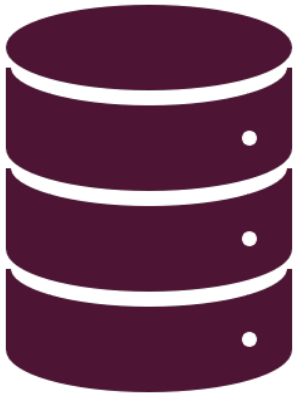


DATA STRUCTURES

WENYE LI
CUHK-SZ

OUTLINE

- Basics in Computer Memory
  - Partially go beyond the current scope.
  - Will come back to detailed discussion after a few weeks.
- Arrays
- Implementation
- Examples

STRUCTURE OF MEMORY

- The basic unit of memory is called a **bit**, either 0 or 1.
- In most modern architectures, the smallest unit on which the hardware operates is a sequence of eight consecutive bits called **byte**.

  a binary (executable) file

  0 1 2 3

  01011001100001001001111011000011...

- Numbers and instructions are stored in still larger units, mostly common a **word**. Because machines have different architectures, the number of bytes and the order of bytes in a word vary from machine to machine.

NUMBERS, BASES, AND CONVERSION

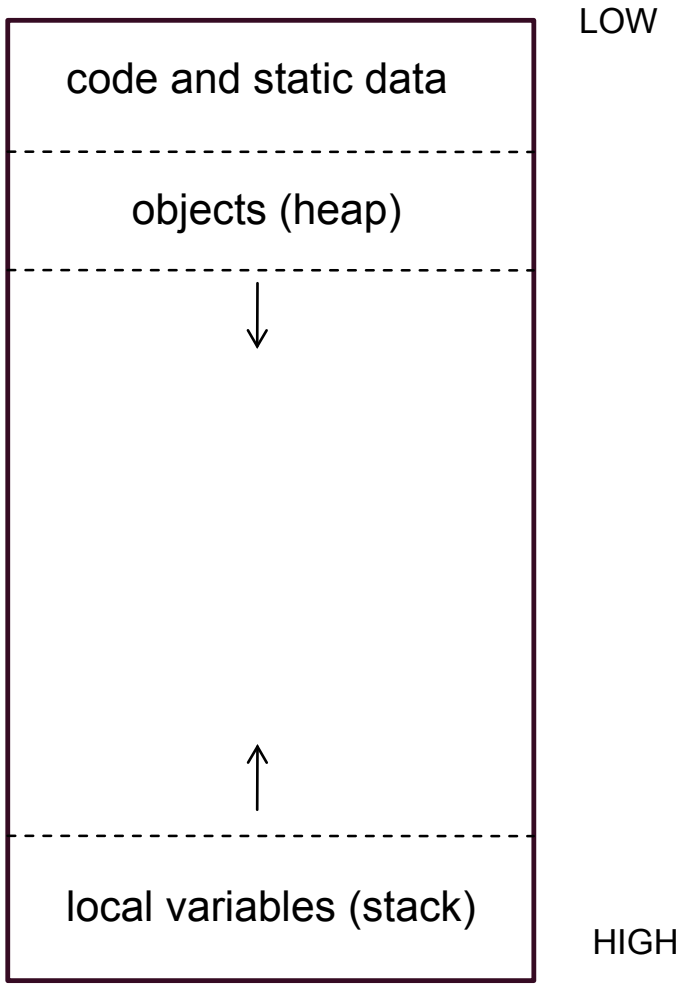
- $(21)_{10} = (10101)_2$
- $(0.65625)_{10} = (0.10101)_2$

- Octal (0,1,2,3,4,5,6,7)
  $(10101)_2 = (010101)_2 = (25)_8$
  $(0.10101)_2 = (0.101010)_2 = (0.52)_8$

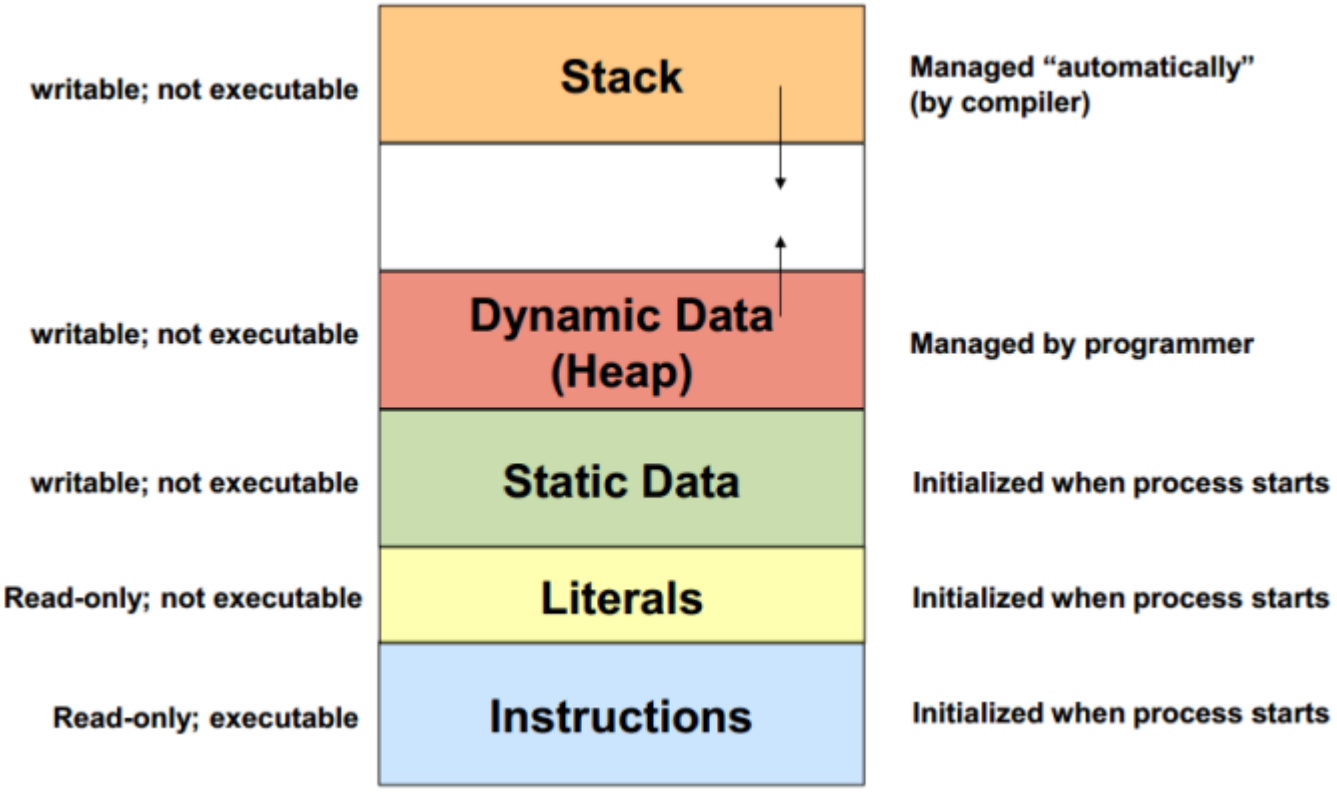
- Hexadecimal (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F)
  $(10101)_2 = (00010101)_2 = (15)_{16}$
  $(0.10101)_2 = (0.10101000)_2 = (0.A8)_{16}$

- Useful numbers
  $(10000000000)_2 = (1024)_{10}$ (about 1K)

MEMORY ALLOCATION



If you prefer the other direction



STACK VS HEAP

Parameter	STACK	HEAP
Basic	Memory is allocated in a contiguous block.	Memory is allocated in any random order.
Allocation and De-allocation	Automatic by compiler instructions.	Manual by the programmer.
Cost	Less	More
Implementation	Easy	Hard
Access time	Faster	Slower
Main Issue	Shortage of memory	Memory fragmentation
Locality of reference	Excellent	Adequate
Safety	Thread safe, data stored can only be accessed by owner	Not Thread safe, data stored visible to all threads
Flexibility	Fixed-size	Resizing is possible
Data type structure	Linear	Hierarchical

Will discuss with more details later.

MEMORY ALLOCATION TO VARIABLES

- One region of memory is reserved for **static data**.
  - never created or destroyed as program runs, such as named constants.
- When a new object is created, Java allocates space from **heap**.
- When a method is called, Java allocates a new block of memory called a stack frame to hold its local variables.
- When a method returns, its stack frame is erased. Stack frames come from **stack**.

OBJECT REFERENCES

- Java identifies an object by its address in memory. That address is called a reference.

- Eg., when Java executes

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

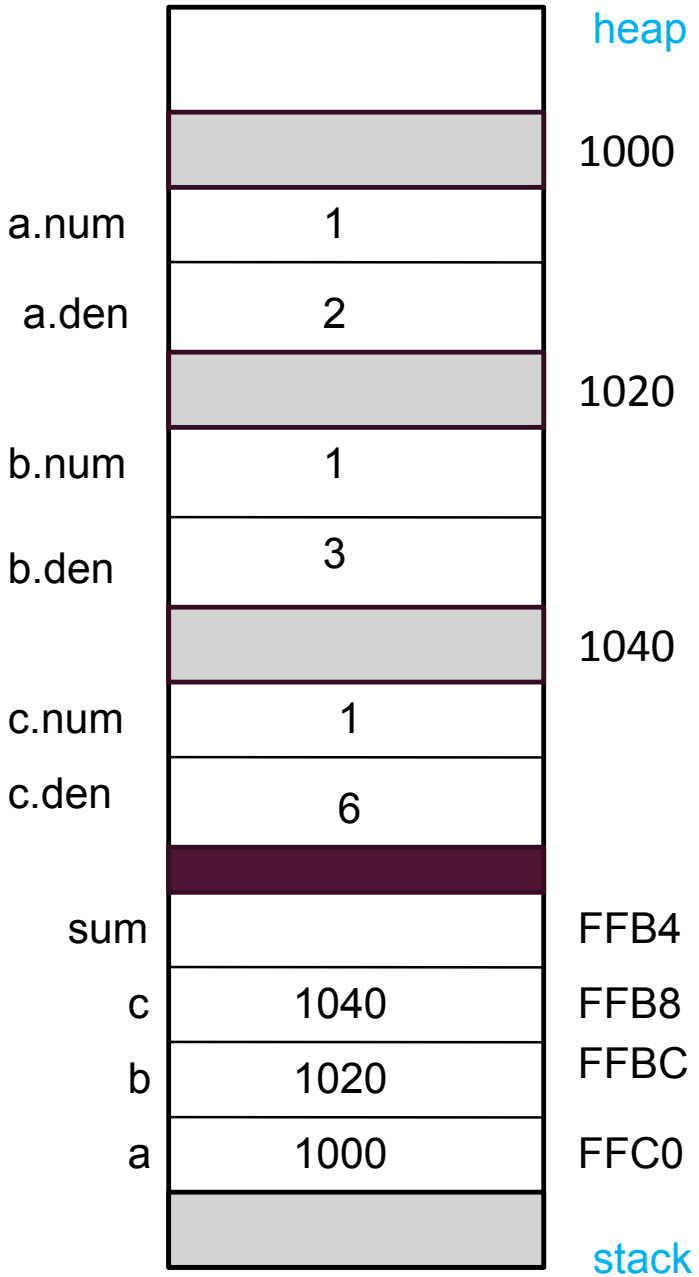
  it allocates heap space for the new Rational object. For this example, imagine that the object is allocated at address 1000.

- The local variable `a` is allocated in the current stack frame and is assigned the value (address), which identifies the object.

```java
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);
    }
    /**
     * 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);
    }
    /**
     * Subtracts the rational number r from this one and returns
     * the difference.
     * @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);
    }
    /**
     * Divides this number by the nonzero rational number r.
     * @param r The nonzero 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;
        }
    }
    /**
     * Calculates the greatest common divisor using Euclid's algorithm.
     * @param First integer
     * @param 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 */
} /* class Rational */
```

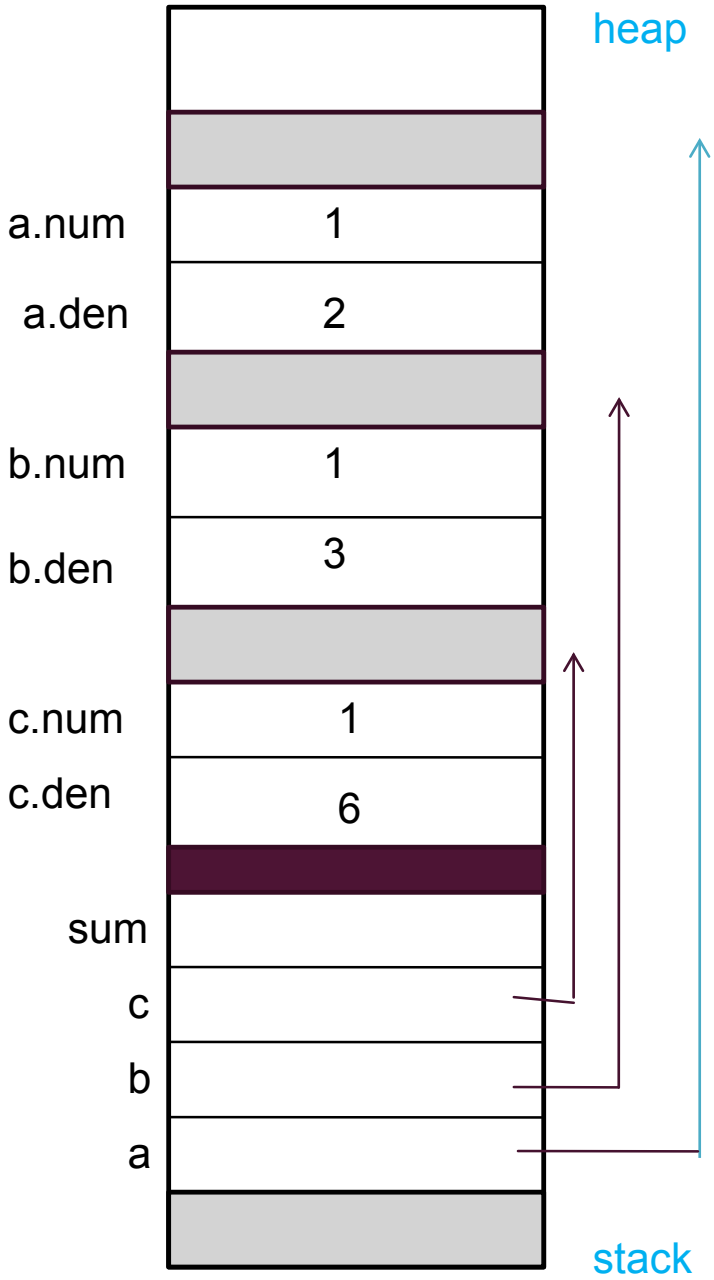
Address Model

```
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);
}
```

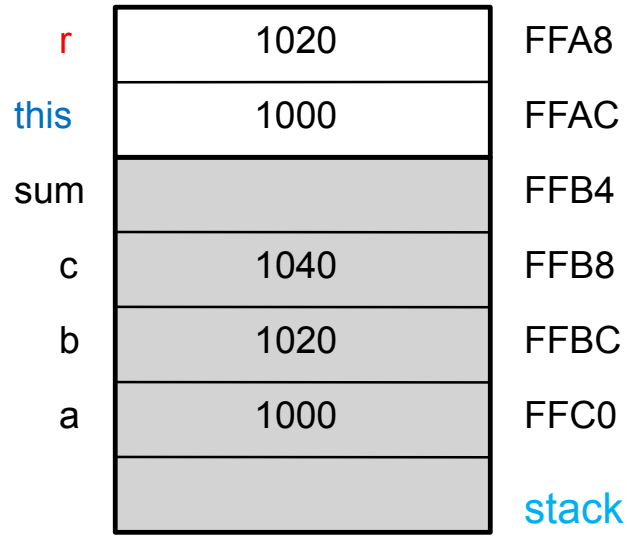


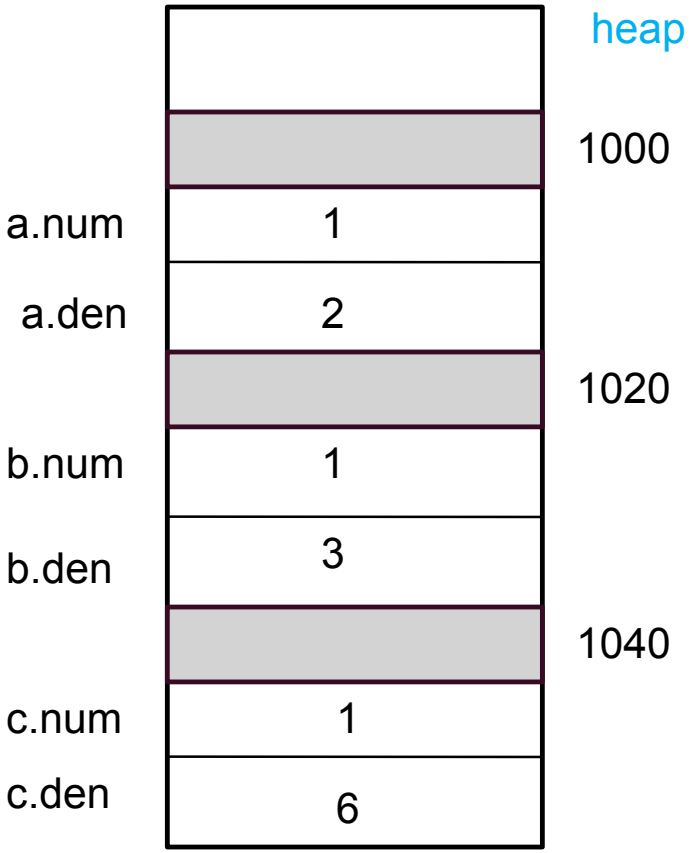
Pointer Model

```
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);
}
```

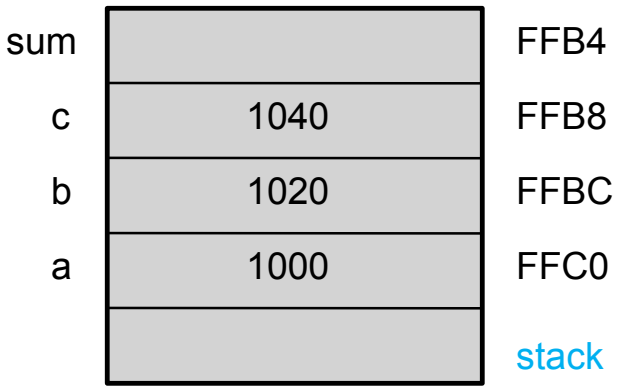


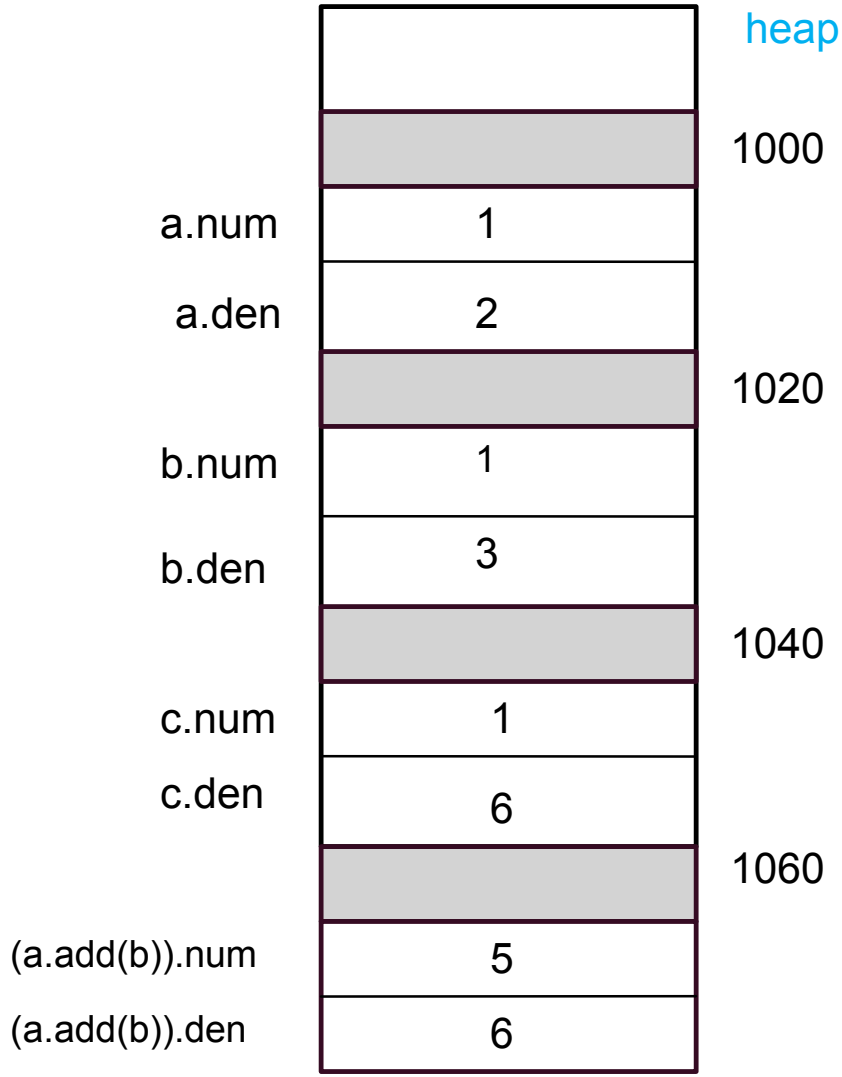
```
public void run() {
        Rational  a = new Rational(1, 2);
        Rational  b = new Rational(1, 3);
        Rational  c = new Rational(1, 6);
        Rational  sum = (a.add(b)).add(c);
}

public  Rational  add(Rational  r) {
      return new Rational(this.num*r.den
                  + r.num*this.den,
                 this.den*r.den);
}
```



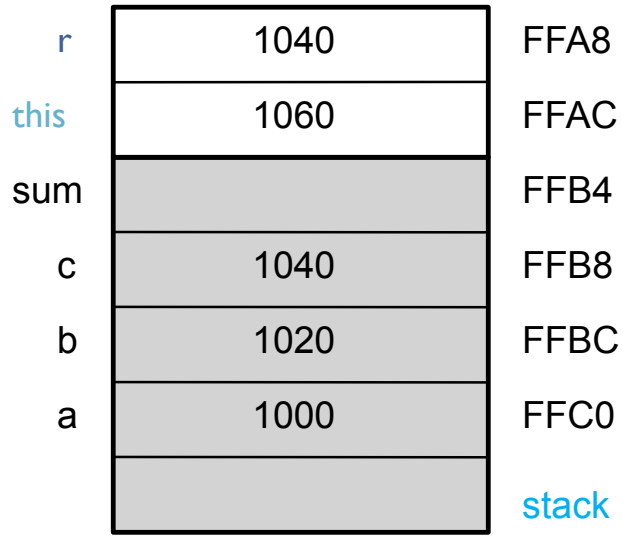


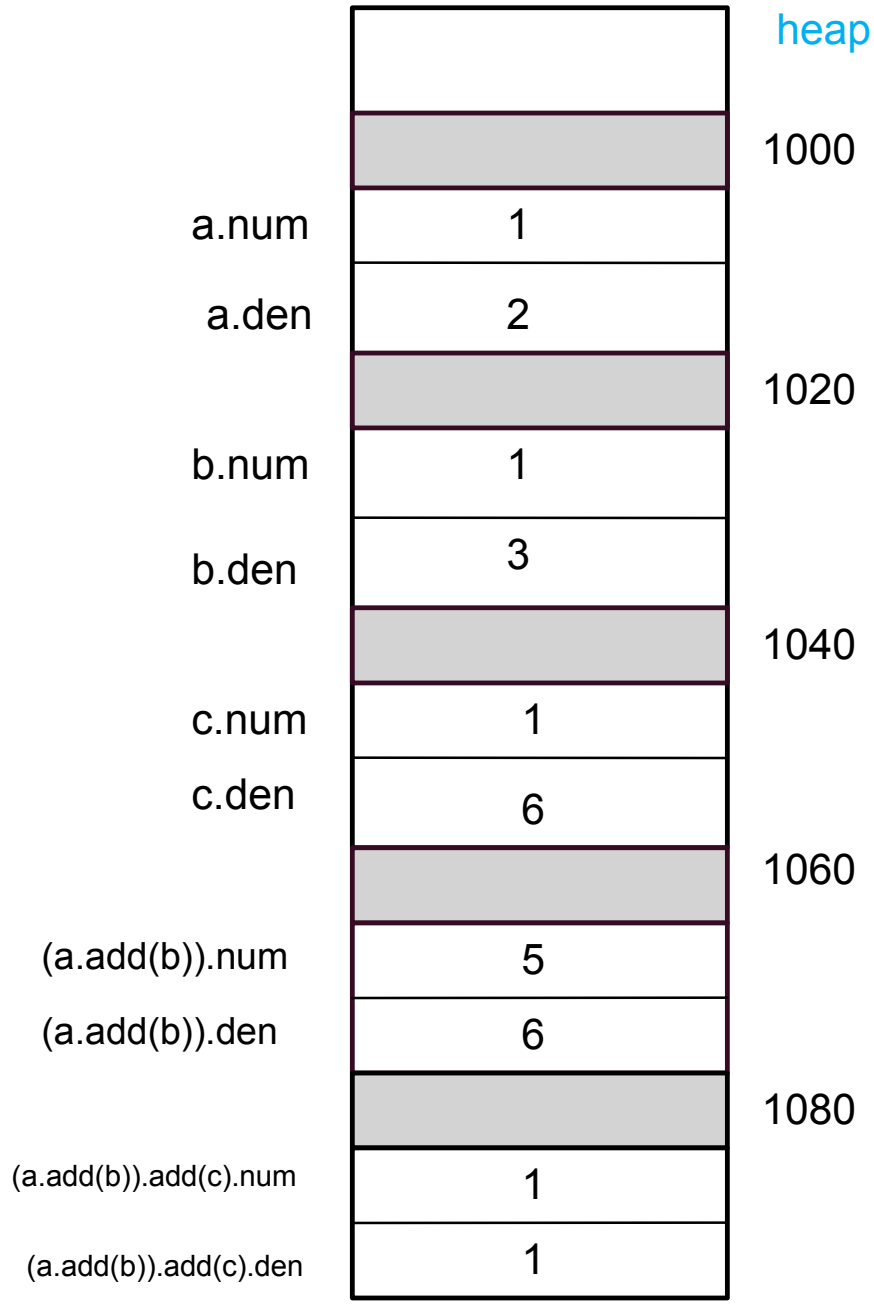
```
public void run() {
        Rational  a = new Rational(1, 2);
        Rational  b = new Rational(1, 3);
        Rational  c = new Rational(1, 6);
        Rational  sum = (a.add(b)).add(c);
}

public  Rational  add(Rational  r) {
     return new Rational(this.num*r.den
                    + r.num*this.den,
                   this.den*r.den);
}
```



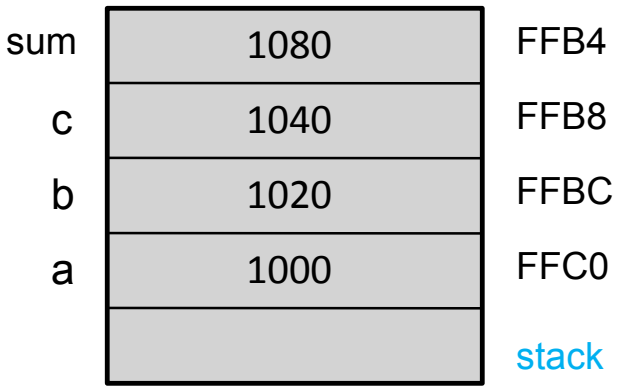


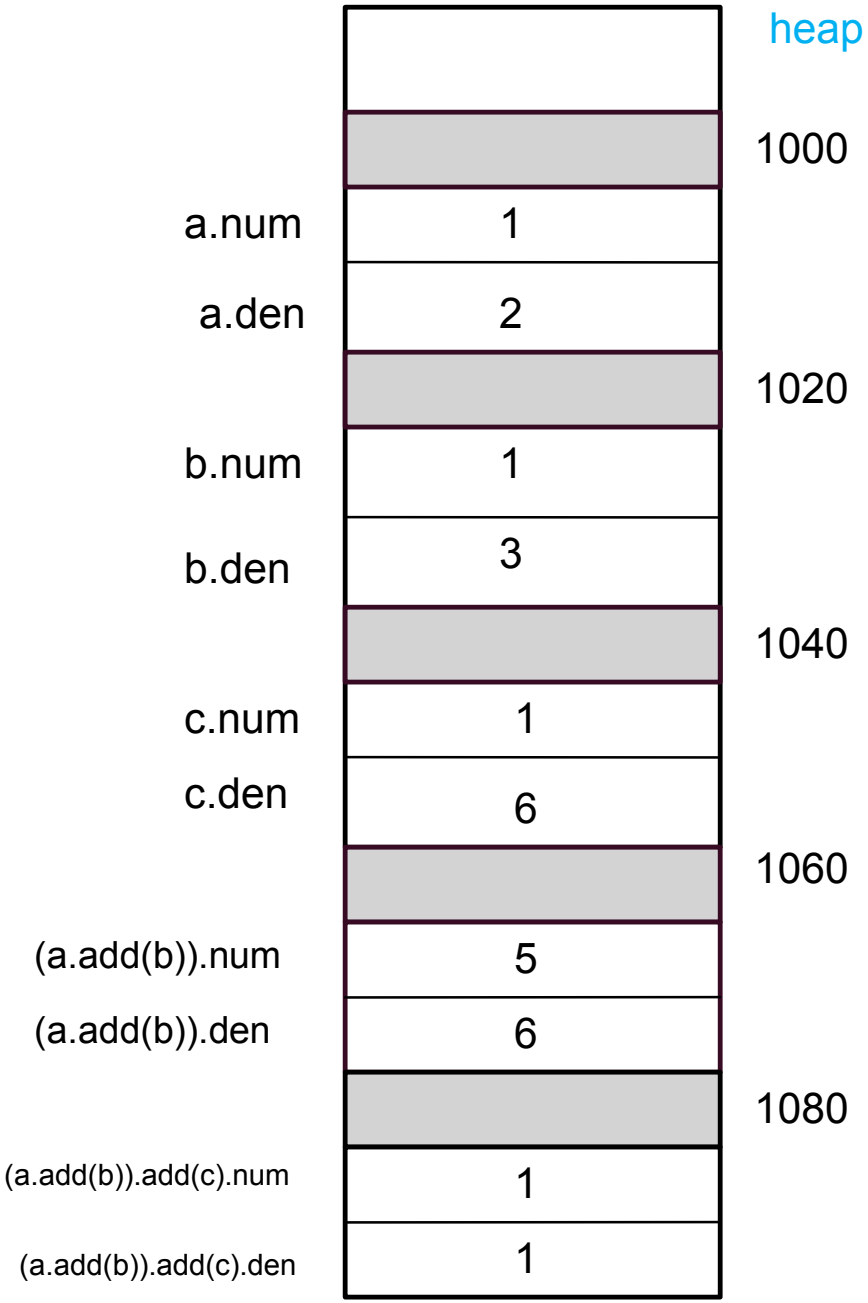
```
public void run() {
        Rational  a = new Rational(1, 2);
        Rational  b = new Rational(1, 3);
        Rational  c = new Rational(1, 6);
        Rational  sum = (a.add(b)).add(c);
}

public  Rational  add(Rational  r) {
     return new Rational(this.num*r.den
                  + r.num*this.den,
                 this.den*r.den);
}
```



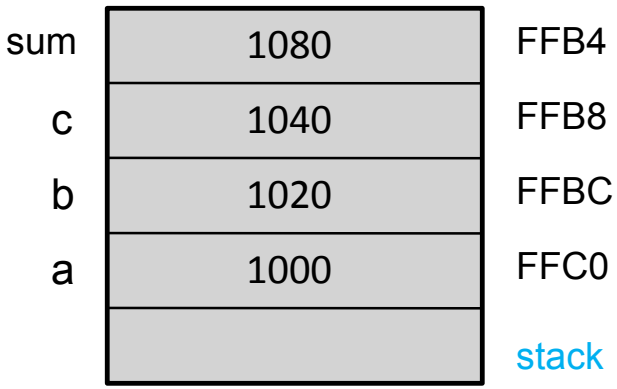


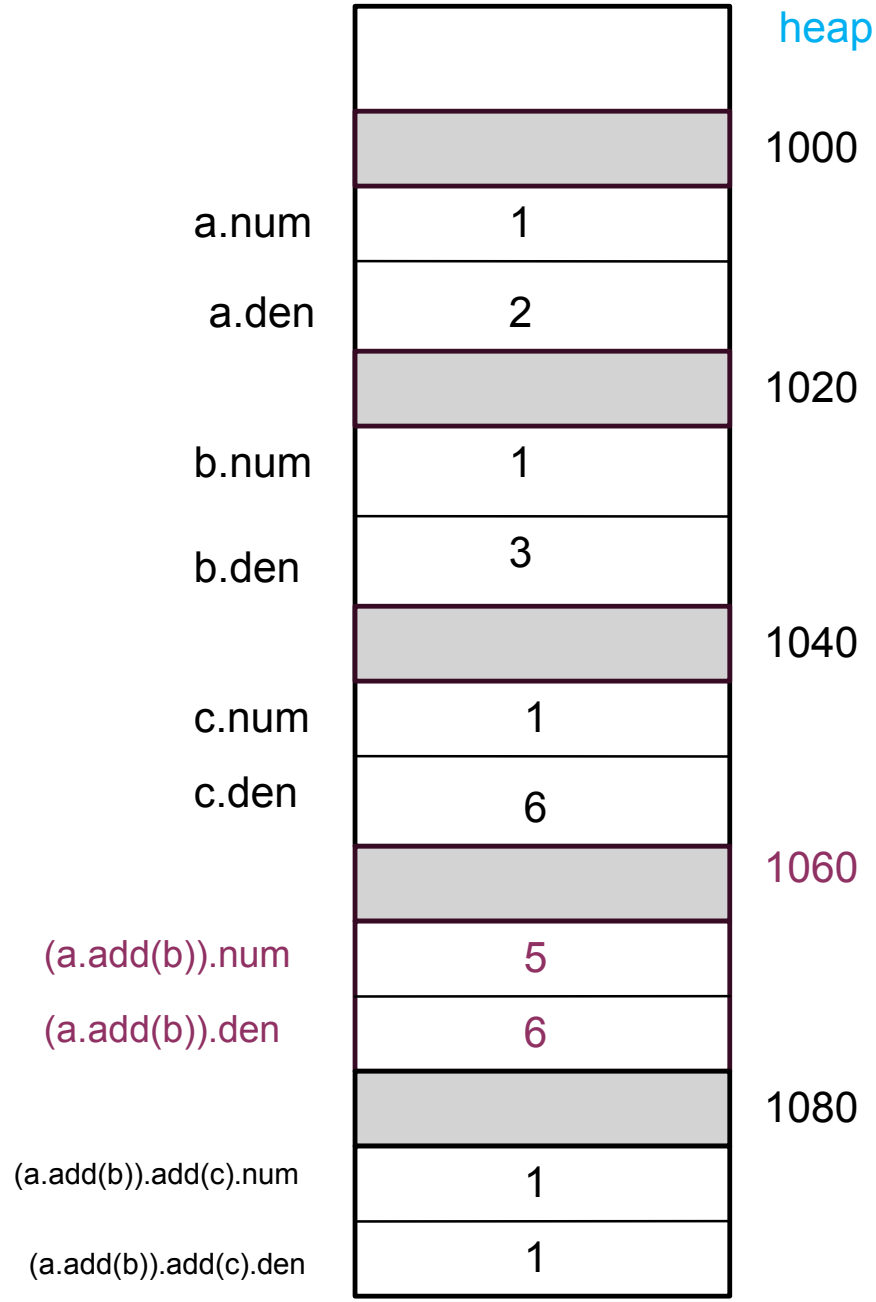
```
public void run() {
        Rational  a = new Rational(1, 2);
        Rational  b = new Rational(1, 3);
        Rational  c = new Rational(1, 6);
        Rational  sum = (a.add(b)).add(c);
}

public  Rational  add(Rational  r) {
     return new Rational(this.num*r.den
                    + r.num*this.den,
                   this.den*r.den);
}
```





```
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);
}

public  Rational  add(Rational  r) {
     return new Rational(this.num*r.den
                    + r.num*this.den,
                   this.den*r.den);
}
```





GARBAGE COLLECTION

- In the example, the object a.add(b) was created in the intermediate step but not referenced by the final stack. It is now **garbage**.
- When memory is running short, Java does garbage collection
  - Mark the objects referenced by variables on stack or in static storage.
  - Sweep all objects in the heap, reclaim unmarked objects (garbage).
- This process is called **garbage collection**.

EXERCISE: STACK-HEAP DIAGRAM

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

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

```
public class Line {
    public Line(Point p1, Point p2) {
        start = p1;
        finish = p2;
    }

    private Point start;
    private Point finish;
}
```

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

Draw a heap-stack diagram (pointer model) showing the state of memory just before the run() method returns.

PRIMITIVE TYPE VERSUS OBJECTS

- Primitive type

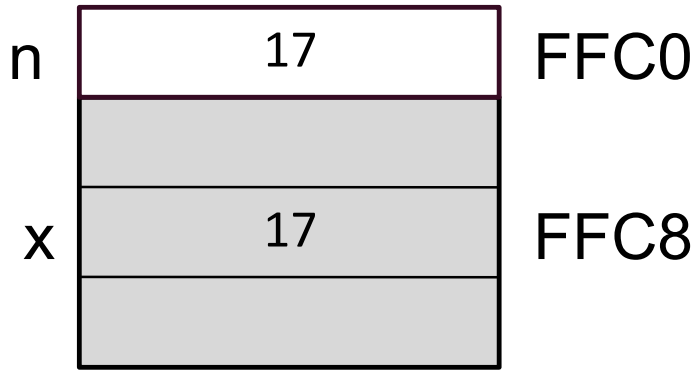
```
public  void run() {
      int  x = 17;
      increment(x);
      println("x = " + x);
}

private void increment(int  n)  {
    n++;
    println("n = " + n);
}
```

Output
n = 18
n = 17

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.

Passing x of primitive type int, a value

increment(x);



x (a value) is copied into n

EMBEDDEDINTEGER CLASS

```
public class EnbeddedInteger {

    public EmbeddedInteger(int n) {
        value = n;
    }

    public void setValue(int n) {
        value = n;
    }

    public  int  getValue() {
        return value;
    }

    public  String  toString() {
        return “” + value;
    }

    private int value;
}
```

- Object

```
public  void  run() {
      EmbeddedInteger  x = new EmbeddedInteger(17);
      increment(x);
      println("x = " + x);
}

private void increment(EmbeddedInteger n) {
    n.setValue(n.getValue()  + 1);
    println("n = " + n);
}
```

Output

```
n = 18
n = 18
```

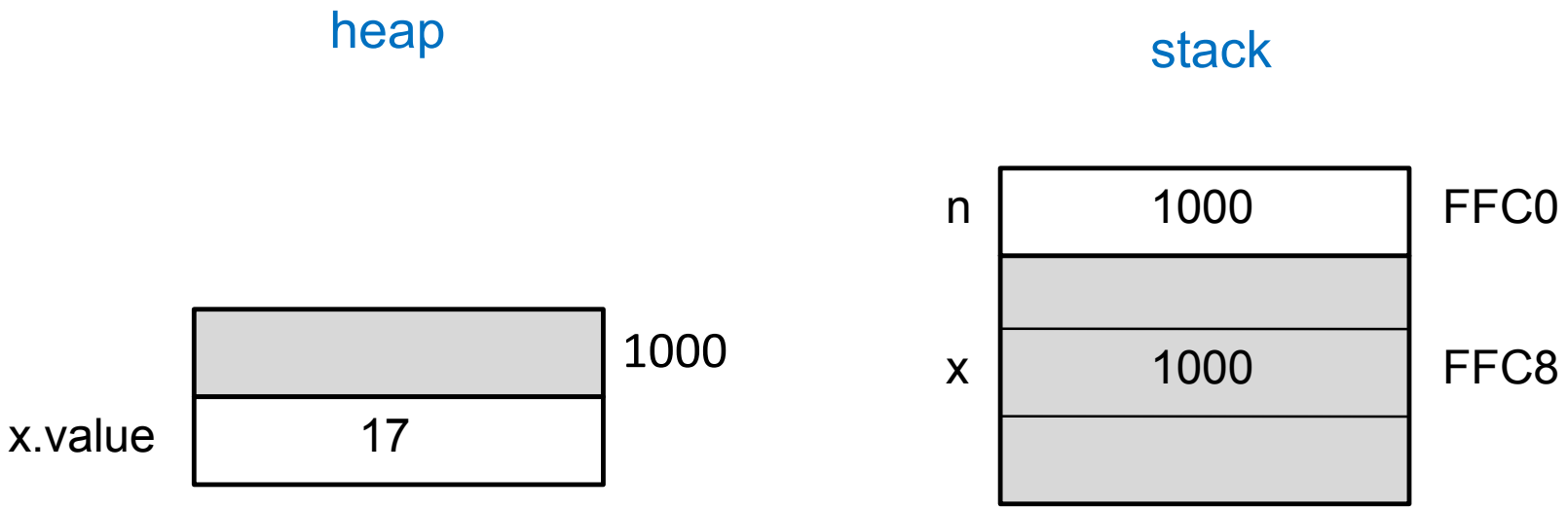
PRIMITIVE TYPES VS OBJECTS

- When you pass an object as an argument, there seems to be some form of sharing going on. However, any changes that you make to the instance variables *inside* an object 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*, not the object itself.

Passing object x, a reference (address)

increment(x)

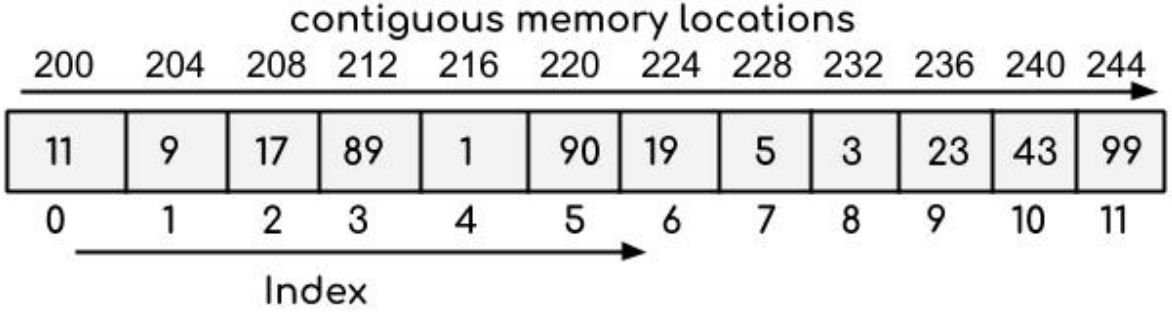


x (a reference to an object) is copied into n
x and n share the same object

OUTLINE

- Basics in Computer Memory
- Arrays
- Implementation
- Examples

ARRAY



- Array: An ordered collection of values
  - Ordered and fixed length
  - Homogeneous: Each value in the array is of the same type
- The individual values in an array are called **elements**.
- The number of elements is called the **length** of the array
- Each element is identified by its position in the array, which is called **index**.
  - In Java, the index numbers begin with 0.

ILLUSTRATION FROM WIKIPEDIA

- Array: a data structure consisting of a collection of elements (values or variables)
  - Each element is identified by at least one array index or key.
  - The memory position of each element can be computed from its index tuple.
  - The simplest type of data structure is a linear array, also called one-dimensional array.

- Example: an array of 10 32-bit (4-byte) integer variables, with indices 0 through 9,
  - May be stored as 10 words at memory addresses 2000, 2004, 2008, ..., 2036, (in hexadecimal: 0x7D0, 0x7D4, 0x7D8, ..., 0x7F4)
  - The element with index i has the address $2000 + (i \times 4)$.

ARRAY DECLARATION

- An array is characterized by
  - Element type
  - Length

```
type[ ]  identifier  =  new type[length];
```

- Default values in initialization
  - numerics 0
  - boolean false
  - objects null

AN ARRAY OF OBJECTS

Elements of an array can be objects of any Java class.

Example: An array of 5 instances of the student class

```
Student [ ]
topStudents = new Student[5];
```

DEFINING LENGTH

- Use named constant to declare the length of an array.

```
private static final int N_JUDGES  =  5;
double[ ]  scores  =  new double[N_JUDGES];
```

- Or read the length of an array from the user.

SELECTING ELEMENTS

- Identifying an element

```
array[index]
```

- Index can be an expression

- Cycling through array elements

```
for (int i = 0; i < array.length; i++) {
    operations involving the ith element;
}
```

What will happen in stack & heap: int[ ] numbers = new int[10];

HUMAN-READABLE INDEX VALUES

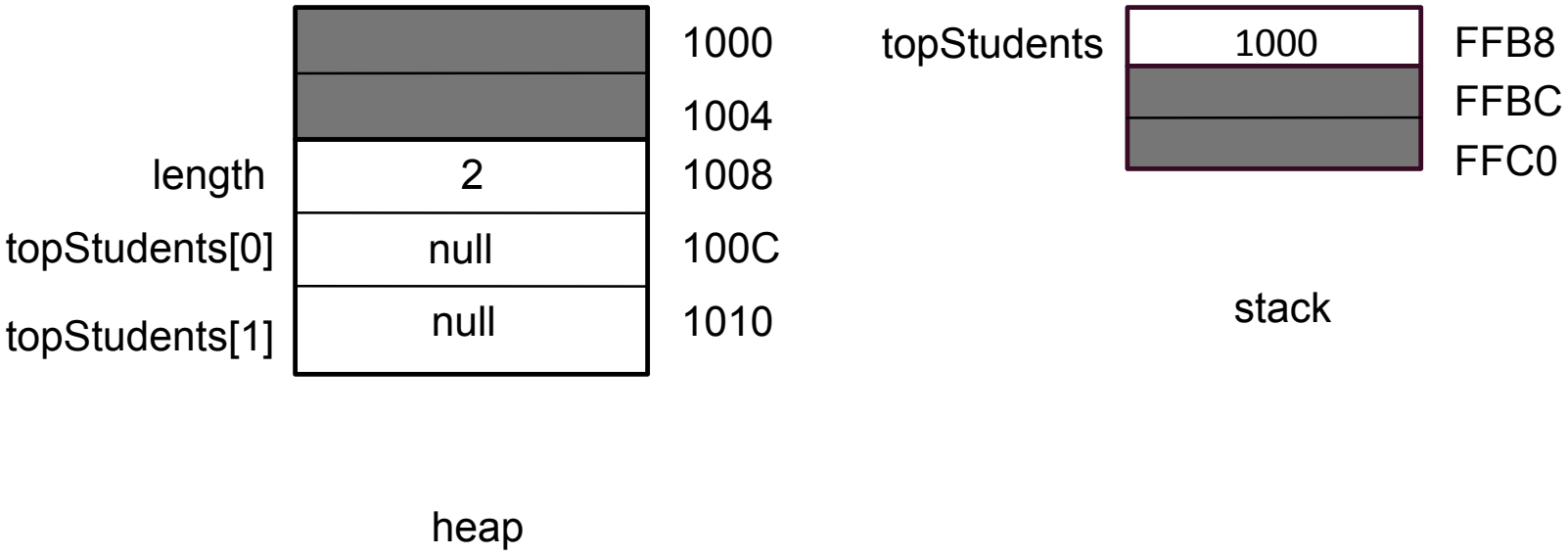
- Starting index numbering at 0 can be confusing.
  - Sometimes, it makes sense to work with index that begins with 1.

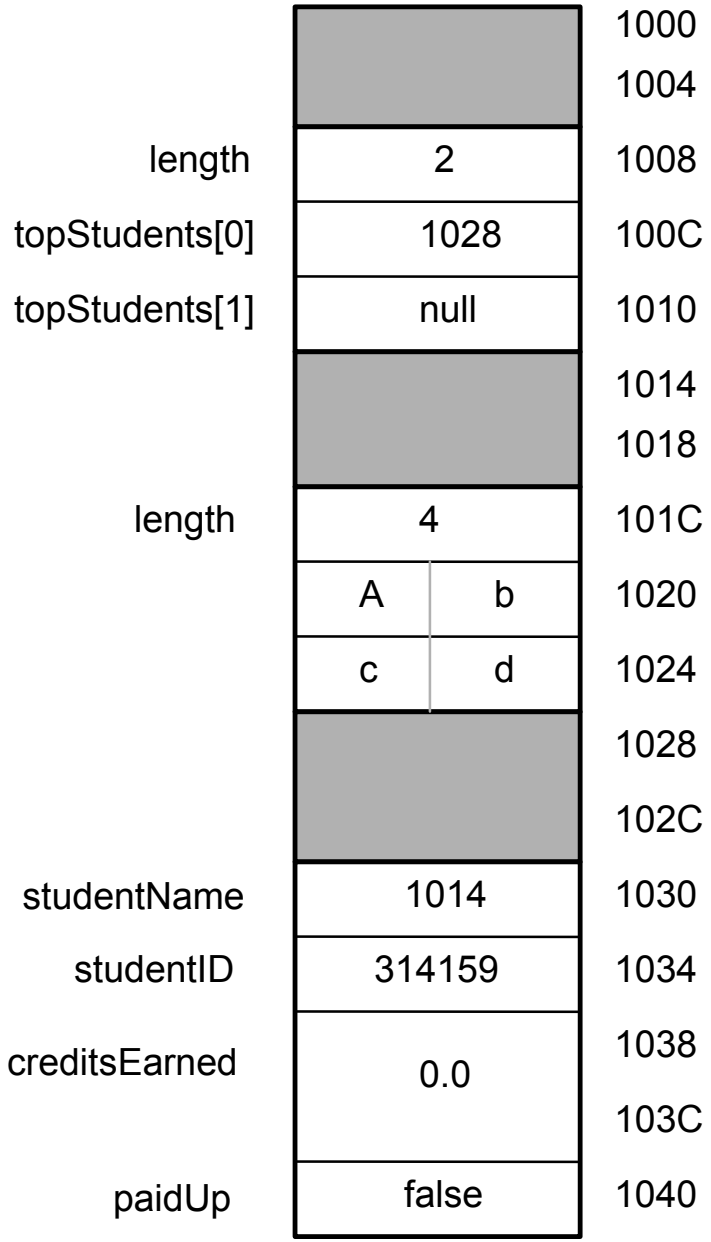
- Two standard ways:
  - Use Java's index number internally and then add one when presenting to the user.
  - Use index values beginning at 1 and ignore the first (0) element in each array.

```java
/**
 * The student class is basic class.
 */
public class Student
{
    /**
     * @param name The student's name
     * @param id student's id
     */
    public Student(String name, int id) {
        studentName = name;
        studentId = id;
    }
    /**
     * Gets name of student
     * @return the name of student
     */
    public String getName() {
        return studentName;
    }
    /**
     * Gets id of student
     * @return the id of student
     */
    public int getId() {
        return studentId;
    }
    /**
     * sets the number of credits enarned.
     * @param credits The new number of credits earned
     */
    public void setCredits(double credits) {
        creditsEarned = credits;
    }
    /**
     * Gets the number of credits earned.
     * @return The number of credits this student has earned
     */
    public double getCredits() {
        return creditsEarned;
    }
    /**
     * Sets whether the student is paid up.
     * @param flag The value true or false indicating paid-up status
     */
    public void setPaidUp(boolean flag) {
        paidUp = flag;
    }
    /**
     * Returns whether the sutdent is paid up.
     * @return Whether the student is paid up.
     */
    public boolean isPaidUp() {
        return paidUp;
    }
    /**
     * Creates a string identifying this student.
     * @return The string used to display this student.
     */
    public String toString() {
        return studentName + " (#" + studentId + ")";
    }

    /* public constants */
    public static final double CREDITS_TO_GRADUATE = 32.0;
    /* Private instance variables */
    private String studentName;        /* The student's name */
    private int studentId;             /* The student's id */
    private double creditsEarned;      /* The number of credits earned */
    private boolean paidUp;            /* Whether student is paid up */
}
```

INTERNAL REPRESENTATION OF ARRAYS

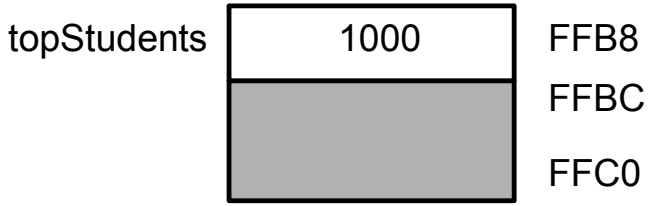
```
Student[ ]  topStudents  =  new Student[2];
topStudents[0] = new Student("Abcd", 314159);
```





Student[] topStudents = new Student[2];

topStudents[0] = new Student("Abcd", 314159);



PASSING ARRAYS AS PARAMETERS

- Recall: Passing objects (references) versus primitive type (values) as parameters.

- Java defines all arrays as objects, implying that the elements of an array are shared between the callee and the caller.

  swapElements(array[i], array[n – i – 1]) (wrong)

  swapElements(array, i, n – i – 1)

```
private void swapElements(int[] array, int p1, int p2) {
        int  tmp = array[p1];
        array[p1] = array[p2];
        array[p2] = tmp;
}
```

- Every array in Java has a length field.

```
private void reverseArray(int[] array)  {
        for (int  i = 0; i < array.length / 2; i++) {
                swapElements(array, i, array.length – i – 1);
        }
}
```

USING ARRAYS

- Example: Letter frequency table

Array: letterCounts[ ]

index: distance from ‘A’

index = Character.toUpperCase(ch) – ‘A’

letterCounts[0] is the count for ‘A’ or ‘a’

A convenient way of initializing an array:

```
int[ ] digits = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

private static final String[ ] US_CITIES_OVER_ONE_MILLION = {
      “New York”,
      “Los Angeles”,
      “Chicago”,
      “Huston”,
      “Philadelphia”,
      “Phoenix”,
      “San Diego”,
      “San Antonio”,
      “Dallas”,
}
```

TWO-DIMENSIONAL ARRAYS

- Each element of an array is an array (of the same dimension)

  ```
  int[][] A = new int[3][2];
  ```

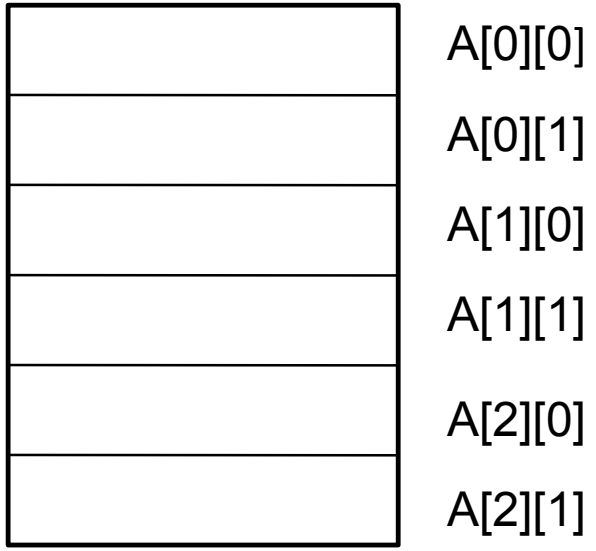
- An array of three arrays of dimension two

  A[0][0] A[0][1]

  A[1][0] A[1][1]

  A[2][0] A[2][0]

Memory allocation (row orientation)



INITIALIZING A TWO-DIMENSIONAL ARRAY

```
static int A[3][2] =  {
    {1,  4},
    {2,  5},
    {3,  6}
};
```

A 3-by-2 matrix

THE `ARRAYLIST` CLASS

- The **`java.util`** package includes a class called **`ArrayList`**
  - Provide standard array behaviors along with other useful operations.

- **`ArrayList`** is a Java class rather than a special form in the language. All operations on **`ArrayList`**s are indicated using method calls.
  - Create a new **`ArrayList`** by calling the **`ArrayList`** constructor.
  - Get the number of elements by calling the **`size`** method.
  - Use the **`get`** and **`set`** methods to select individual elements.

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

OUTLINE

- Basics in Computer Memory
- Arrays
- Implementation
- Examples

OUTLINE

- Basics in Computer Memory
- Arrays
- Implementation
- Examples