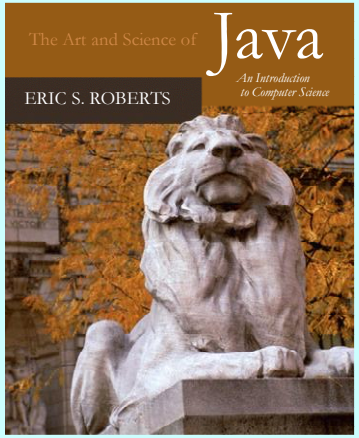
CHAPTER 7

Objects and Memory

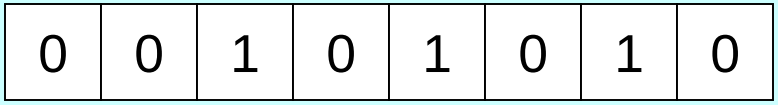
Yea, from the table of my memory
I'll wipe away all trivial fond records.
—William Shakespeare, *Hamlet*, c. 1600



Edited by:
S. M. Ghaffarian

The Structure of Memory

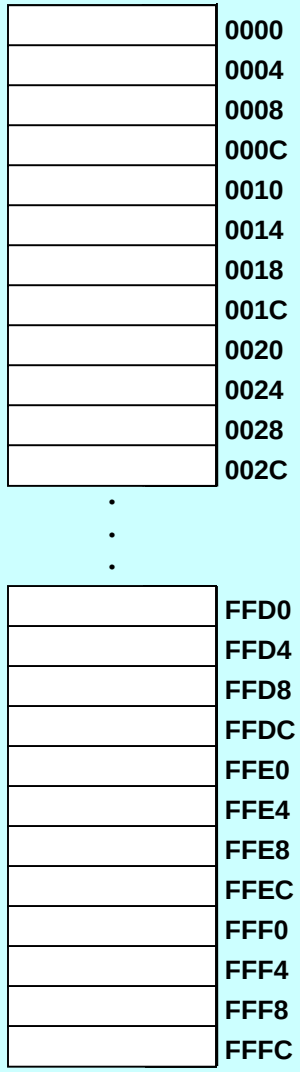
- The fundamental unit of memory inside a computer is called a **bit**, which is a contraction of the words *binary digit*. A bit can be in either of two states, usually denoted as 0 and 1.
- The hardware structure of a computer combines individual bits into larger units. In most modern architectures, the smallest unit on which the hardware operates is a sequence of eight consecutive bits called a **byte**. The following diagram shows a byte containing a combination of 0s and 1s:



- Numbers are stored in still larger units that consist of multiple bytes. The unit that represents the most common integer size on a particular hardware is called a **word**. Because machines have different architectures, the number of bytes in a word may vary from machine to machine.

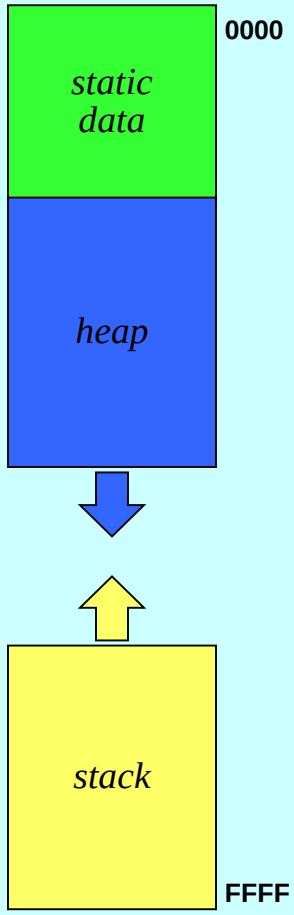
Memory and Addresses

- 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.



Rational Numbers

- As a more elaborate example of class definition, we define a class called **Rational** that represents **rational numbers**, which are simply the quotient of two integers.
- Rational numbers can be useful in cases in which you need exact calculation with fractions. Even if you use a **double**, the floating-point number 0.1 is represented internally as an approximation. The rational number 1 / 10 is exact.
- Rational numbers support the standard arithmetic operations:

Addition:

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

Subtraction:

$$\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$$

Multiplication:

$$\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$$

Division:

$$\frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc}$$

Implementing the `Rational` Class

- The next five slides show the code for the **Rational** class along with some brief annotations.
- As you read through the code, the following features are worth special attention:
  - The constructors for the class are overloaded. Calling the constructor with no argument creates a **Rational** initialized to 0, calling it with one argument creates a **Rational** equal to that integer, and calling it with two arguments creates a fraction.
  - The constructor makes sure that the numerator and denominator of any **Rational** are always reduced to lowest terms. Moreover, since these values never change once a new **Rational** is created, this property will remain in force.
  - The **add**, **subtract**, **multiply**, and **divide** methods are written so that one of the operands is the receiver (signified by the keyword **this**) and the other is passed as an argument. Thus to add **r1** and **r2** you would write:

```
r1.add(r2)
```

The `Rational` Class

```
/**
 * The Rational class is used to represent rational numbers, which
 * are defined to be the quotient of two integers.
 */
public class Rational {

/** Creates a new Rational initialized to zero. */
   public Rational() {
      this(0);
   }

/**
 * Creates a new Rational from the integer argument.
 * @param n The initial value
 */
   public Rational(int n) {
      this(n, 1);
   }
```


skip code

The `Rational` Class

```
/**
 * Creates a new Rational with the value x / y.
 * @param x The numerator of the rational number
 * @param y The denominator of the rational number
 */
   public Rational(int x, int y) {
      int g = gcd(Math.abs(x), Math.abs(y));
      num = x / g;
      den = Math.abs(y) / g;
      if (y < 0) num = -num;
   }

/**
 * Adds the rational number r to this one and returns the sum.
 * @param r The rational number to be added
 * @return The sum of the current number and r
 */
   public Rational add(Rational r) {
      return new Rational(this.num * r.den + r.num * this.den,
                          this.den * r.den);
   }
```


skip code

The `Rational` Class

```
/**
 * Subtracts the rational number r from this one.
 * @param r The rational number to be subtracted
 * @return The result of subtracting r from the current number
 */
   public Rational subtract(Rational r) {
      return new Rational(this.num * r.den - r.num * this.den,
                          this.den * r.den);
   }

/**
 * Multiplies this number by the rational number r.
 * @param r The rational number used as a multiplier
 * @return The result of multiplying the current number by r
 */
   public Rational multiply(Rational r) {
      return new Rational(this.num * r.num, this.den * r.den);
   }
```


skip code

The `Rational` Class

```
/**
 * Divides this number by the rational number r.
 * @param r The rational number used as a divisor
 * @return The result of dividing the current number by r
 */
   public Rational divide(Rational r) {
      return new Rational(this.num * r.den, this.den * r.num);
   }

/**
 * Creates a string representation of this rational number.
 * @return The string representation of this rational number
 */
   public String toString() {
      if (den == 1) {
         return "" + num;
      } else {
         return num + "/" + den;
      }
   }
```


skip code

The `Rational` Class

```
/**
 * Calculates the greatest common divisor using Euclid's algorithm.
 * @param x First integer
 * @param y Second integer
 * @return The greatest common divisor of x and y
 */
   private int gcd(int x, int y) {
      int r = x % y;
      while (r != 0) {
         x = y;
         y = r;
         r = x % y;
      }
      return y;
   }

/* Private instance variables */
   private int num;    /* The numerator of this Rational   */
   private int den;    /* The denominator of this Rational */

}
```


skip code

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.

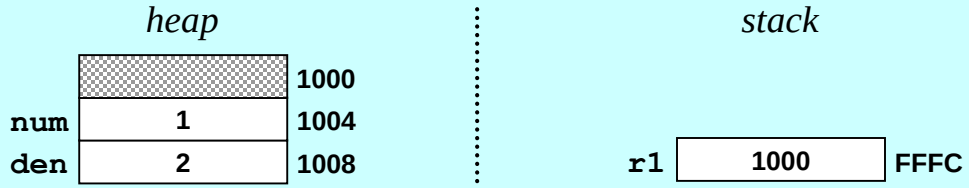
Object References

- 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.

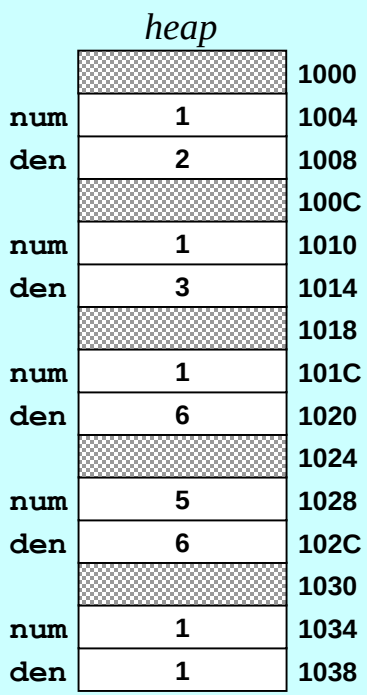
- The local variable **`r1`** is allocated in the current stack frame and is assigned the value 1000, which identifies the object.

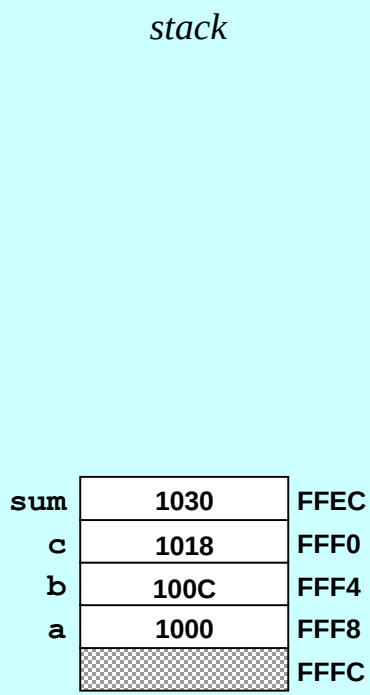


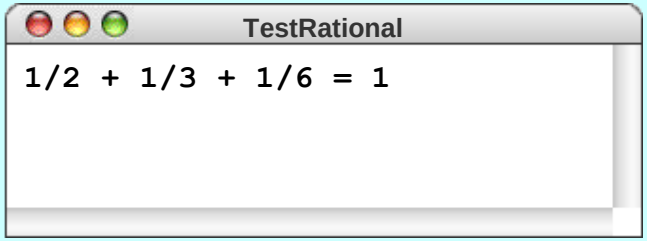
- The next slide traces the execution of the **`TestRational`** program from Chapter 6 using heap-stack model.

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);
}
```

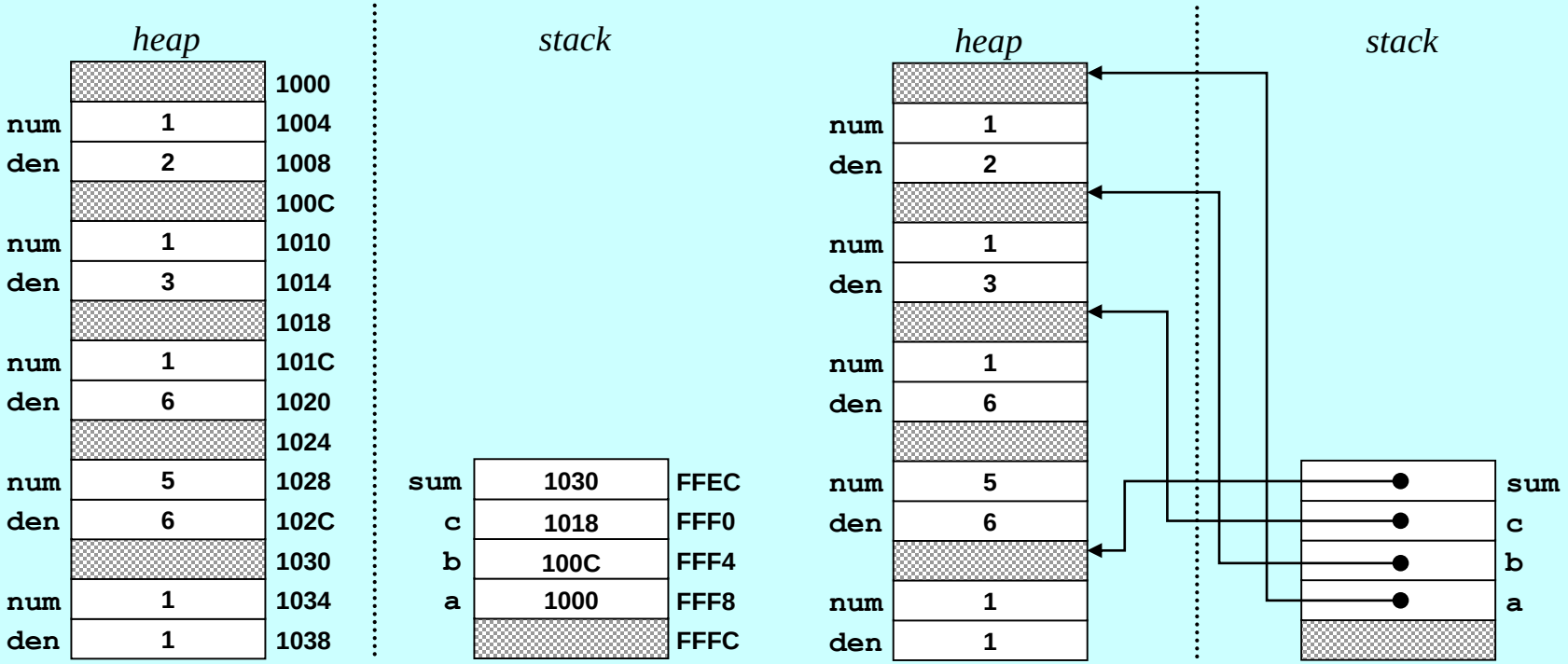






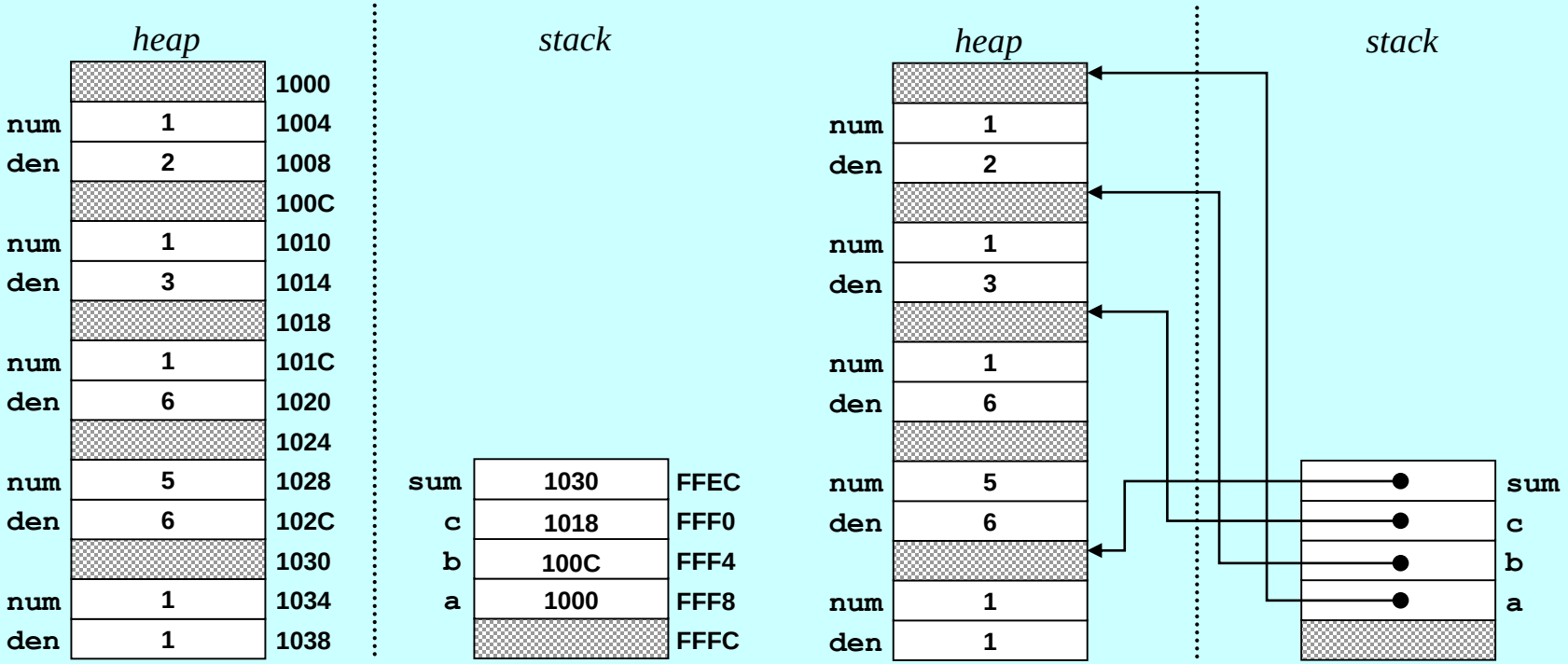
The Pointer Model

- 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**.



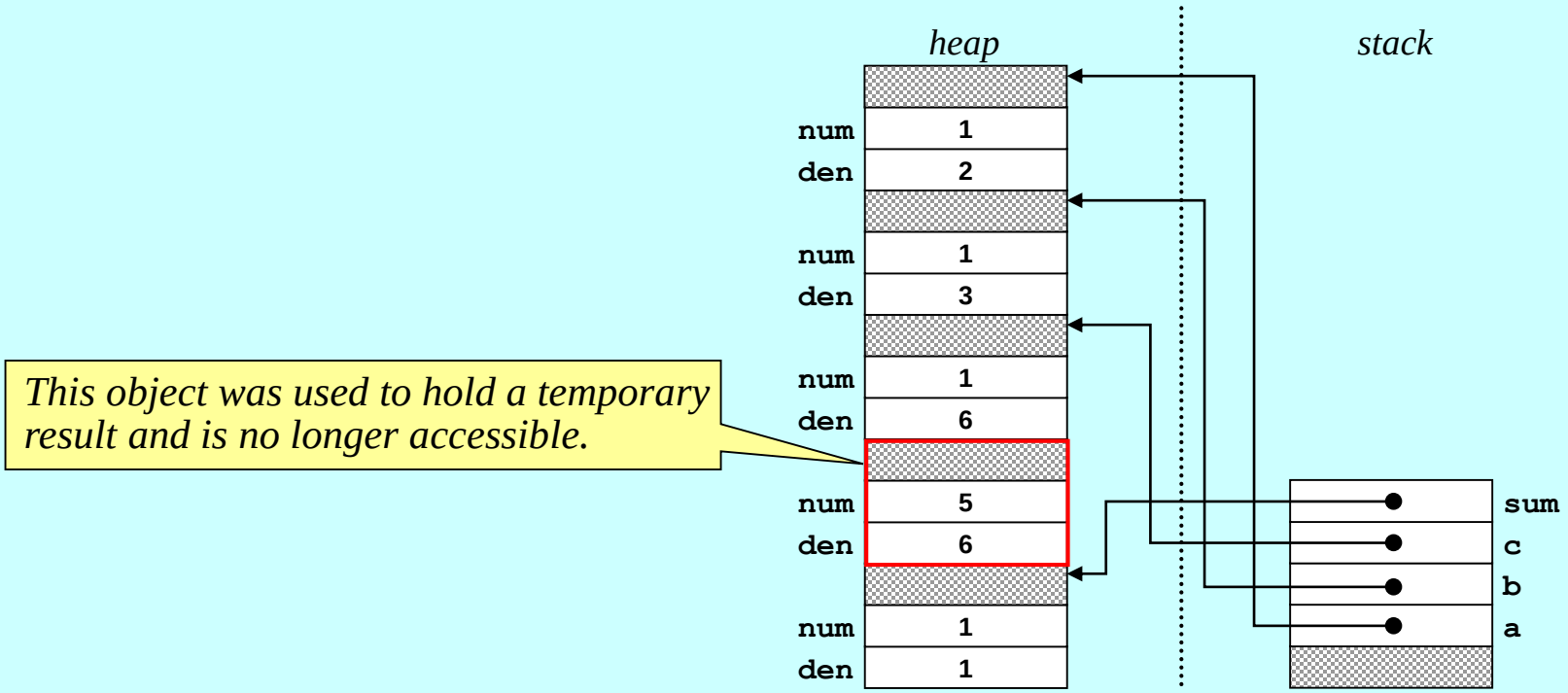
Addresses *vs.* Pointers

- 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:

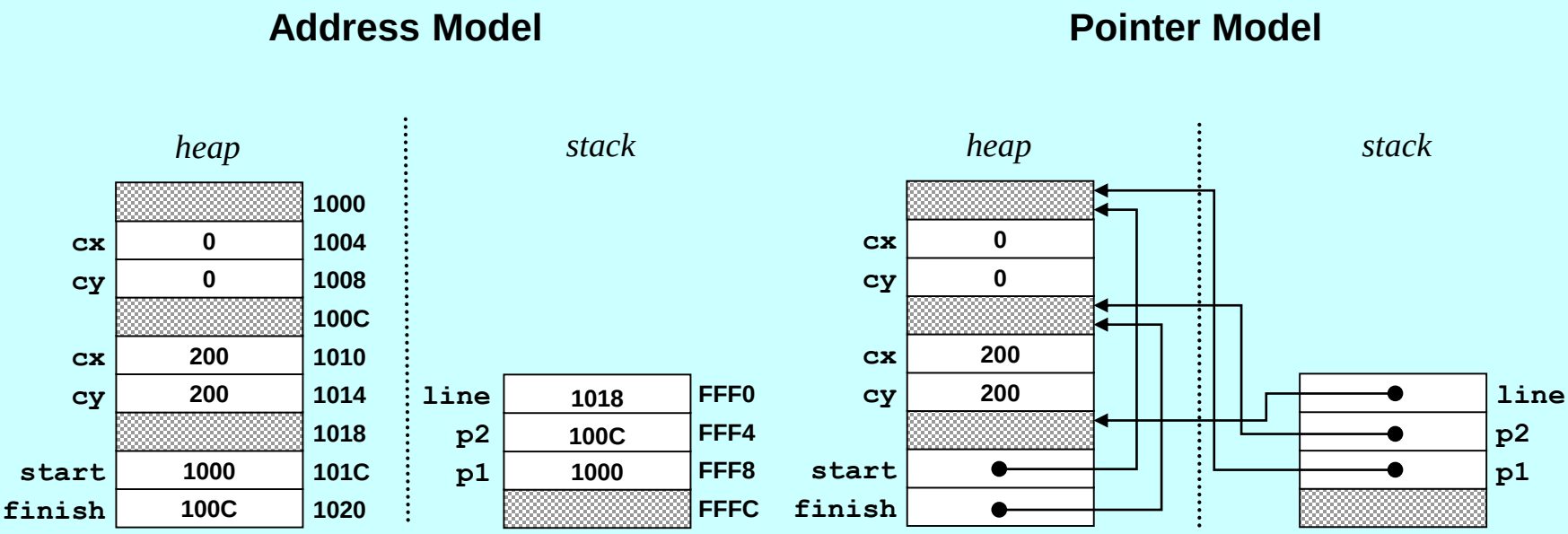
```
public class Point {
   public Point(int x, int y) {
      cx = x;
      cy = y;
   }

   . . . 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.

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

Solution: Stack-Heap Diagrams



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.

Wrapper Classes

- The designers of Java chose to separate the primitive types from the standard class hierarchy mostly for efficiency. Primitive values take less space and allow Java to use more of the capabilities provided by the hardware.
- Even so, there are times in which the fact that primitive types are *not* objects gets in the way. There are many tools in the Java libraries—several of which you will encounter later in the book—that work only with objects.
- To get around this problem, Java includes a **wrapper class** to correspond to each of the primitive types:

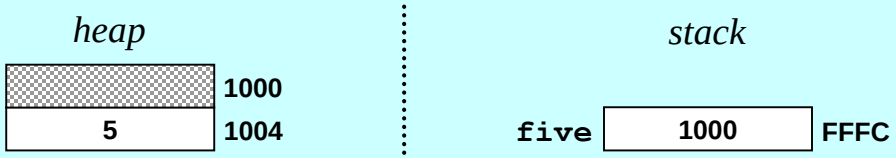
`boolean`	`Boolean`	`float`	`Float`
`byte`	`Byte`	`int`	`Integer`
`char`	`Character`	`long`	`Long`
`double`	`Double`	`short`	`Short`

Using Wrapper Classes

- You can create an instance of a wrapper class by calling its constructor with the primitive value. For example, the line

```
Integer five = new Integer(5);
```

creates a new **`Integer`** object containing the value 5:



- To value stored in the variable **`five`** is a real object, and you can use it in any contexts that require objects.
- For each of the wrapper classes, Java defines a method to retrieve the primitive value, as illustrated below:

```
int underlyingValue = five.intValue();
```

Boxing and Unboxing

- As of Java Standard Edition 5.0, Java automatically converts values back and forth between a primitive type and the corresponding wrapper class. For example, if you write

```
Integer five = 5;
```

  Java will automatically call the **`Integer`** constructor.

- Similarly, if you then write

```
int six = five + 1;
```

  Java will automatically call **`intValue`** before the addition.

- These operations are called **boxing** and **unboxing**.
- Although boxing and unboxing can be quite convenient, this feature can generate confusion and should be used with care.

The End