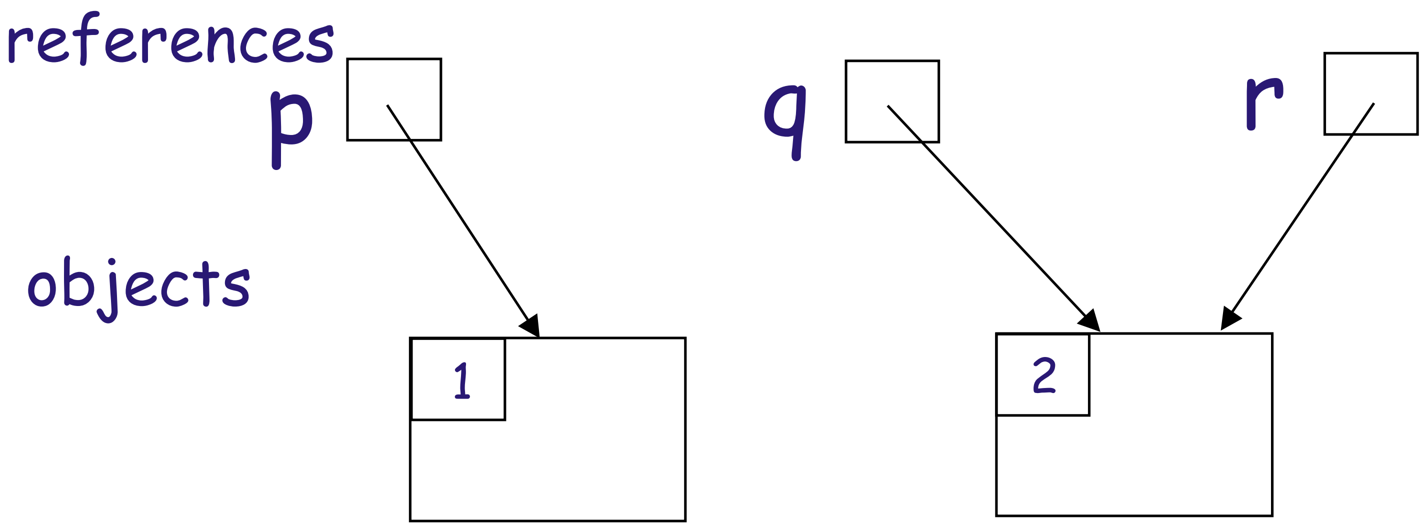
Reference Counting

- Each object has a reference count, not normally shown.
- An **invariant** is maintained:

  Reference count =

  # of references pointing to this object

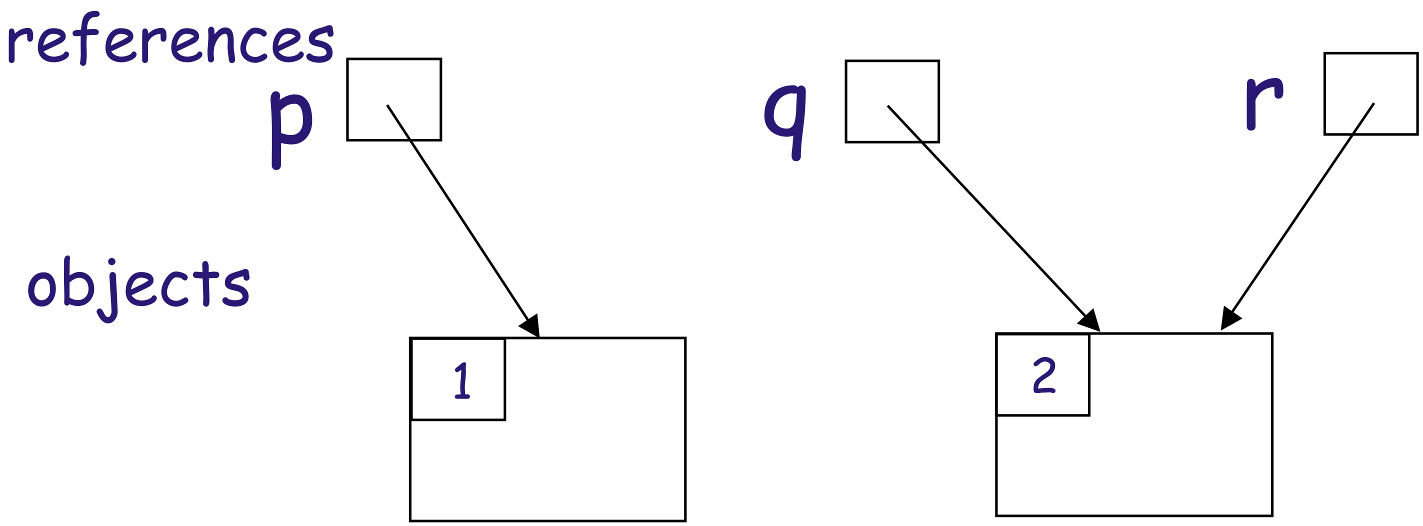
Reference Counting



What happens when we execute:

q = p; // make q point to where p points

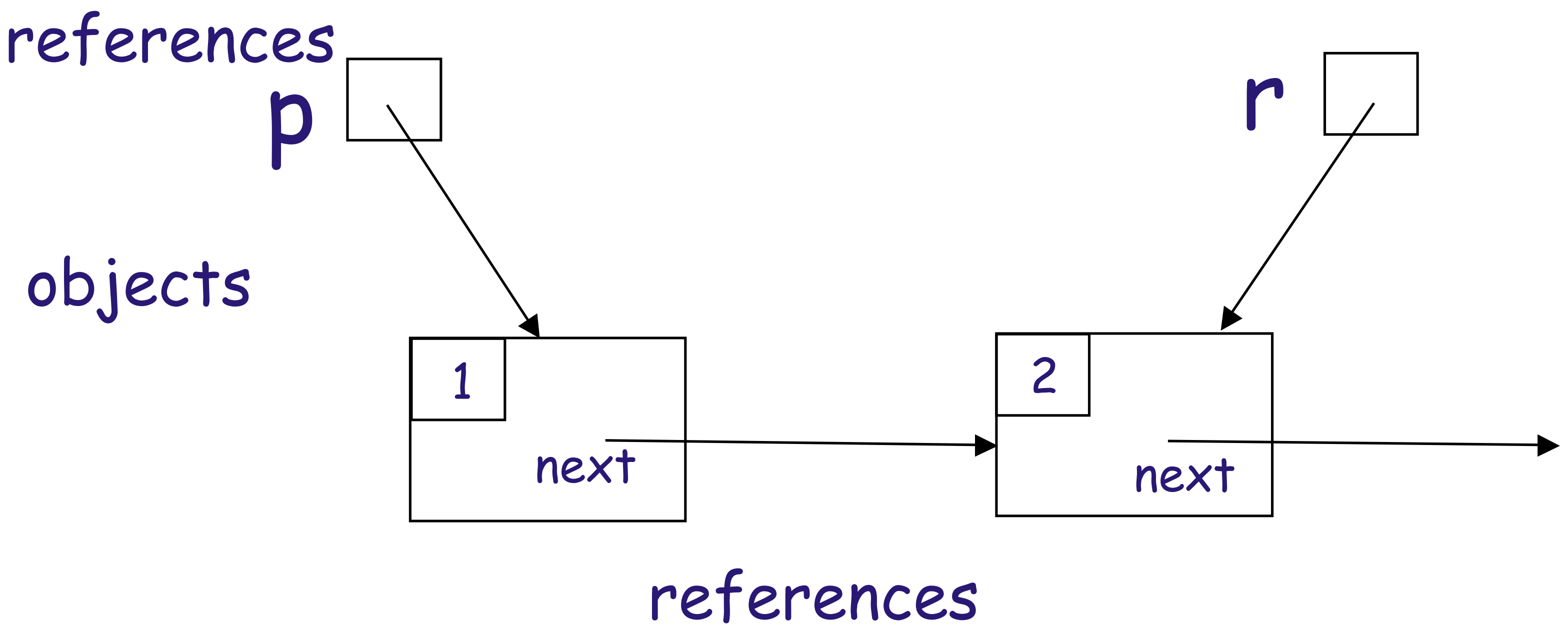
Reference Counting



What happens when we execute:

p = q;

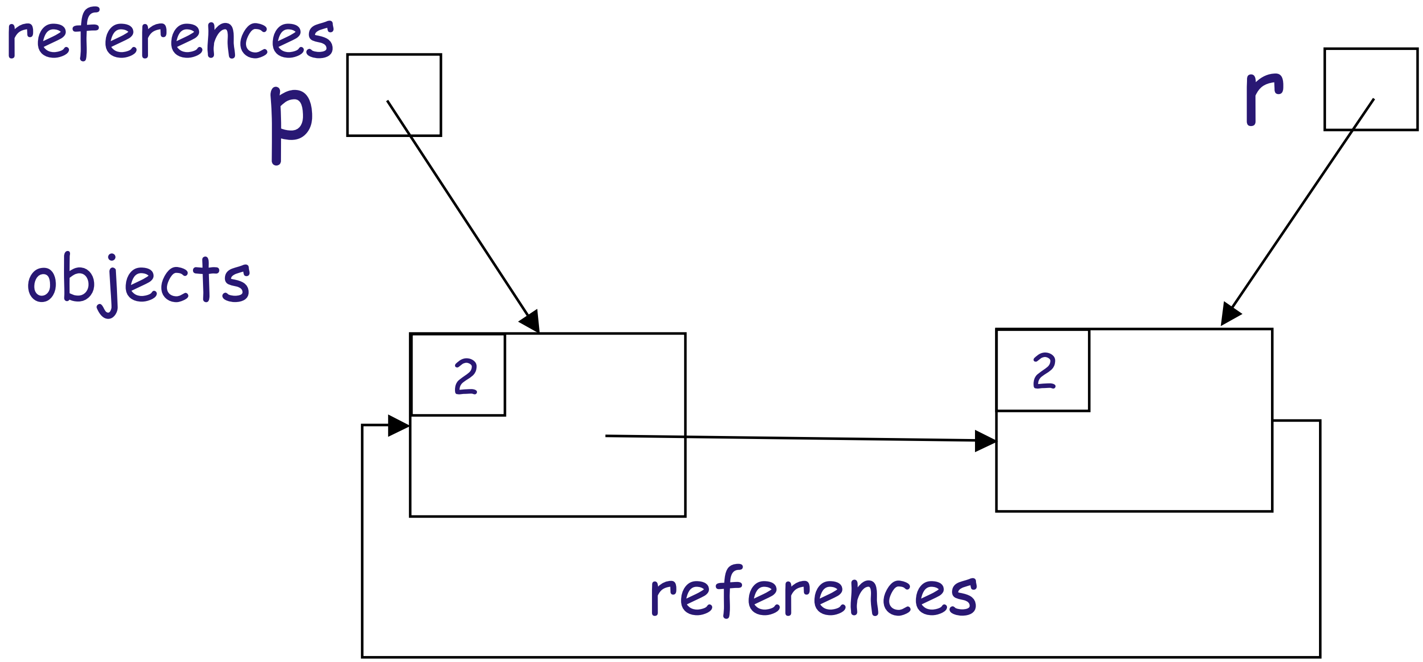
Linked List



What happens when we execute:

$$p = p.next;$$

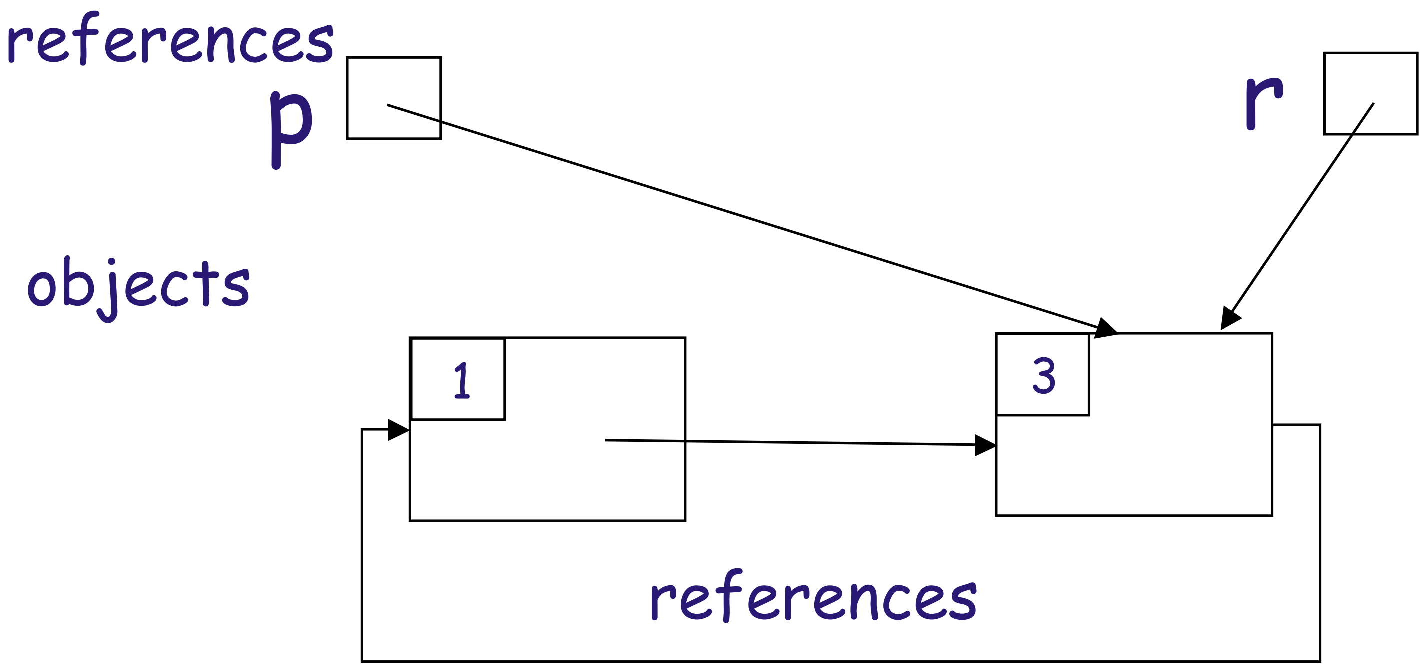
Circular Linked List



What happens when we execute:

$$p = r;$$

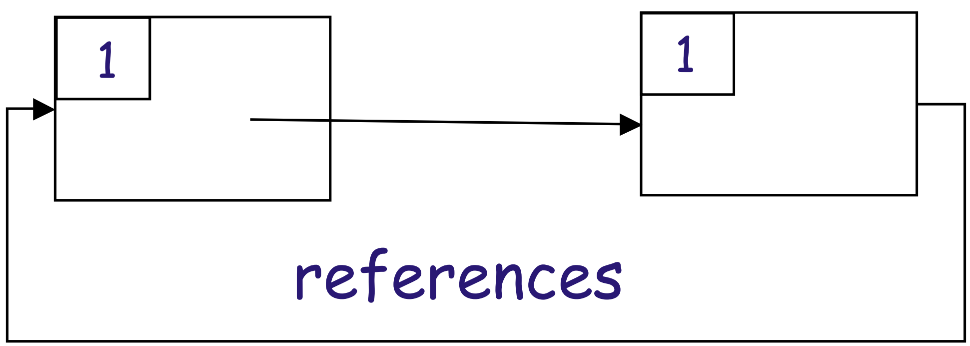
Circular Linked List



What happens when we execute:

```
p = null;
r = null;
```

Circular Linked List



Oops!

Garbage Collection (GC)

- Reference counting keeps track of what is accessible by modifying reference counts at each operation.
- Garbage collection does waits until memory is scarce, then determines what is accessible. The rest can become free memory.
- Garbage collection does not suffer from the cyclic problem of reference counting.
- GC was invented by John McCarthy in conjunction with early Lisp implementation.

Garbage Collection

- is essentially a graph-search problem.
- Determine what is accessible from one or more "roots" of a directed graph.
- The **complement** of accessible is inaccessible, i.e. garbage.

Some GC Techniques

- Mark/Sweep
- Copying
- Generational

Mark/Sweep Garbage Collection

- Do a search (say depth-first) from the roots of the “memory graph”.
- Mark any reachable nodes as you go.
- (By making a pass over *all* nodes linearly through memory) Sweep up any unmarked nodes into the free list.
- (This all supposes that node entities are clearly identifiable. It requires that memory be maintained appropriately.)

Memory Overhead Comparison

- Reference counting:
  - One integer per node
- Mark/sweep:
  - One bit per node

Time Overhead Comparison

- Reference counting:
  - A small tax on every operation
- Mark/sweep:
  - A big tax when memory runs out

Copying Garbage Collection

- Divide the memory into two **half-spaces**, say A and B.
- Work within one half-space (A or B) at any given time.
- Assuming using A now. When it comes time to collect garbage, perform a depth-first search, **copying** each used record from A to B. Then switch to using B.

Copying Advantages

- Trivial to **compact** as you copy. Relative locations do not get maintained.
- Caution: Cannot rely on any absolute addresses, because memory is generally **relocated**.
- This results in a single large unused chunk of memory on each collection.
- **Real-time** versions of copying have been devised (cf. Henry Baker article).

Copying Disadvantages?

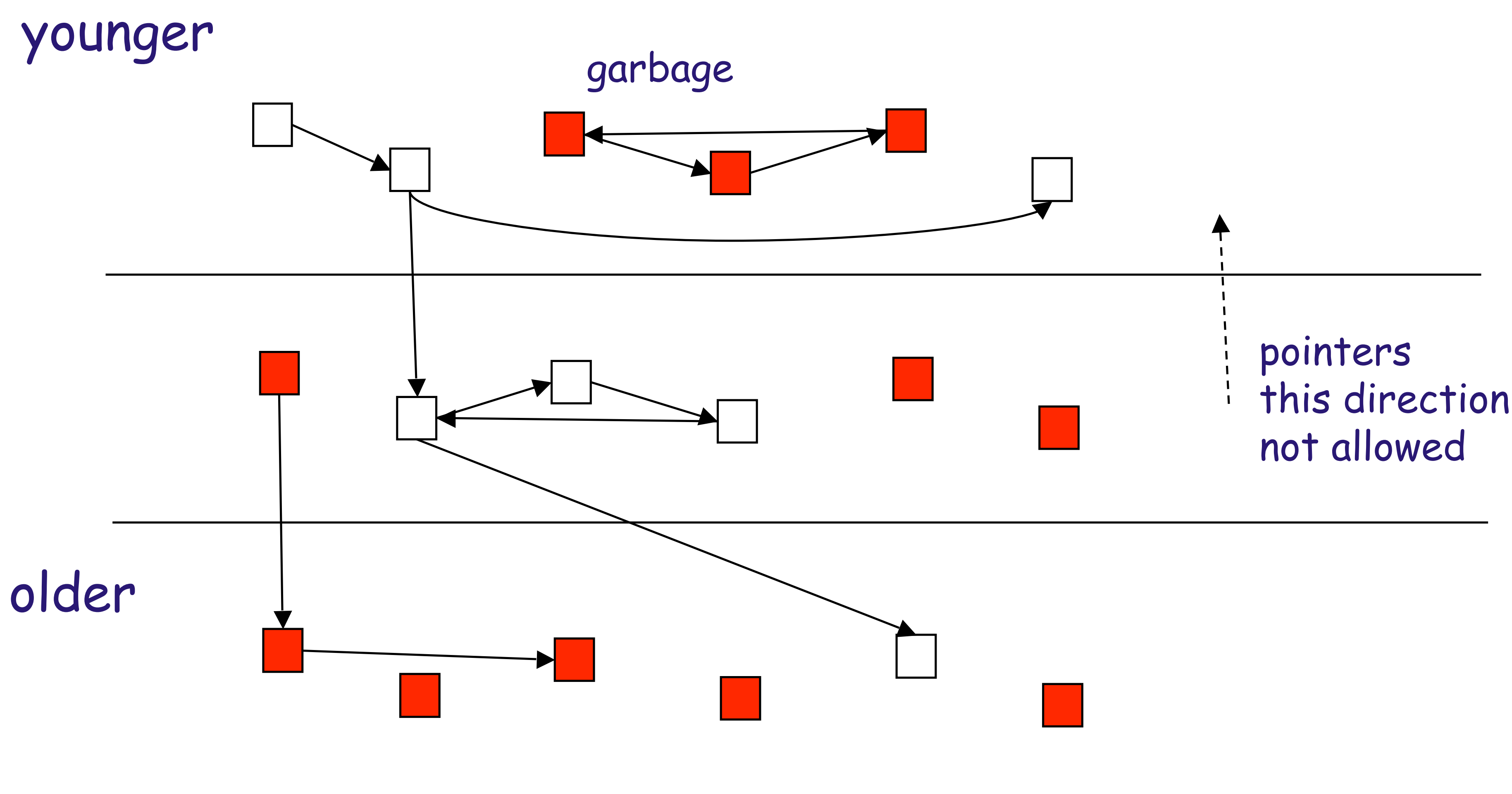
Generational Garbage Collection

- This is one of many heuristics used to reduce overhead in GC.
- It can be observed that some nodes are **ephemeral** (temporary and quickly become garbage), while others have great longevity.
- Thus devise a way to do a quick **partial** collection to pick up the ephemeral nodes, reserving a full GC until more desperate.

Generational Garbage Collection

- The extension of the dichotomy ephemeral vs. long-lived is achieved by assigning nodes to **generations**.
- A node in the **youngest** generation is viewed as the most likely to become garbage.
- References can point from a younger to an older generation, but not vice-versa. (Attempting to do so results in moving the target to the same generation as the source.)
- If a node survives collection at one generation, it is **promoted** to the next generation.
- One generation is collected only if there is not enough memory freed by collecting younger generations.

Generational Garbage Collection



Food for Thought
(or indigestion?)

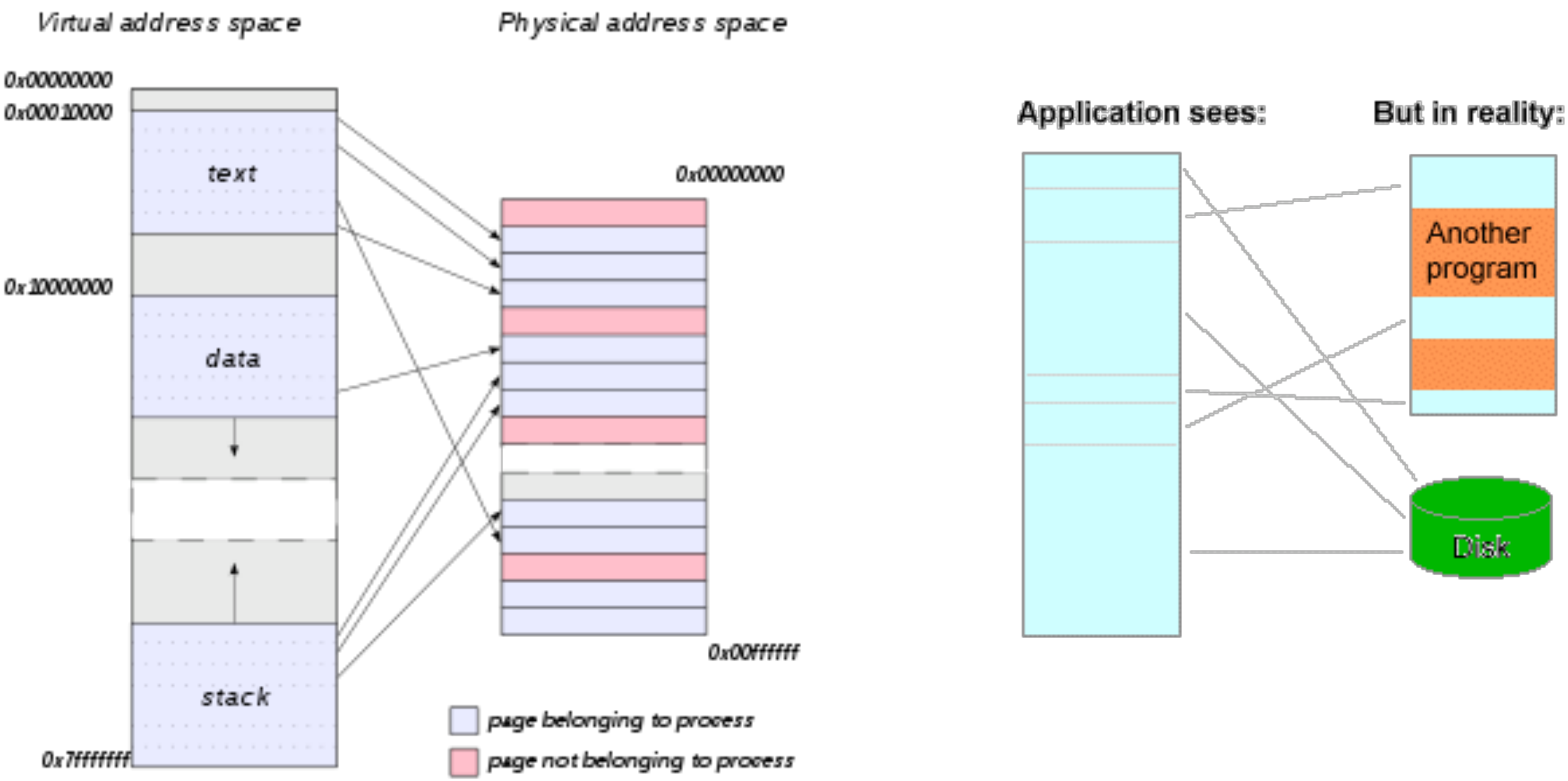
- How does the need for memory recycling impact the need to use threads?

More Considerations

- Approaches to making memory larger/faster:
  - Paging, Virtual Memory
  - Caching
  - Each comes with its own set of issues

Paging

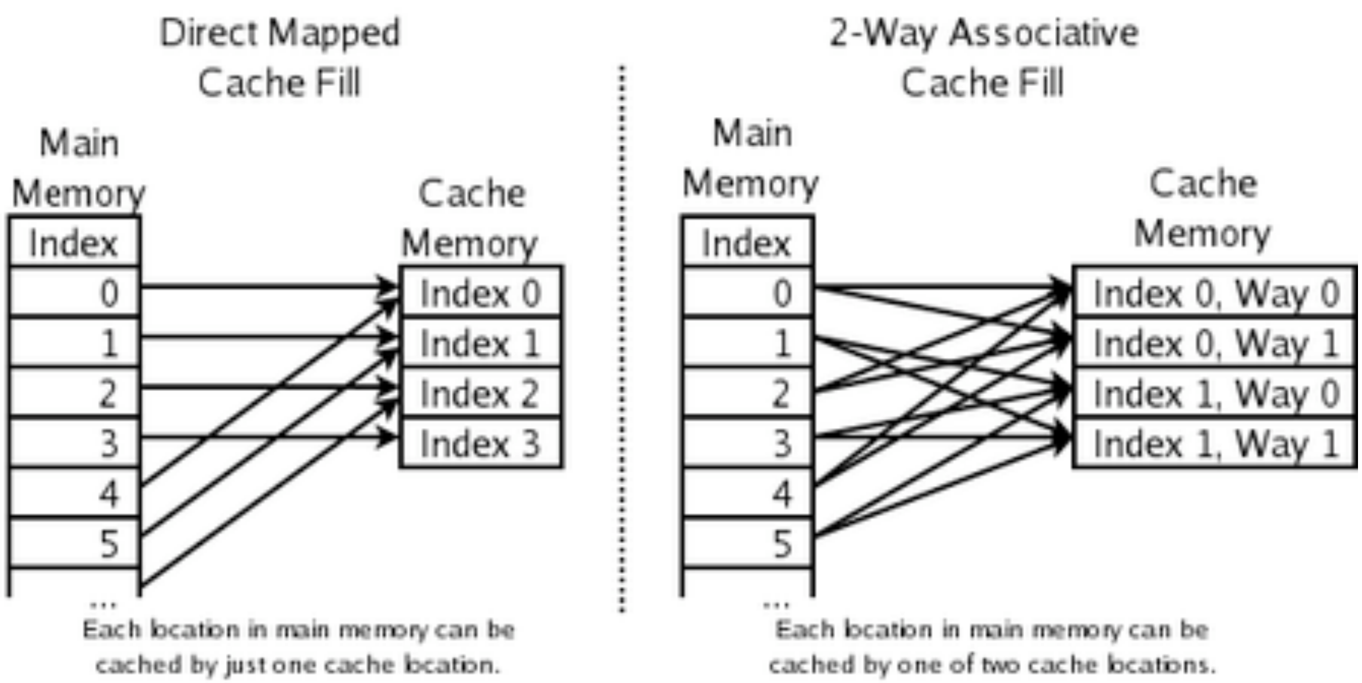
Purpose: Virtual memory, Sharing physical memory



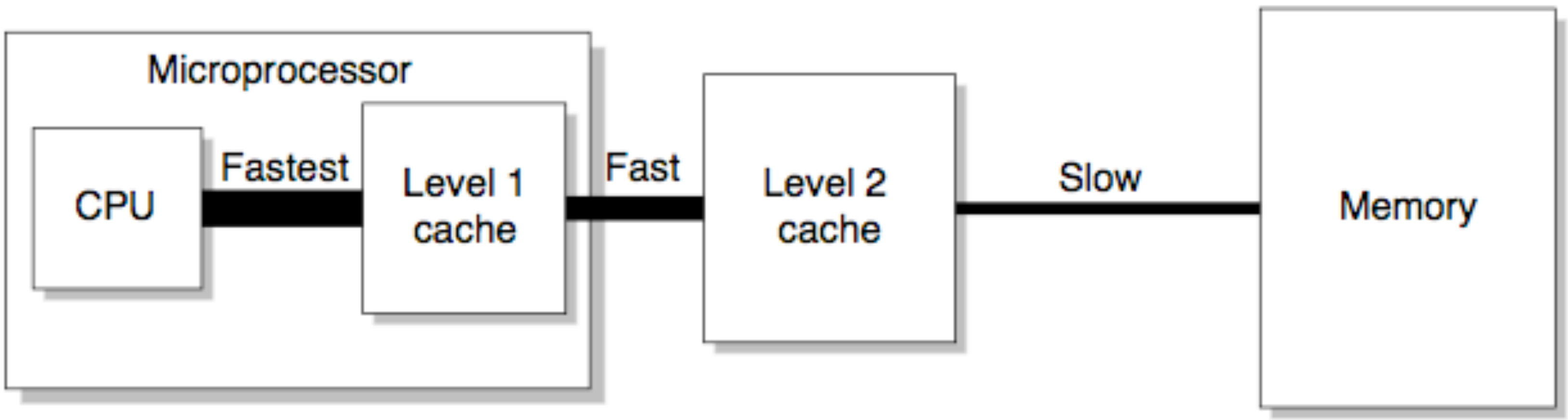
Caching

Purpose: Faster memory for faster execution

Analogous to hashing in some ways, but done in hardware.



Multi-Level Cache



Computer Memory Hierarchy

by Dan Lash (.com)