Eric Roberts
CS 106A

Handout #32
May 2, 2012

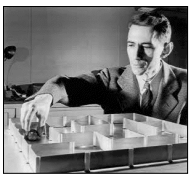
Objects and Memory

Objects and Memory

Eric Roberts
CS 106A
May 2, 2012

Claude Shannon

Claude Shannon was one of the pioneers who shaped computer science in its early years. In his master's thesis, Shannon showed how it was possible to use Boolean logic and switching circuits to perform arithmetic calculations. That work led directly to the use of binary arithmetic inside computers—an idea that initially generated considerable resistance among more established scientists.

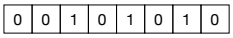


Claude Shannon (1916-2001)

Shannon is best known as the inventor of ***information theory,*** which provides the mathematical foundation for understanding data communication. To many, however, Shannon's great contribution was that he never lost his sense of fun throughout his long career.

The Structure of Memory

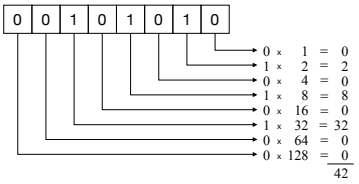
- The fundamental unit of memory inside a computer is called a ***bit***, which is a contraction of the words ***binary digit***. A bit can be in either of two states, usually denoted as 0 and 1.
- The hardware structure of a computer combines individual bits into larger units. In most modern architectures, 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 varies from machine to machine.

Binary Notation

- Bytes and words can be used to represent integers of different sizes by interpreting the bits as a number in ***binary notation***.
- Binary notation is similar to decimal notation but uses a different ***base***. Decimal numbers use 10 as their base, which means that each digit counts for ten times as much as the digit to its right. Binary notation uses base 2, which means that each position counts for twice as much, as follows:

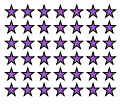


Numbers and Bases

- The calculation at the end of the preceding slide makes it clear that the binary representation 00101010 is equivalent to the number 42. When it is important to distinguish the base, the text uses a small subscript, like this:

$$00101010_2 = 42_{10}$$

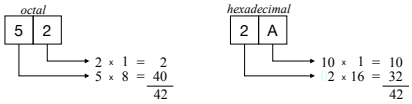
- Although it is useful to be able to convert a number from one base to another, it is important to remember that the number remains the same. What changes is how you write it down.
- The number 42 is what you get if you count how many stars are in the pattern at the right. The number is the same whether you write it in English as *forty-two,* in decimal as 42, or in binary as 00101010.



- Numbers do not have bases; representations do.

Octal and Hexadecimal Notation

- Because binary notation tends to get rather long, computer scientists often prefer ***octal*** (base 8) or ***hexadecimal*** (base 16) notation instead. Octal notation uses eight digits: 0 to 7. Hexadecimal notation uses sixteen digits: 0 to 9, followed by the letters A through F to indicate the values 10 to 15.
- The following diagrams show how the number forty-two appears in both octal and hexadecimal notation:



- The advantage of using either octal or hexadecimal notation is that doing so makes it easy to translate the number back to individual bits because you can convert each digit separately.

Exercises: Number Bases

- What is the decimal value for each of the following numbers?

  10001_2 177_8 AD_{16}

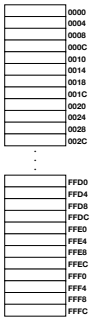
- As part of a code to identify the file type, every Java class file begins with the following sixteen bits:

1	1	0	0	1	0	1	0	1	1	1	1	1	1	1	0

  How would you express that number in hexadecimal notation?

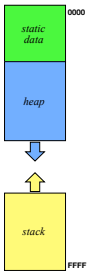
Memory and Addresses

- 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 diagram on the right 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

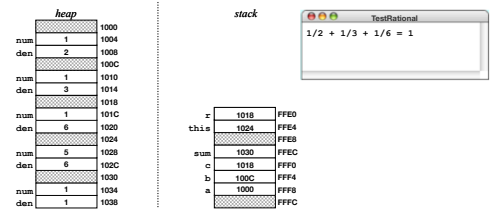
```
Rational r1 = new Rational(1, 2);
```

  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 the heap-stack model.

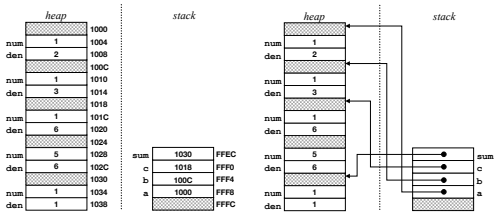
A Complete Heap-Stack Trace

```
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);
}
```



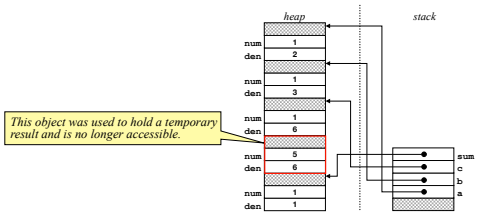
The Pointer Model

- 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***.



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);
}
```

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 CS 106A—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`