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 `ArrayList`s, 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

Memory

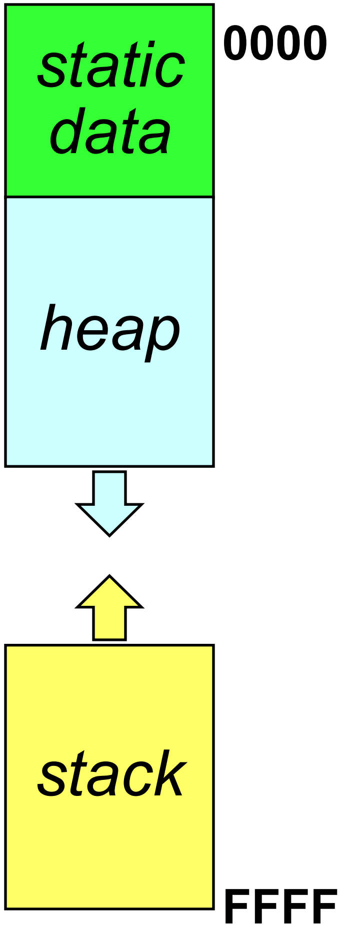
Bits, bytes, words

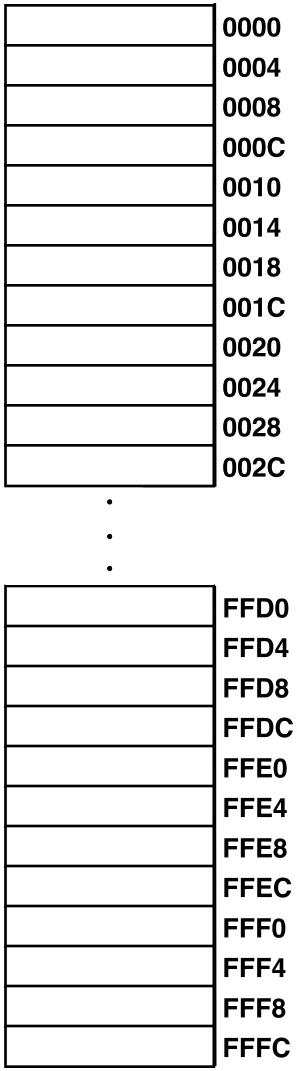
Variable storage

Static data: class var's

Heap: instance var's

Stack: local var's





Initialization

Automatic initialization to default value for

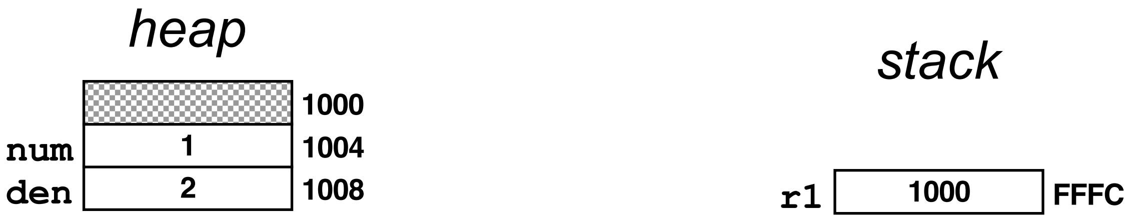
- Class var’s
- Instance var’s
- Array elements

Not for local var’s!

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

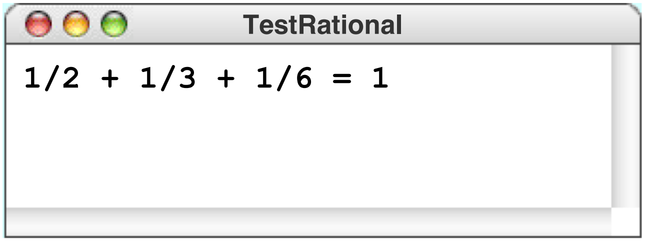
```
public void run() {
   Rational a = new Rational(1, 2);
   Rational b = new Rational(1, 3);
   Rational c = new Rational(1, 6);
   Rational sum = a.add(b).add(c);
   println(a + " + " + b + " + " + c + " = " + sum);
}
```

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



```
1/2 + 1/3 + 1/6 = 1
```

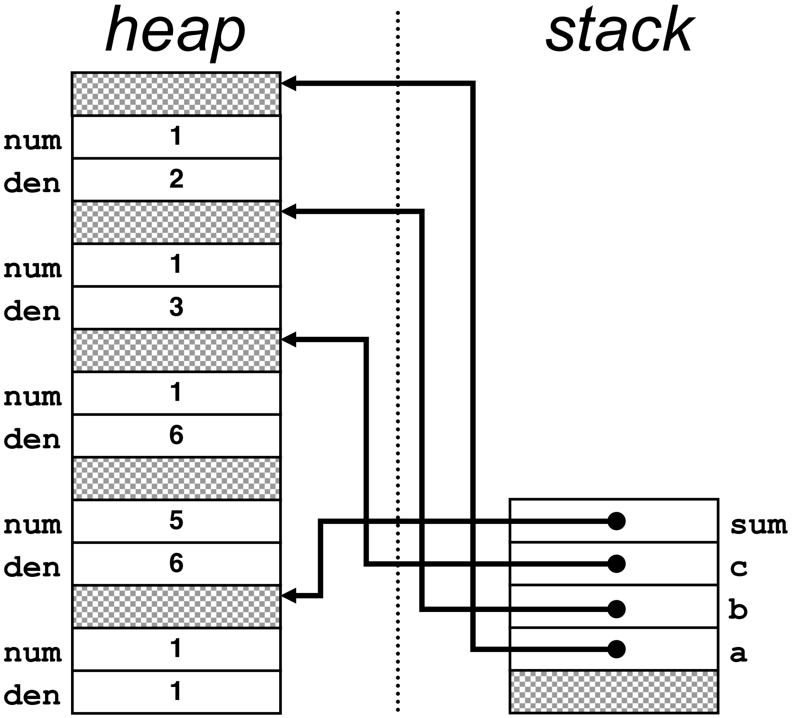
Address Model vs. *Pointer Model*

heap

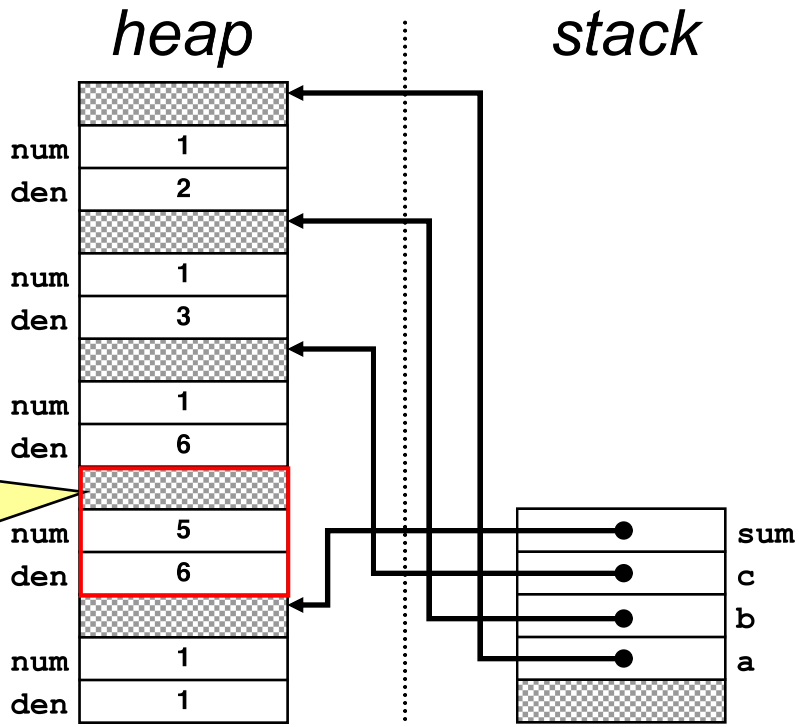
	▨	1000
num	1	1004
den	2	1008
	▨	100C
num	1	1010
den	3	1014
	▨	1018
num	1	101C
den	6	1020
	▨	1024
num	5	1028
den	6	102C
	▨	1030
num	1	1034
den	1	1038

stack

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



Garbage Collection



This object was used to hold a temporary result and is no longer accessible

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

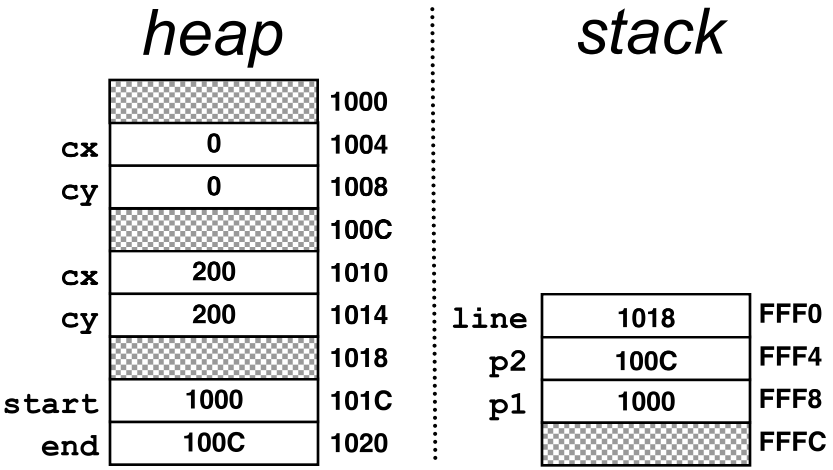
Exercise: Stack-Heap Diagrams

```
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



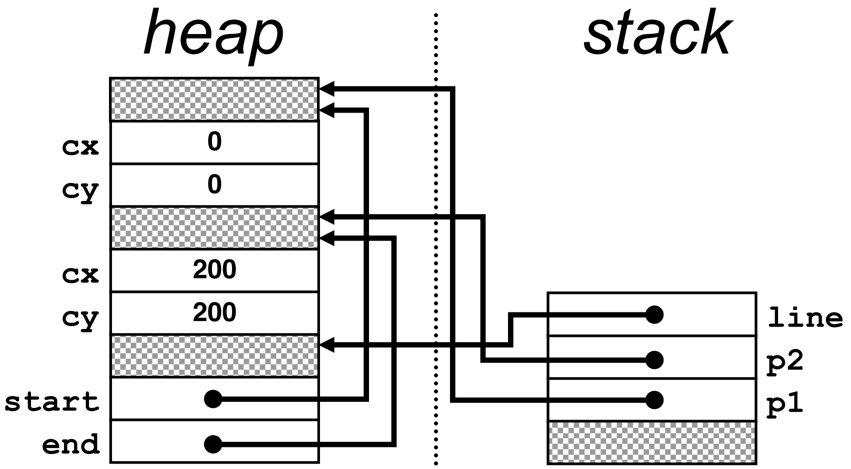
heap

	▨	1000
cx	0	1004
cy	0	1008
	▨	100C
cx	200	1010
cy	200	1014
	▨	1018
start	1000	101C
end	100C	1020

stack

line	1018	FFF0
p2	100C	FFF4
p1	1000	FFF8
	▨	FFFC

Pointer Model



heap

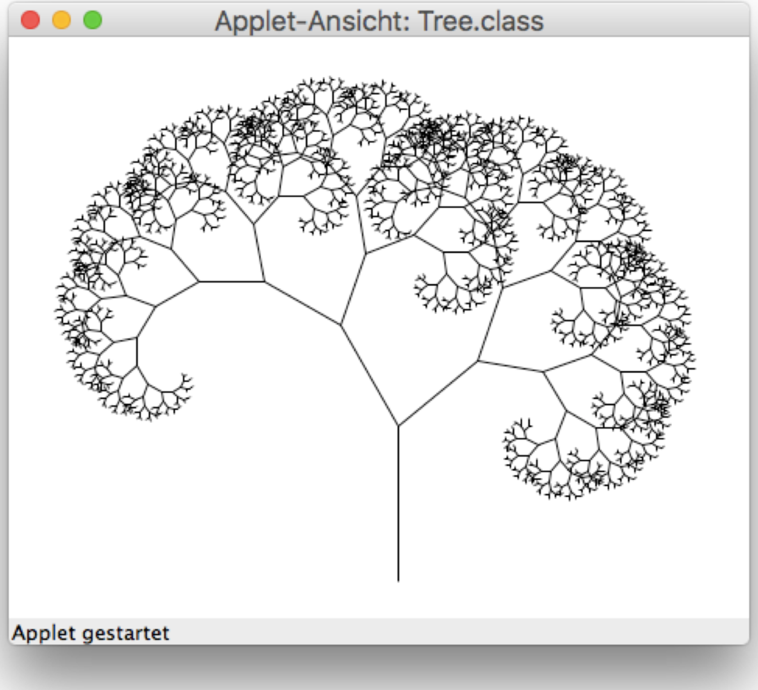
stack

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

MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
0x3A28213A 0x6339392C, 0x7363682E.
I HATE YOU.

https://xkcd.com/138/

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

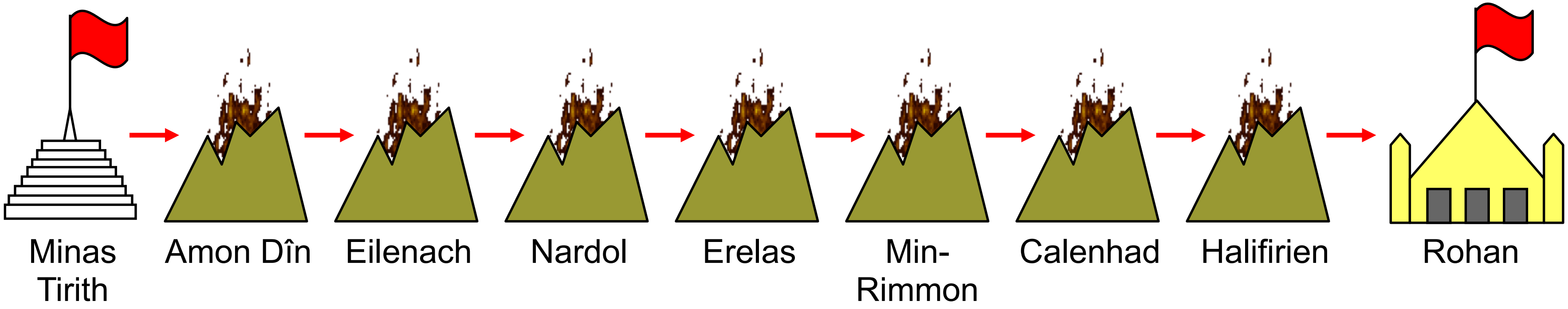
Carl Burch, Programming with Java (Online book)

http://www.toves.org/books/java/ch18-recurex/

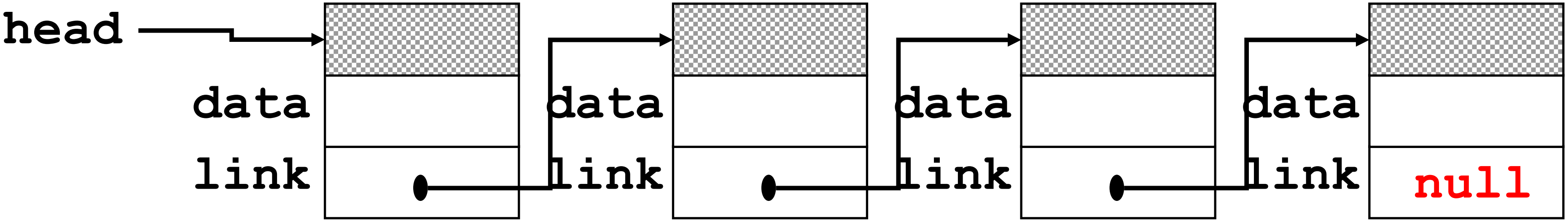
```
import java.awt.*;
import acm.program.*;
import acm.graphics.*;

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

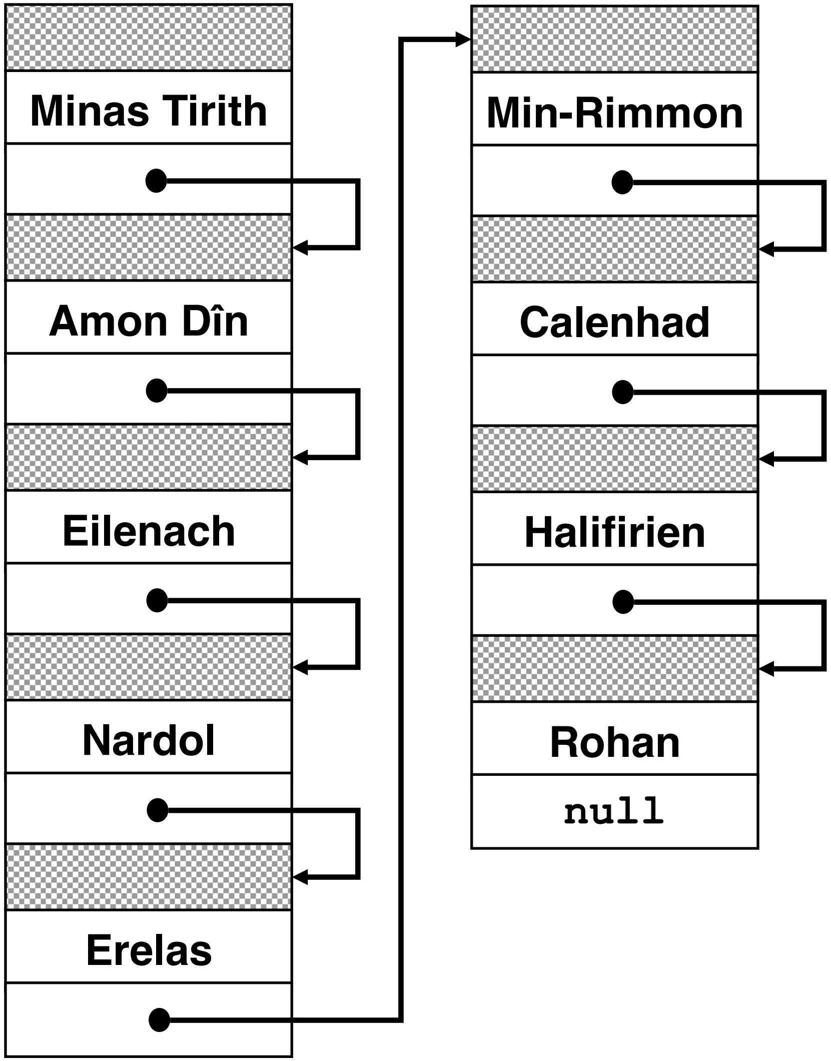
Linking Objects Together



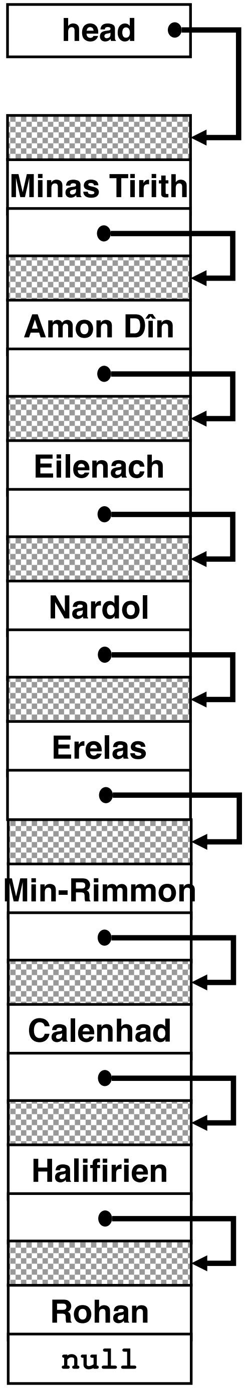
Linked list:



`head == null` means that list is empty



```
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 . . .
   }
}
```



```
public class SignalTowerTest {

  public void run() {
    createSignalTowers();
    minasTirith.signal();
  }

  /* Creates the list of signal towers */
  private void createSignalTowers() {
    rohan = new SignalTower("Rohan", null);
    halifirien = new SignalTower("Halifirien", rohan);
    calenhad = new SignalTower("Calenhad", halifirien);
    minRimmon = new SignalTower("Min-Rimmon", calenhad);
    erelas = new SignalTower("Erelas", minRimmon);
    nardol = new SignalTower("Nardol", erelas);
    eilenach = new SignalTower("Eilenach", nardol);
    amonDin = new SignalTower("Amon Din", eilenach);
    minasTirith = new SignalTower("Minas Tirith", amonDin);
    head = minasTirith;
  }

  /* Private instance variables */
  private SignalTower head, minasTirith, amonDin, eilenach,
    nardol, erelas, minRimmon, calenhad, halifirien, rohan;
}
```

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
- 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*
- *Garbage collection* reclaims unused memory in heap (*mark-and-sweep*)
- In method calls, primitive types 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*