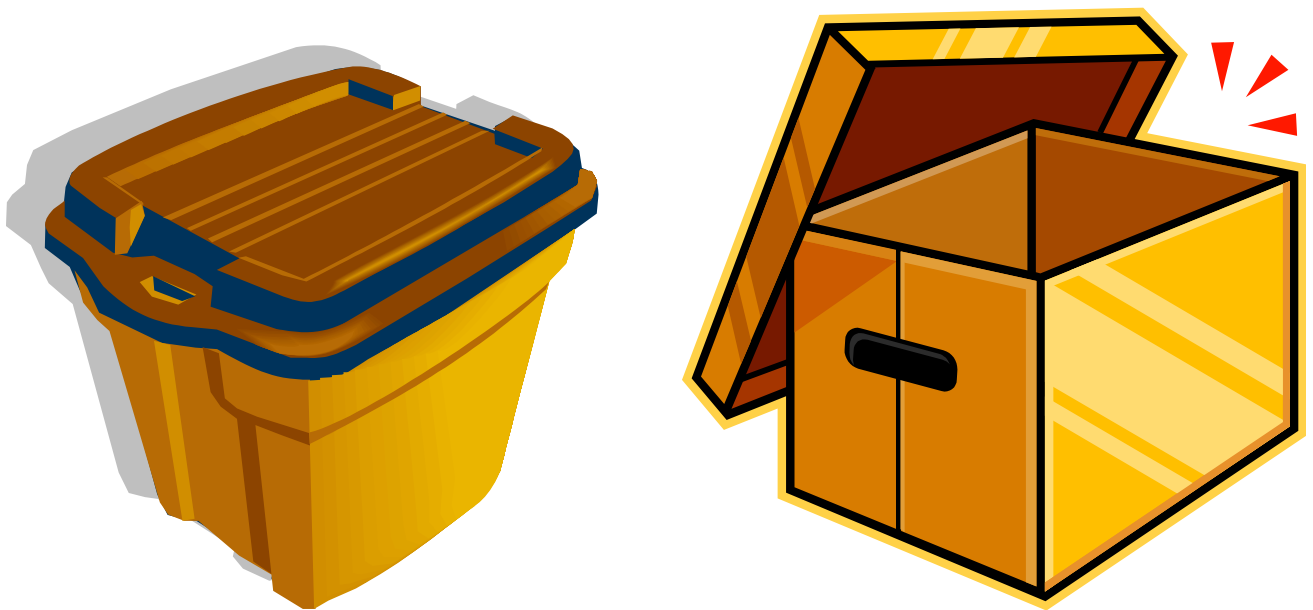

Chapter 2

Creation and Storage of JAVA Objects

What is in This Chapter ?

When beginning object-oriented programming, students often have difficulty understanding how objects interact. As a result, students sometimes struggle to write code in an object-oriented manner. In this chapter we discuss how objects are created, stored and used in JAVA. In order to properly understand object-oriented programming, it is important for you to understand where data is being stored and how to access the data that is within another object. Once you understand this simple concept, your life as an object-oriented programmer will be easier. We will also discuss memory allocation so that you fully understand what an object actually is. This will help you in 2nd year when you have to allocate memory on your own.



2.1 Using Existing JAVA Objects

Until now, we have discussed creating programs by creating a class and inserting all of our code into a **main()** procedure/method. This means that our programs are considered procedural. **Object-Oriented Programming (OOP)** is *similar* to that of procedural programming in that it also involves executing a set of instructions in some specified order. However, it differs from procedural programming in the way that your code is organized.



Programming using object-oriented *style*, involves organizing your code in "chunks" that logically correspond to real-world objects. For example, you may group all of your code related to a **person** into one file (called a **class**) while code related to a **car** or a **bank account** would be grouped together in separate files (i.e., classes).

JAVA actually has a **lot** of pre-defined objects that are all organized into various **packages**. A package is essentially equivalent to a folder that contains your **.java** files. There are many standard packages in JAVA, each with many classes.

Here are just some of the standard packages that you will likely use in this course:

<code>java.lang</code>	Basic classes and interfaces required by many JAVA programs. It is automatically imported into all programs.
<code>java.util</code>	Utility classes and interfaces such date/time manipulations, random numbers, string manipulation, collections ...
<code>java.io</code>	Classes that enable programs to input and output data.
<code>java.text</code>	Classes and interfaces for manipulating numbers, dates, characters and strings. Provides internationalization capabilities as well.

When you want to make use of some of these classes, you will use the **import** keyword to tell JAVA that you want to use a class so that it knows where to find it:

```
import <packageName>.*;
```

We did this already when we used the **Scanner** class, which is in the **java.util** package. Basically, the **import** statement is used to tell the compiler which package (i.e., directory) the class files are sitting in. You can always replace the ***** by a class name (where the class name is in the package) so that the readers of your code are more clear on which classes you are actually using. **Keep in mind though that the **import** statement *does not load* any classes, it merely instructs the compiler *where to find them* when you run your code. The code is only imported/loaded by the JVM from those libraries as it is needed.**

Here is a simple example that makes use of the pre-defined **Object**, **String**, **Date**, **Point** and **Rectangle** object classes in JAVA, making sure to import the correct package:

```

import java.lang.Object;
import java.lang.String;
import java.util.Date;
import java.awt.Point;
import java.awt.Rectangle;

public class ObjectTestProgram {
    public static void main(String[] args) {
        System.out.println(new Object());           // general object
        System.out.println(new String());           // blank string
        System.out.println(new Date());             // date object
        System.out.println(new Point(50, 75));       // point object
        System.out.println(new Rectangle(5,10,20,30)); // rectangle
    }
}

```

If we do not specify where to find the objects via the **import** statement, JAVA will become confused when compiling our code and will generate compile errors such as this:

Error: C:\...\ObjectTestProgram.java:12: cannot find symbol class Date

In fact, all classes in the **java.lang** package are automatically imported so we do not need the first two **import** statements. Also, when we have multiple classes being imported from the same package (e.g., **Point**, **Rectangle**), we can use a single **import** statement with the ***** wildcard character to tell JAVA to import any needed classes from that package. So here is the simplest form of the code:

```

import java.util.*;
import java.awt.*;

public class ObjectTestProgram {
    public static void main(String[] args) {
        System.out.println(new Object());           // general object
        System.out.println(new String());           // blank string
        System.out.println(new Date());             // date object
        System.out.println(new Point(50, 75));       // point object
        System.out.println(new Rectangle(5,10,20,30)); // rectangle
    }
}

```

In this example, we are simply creating the objects and then displaying them. Notice how these objects are displayed in the output:

```

java.lang.Object@d93b30

Mon Jan 09 20:30:35 EST 2017
java.awt.Point[x=50,y=75]
java.awt.Rectangle[x=5,y=10,width=20,height=30]

```

Each object displays itself differently. Notice that the **Date** object that was created actually corresponds to today's date and time (i.e., on January 9, 2017 when I ran the code). Also, notice that the **String** object was actually an empty string (i.e., no characters were displayed).

2.2 Creating Your Own Objects in JAVA

In the previous course, you may have already gained experience in **defining your own data structures** (a.k.a. **data types, objects**) that you used within your program in order to group various data elements together. For example, you may have created a data structure that represents someone's address in a similar manner as shown here.

```
class Address {
    String    name;
    int       streetNumber;
    String    streetName;
    String    city;
    String    province;
    String    postalCode;
}
```

In JAVA, we create this object by defining a **class**. Each class that we define represents a new **type** (or **category**) of object. So, the above class represents an **Address object** that we have defined. Here is a simple definition of an object as we know it so far:

*A **object** represents multiple pieces of information that are grouped together.*

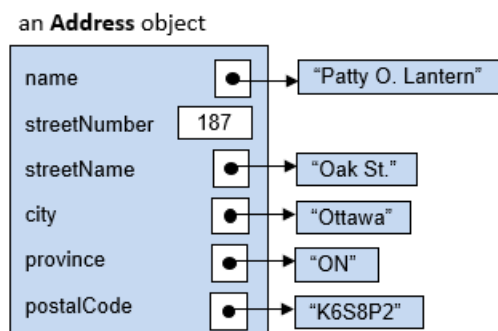
A primitive data type (e.g., **int**, **float**, **char**) represents a **single** simple piece of information. An **object**, however, is a **bundle** of data, which can be made up of multiple primitives or possibly other objects as well. You can think of an object as a bunch of small pieces of information with elastic bands around them to hold them together as a single object (as shown here). Once we define this class/object, then we were allowed to create **Address** objects and use them within our programs. For example, here is how we can create a new **Address** object and fill in its values:



```
Address  addr;

addr = new Address();
addr.name = "Patty O. Lantern";
addr.streetNumber = 187;
addr.streetName = "Oak St.";
addr.city = "Ottawa";
addr.province = "ON";
addr.postalCode = "K6S8P2";
```

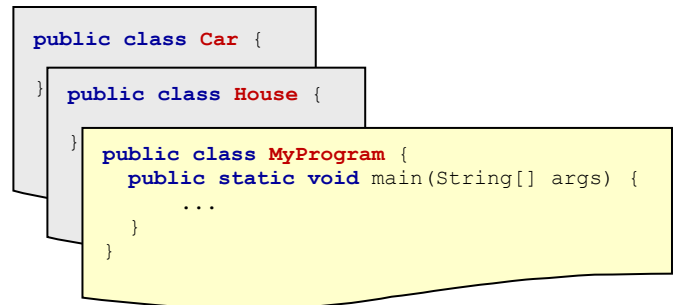
```
System.out.print(addr.name + " lives at ");
System.out.println(addr.streetNumber + " " + addr.streetName);
```



The code above prints out: "Patty O.Lantern lives at 187 Oak St.".

In JAVA, we generally define all of our own objects in separate **.java** files which will reside in the same folder as the main program class:

Even though the **Car** and **House** objects are defined in their own individual **.java** files, they cannot be **run** as programs. You can only run classes that have the **public static void main(...)** method defined. So, a JAVA program will typically consist of multiple **.java** files ... many of them being object definitions, and one of them being the actual program itself.



```
public class Car {
}
public class House {
}
public class MyProgram {
    public static void main(String[] args) {
        ...
    }
}
```

For example, we can define very simple **Car** and **Person** objects along with a test program as follows (remember that each class is defined in its own file):

```
public class Car {
    String make;
    String model;
    int    year;
}
```

```
public class Person {
    String    name;
    String    phoneNumber;
}
```

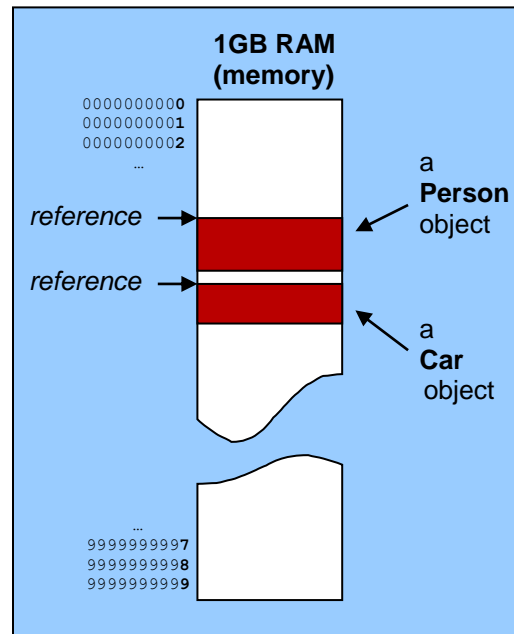
```
public class MyObjectTestProgram {
    public static void main(String[] args) {
        System.out.println(new Car());
        System.out.println(new Person());
    }
}
```

Notice now the output from the program:

```
Car@19821f
Person@42e816
```

This is what objects look like by default. They show the name of the class, then an **@** symbol, and finally a strange combination of numbers and letters.

This number/letter combination represents the location (or **address**) of the object in the computer's memory. We call this the **reference**, because this memory address "*refers to*" the object. The actual value of the address is unimportant to us, however, it is important for you to understand that each time we make an object, it "uses up" a portion of the computer's memory.



Later we will see how to change the appearance of our objects so that they show more meaningful information when displayed.

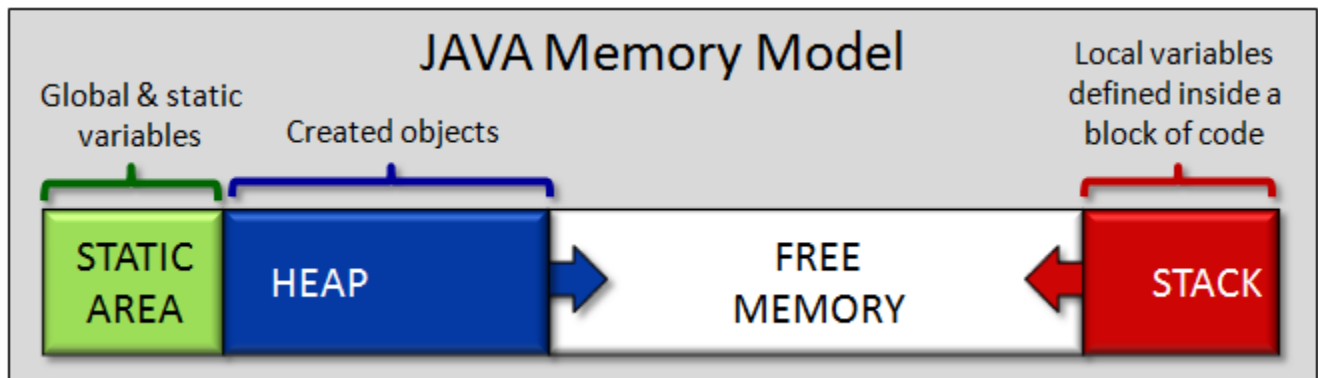
The next section of notes will clarify in more detail exactly how these objects are stored in memory.

2.3 Memory Allocation and Object Storage

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 (JVM)** is allotted a certain amount of memory space on your computer when your program begins to run. The amount of memory allotted is adjustable via command-line arguments.

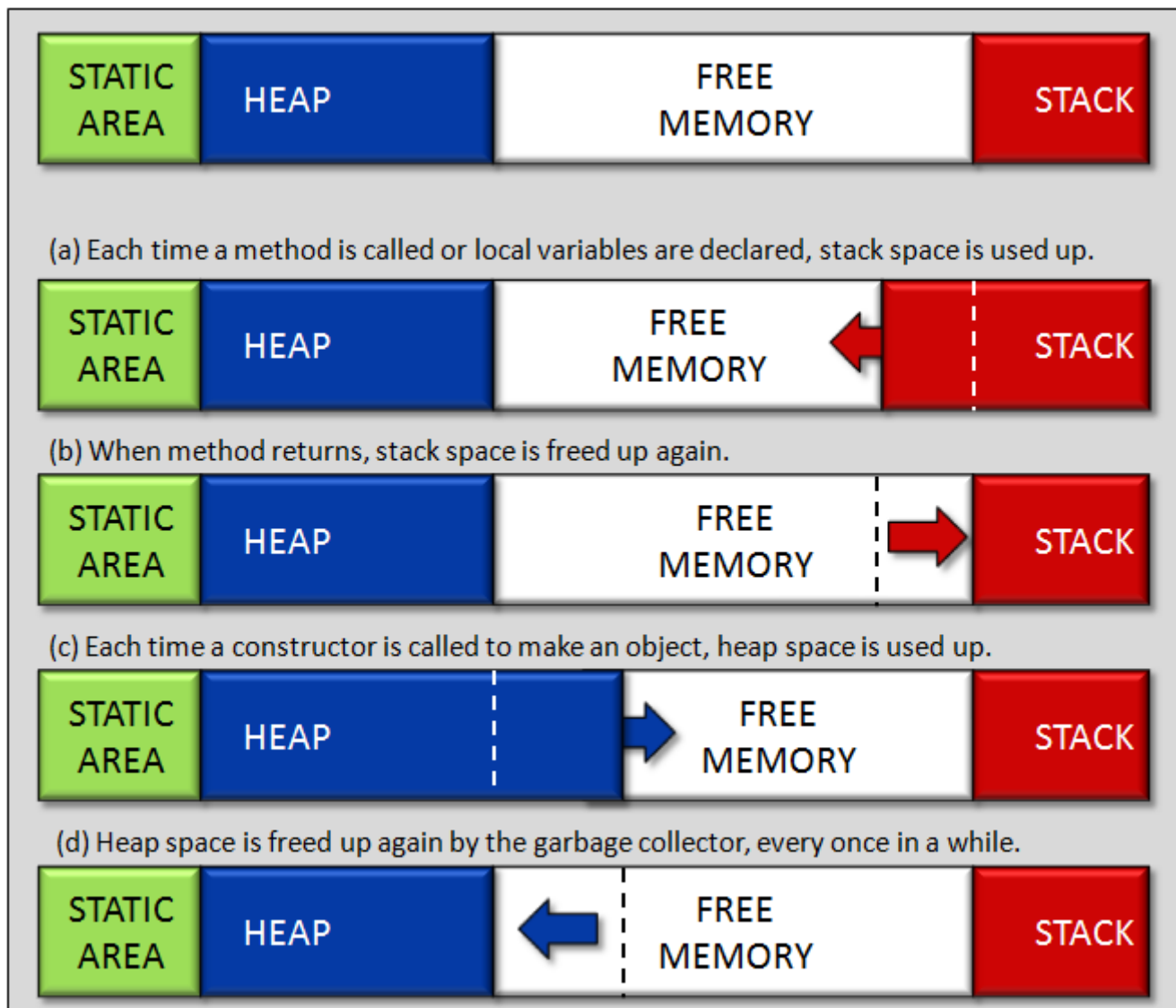
Upon start-up, some of this allotted memory is used up by the JVM. The remaining memory that is available for your program is denoted as "free memory". As your program runs, it will **allocate** (i.e., **use up**) some of this free memory at various times throughout the runtime of the program. Your program 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 & 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 (gc)** 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.

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

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

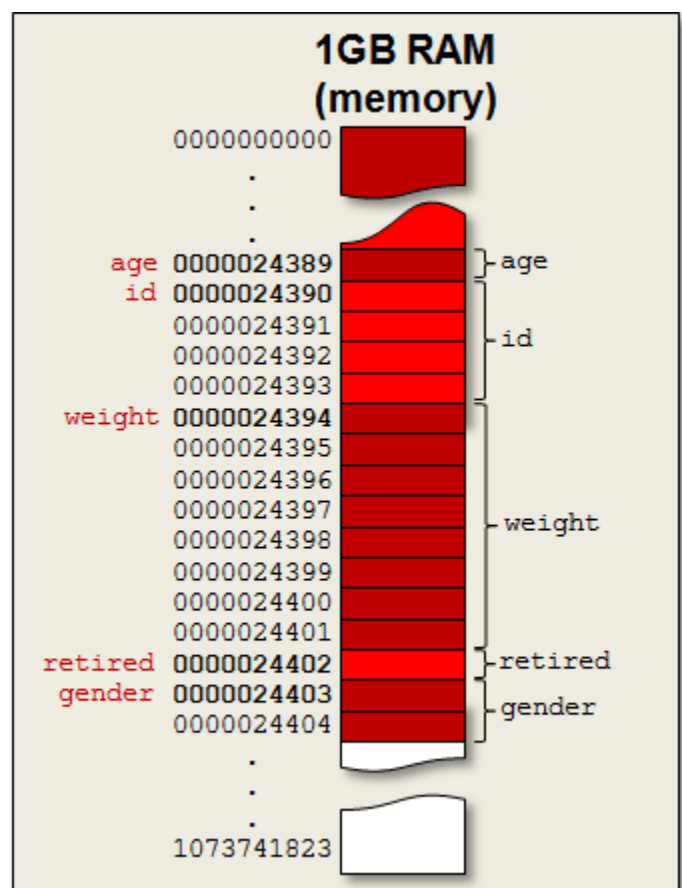
Each time we declare a variable within a method, it reserves enough space in the **STACK** memory to store the data.



For example, consider the following variables declared within a method (i.e., these are NOT object attributes) and notice the amount of memory that it consumes in the stack memory:

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

JAVA automatically reserves this space for us when we declare these variables. Each variable begins at a unique address in the computer's memory (i.e., the number shown on the left side). 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 type attributes as follows:

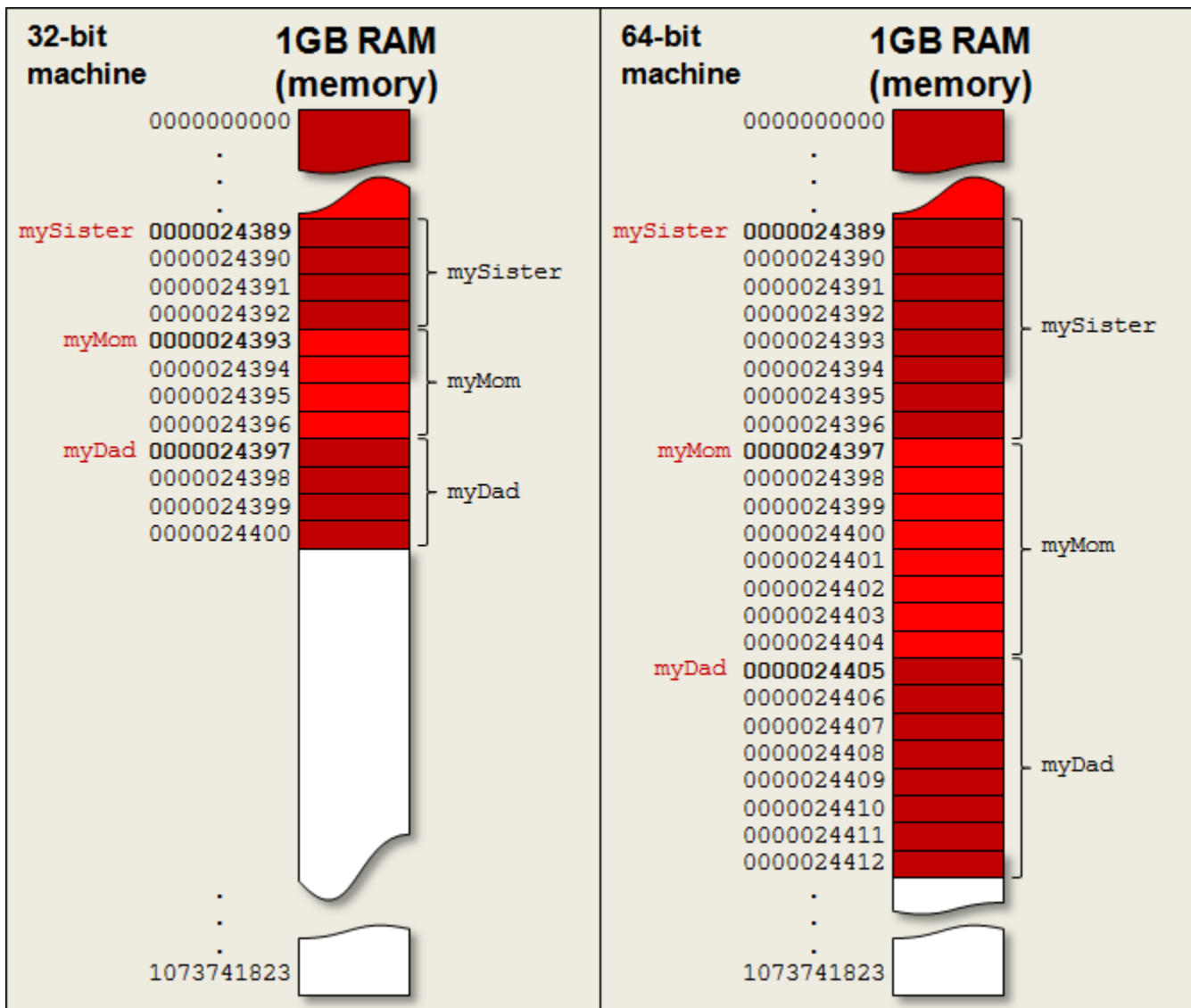
```
public class Person {
    byte    age;
    int     id;
    double  weight;
    boolean retired;
    char    gender;
}
```

a **Person** object

age	19
id	1089
weight	153
retired	false
gender	'M'

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., just the variable that holds the object ... not the object's contents). 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.

By declaring the variable: `Person mySister;` we 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:

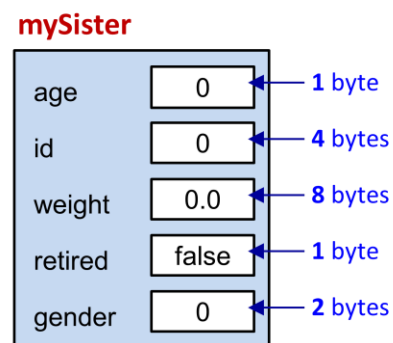
```
mySister = new Person();
```

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.

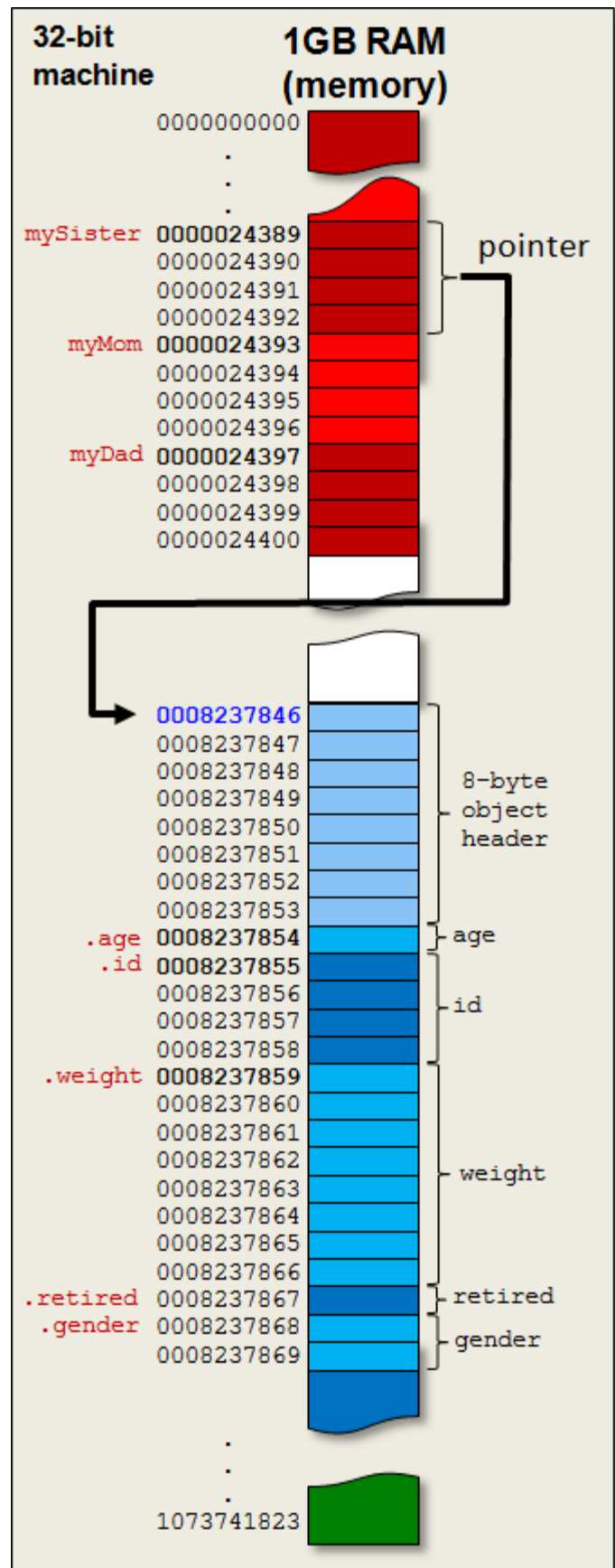
```
public class Person {
    byte    age;        // 1 byte
    int     id;         // 4 bytes
    double  weight;     // 8 bytes
    boolean retired;    // 1 byte
    char    gender;     // 2 bytes
}
```



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 bytes** more 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.

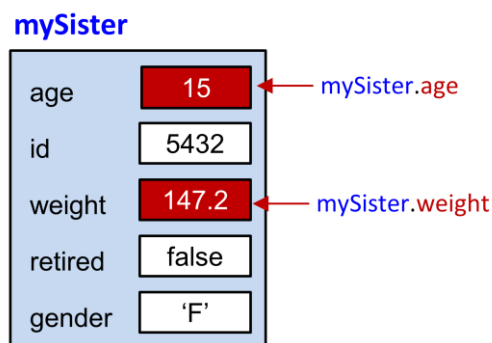
So, each **Person** object that we create will require $(16 + 8 = 24)$ bytes of storage as shown in the diagram here. Notice that the **mySister** variable now points to the location where the **Person** object is being stored (i.e., address 0008237846 in our example). So, the integer value of **0008237846** 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.weight > 150) {
    ...
}
```

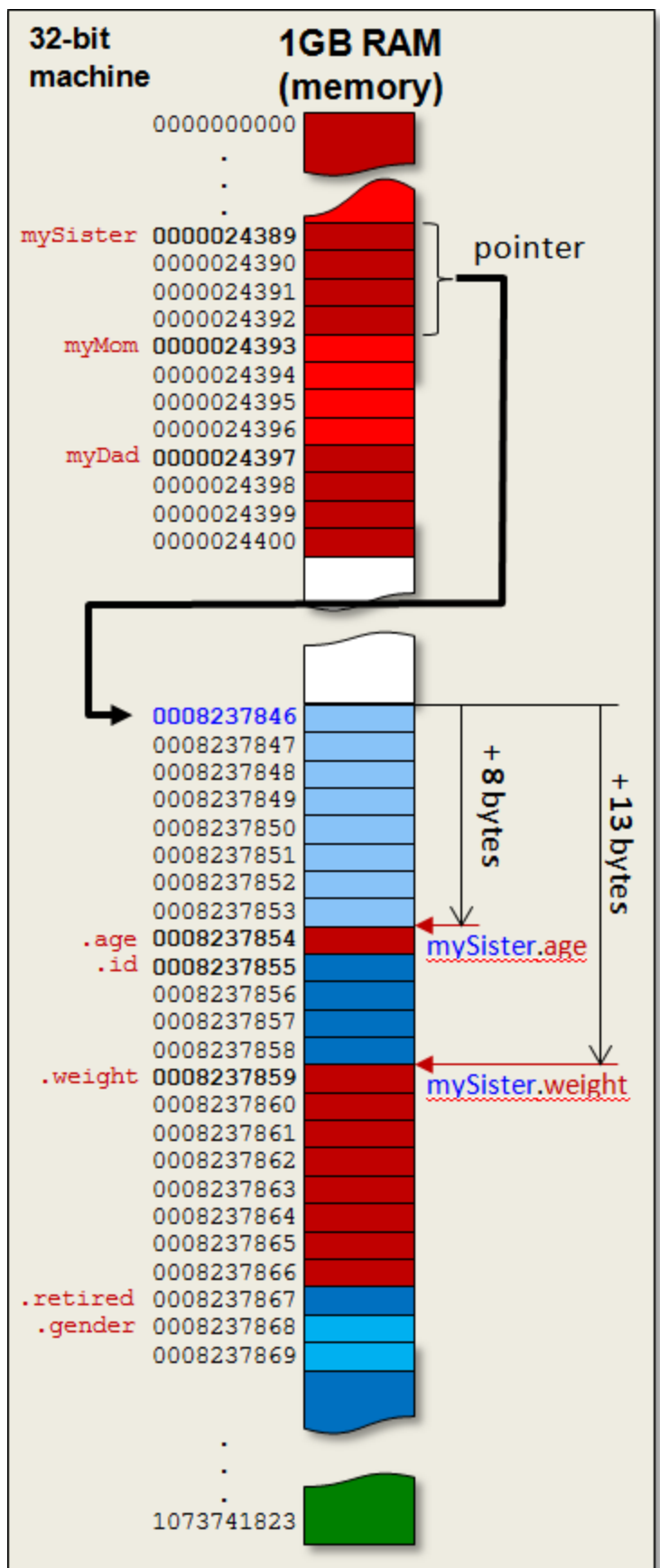
The code above requires us to go inside the object to modify its internal **age** variable/attribute and also to access the internal **weight** variable/attribute:



In order to do this, JAVA needs to determine the memory location of the **age** and **weight** attributes relative to the location of the **mySister** Person object.

JAVA begins with the address stored in the **mySister** variable (i.e., 008237846) and then adds to that value the fixed offset that the **.age** portion of the object is with respect to the start of the object (i.e., adds **8 bytes** more to bypass the header). The result is $0008237846 + 8 = 0008237854$. Once it has this location computed, it can then change the byte value there to 15 as the code instructed.

Similarly, when accessing the **.weight** portion of the object, the offset from the start of the object is $8 + 1 + 4 = 13$ bytes. Hence the weight value is found at address location $0008237846 + 13 = 0008237859$. Accessing the **double** at this address requires the 8 bytes from address 0008237859 to 0008237866 to be interpreted as a **double** value.



Now what happens when an object is contained within another object ? Consider two simple objects defined as follows:

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

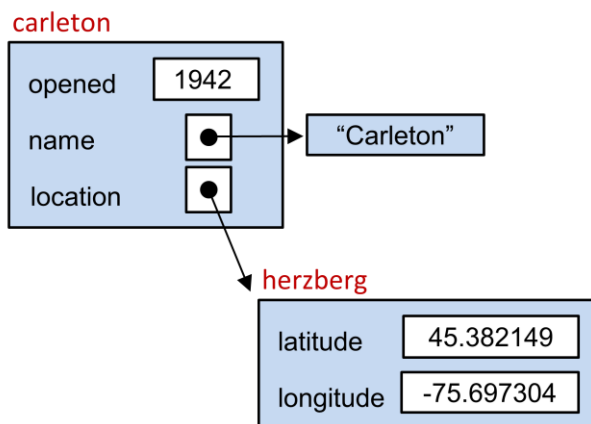
public class University {
    short     opened;
    String    name;
    GPSLocation location;
}
```

Now consider the following code:

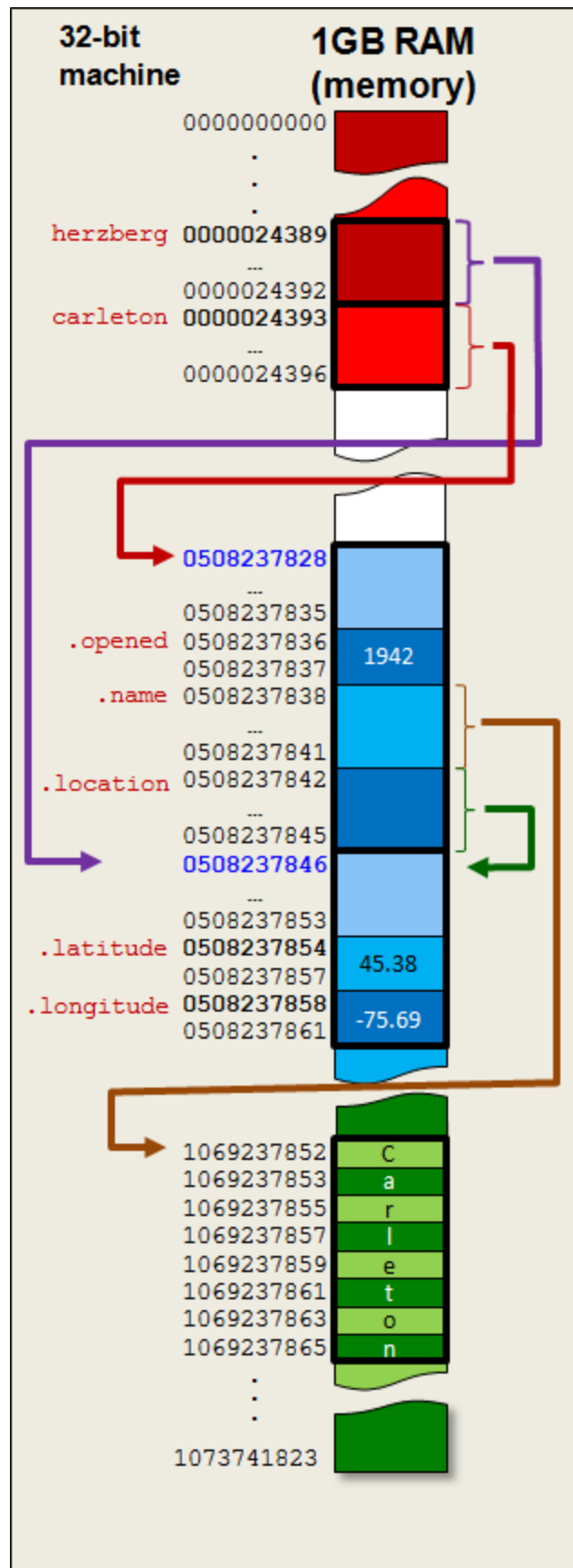
```
GPSLocation    herzberg;
University     carleton;

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 involved:

- the String literal "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 **0508237842** are the same ... which is **0508237846**.

String literals (i.e., String created using double quotes in your code) not stored in the Heap memory but are actually stored in the STATIC AREA as constants. Any other created Strings are stored in the Heap memory.

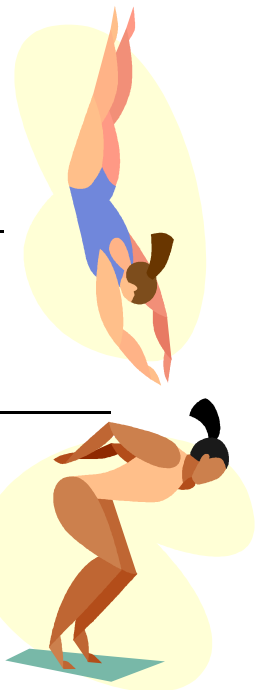
Example:

Consider writing a program that simulates a diving competition. The program will keep track of various athletes who perform dives. Assume that the following objects have been defined, each in their own files:

```
public class Dive {
    String    name;
    int       difficulty;
}
```

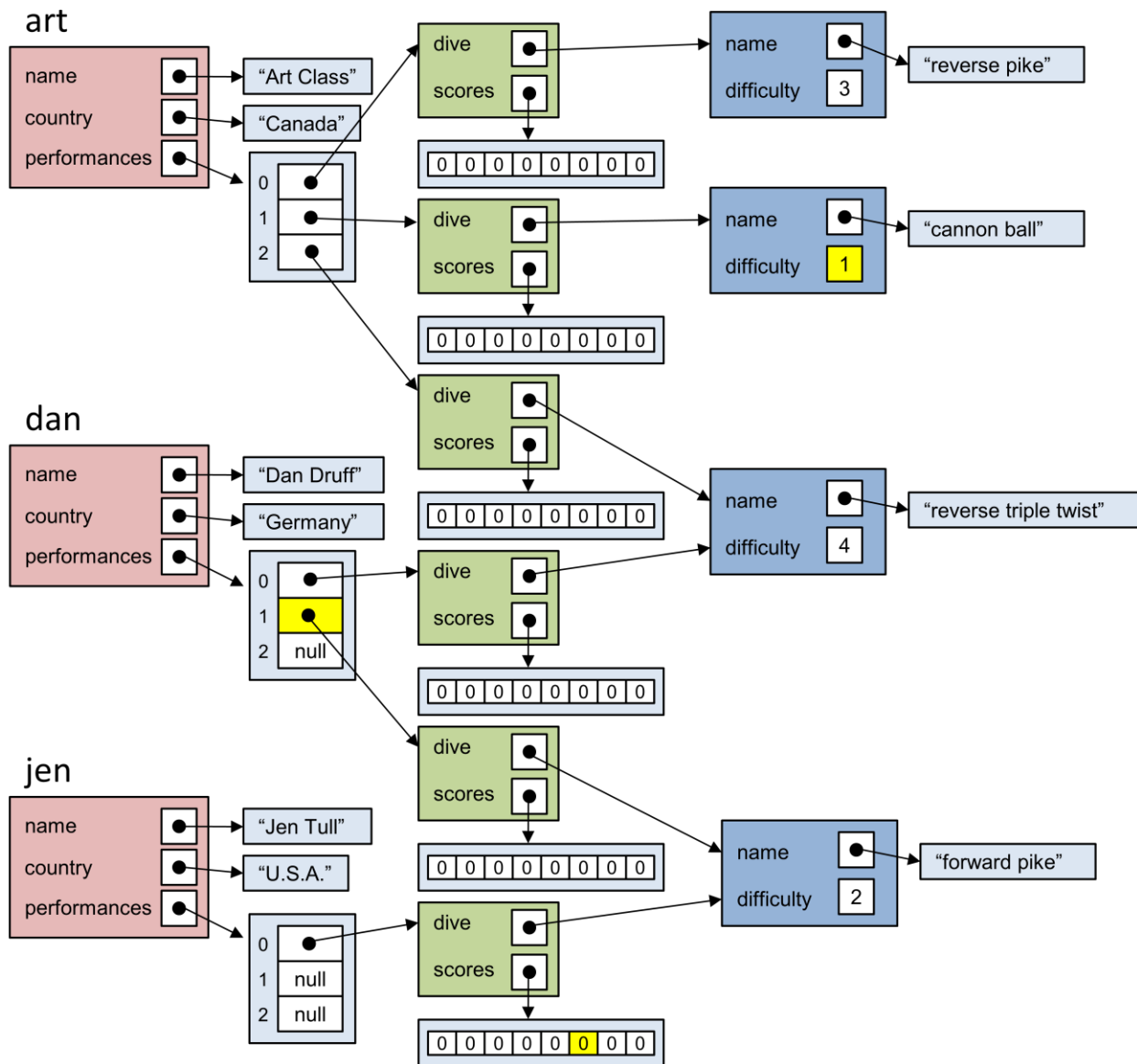
```
public class Performance {
    Dive      dive;
    float[]   scores;
}
```

```
public class Athlete {
    String     name;
    String     country;
    Performance[] performances;
}
```



Notice that each **Performance** object contains a **Dive** object. That means, each performance corresponds to a single dive (i.e., an athlete performs one dive at a time). Also, you will notice that the **Athlete** keeps an array of **Performance** objects. That is, as the athlete performs dives over time, new performances will be added to this array ... each performance representing a particular dive.

In order to make sure that we understand how objects are stored inside of one another, let us see if we can write code that constructs a particular arrangement of these objects. Here is a diagram showing the arrangement of objects that we would like to construct. Try to write the code that will produce this picture.



Looking at the picture, there are 3 **Athlete** objects, 6 **Performance** objects and 4 **Dive** objects. The remaining objects are Strings and arrays. Since the **Dive** objects don't contain the other objects that we created (i.e., neither **Athlete** nor **Performance**) we should start by making those first. We can store them into variables **d1**, **d2**, **d3** and **d4** for later use.

```
Dive d1, d2, d3, d4;
```

```
d1 = new Dive();
d1.name = "reverse pike";
d1.difficulty = 3;
```

```
d2 = new Dive();
d2.name = "cannon ball";
d2.difficulty = 1;
```

```
d3 = new Dive();
d3.name = "reverse triple twist";
d3.difficulty = 4;
```

```
d4 = new Dive();
d4.name = "forward pike";
d4.difficulty = 2;
```


Now, we should create the **Performance** objects, making sure to point them to the correct **Dive** objects.

```

Performance    p1, p2, p3, p4, p5, p6;

p1 = new Performance();           p4 = new Performance();
p1.dive = d1;                       p4.dive = d3;
p1.scores = new float[8];           p4.scores = new float[8];

p2 = new Performance();           p5 = new Performance();
p2.dive = d2;                       p5.dive = d4;
p2.scores = new float[8];           p5.scores = new float[8];

p3 = new Performance();           p6 = new Performance();
p3.dive = d3;                       p6.dive = d4;
p3.scores = new float[8];           p6.scores = new float[8];

```

Finally, we create the **Athlete** objects:

```

Athlete    art, dan, jen;

art = new Athlete();
art.name = "Art Class";
art.country = "Canada";
art.performances = new Performance[3];

dan = new Athlete();
dan.name = "Dan Druff";
dan.country = "Germany";
dan.performances = new Performance[3];

jen = new Athlete();
jen.name = "Jen Tull";
jen.country = "U.S.A.";
jen.performances = new Performance[3];

```

Of course, we need to simulate these athletes doing their performances. So we need to add the performances for each athlete:

```

art.performances[0] = p1;
art.performances[1] = p2;
art.performances[2] = p3;
dan.performances[0] = p4;
dan.performances[1] = p5;
jen.performances[0] = p6;

```

It seems like a lot of code, but we will see later how to shorten it. For now, it is just important that you understand how various objects are stored inside others.

As a further test of your understanding, see if you can write code to access and print out the following:

1. the difficulty level of Art's 2nd performance
2. the object representing Dan's 2nd performance
3. the 6th judge's score for Jen's first performance

Here are the solutions:

1. `System.out.println(art.performances[1].dive.difficulty);`
2. `System.out.println(dan.performances[1]);`
3. `System.out.println(jen.performances[0].scores[5]);`

Can you write code to determine the average judges' score for **Art** (consider all performances) ? Assume that there are interesting scores stored in the data, because at the moment they are all 0.

```
float    sum = 0;

for(int p=0; p<3; p++) {
    for(int s=0; s<8; s++) {
        sum = sum + art.performances[p].scores[s];
    }
}
System.out.println(sum/24);
```

Do you understand this now ? If not, you may want to come for further help during office hours. It is VERY important that you understand how to do this stuff.