

---

## Chapter 13

# Memory Usage and Pointers

---

### What is in This Chapter ?

In this chapter we discuss the memory usage in JAVA when using both primitive and object data types. It is important to understand where data is being stored and how to access data that is within another object. Ultimately, it all boils down to knowing which memory address contains the data that we want to access or modify. The notion of ***pointers*** will be discussed and an example using a **doubly-linked list** will help you understand the use of pointers (or references) in JAVA.

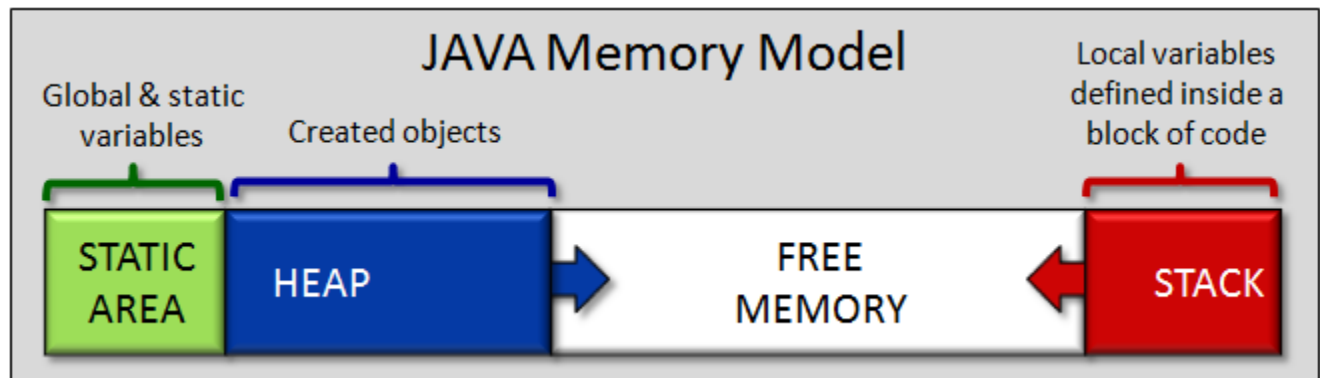


## 13.1 Memory Allocation

In order to understand how objects are stored, it is first necessary to understand how your computer's memory gets "used-up" as your program runs. The Java Virtual Machine is allotted a certain amount of memory space on your computer when your program begins to run. (This amount of memory allotted is adjustable via command-line arguments).

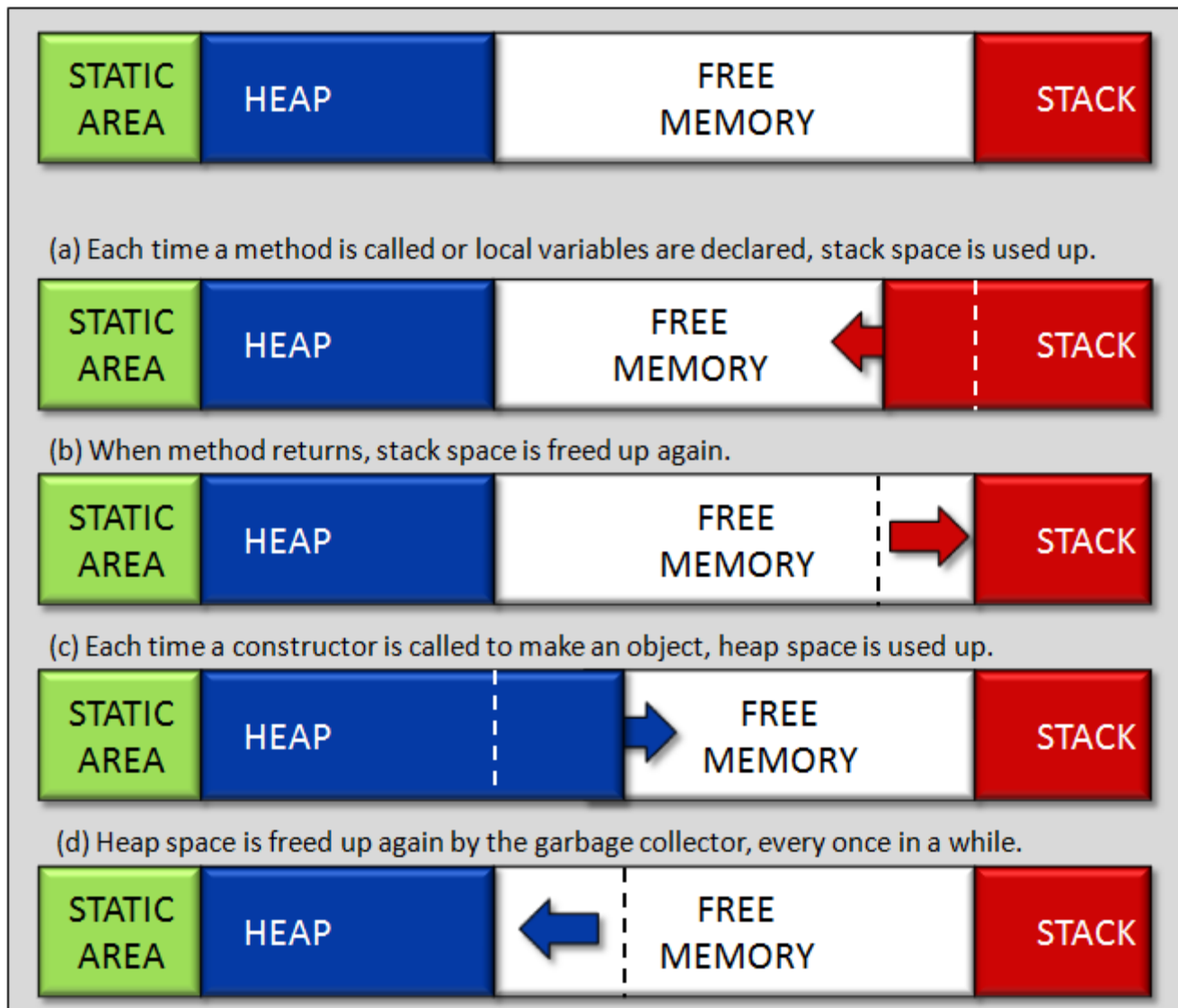
Upon start-up, some of this allotted memory is used up by the JVM. The remaining memory that is available for your program is denoted as "free memory". As your program runs, it will **allocate** (i.e., **use up**) some of this free memory at various times throughout the runtime of the program. Your program will also **return** (i.e., **free up**) this used memory at various times as it completes portions of your program. Hence the amount of available free memory will shrink and grow throughout the execution of the program.

If we consider a snapshot at any time, the memory is broken up into 4 main logical portions as shown here:



1. The **Static Area** of memory is memory that is used by the global and static variables that are defined by your program. This memory usage is fixed and does not change as the program runs.
2. The **Free Memory** is the memory that is not currently being used by your program. If this memory ever gets used up during your program, you will get an "Out Of Memory" error and your program will stop running.
3. The **Stack** memory is the memory that is used to store local variables. It also gets used up a little each time you call a method or run code within a block of code (i.e., a block is any code within braces). The amount of memory used during a method call depends on the number and size of the local variables defined in the method as well as its parameters.
4. The **Heap** memory is the memory that stores all the objects that you create. Each time that you call a constructor by using the new keyword, the **Heap** memory will increase.

The following diagram shows how the Stack and Heap memory grows and shrinks over time:



Interestingly, in JAVA, there is no way explicitly to free up heap memory from objects that you no longer want to use. The garbage collector handles this for you. You can "suggest" that the garbage collector free up memory at any time in your program by using **System.gc()**. However, this does not ensure that garbage collection will take place immediately. It is often suggested to set object-type (i.e., non primitive type) variables to **null** so that the garbage collector will realize that you are no longer holding on to an object and can free it sooner. Ultimately, the success of this strategy depends on how the garbage collector has been implemented.

For now, let us try to understand how data is stored in the stack memory.

Recall the 8 primitive data types in JAVA and the amount of memory that each requires:

Type	Bytes Used	Can Store Values Within This Range
<b>byte</b>	1	-128 to +127
<b>short</b>	2	-32,768 to +32,767
<b>int</b>	4	-2,147,483,648 to +2,147,483,647
<b>long</b>	8	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
<b>float</b>	4	$-10^{38}$ to $+10^{38}$
<b>double</b>	8	$-10^{308}$ to $+10^{308}$
<b>char</b>	2	any ASCII or UNICODE character (e.g., 'A','a','1','*','>', etc..)
<b>boolean</b>	1	true or false

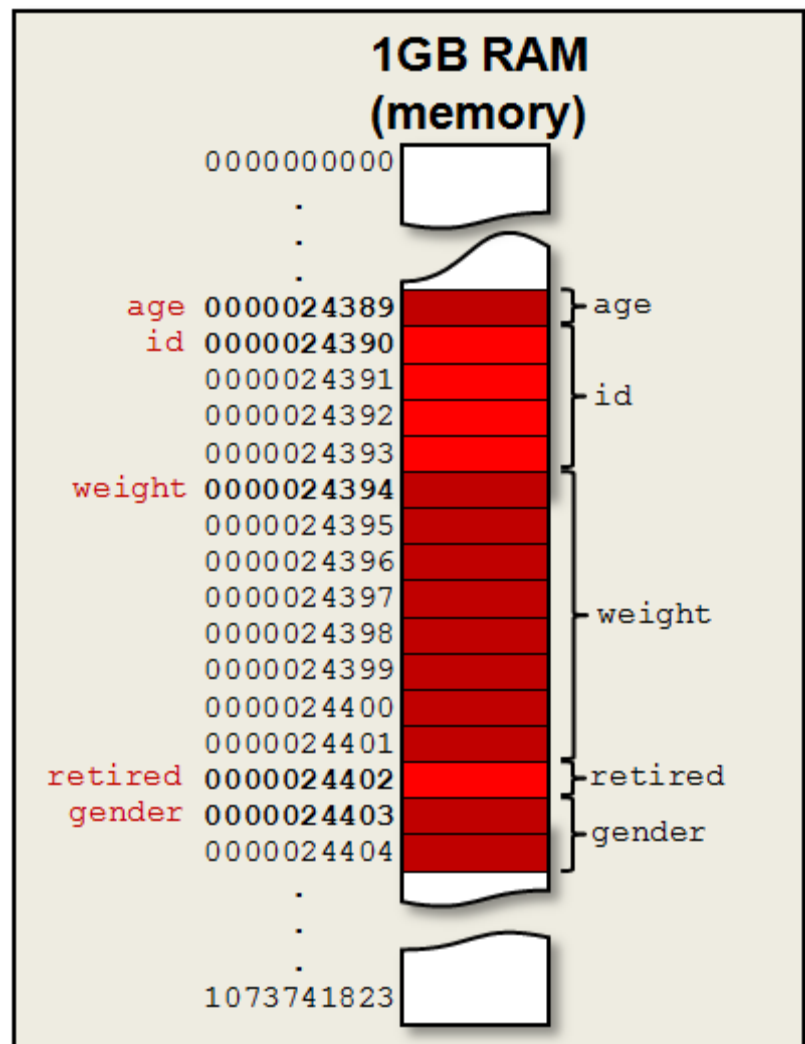
Each time we declare a variable, it reserves enough space in the stack memory to store the data.

For example, consider the following code and notice the amount of memory that it consumes in the stack memory:

```
byte    age;
int     id;
float   weight;
boolean retired;
char    gender;
```

Usually, the space for the variables is declared consecutively as shown here. JAVA automatically reserves this space for us when we declare these variables. Each variable begins at a unique address in the computer's memory.

When using the variables, in our program, the value for the variable is obtained by simply looking at the address location to obtain the information. Similarly, when assigning values to the variables, the address is used to know where to start storing the information.

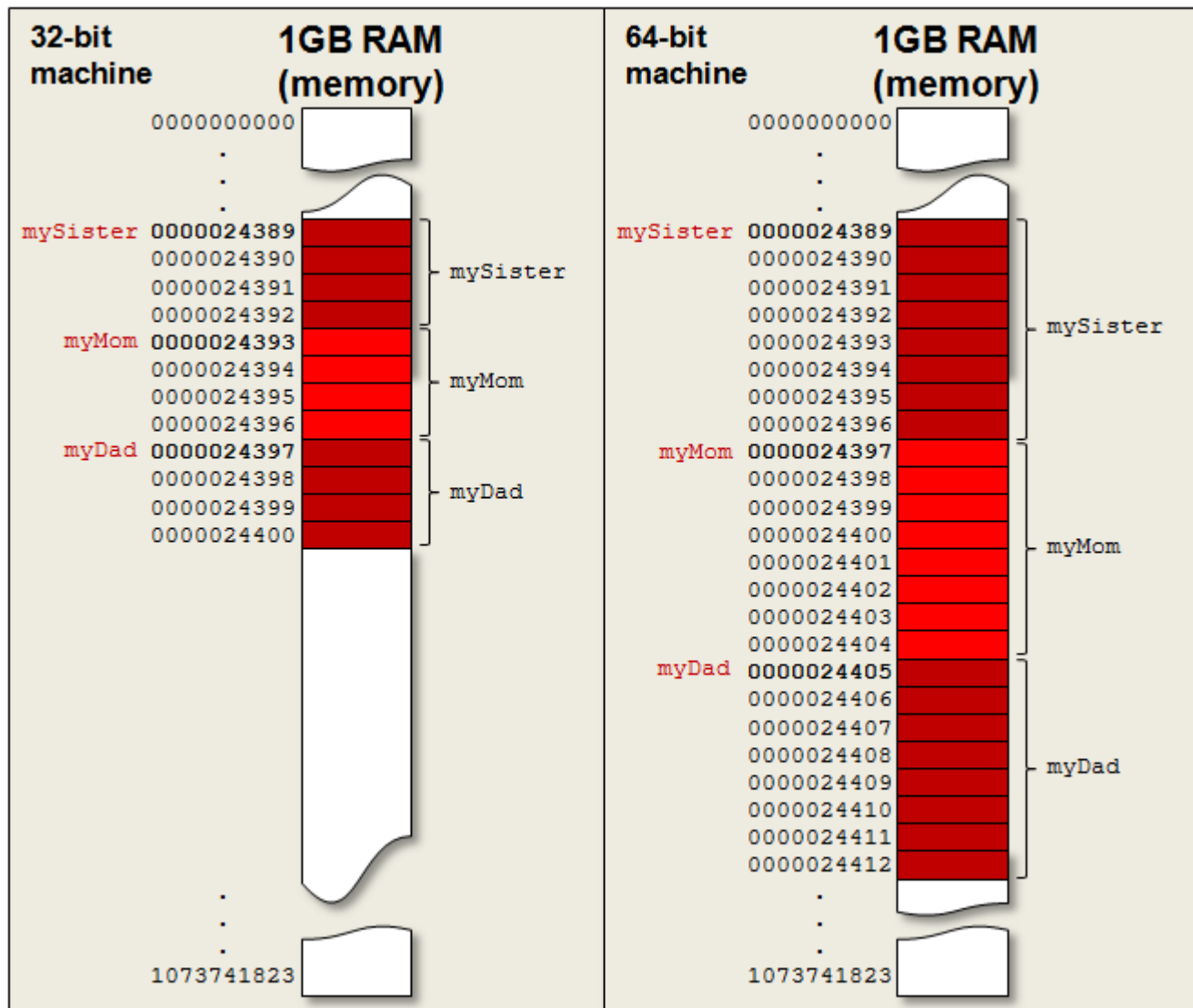


Consider now a **Person** object that stores only primitive data types as follows:

```
public class Person {
    byte    age;
    int     id;
    double  weight;
    boolean retired;
    char    gender;
}
```

Notice how the object would be stored in memory on 32-bit and 64-bit machines if we were to declare a few variables of type **Person** as follows:

```
Person mySister;
Person myMom;
Person myDad;
```



Notice that on a 32-bit machine each variable requires just 4 bytes ... and requires 8 bytes on a 64-bit machine. These bytes represent **pointers** to the location in memory that the object will actually reside. A 32-bit machine has a 32-bit address space ... and so 4 bytes are required to store each address reference (i.e., each object variable). A 64-bit machine has a 64-bit address space ... and so each object variable requires an extra 4 byte overhead (i.e., double the space). So 64-bit machines, although they may be faster for CPU-related operations, may require more space allocation by default (there are ways to "compress" the pointer references...but this is not discussed here). From this point onwards in the notes, unless otherwise stated explicitly, we will assume that we are using a 32-bit machine in order to simplify the discussion.

You will also notice that the space is not reserved for storing any of the actual data inside the object. Storing the data inside the object would require **16** bytes of storage to store the **age** (1 byte), **id** (4 bytes), **weight** (8 bytes), **retired** (1 byte) and **gender** (2 bytes) information. However, this space is not reserved until the object is created by calling its constructor.

At this point, we just get a **reference** (or pointer) to the location of the object in memory. Since we have yet to create the object ... the value of the pointer is **null**. So, **null** actually represents an undefined memory address which requires **4** bytes of storage at all times (64-bit machines require **8** bytes to store each pointer). That means, each time that we will use objects in java, there is always a **4**-byte overhead (**8**-byte for 64-bit machines) to store the reference to the object.

Now what about the object itself ? Consider what happens when we create the object via a constructor as follows:

```
mySister = new Person();
```

This is now a constructor call, so the memory that will be used to store the object's data will be the Heap memory. The amount of memory used up depends on the object's data values. Looking at the class definition of the Person object, you will notice that it contains only primitive data types ... each of which has a fixed size. The object requires 16 bytes to store your data. However, each created object in JAVA requires an additional storage overhead of **8** bytes to store an **object header**. The data contained in the header is implementation-specific ... so it depends on the particular java implementation that you are using. In fact, it is possible that other java implementations may even vary the amount of space used in the header.

Also, in some cases, additional bytes are allocated in memory to ensure that the entire object uses a multiple of **8** bytes. That is, our current **Person** object stores **16** bytes of data ... a nice multiple of **8**. However, if we were to add an additional boolean attribute to the **Person** object definition, for example, then it would take up **17** bytes. In that case, java will probably reserve an additional **7** more bytes to bring the total up to **24** bytes so that the entire object again uses a multiple of **8** bytes. These extra **7** bytes would be unused, but nevertheless allocated.

So, each Person object that we create will require (16 + 8 = 24) bytes of storage as shown in the diagram on the following page.

Notice as well that the **mySister** variable now points to the location where the **Person** object is being stored (i.e., address 0008237846 in our example).

So, the integer value of **0008237846** will be stored as the pointer at address location **0000024389**. Whenever we therefore use the **mySister** variable, JAVA just looks at its value and follows the pointer to find the object.

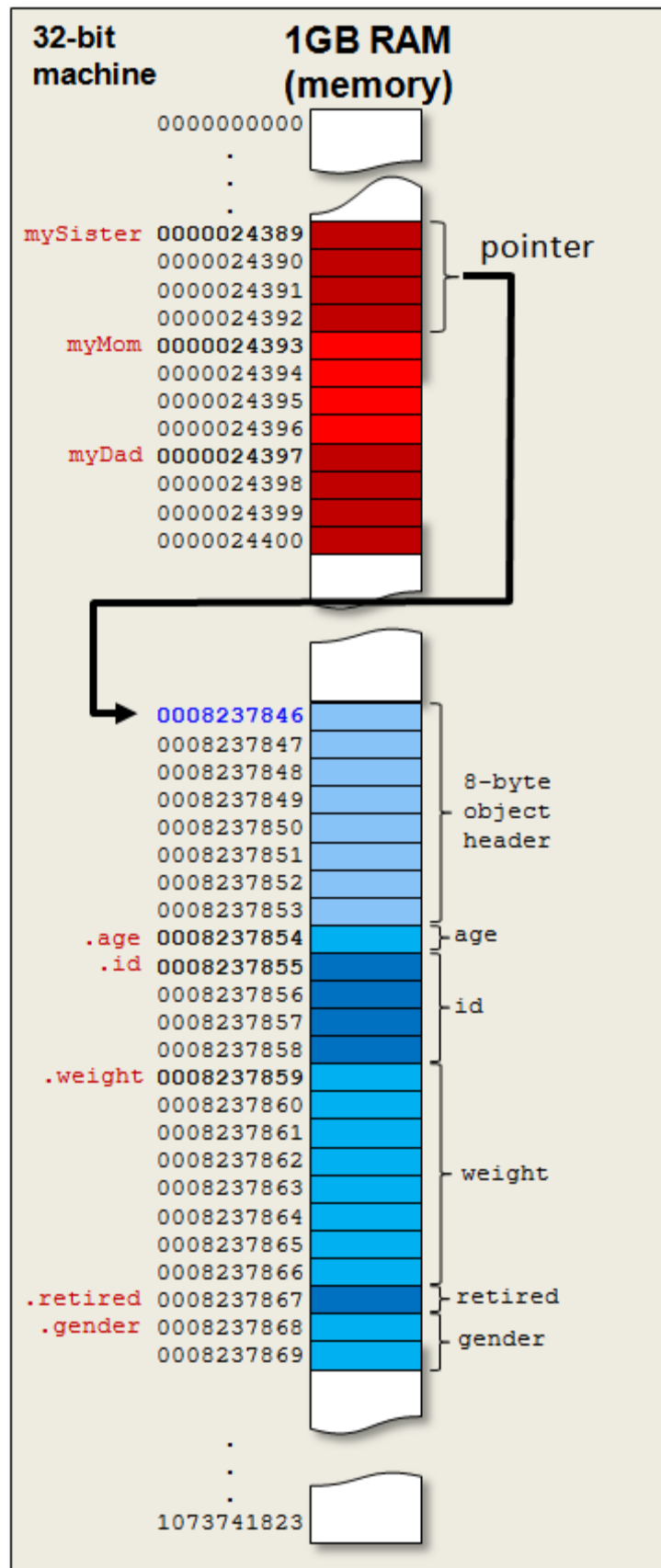
What happens when we access the internals of the **Person** object by using the dot operator ?

```
mySister.age = 15;
if (mySister.retired) {
    ...
}
```

Well, JAVA begins with the address stored in the **mySister** variable (i.e., 0008237846) and then adds to that value the offset that the **.age** portion of the object is with respect to the start of the object (i.e., adds 8 more bytes to bypass the header). The result is  $0008237846 + 8 = 0008237854$ .

Also when accessing the **.retired** portion of the object, the offset from the start of the object is  $8 + 1 + 4 + 8 = 21$  bytes. Hence the retired value is found at address location  $0008237846 + 21 = 0008237867$ .

Now what happens when an object is contained within another object ? Well, there is nothing really new that is going on. Let's look at an example.



Consider two simple objects defined as follows:

```
public class GPSLocation {
    float    latitude;
    float    longitude;
}

public class University {
    short     opened;
    String     name;
    GPSLocation location;
}
```

Now consider the following code:

```
GPSLocation    herzberg;
University     carleton;

herzberg = new GPSLocation();
herzberg.latitude = 45.382149;
herzberg.longitude = -75.697304;

carleton = new University();
carleton.opened = 1942;
carleton.name = "Carleton";
carleton.location = herzberg;
```

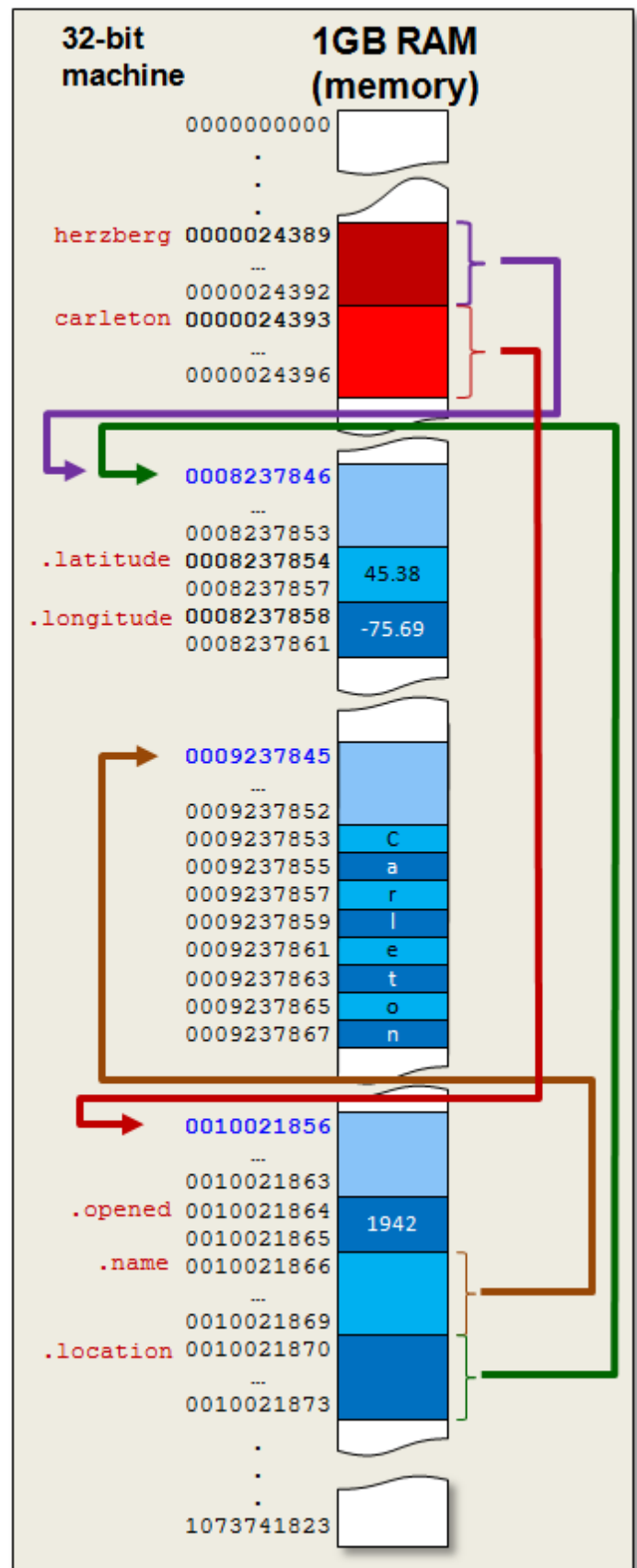
Notice what the memory allocation will look like for this example (the picture is condensed a little vertically to fit onto the page) ---->

You can see that there are three objects created:

- the String "Carleton"
- the GPSLocation
- the University

The **carleton** variable points to the **University** object which itself contains pointers to the **String** object and the **GPSLocation** object.

The **herzberg** variable also points to the **GPSLocation** object and so the address value stored at locations **0000024389** and **0010021870** are the same ... which is **0008237846**.





## 13.1 Doubly-Linked Lists

Consider allocating a large byte array:

```
byte[] myArray;

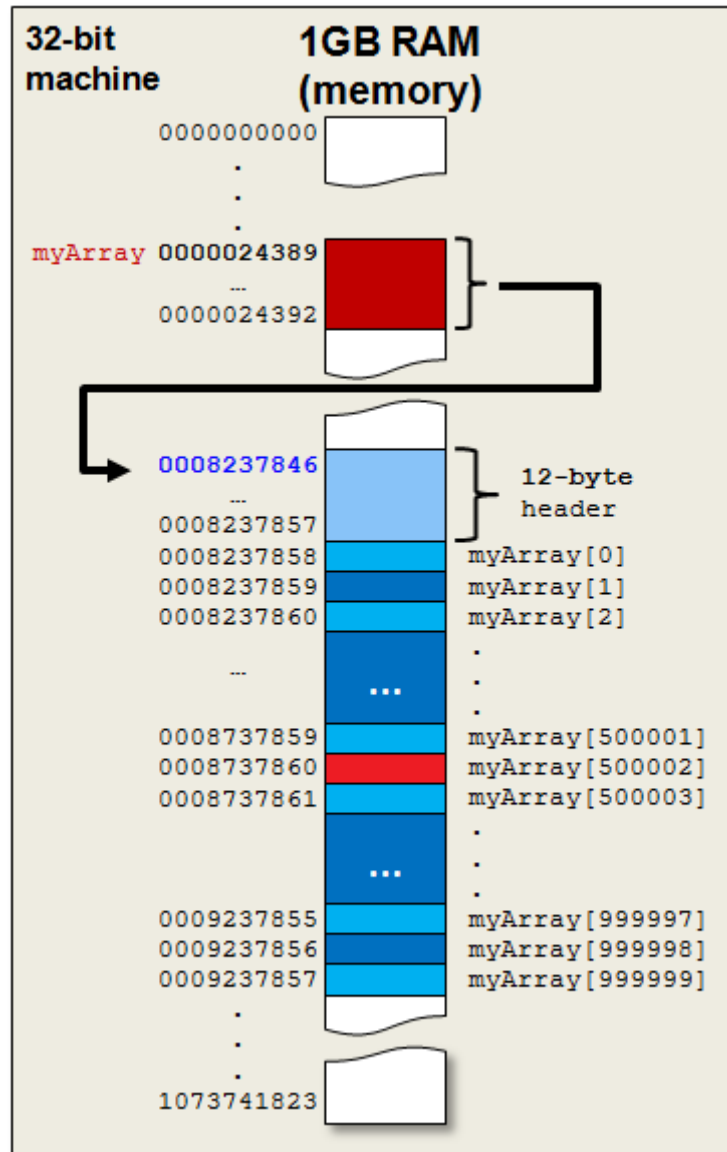
myArray = new byte[100000];
```

An array is an object and the values of the array are kept in consecutive memory locations. Note that usually the array length is also kept along with the array, so we have shown this as an extra 4 bytes in the object header ... making it a 12-byte header (although the exact size depends on the java implementation).

Now, consider filling up this array with some appropriate values. Now consider what happens when we want to remove the item at position 500002 in the array (shown red).

Remember that we cannot simply remove something from an array but that we have 2 choices:

1. replace the item at index 500002 with some clearly identifiable value (e.g., -1).
2. The second option is to actually remove the item at that position by copying all the items from index 500003 to 999999 back 1 position in the array and then reduce the size of the array by 1.



Solution (1) is quick to do a remove operation, but then we will leave "gaps" in the array so that when we process it later we will need to consider the fact that there may be a lot of invalid data stored in the array at any time. In fact, after a while ... the array may be filled with mostly invalid data!

Solution (2) would take a lot more time to remove an item because we would potentially need to move large portions of the array back one position in memory each time we do a remove operation. In addition, we can "logically" reduce the array size by one, but in reality, JAVA has already allocated the memory for the 100,000 elements ... so that will not change. In other words, we are essentially classifying the "end portion" of the array as garbage data as time goes on. We are not saving any space ... the garbage/wasted data is still taking up memory.

This problem gets worse as we consider adding items to the array beyond the 100,000 capacity. In that case, we would need to create a whole new bigger array and copy all the items from the "old" array into the "new" array ... then discard the old array. To do this, we would simply move the **myArray** variable pointer to point to the new array:

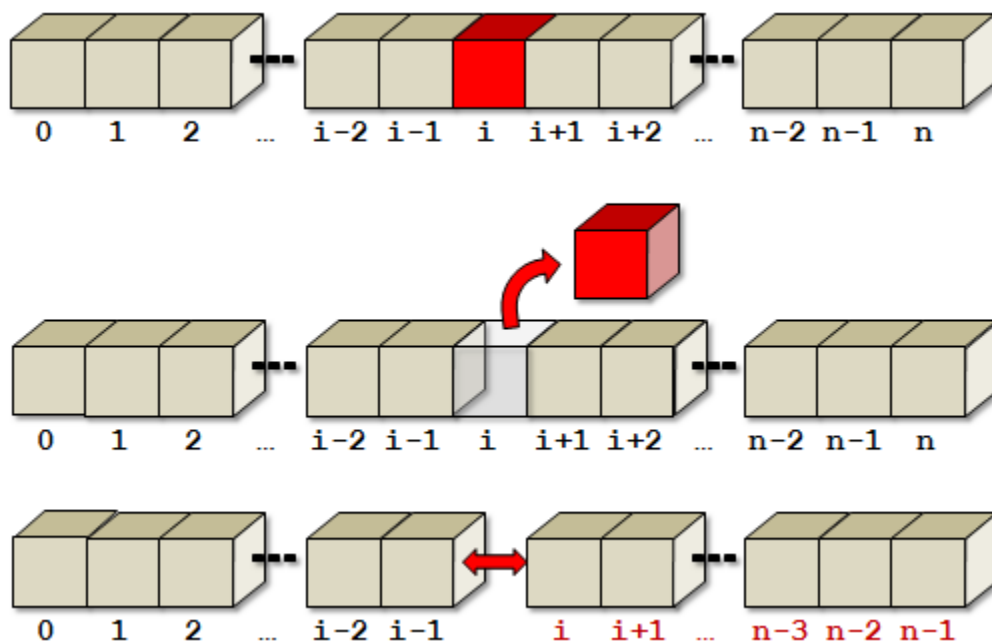
```
myArray = new byte[200000];
```

This would create a whole new object which takes another 200,000 bytes (+12 for header). The Java VM would then realize that the data in memory from address 0008237846 to 0009237857 would no longer be needed and it would be scheduled for a future garbage collection operation. In languages such as C or C++, there is no garbage collector, so we would have to remember to free up that memory on our own.

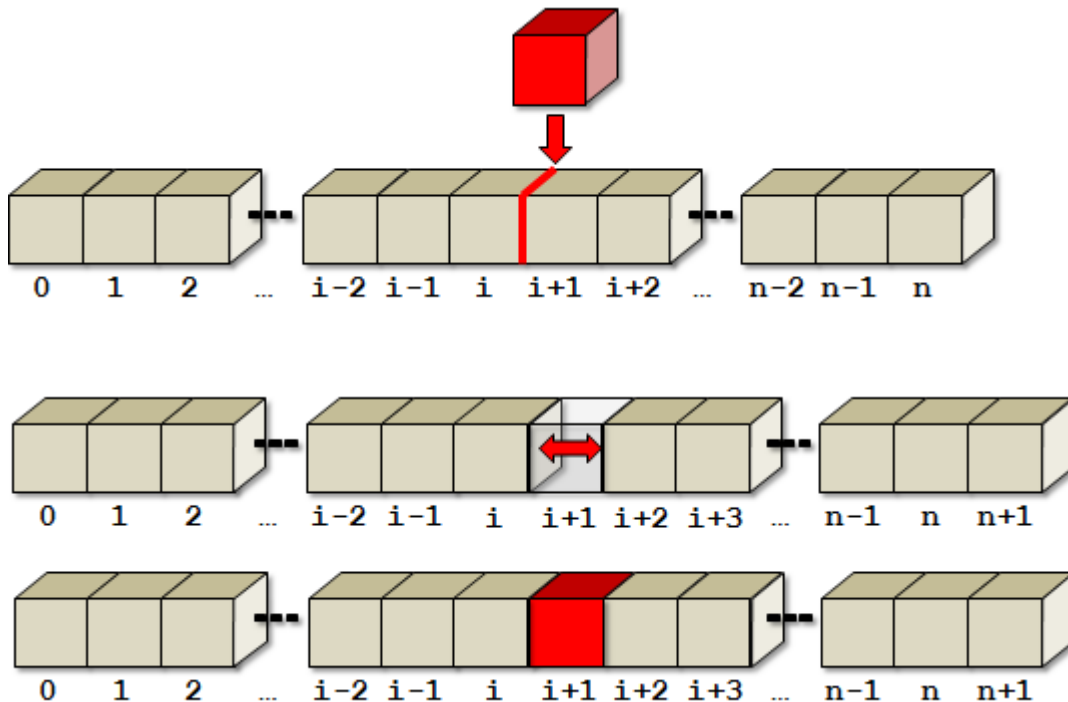
One huge danger of this "new-array-reallocation-and-copy-over" strategy is that if any other objects are pointing to the old array ... then it is not garbage collected and potentially we have two places in our code that at one time may have been pointing to the same array but are now pointing to different arrays !!

One solution to this problem is to store data in what is called a **doubly-linked list**. What we would like to be able to do is to:

1. cut out a single piece of data and stitch the remaining data back together:



2. cut open a spot in the data and insert a single piece of data inside:



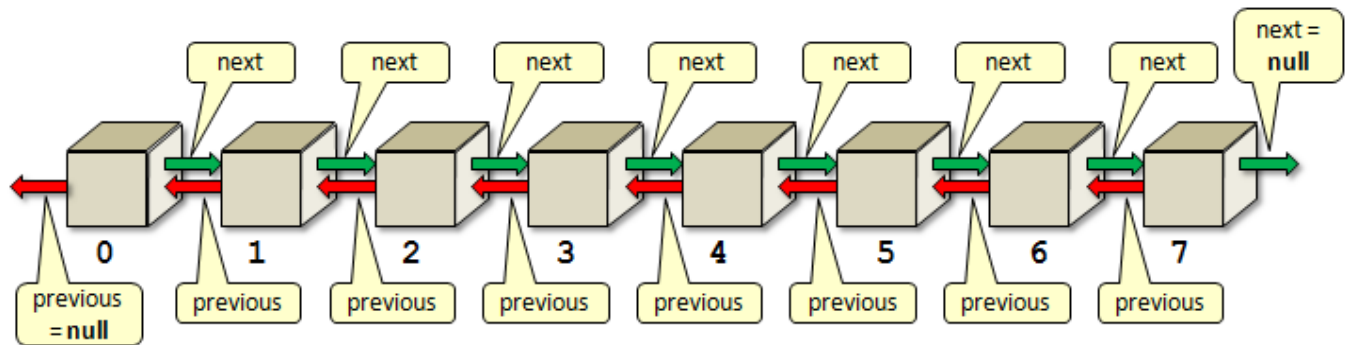
To do this, we need to allow the data to be split and merged anywhere within the list of data. We can do this by allowing each piece of data to be its own object. As long as each of these objects knows the object before it in the list as well as the object after it in the list, then we can make this happen.

Consider a single item in the list represented as follows:

```
public class Item {
    byte data;
    Item previous;
    Item next;

    public Item(byte d) {
        data = d;
        previous = null;
        next = null;
    }
}
```

Notice that this **Item** class represents a recursive data structure definition since the item before (i.e., *previous*) this item is an **Item** object and the item after (i.e., *next*) it is also an **Item** object. That means, the Items each keep a pointer to the object before it in the list and the object after it in the list. So we can re-draw our n-element list now as follows:



Consider a simple array created as follows:

```

byte[]    myList = new byte[8];
myList[0] = 23;
myList[1] = 65;
myList[2] = 87;
myList[3] = 45;
myList[4] = 56;
myList[5] = 34;
myList[6] = 95;
myList[7] = 71;
  
```

Here is how we would create the linked-list version for this list of data:

```

Item myList = new Item(23);
Item myList1 = new Item(65);
Item myList2 = new Item(87);
Item myList3 = new Item(45);
Item myList4 = new Item(56);
Item myList5 = new Item(34);
Item myList6 = new Item(95);
Item myList7 = new Item(71);

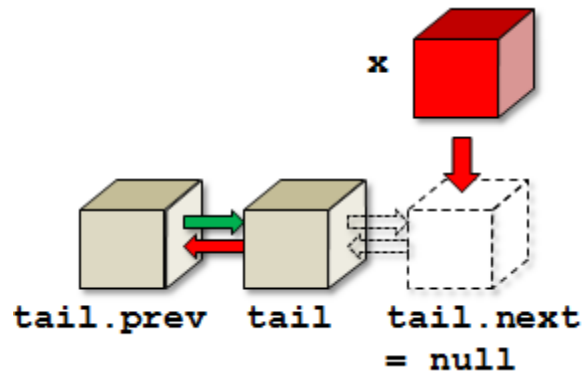
myList.previous = null;
myList.next = simple1;
myList1.previous = simple;
myList1.next = simple2;
myList2.previous = simple1;
myList2.next = simple3;
myList3.previous = simple2;
myList3.next = simple4;
myList4.previous = simple3;
myList4.next = simple5;
myList5.previous = simple4;
myList5.next = simple6;
myList6.previous = simple5;
myList6.next = simple7;
myList7.previous = simple6;
myList7.next = null;
  
```

This code is a bit ugly because it uses many variable names. However, typically we would create operations for adding and removing Items. Also, we usually want to keep track of the first and last items in the linked list ... which are known as the **head** and the **tail**. So, often we create another class to keep track of this information as follows:

```
public class LinkedList {
    Item head;
    Item tail;

    public LinkedList() {
        head = null;
        tail = null;
    }
}
```

Then we would make operations in this class. One useful operation would be to add an item to the end (i.e., tail) of the list as follows:



Here is the code that will do this:

```
public void add(Item x) {
    if (tail == null) {
        tail = x;
        head = x;
    }
    else {
        tail.next = x;
        x.previous = tail;
        tail = x;
    }
}
```

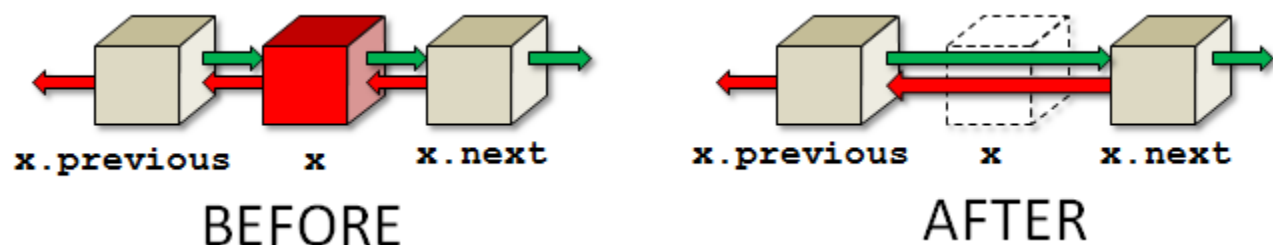
Notice that we had to handle the case where we called the method the very first time. In that case, the head and the tail would both be **null**. So, when adding in that case, the item being added becomes the sole item in the list ... making it both the head and the tail at the same time. From then on, all additions occur at the tail end of the list.

Once we have this method available, the code to construct the list becomes simplified:

```
LinkedList myList = new LinkedList();
myList.add(new Item(23));
myList.add(new Item(65));
myList.add(new Item(87));
myList.add(new Item(45));
myList.add(new Item(56));
myList.add(new Item(34));
myList.add(new Item(95));
myList.add(new Item(71));
```

Is this better than an array ? It seems like a lot of overhead! Well ... it may indeed take up more space ... but the size of the list is unlimited (except for running out of computer memory). That is ... we never have to worry about going past an array bounds. Also, we never have to worry about re-allocating a new larger array and copying elements over into it.

What about the removal of an item from the list ? It too just involves moving a couple of pointers around.



Here is the code to remove an item ... assuming that **x** is in the list:

```
public void remove(Item x) {
    if (x == head) {
        if (x == tail) {
            head = tail = null;
        }
        else {
            head = x.next;
            head.previous = null;
        }
    }
    else {
        if (x == tail) {
            tail = x.previous;
            tail.next = null;
        }
        else {
            x.previous.next = x.next;
            x.next.previous = x.previous;
        }
    }
}
```

The code looks a little long because we need to handle the special cases in which the removed item is the head of the list or the tail of the list. However, you will notice that the code for removal simply involves the changing of two pointers. There is no need to copy items back in the array, nor is there any concern about garbage data lying around. The code is quite simple and elegant.

How would we write a method to add up all of the byte data in the list? He would need to start with the head of the list and keep traversing successive **.next** pointers until we reached the tail.

Here is code that will do this:

```
public int totalData() {
    if (head == null)
        return 0;

    int total = 0;
    Item currentItem = head;
    while (currentItem != null) {
        total += currentItem.data;
        currentItem = currentItem.next;
    }
    return total;
}
```

Do you understand why a **for** loop was not used ?

Can you write an **insert(x, i)** method that will insert item x after position i in the list ? Try it. You will need to start at the head of the list and count past i items before you start changing pointers. Can you do a **remove(i)** method that will remove the i'th item from the list ?

It is important to understand how to manipulate pointers like this because some languages (e.g., C and C++) require a lot of memory allocation and pointer manipulation. The more practice you get ... the better!!

You should realize that although our list contained simple data in the form of a single byte, you can simply change the type of the data to any data type. In this way, the list can store any kind of data that you want. Here is a general definition for a list Item that can store any object:

```
public class Item {
    Object data;
    Item previous;
    Item next;

    public Item(Object obj) {
        data = obj;
        previous = null;
        next = null;
    }
}
```

Notice what the memory allocation would look like for a simple 3 item list when simple byte data is used (left side diagram) and when **Person** object data is used (right side diagram):

