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.

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

```java
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.

A Complete Heap-Stack Trace

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

This object is a temporary value used only during the calculation.

```
1/2 + 1/3 + 1/6 = 1
```
The Pointer Model

- The heap-stack diagram at the lower left shows the state of memory at the end of the run method from TestRational.
- The diagram at the lower right shows exactly the same state using arrows instead of numeric addresses. This style of diagram is said to use the 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

- One fact that the pointer model makes clear in this diagram is that there are no longer any references to the `Rational` value 5/6. That value has now become garbage.
- From time to time, Java runs through the heap and reclaims any garbage. This process is called garbage collection.

Exercise: Stack-Heap Diagrams

Suppose that the classes `Point` and `Line` are defined as follows:

```java
public class Point {
    public Point(int x, int y) {
        cx = x;
        cy = y;
    }
    ... other methods appear here ...
    private int cx;
    private int cy;
}

public class Line {
    public Line(Point p1, Point p2) {
        start = p1;
        finish = p2;
    }
    ... other methods appear here ...
    private Point start;
    private Point finish;
}
```

Draw a heap-stack diagram showing the state of memory just before the following `run` method returns:

```java
public static void main(String[] args) {
    Point p1 = new Point(0, 0);
    Point p2 = new Point(200, 200);
    Line line = new Line(p1, p2);
}
```
**Solution: Stack-Heap Diagrams**

**Address Model**

- `cx` starts at 1000, `cy` starts at 1000.
- `cx` moves to 1004, `cy` remains at 1000.
- `cx` moves to 200, `cy` remains at 1000.
- `cx` moves to 200, `cy` remains at 1000.
- `cx` moves to 0, `cy` remains at 0.

**Pointer Model**

- `cx` starts at 1018, `cy` starts at 1018.
- `cx` moves to 1014, `cy` remains at 1018.
- `cx` moves to 200, `cy` remains at 200.
- `cx` moves to 200, `cy` remains at 200.
- `cx` moves to 0, `cy` remains at 0.

---

**Primitive Types vs. Objects**

- At first glance, Java’s rules for passing objects as arguments seem to differ from the rules Java uses with arguments that are primitive types.

- When you pass an argument of a primitive type to a method, Java copies the value of the argument into the parameter variable. As a result, changes to the parameter variable have no effect on the argument.

- When you pass an object as an argument, there seems to be some form of sharing going on. Although changing the parameter variable itself has no effect, any changes that you make to the instance variables inside an object—usually by calling setters—have a permanent effect on the object.

- Stack-heap diagrams make the reason for this seeming asymmetry clear. When you pass an object to a method, Java copies the reference but not the object itself.
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:

  ![Linked List Diagram](image)

- 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.”


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.
Message Passing in Linked Structures

To represent this message-passing image, you might use a definition such as the one shown on the right.

You can then initialize a chain of SignalTower objects, like this:

Calling signal on the first tower sends a message down the chain.

```
public class SignalTower {
    /* Constructs a new signal tower */
    public SignalTower(String name, SignalTower link) {
        towerName = name;
        nextTower = link;
    }
    /* 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 . . .
    /* Private instance variables */
    private String towerName;
    private SignalTower nextTower;
}
```

```
public static void main(String[] args) {
    SignalTower rohan = new SignalTower("Rohan", null);
    SignalTower halifirien = new SignalTower("Halifirien", rohan);
    SignalTower calenhad = new SignalTower("Calenhad", halifirien);
    SignalTower minRimmon = new SignalTower("Min-Rimmon", calenhad);
    SignalTower erelas = new SignalTower("Erelas", minRimmon);
    SignalTower nardol = new SignalTower("Nardol", erelas);
    SignalTower eilenach = new SignalTower("Eilenach", nardol);
    SignalTower amonDin = new SignalTower("Amon Din", eilenach);
    SignalTower minasTirith = new SignalTower("Minas Tirith", amonDin);
    minasTirith.signal();
}
```