Please visit http://pingo.upb.de/643250

https://xkcd.com/138/

Five-Minute Review

1. What are *local/instance/class variables*? What about *constants*?
2. What is an *array*?
3. How do we locally declare an array of 5 integers?
4. How is the above array *stored*?
5. What are *multi-dimensional arrays*?

Five-Minute Review

1. What do `&`, `|`, `~`, `^` mean for integers?
2. What are typical uses for `&` and `|`?
3. What is a *generic class*?
4. What are **`ArrayLists`**, when should they be used?
5. What is a *pixel*, how is it encoded?

Programming – Lecture 8

Objects and Memory (**Chapter 7**)

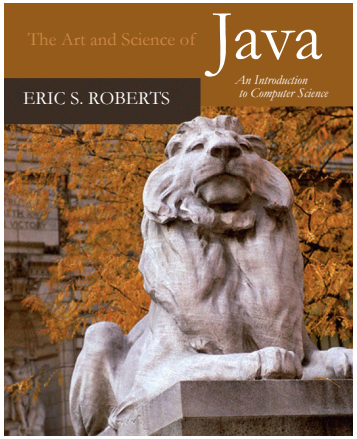
- Memory structure
- Allocation of memory to variables – Heap, Stack
- Recursion (for this only: **Chapter 14**)
- Linking objects together

CHAPTER 7

Objects and Memory

Yea, from the table of my memory
I'll wipe away all trivial fond records.

—William Shakespeare, *Hamlet,* c. 1600

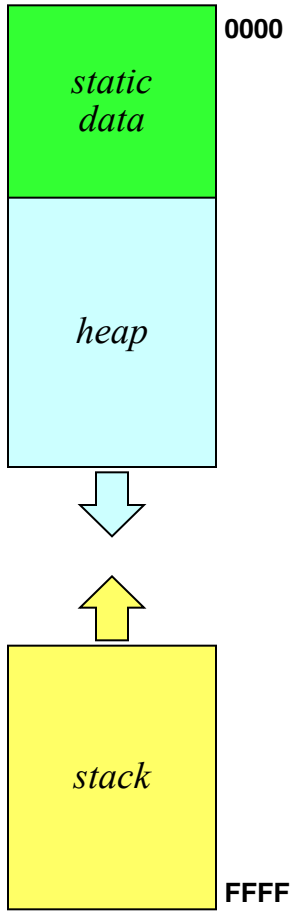


Memory and Addresses

- Every byte inside the primary memory of a machine is identified by a numeric address. The addresses begin at 0 and extend up to the number of bytes in the machine, as shown in the diagram on the right.
- In these slides as well as in the diagrams in the text, memory addresses appear as four-digit hexadecimal numbers, which makes addresses easy to recognize.
- In Java, it is impossible to determine the address of an object. Memory addresses used in the examples are therefore chosen completely arbitrarily.
- Memory diagrams that show individual bytes are not as useful as those that are organized into words. The revised diagram on the right now includes four bytes in each of the memory cells, which means that the address numbers increase by four each time.

The Allocation of Memory to Variables

- When you declare a variable in a program, Java allocates space for that variable from one of several memory regions.
- One region of memory is reserved for variables that are never created or destroyed as the program runs, such as named constants and other class variables. This information is called **static data**.
- Whenever you create a new object, Java allocates space from a pool of memory called the **heap**.
- Each time you call a method, Java allocates a new block of memory called a **stack frame** to hold its local variables. These stack frames come from a region of memory called the **stack**.
- In classical architectures, the stack and heap grow toward each other to maximize the available space.



Memory

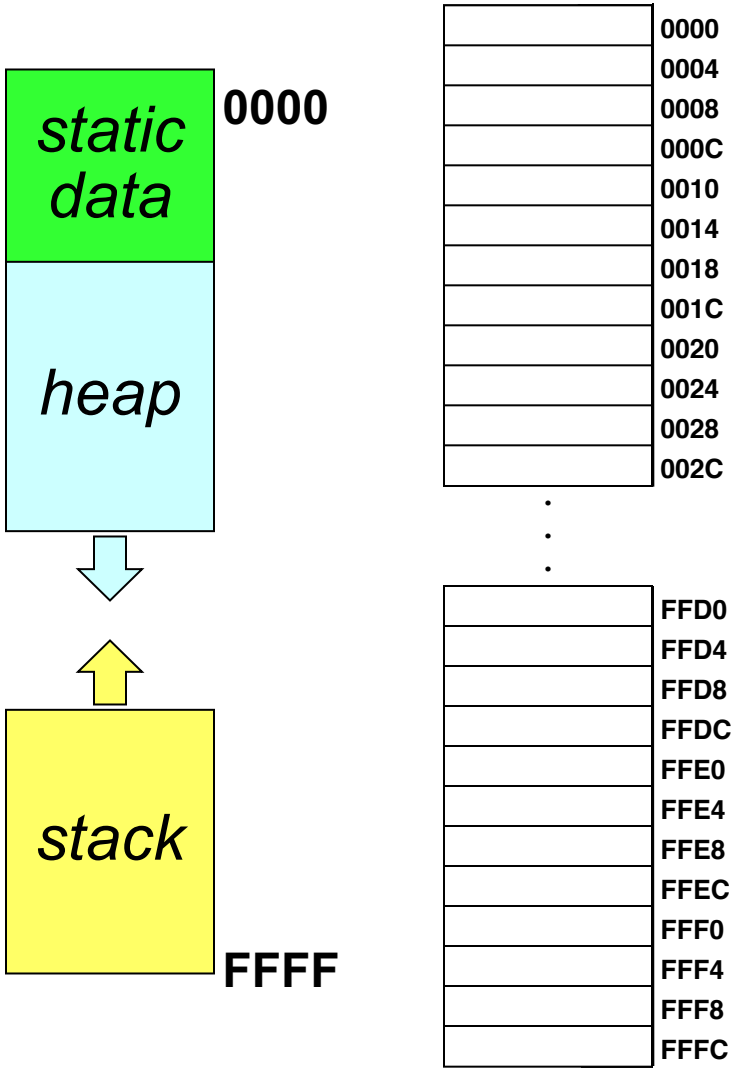
Bits, bytes, words

Variable storage

Static data: class var's

Heap: instance var's

Stack: local var's



Initialization

Automatic initialization to default value for

- Class var's
- Instance var's
- Array elements

Not for local var's!

Heap-Stack Diagrams

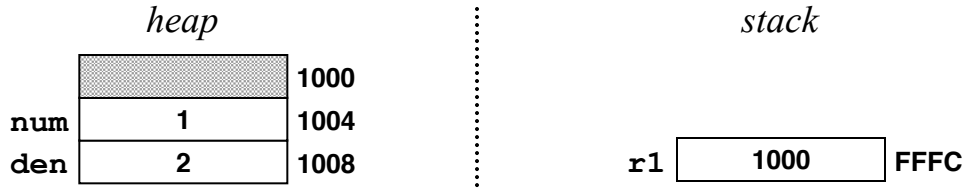
- It is easier to understand how Java works if you have a good mental model of its use of memory. The text illustrates this model using **heap-stack diagrams**, which show the heap on the left and the stack on the right, separated by a dotted line.
- Whenever your program creates a new object, you need to add a block of memory to the heap side of the diagram. That block must be large enough to store the instance variables for the object, along with some extra space, called **overhead**, that is required for any object. Overhead space is indicated in heap-stack diagrams as a crosshatched box.
- Whenever your program calls a method, you need to create a new stack frame by adding a block of memory to the stack side. For method calls, you need to add enough space to store the local variables for the method, again with some overhead information that tracks what the program is doing. When a method returns, Java reclaims the memory in its frame.

Object References

- Internally, Java identifies an object by its address in memory. That address is called a **reference**.
- As an example, when Java evaluates the declaration

```
Rational r1 = new Rational(1, 2);
```

  it allocates heap space for the new **`Rational`** object. For this example, imagine that the object is created at address 1000.
- The local variable **`r1`** is allocated in the current stack frame and is assigned the value 1000, which identifies the object.



- The next slide traces the execution of the **`TestRational`** program from Chapter 6 using heap-stack model.

Object References

- *Reference* of object: address where object is stored
- Object var's store object references
- Object var's *reference* objects, or *point to* objects
- In general, var's containing memory addresses are also referred to as *pointers*

```
Rational r1 = new Rational(1, 2);
```

Note: this assumes `r1` to be local var

A Complete Heap-Stack Trace

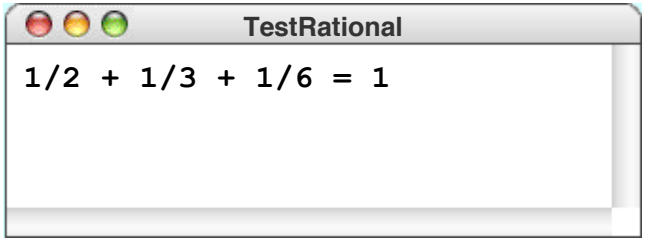
```
public void run() {
   Rational a = new Rational(1, 2);
   Rational b = new Rational(1, 3);
   Rational c = new Rational(1, 6);
   Rational sum = a.add(b).add(c);
   println(a + " + " + b + " + " + c + " = " + sum);
}
```

heap

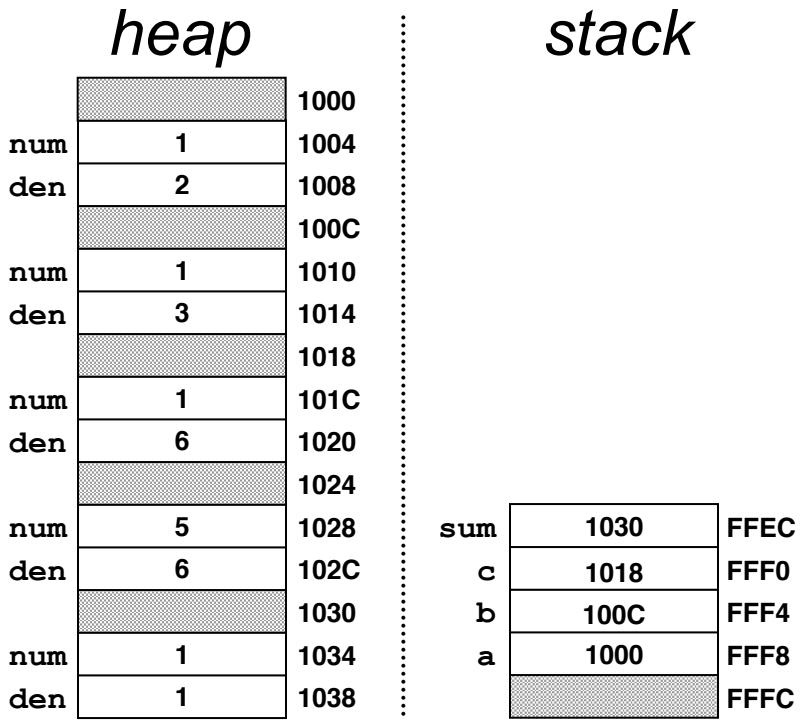
		1000
num	1	1004
den	2	1008
		100C
num	1	1010
den	3	1014
		1018
num	1	101C
den	6	1020
		1024
num	5	1028
den	6	102C
		1030
num	1	1034
den	1	1038

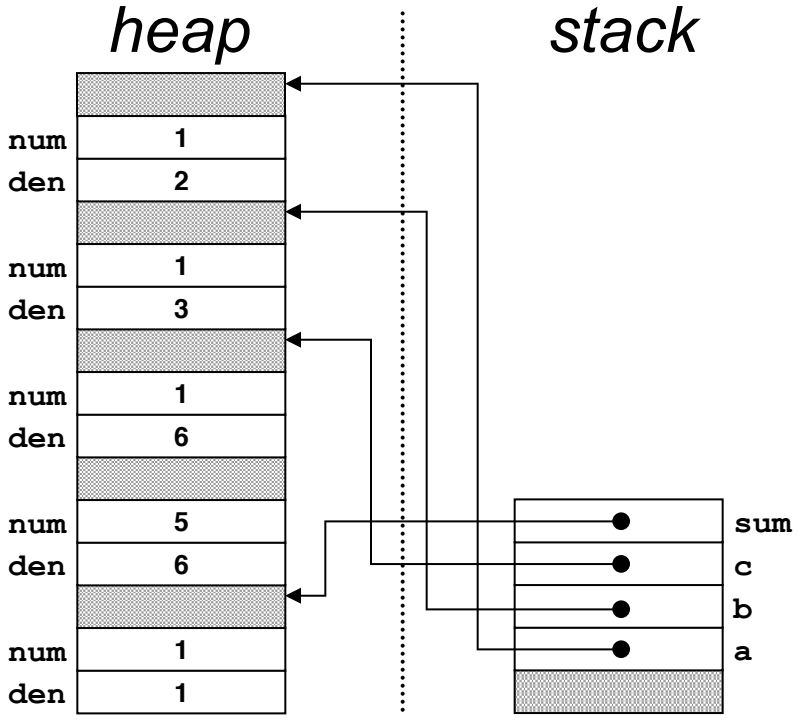
stack

sum	1030	FFEC
c	1018	FFF0
b	100C	FFF4
a	1000	FFF8
		FFFC



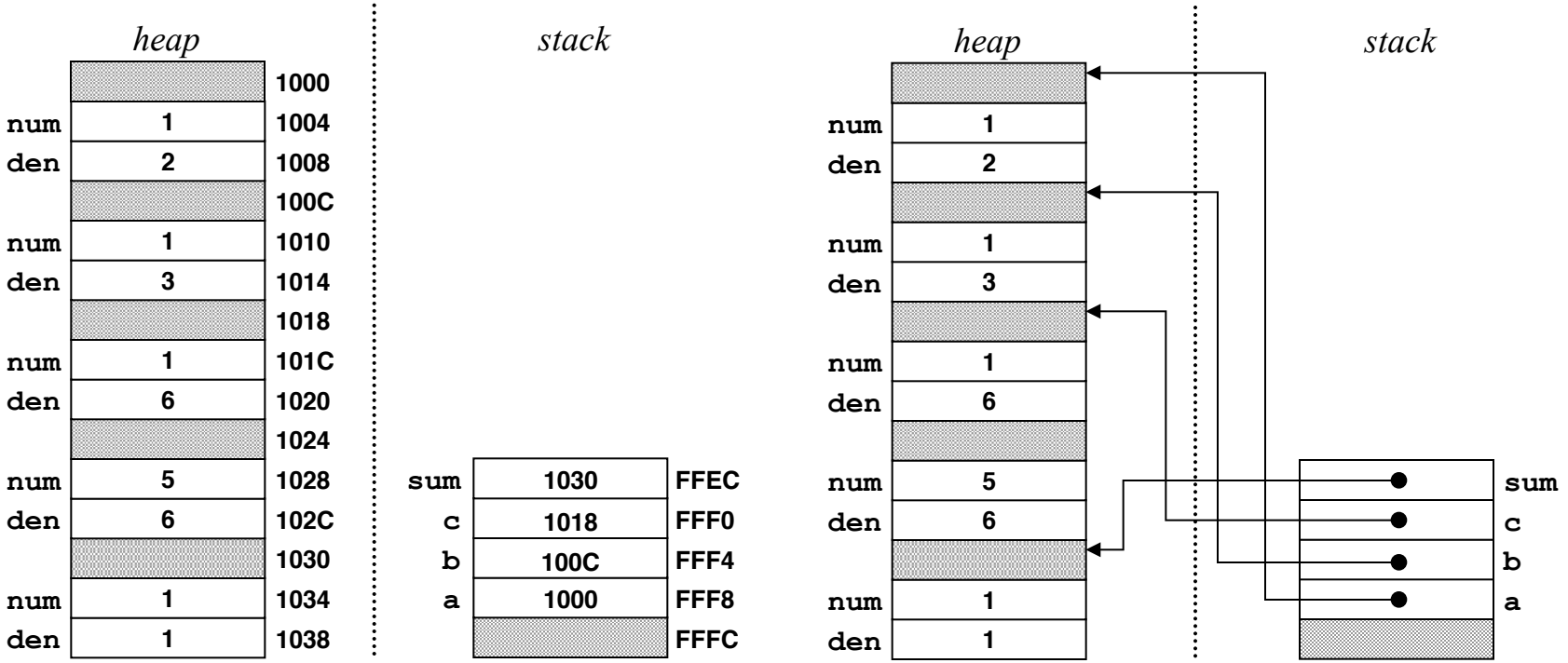
Address Model vs. *Pointer Model*



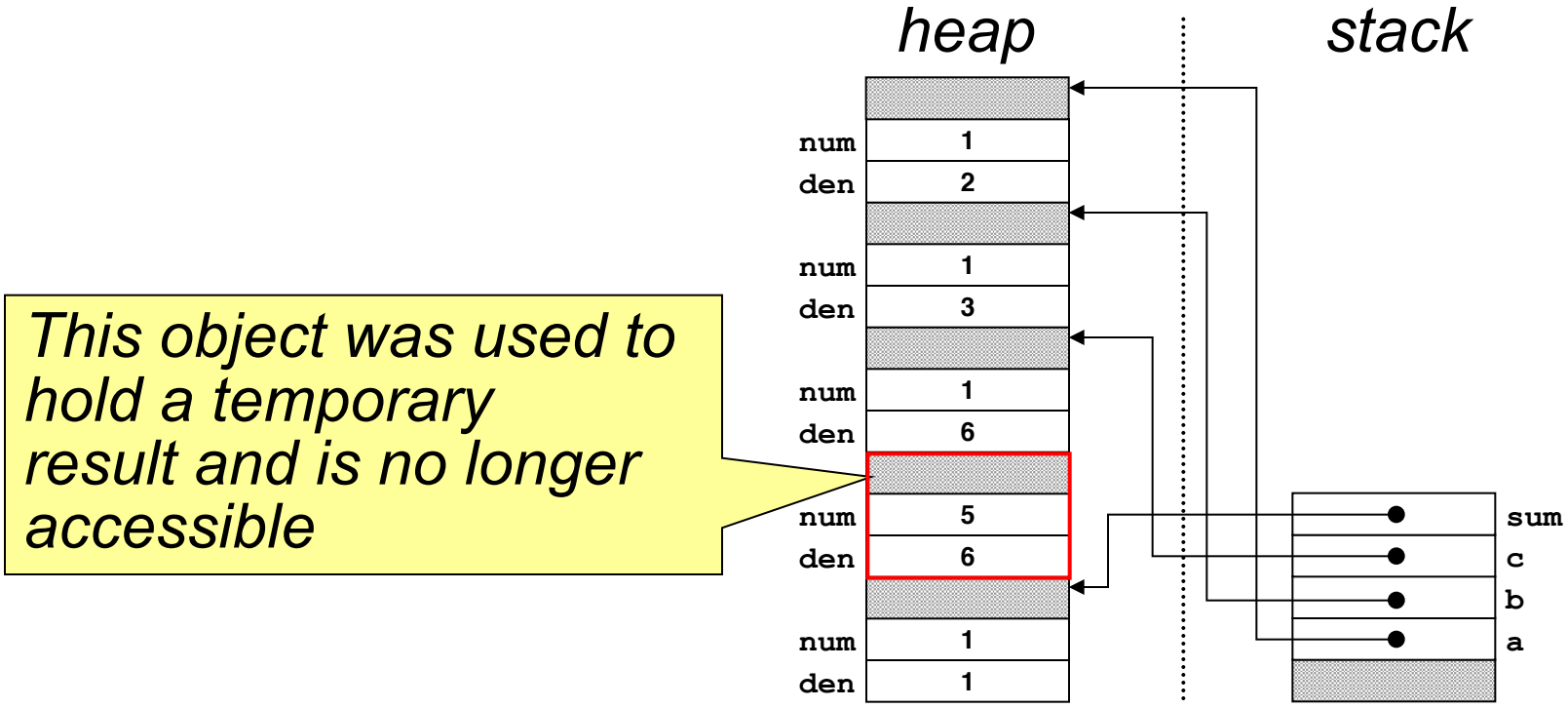


Addresses *vs.* Pointers

- The two heap-stack diagram formats—the address model and the pointer model—describe exactly the same memory state. The models, however, emphasize different things:
  - The address model makes it clear that references have numeric values.
  - The pointer model emphasizes the relationship between the reference and the object and makes the diagram easier to follow.



Garbage Collection



Mark-and-sweep collection, *in-use* flags

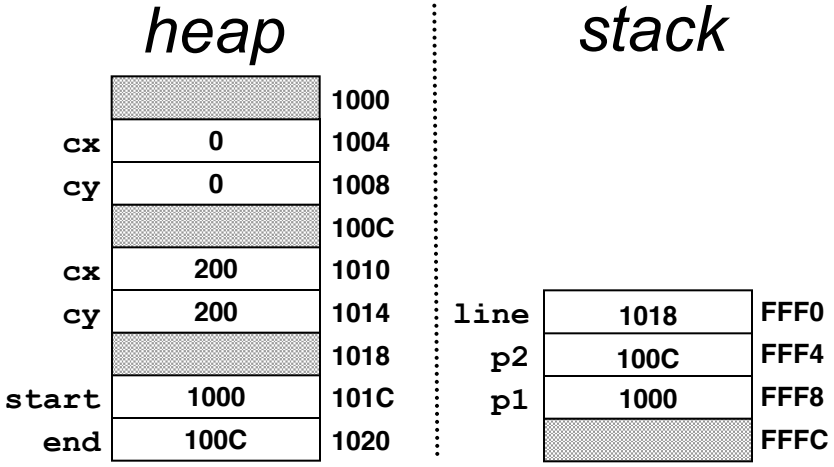
Exercise: Stack-Heap Diagrams

```
public class Point {
   public Point(int cx,
                int cy) {
      this.cx = cx;
      this.cy = cy;
   }

   . . . other methods appear here . . .

   private int cx;
   private int cy;
}
```

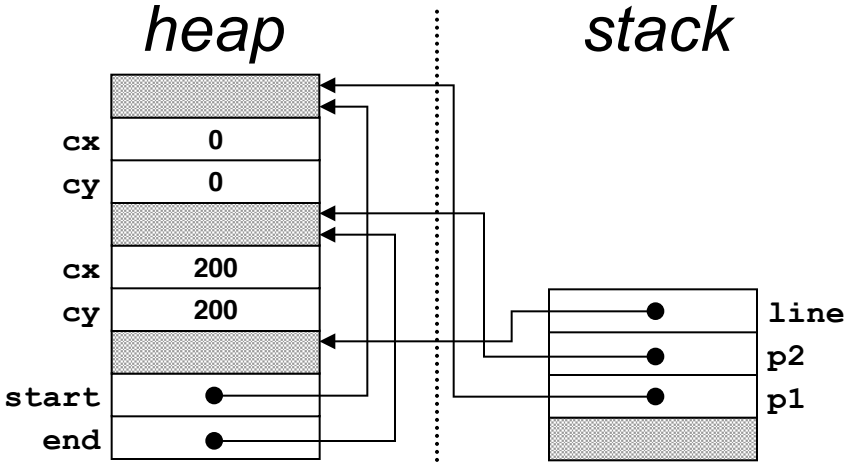
```
public class Line {
   public Line(Point start,
               Point end) {
      this.start = start;
      this.end = end;
   }

   . . . other methods appear here . . .

   private Point start;
   private Point end;
}
```

```
public void run() {
   Point p1 = new Point(0, 0);
   Point p2 = new Point(200, 200);
   Line line = new Line(p1, p2);
}
```

Address Model



Pointer Model



Recursion

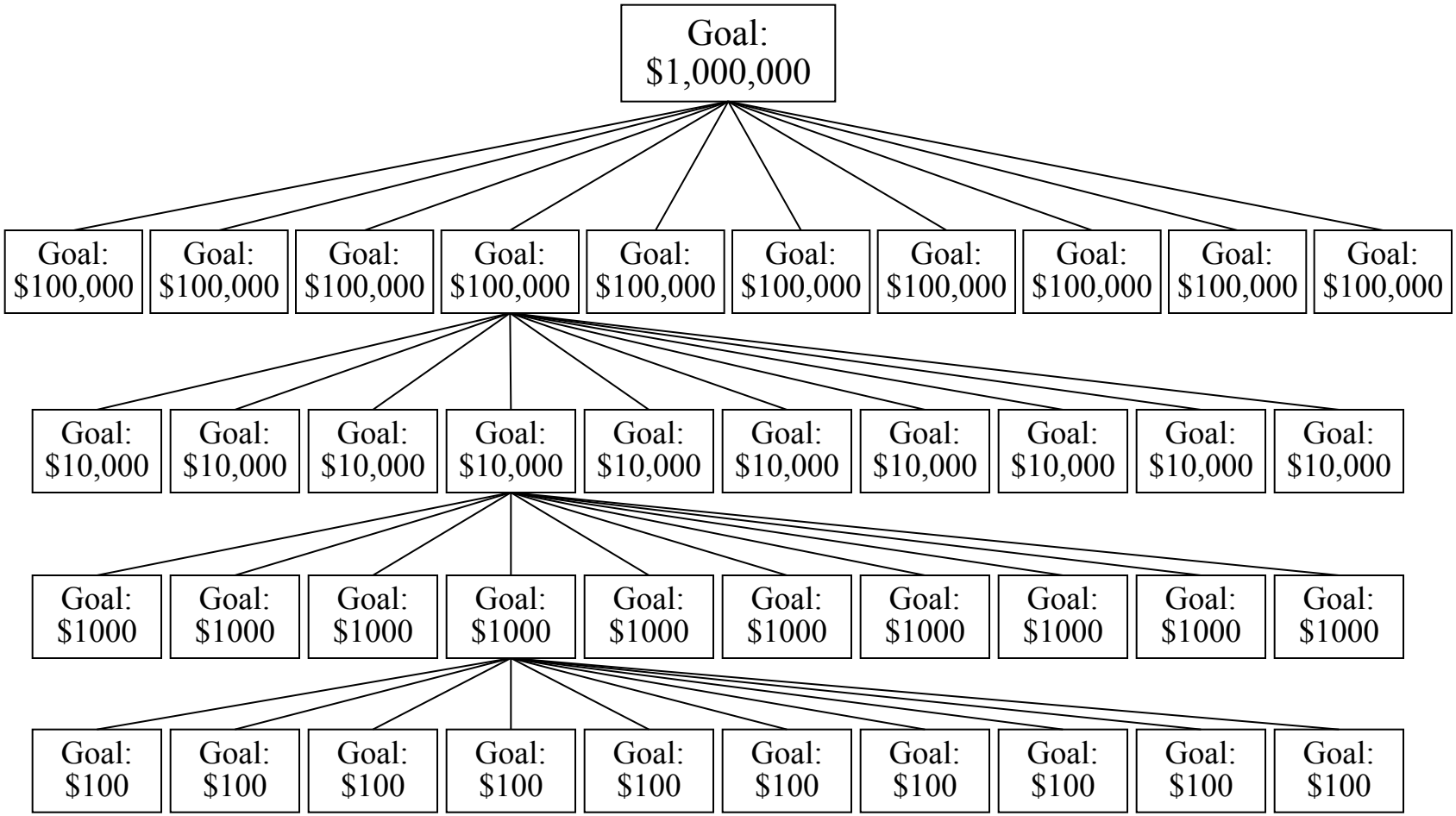
- Note: the slides on recursion are for Chapter 14 of the book (not contained in the on-line draft-book)
- One programming pattern made possible by the use of a dynamic stack for data associated with a method is recursion, which may involve multiple instances of the same method.
- **Recursion** is the process of solving a problem by dividing it into smaller subproblems *of the same form*. The italicized phrase is the essential characteristic of recursion; without it, all you have is a description of stepwise refinement as discussed in Chapter 5.
- The fact that recursive decomposition generates subproblems that have the same form as the original problem means that recursive programs will use the same method to solve subproblems at different levels of the solution. In terms of the structure of the code, the defining characteristic of recursion is having methods that call themselves, directly or indirectly, as the decomposition process proceeds.

A Simple Illustration of Recursion

- Suppose that you are the national fundraising director for a charitable organization and need to raise $1,000,000.
- One possible approach is to find a wealthy donor and ask for a single $1,000,000 contribution. The problem with that strategy is that individuals with the necessary combination of means and generosity are difficult to find. Donors are much more likely to make contributions in the $10 range.
- Another strategy would be to ask 100,000 friends for $10 each. Unfortunately, most of us don't have 100,000 friends.
- There are, however, more promising strategies. You could, for example, find ten regional coordinators and charge each one with raising $100,000. Those regional coordinators could in turn delegate the task to local coordinators, each with a goal of $10,000, continuing the process reached a manageable contribution level.

A Simple Illustration of Recursion

The following diagram illustrates the recursive strategy for raising $1,000,000 described on the previous slide:



A Pseudocode Fundraising Strategy

If you were to implement the fundraising strategy in the form of a Java method, it would look something like this:

```
private void collectContributions(int n) {
    if (n <= 100) {
        Collect the money from a single donor.
    } else {
        Find 10 volunteers.
        Get each volunteer to collect n/10 dollars.
        Combine the money raised by the volunteers.
    }
}
```

What makes this strategy recursive is that the line

Get each volunteer to collect n/10 dollars.

will be implemented using the following recursive call:

```
collectContributions(n / 10);
```

Recursive Functions

- The easiest examples of recursion to understand are functions in which the recursion is clear from the definition. As an example, consider the factorial function from Chapter 5, which can be defined in either of the following ways:

$$n! = n \times (n - 1) \times (n - 2) \times \cdots \times 3 \times 2 \times 1$$

$$n! = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ n \times (n - 1)! & \text{otherwise} \end{cases}$$

- The second definition leads directly to the following code, which is shown in simulated execution on the next slide:

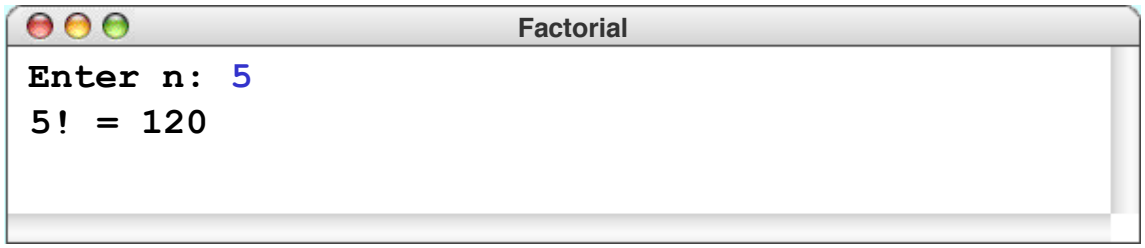
```
private int factorial(int n) {
   if (n == 0) {
      return 1;
   } else {
      return n * factorial(n - 1);
   }
}
```

Simulating the `factorial` Method

```
public void run() {
 private int factorial(int n) {
  private int factorial(int n) {
   private int factorial(int n) {
    private int factorial(int n) {
     private int factorial(int n) {
      private int factorial(int n) {
         if (n == 0) {
            return 1;
         } else {
            return n * factorial(n - 1);
         }
      }
```

n

0



skip simulation

The Recursive "Leap of Faith"

- The purpose of going through the complete decomposition of the calculation of `factorial(5)` is to convince you that the process works and that recursive calls are in fact no different from other method calls, at least in their internal operation.
- The danger with going through these details is that it might encourage you to do the same when you write your own recursive programs. As it happens, tracing through the details of a recursive program almost always makes such programs harder to write. Writing recursive programs becomes natural only after you have enough confidence in the process that you don't need to trace them fully.
- As you write a recursive program, it is important to believe that any recursive call will return the correct answer as long as the arguments define a simpler subproblem. Believing that to be true—even before you have completed the code—is called the **recursive leap of faith**.

The Recursive Paradigm

- Most recursive methods you encounter in an introductory course have bodies that fit the following general pattern:

```
if (test for a simple case) {
    Compute and return the simple solution without using recursion.
} else {
    Divide the problem into one or more subproblems that have the same form.
    Solve each of the problems by calling this method recursively.
    Return the solution from the results of the various subproblems.
}
```

- Finding a recursive solution is mostly a matter of figuring out how to break it down so that it fits the paradigm. When you do so, you must do two things:
  1. Identify **simple cases** that can be solved without recursion.
  2. Find a **recursive decomposition** that breaks each instance of the problem into simpler subproblems of the same type, which you can then solve by applying the method recursively.

Exercise: A Recursive `gcd` Function

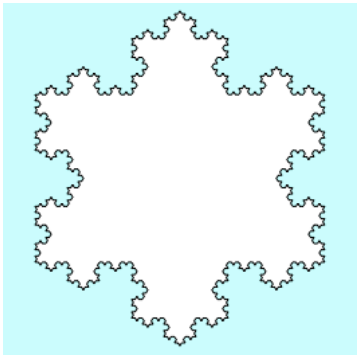
In the discussion of algorithmic methods in Chapter 5, one of the primary examples was Euclid's algorithm for computing the greatest common divisor of two integers, *x* and *y*. Euclid's algorithm can be implemented using the following code:

```
public int gcd(int x, int y) {
   int r = x % y;
   while (r != 0) {
      x = y;
      y = r;
      r = x % y;
   }
   return y;
}
```

Rewrite this method so that it uses recursion instead of iteration, taking advantage of Euclid's insight that the greatest common divisor of *x* and *y* is also the greatest common divisor of the *y* and the remainder of *x* divided by *y*.

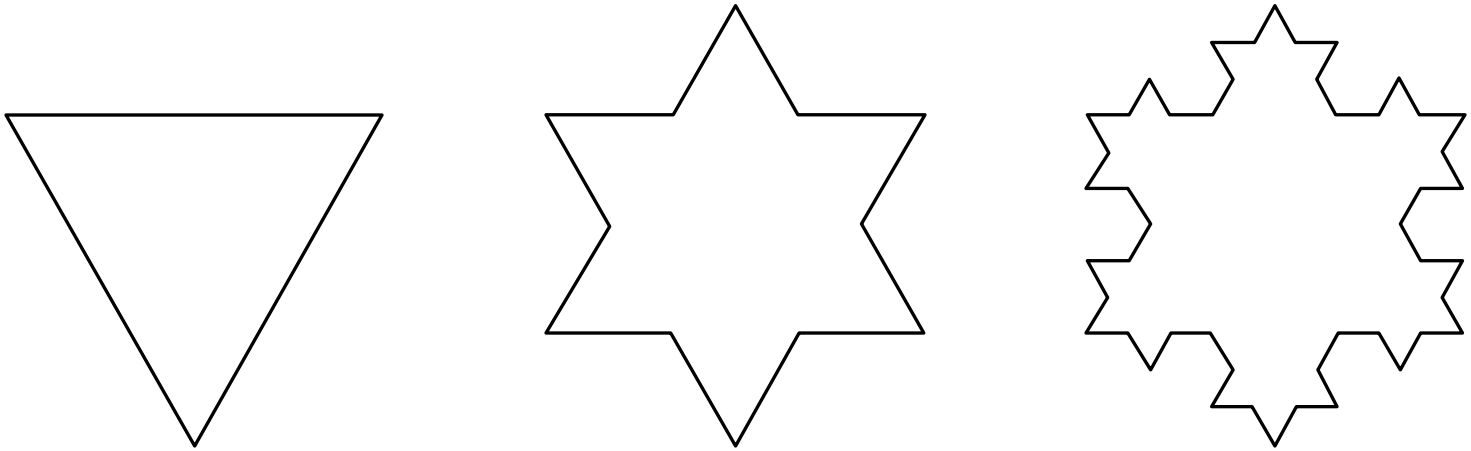
Graphical Recursion

- Recursion comes up in certain graphical applications, most notably in the creation of **fractals**, which are mathematical structures that consist of similar figures repeated at various different scales. Fractals were popularized in a 1982 book by Benoit Mandelbrot entitled *The Fractal Geometry of Nature*.
- One of the simplest fractal patterns to draw is the **Koch fractal**, named after its inventor, the Swedish mathematician Helge von Koch (1870-1924). The Koch fractal is sometimes called a **snowflake fractal** because of the beautiful, six-sided symmetries it displays as the figure becomes more detailed. as illustrated in the following diagram:



Drawing Koch Fractals

- The process of drawing a Koch fractal begins with an equilateral triangle, as shown in the diagram on the lower left.
- From the initial position (which is called a fractal of **order 0**), each higher fractal order is created by replacing each line segment in the figure by four segments that connect the same endpoints but include an equilateral wedge in the middle.



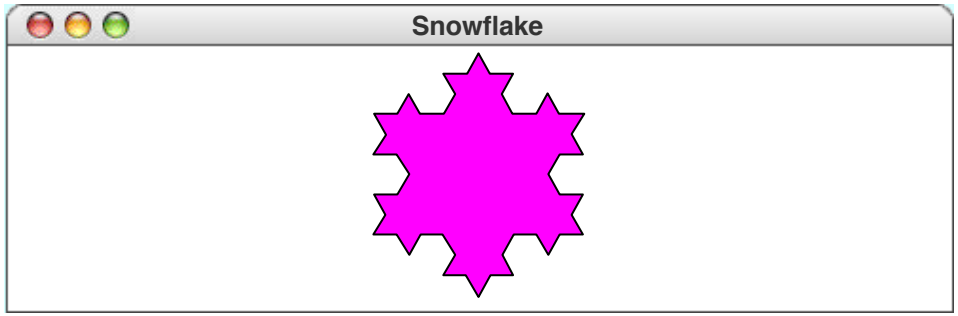
The figure on the previous slide is the Koch fractal of order 4.

Simulating the **Snowflake** Program

```
public void run() {
```

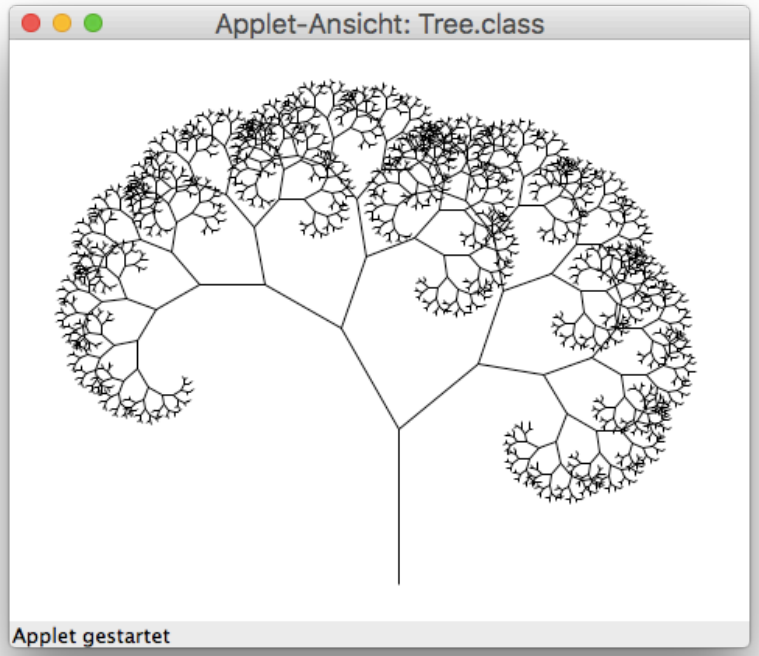
```
public SnowflakeFractal(double edge, int order) {
```

```
private void addFractalLine(double r, int theta, int order) {
   if (order == 0) {
      addPolarEdge(r, theta);
   } else {
      addFractalLine(r / 3, theta, order - 1);
      addFractalLine(r / 3, theta + 60, order - 1);
      addFractalLine(r / 3, theta - 60, order - 1);
      addFractalLine(r / 3, theta, order - 1);
   }
}
```



skip simulation

Recursion



- *Recursion*: method calls itself
- *Direct recursion*: `a()` calls `a()`
- *Indirect recursion*: `a()` calls `b()`, which calls `a()`
- Allowing recursion is motivation to use a stack for method calls; stack permits multiple stack frames for the same method

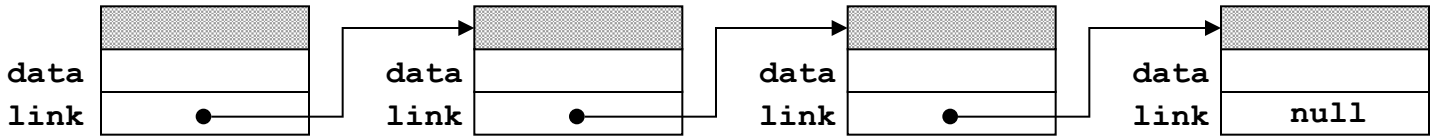
Carl Burch, Programming with Java (Online book)
http://www.toves.org/books/java/ch18-recurex/

```
import java.awt.*;
import acm.program.*;
import acm.graphics.*;

public class Tree extends GraphicsProgram {
  public void run() {
    setSize(500, 350);
    drawTree(250, 350, 100, 90);
  }

  public void drawTree(...) { ... }
}
```

```
public void drawTree(double x0, double y0,
    double len, double angle) {
  if (len > 2) {
    double x1 = x0 +
        len * GMath.cosDegrees(angle);
    double y1 = y0 -
        len * GMath.sinDegrees(angle);
    add(new GLine(x0, y0, x1, y1));
    drawTree(x1, y1, len * 0.75, angle + 30);
    drawTree(x1, y1, len * 0.66, angle - 50);
  }
}
```

Linking Objects Together

- Although most examples of this technique are beyond the scope of a first course, references are particularly important in computer science because they make it possible to represent the relationship among objects by linking them together in various ways.
- One common example (which you will encounter again in Chapter 13) is called a **linked list**, in which each object in a sequence contains a reference to the one that follows it:

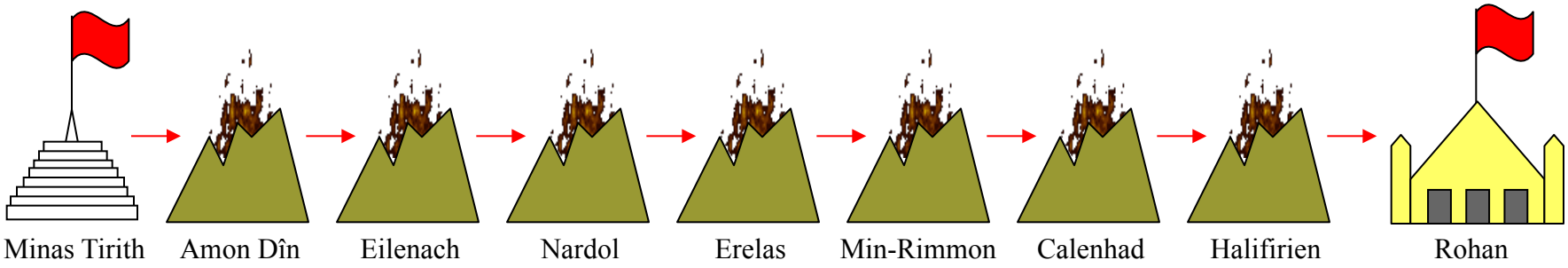


- Java marks the end of linked list using the constant **`null`**, which signifies a reference that does not actually point to an actual object. The value **`null`** has several other uses, as you will discover in the chapters that follow.

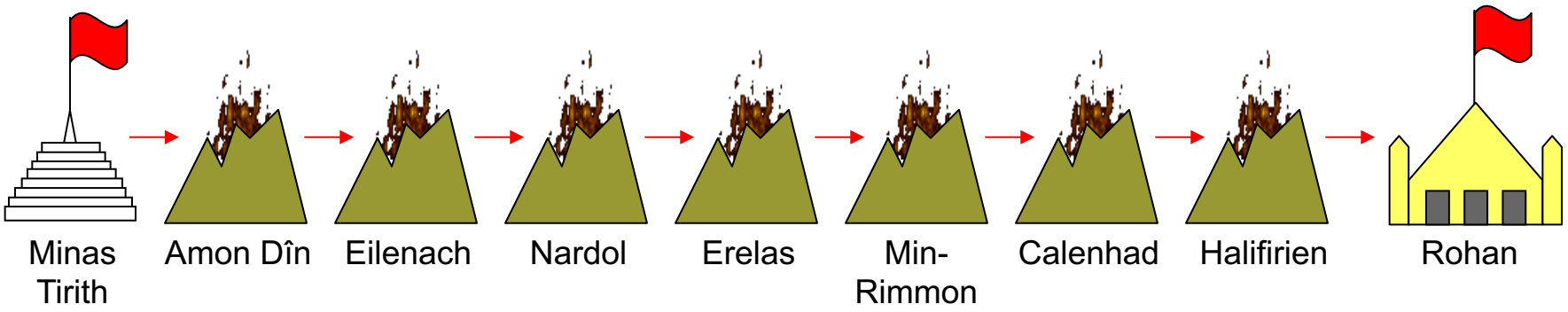
The Beacons of Gondor

> *For answer Gandalf cried aloud to his horse. "On, Shadowfax! We must hasten. Time is short. See! The beacons of Gondor are alight, calling for aid. War is kindled. See, there is the fire on Amon Dîn, and flame on Eilenach; and there they go speeding west: Nardol, Erelas, Min-Rimmon, Calenhad, and the Halifirien on the borders of Rohan."*
>
> —J. R. R. Tolkien, *The Return of the King,* 1955

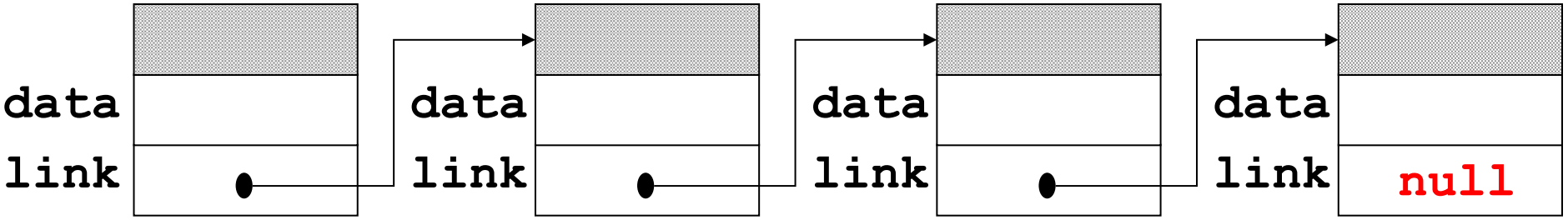
In a scene that was brilliantly captured in Peter Jackson's film adaptation of *The Return of the King,* Rohan is alerted to the danger to Gondor by a succession of signal fires moving from mountain top to mountain top. This scene is a perfect illustration of the idea of message passing in a linked list.

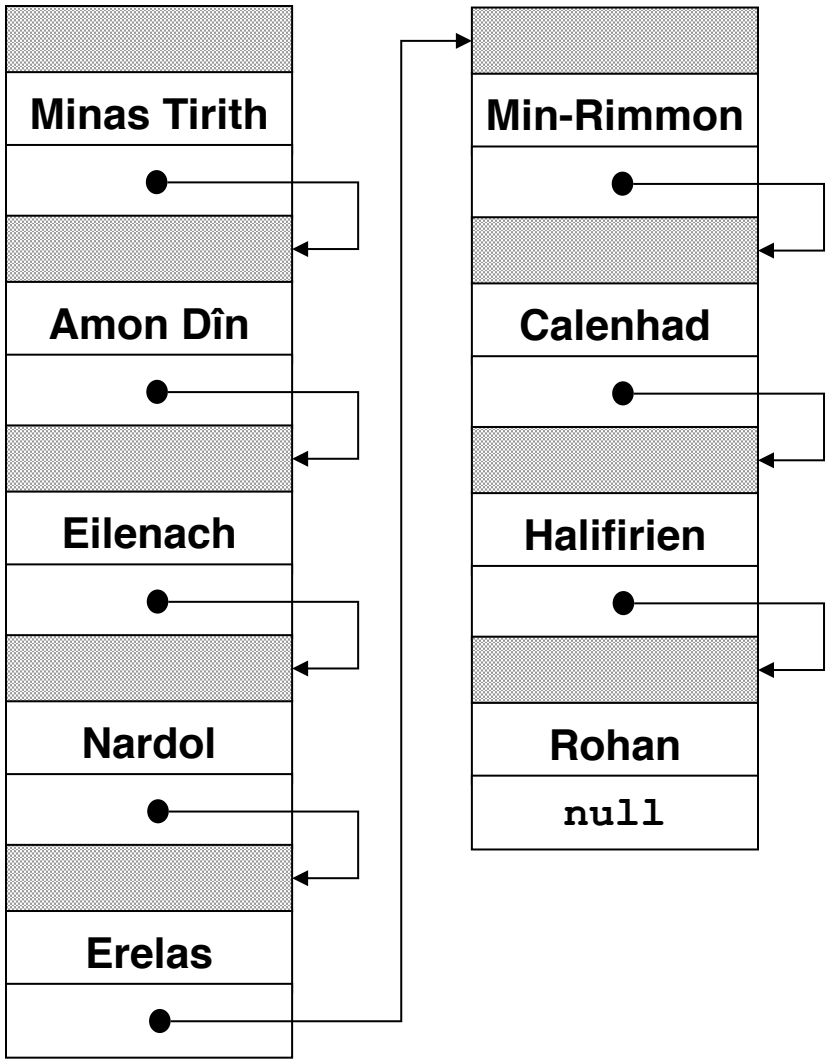


Linking Objects Together



Linked list:






```
public class SignalTower {

    /* Private instance variables */
    private String towerName;
    private SignalTower nextTower;

    /* Constructs a new signal tower */
    public SignalTower(String towerName,
                       SignalTower nextTower) {
        this.towerName = towerName;
        this.nextTower = nextTower;
    }

    /* Signals this tower and passes the
     * message along to the next one.
     */
    public void signal() {
        lightCurrentTower();
        if (nextTower != null) {
            nextTower.signal();
        }
    }

    /* Marks this tower as lit */
    public void lightCurrentTower() {
        . . . code to draw a fire on this tower . . .
    }
}
```

Summary I

- Computer memory is a sequence of *addressable bytes*
- **`char`** / **`int`** / **`double`** require 2 / 4 / 8 bytes
- Memory is organized in three regions:
  1. *Static data*: program code, static variables
  2. *Heap*: objects, instance variables (**`new`**)
  3. *Stack*: local variables, references to objects (pointers)
- Stacks are dynamic *last-in, first-out* (*LIFO*) data structures (*push* + *pop*)

Summary II

- Using a stack for method data allows an arbitrary number of *method instances*, which facilitates *recursion*
- Stack frame of a method call includes **`this`** pointer
- *Garbage collection* reclaims unused memory in heap (*mark-and-sweep*)
- In method calls, primitive type are *passed by value*, objects are *passed by reference*; thus objects are shared between caller and callee
- Automatic *boxing/unboxing* transforms between primitive types and their corresponding *wrapper classes*
- Objects can contain references to other objects – use this e.g. for *linked lists*