Chapter 7

Data Structures and Objects

What is in This Chapter ?

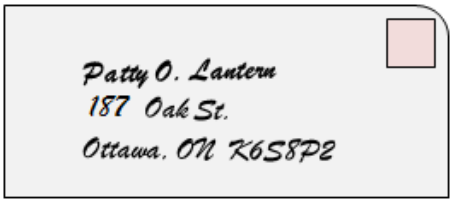
Almost all programs require the use of some data as input to the problem being solved. It is often advantageous to group (or structure) related data together. This chapter discusses the idea of creating ***data structures*** in Processing, which are also known as ***objects***. Objects are used as a way of keeping your data organized in a logical manner. In a follow-up course, we will further develop the various concepts of Object-Oriented Programming.

7.1 Simple Data Structures and Objects

Recall that we used functions and procedures to provide ***control abstraction*** in order to hide low-level *conceptual details* within our algorithms so that they are simpler to read and understand. There is another type of abstraction known as ***data abstraction***. In this type of abstraction we are interested in hiding ***information*** *(or data)* that will unnecessarily clutter up an algorithm. The idea behind data abstraction is to group simple data values together which have a well understood relationship.

For example, if we are mailing out an envelope (within Canada), then an ***address*** is assumed to have this information:

1. name
2. street number
3. street name
4. city
5. province
6. postal Code



Whenever most people hear the word **"address"**, they understand that such information is actually made up of some smaller, specific kinds of information. The address itself is not complete unless it is has all of that information. In a sense, the individual pieces of information *make up* (or ***define the structure of***) the address.

We can create such "more abstract" types of data (e.g., like an *address*) simply by combining or structuring the *more primitive* (i.e., simpler) pieces of data together in meaningful ways:

> *A **data structure** is a particular way of combining, storing and organizing data so that it may be used more efficiently and in a more abstract manner.*

We also use the word ***data type*** which is somewhat analogous to the term ***data structure***. In object-oriented programming languages, such as Java, a data type is also known as a ***class*** or category, and *defining a data type* is also called ***defining an object***.

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

Recall that a primitive data type represents a **single** simple piece of information such as a number or character. 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 an elastic around it →

Perhaps the simplest data structure is called a ***string*** which is a group of one or more characters with a specific ordering. Characters by themselves are not very interesting. However, when we group them together, we form a huge variety of words and seemingly unlimited variety of sentences. Each word in the English language, for example, represents a string data structure, as does each sentence, paragraph, page of text, etc…

In many programming languages (including JAVA), strings are represented by placing double quotes around a set of characters like this:

```
String  name;

name = "Patty O. Lantern";
```

A string is not a *primitive* data type because it is made up of characters … which themselves are primitive data types. In fact, we can abstract out the notion of a string even further by grouping strings together in a meaningful way to create an even more abstract data structure.

For example, consider an address as described above. A full address may be represented using multiple numbers and strings as follows:

```
String  name, streetNumber, streetName, city, province, postalCode;

name = "Patty O. Lantern";
streetNumber = "187B";
streetName = "Oak St.";
city = "Ottawa";
province = "ON";
postalCode = "K6S8P2";
```

Together, all of the variables above represent a full address. It would be advantageous if we could define a single variable, perhaps called **address**, that can store all of the above information:

```
Address    myAddress;

myAddress = … ??? …;
```

Of course, we could combine everything into one big string …

```
String     myAddress;

myAddress = "Patty O. Lantern, 187B Oak St., Ottawa, ON, K6S8P2";
```

… but then it would be more difficult/cumbersome to extract the needed pieces of information (e.g., street number or last name).

Many programming languages allow you to "group" variables together into a structure of some type. The process of defining which variables and types of data should be grouped together is called ***defining a data structure*** (or ***defining a data type***). In object-oriented languages (such as JAVA) this is also called ***defining an object*** (or sometimes defining a ***class***).

When defining such a structure (or class), we need to use the **class** keyword and we also need to specify the name of the new type of data (e.g., **Address**) as well as the names and types of data that is contained within it. The diagram on the right is a visual representation of the class definition:

We will use the following notation to define this Address structure/class/object in JAVA:

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

Notice that we capitalized the data type **Address**. It is proper coding style to ALWAYS capitalize your class names. Note that the class name should also always be singular (i.e., **Address** ... not **Addresses**).

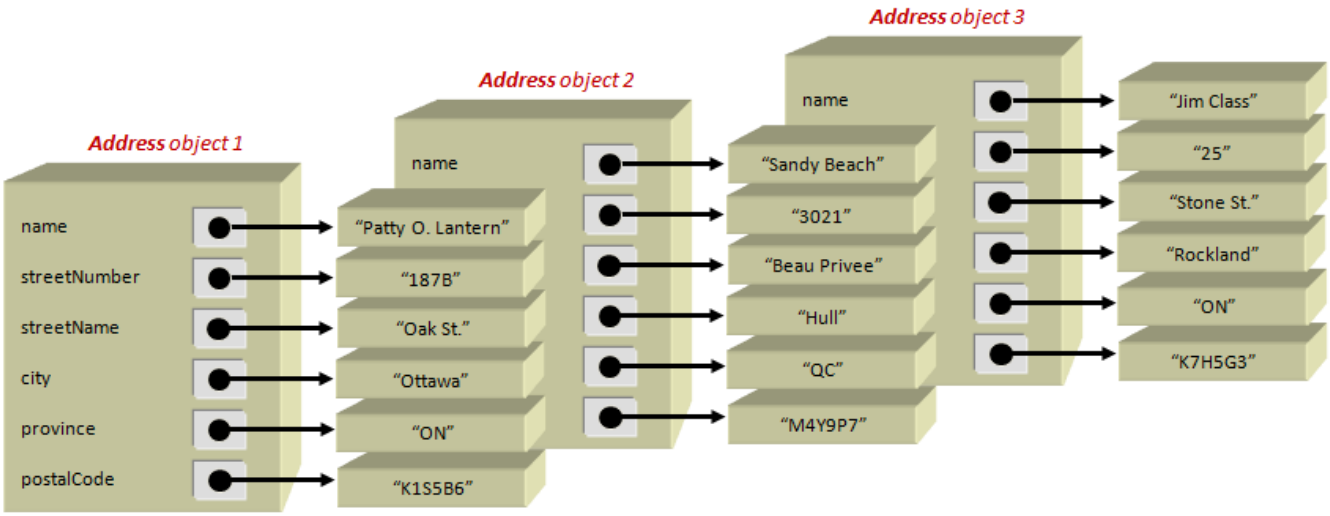
The above notation shows that an address is made up of 6 pieces of data with the given labels. It is as if the **Address** data type is a "blank form" onto which we can fill in appropriate values. It defines a kind of *template* for creating data of this type.

The code should be written all on its own as a new/separate JAVA file. That is how we will define a new data type.

However, *defining* a data type does not actually create any variables, it only creates a *definition*. When we actually want to *use* a data type, we need a way of specifying that we want to create a new ***instance*** of this data type.

> *An **instance** of a data structure (or object) is a particular group of values for each of the individual variables that make up the data structure (or object).*

That means an instance is a particular object belonging to the category of objects defined by the data type. For example, each of the following is an instance of the **Address** data type, because they represent particular addresses:



Notice in the diagram, that the String values are themselves shown as separate objects. That is, the value of the name of the first Address object is a String object with the value "Patty O. Lantern". We use an arrow from one object to another to indicate that the object is ***inside of*** another object. So, in the image above ... we can count the objects simply by counting the boxes ... 21 in total for all three addresses combined.

In order to create one of these **Address** objects, we need to tell JAVA to create a new object for us. This is done by using the **new** keyword in JAVA as follows:

```
new Address();
```

This will create the object by allocating enough memory to hold all of the address' information..

You may want to think of the **Address** class as a ***factory*** that makes **Address** objects (i.e., makes ***instances*** of the **Address** class). In general, every time we use **new**, it is as if we go to the factory for that class and buy a instance of the object. So… the *class* is the "factory", and the *instance* is the particular "object" that we can start using now in our programs.

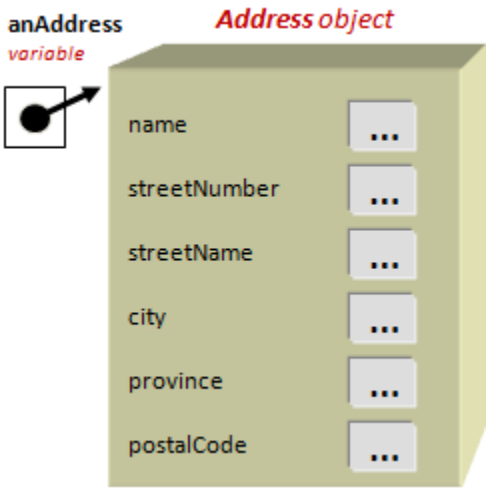
The **Address** Class **new Address()** Instance of type **Address**

A **Person** Class **new Person()** Instance of type **Person**

A **House** Class **new House()** Instance of type **House**



Of course, once the object is created, we need to store it into a variable so that we can access and modify it later. The variable can have any name, but its type MUST be declared with the exact same name as the class as follows:




```
Address     anAddress;

anAddress = new Address();
```

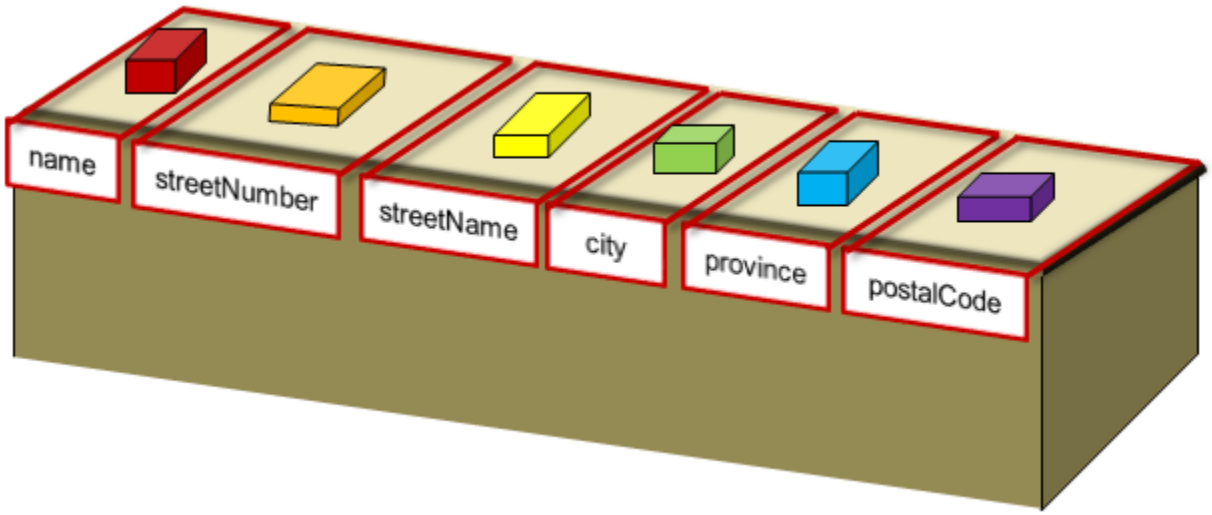
The **anAddress** variable now holds an ***instance*** of the **Address** class.

Just to help you understand how this all works, let us compare what we just did to what we would have done without the Address class definition.

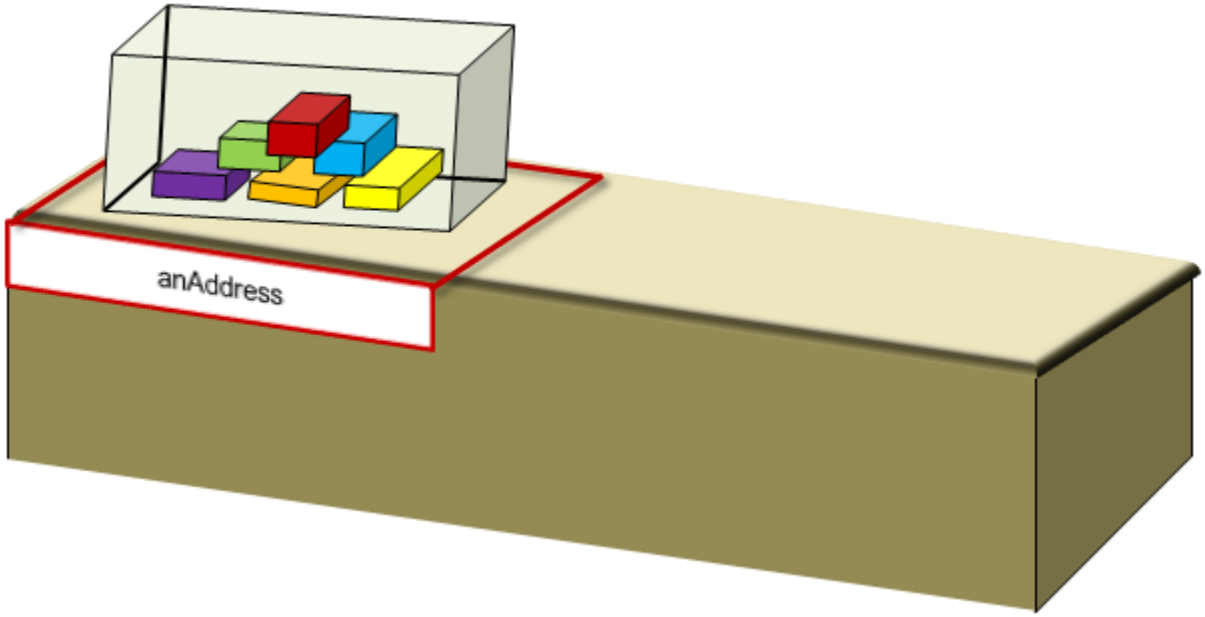
If we were to define 6 separate String variables to store the information like this:

```
String     name;
String     streetNumber;
String     streetName;
String     city;
String     province;
String     postalCode;
```

then we would be reserving space for 6 variables as follows:

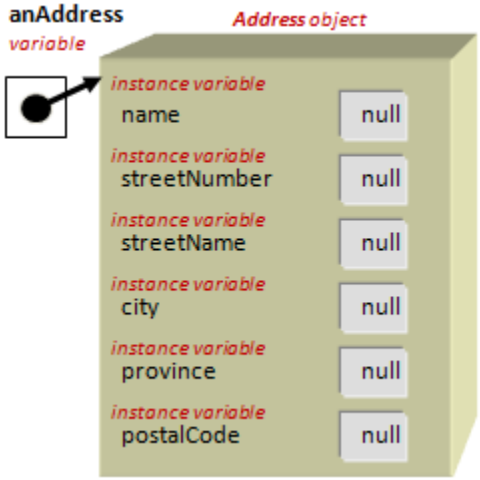


However, when creating an instance of an object, although we are actually storing the same data, we are only defining one variable to hold on to ALL 6 pieces of data. It is as if we are taking all of the data and putting it into a larger box like this:



There is a term that we use to describe the pieces/variables of a particular instance of a class. We call them ***instance variables***, since they are just regular variables … that happen to be grouped together to form a particular instance of the class.

Once we create the object, we can start to use it.



We can access the individual parts (i.e., instance variables) of an object by using the dot operator as follows:

```
Address    anAddress;

anAddress = new Address();
System.out.println(anAddress.name);
System.out.println(anAddress.streetNumber);
System.out.println(anAddress.streetName);
// ... etc ...
```

dot

So, instance variables are used just like regular variables, except that we now precede the variable name by the object's variable name and the dot operator. The dot operator indicates to **"go inside"** the data type to get a piece of information. That is, we are getting more specific as to what particular piece of data we want. Whenever we use, for example, **anAddress.**name, it behaves just like any other variable and refers to the data stored in the "name" part of the address's memory. The above code, however, will not print out anything exciting. In fact, it will print out:

```
null
null
null
```

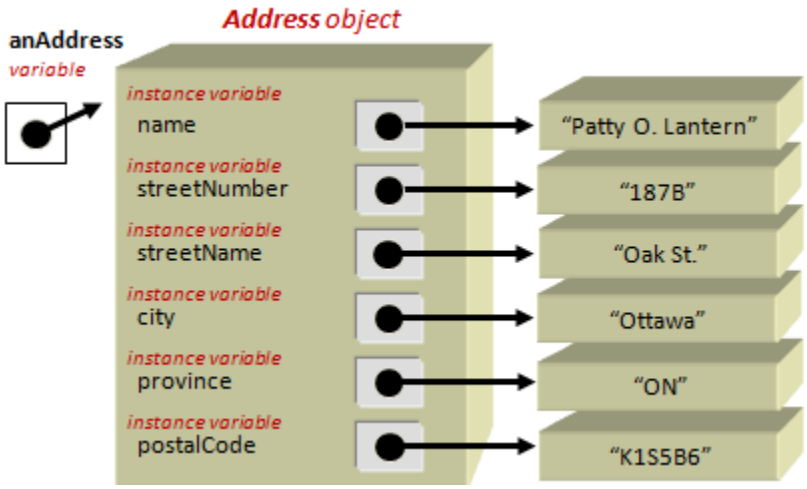
Why ? Well, by default, when we create an object using the **new** keyword, JAVA will allocate space for the object, but will not assign any values to the instance variables. Any instance variable which has a primitive type (e.g., **int**, **float**, **char**, etc...) will have a default value of 0. Any instance variable with an object type (e.g., **String**) will have the default value **null**.

> ***Null*** *is a word that represents an undefined value. If a variable has a value of* ***null****, it usually means that it has not yet been given a value.*

So, in our example above, we saw **null** printed 3 times because none of the three String variables were given a value ... and since Strings are objects (i.e., not primitives), then we see **null** displayed. Regardless of what is displayed, the code does not make sense logically because we are trying to print out the address's name before we have assigned a value to it.

We can then assign values to the individual instance variables of an object by using the dot operator again as follows:

```
Address    anAddress;

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

System.out.println(anAddress.name);
System.out.println(anAddress.streetNumber);
System.out.println(anAddress.streetName);
```

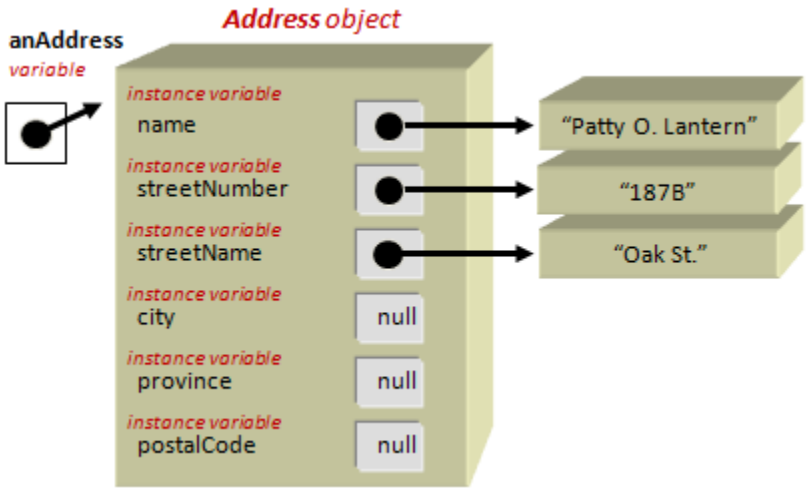


Now, we will see the output that we expected:

```
Patty O. Lantern
187B
Oak St.
```

Sometimes, however, when we create our objects ... some information may be missing. For example, when we give a local person our ***address*** on a piece of paper, it is likely that we'll only give them the street number and name.

In this situation there will be some missing information. Perhaps the missing information is assumed to be particular values. For example, we may assume that the city and province are local to where we received the piece of paper.



As mentioned, the value of the instance variables in this case are **null** (or **0** or **false** in the case of primitives).

When using the object, we always want to be aware that there could be data missing. That is, we need to be aware of cases where **null** may appear.

Why ? Well, **null** is an undefined object. If we try to use an object that is not defined, then our program will crash. For example, consider this code:

```
Address     anAddress;

System.out.println(anAddress.name);
```

The code will not compile but will instead give you this error:

```
Error: variable anAddress might not have been initialized.
```

JAVA noticed that we are trying to access the address's **name** attribute ... but that we have not yet assigned an initial value to our **anAddress** variable. It is reminding us to do so. We had forgotten to do **new** `Address();`.

However, sometimes java will not catch your error during the compiling stage. Consider the following code which is supposed to print out the person's first initial:

```
Address     anAddress;

anAddress = new Address();
System.out.println(anAddress.name.charAt(0));
```

This code compiles but then crashes with an error that looks something like this:

```
java.lang.NullPointerException
      at TestCode.main(TestCode.java:12)
      at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
      at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
      at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
      at java.lang.reflect.Method.invoke(Unknown Source)
      at edu.rice.cs.drjava.model.compiler.JavacCompiler.runCommand(Javac.java:72)
```

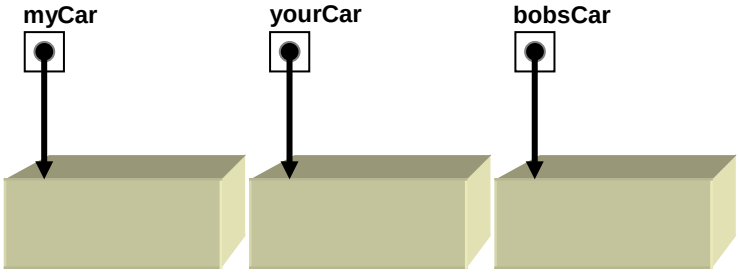
A **NullPointerException** basically means that we are trying to go inside of an object that has not been defined. But didn't we already define the **Address** object ? Yes ... however ... we did not give an initial value to the **name** attribute. Notice that we are trying to go inside the **name** attribute now to call a function on it ... the **charAt(0)** function ... which gets the first character of the String. Since the String itself is **null**, we cannot call the function on it ... so JAVA gives us a **NullPointerException** to tell us that it won't work. It actually, also tells us the method name as well as the line number that caused the problem so that we can find it and fix it. In this case, it is an easy fix ... just assign a value to the **name** attribute of the address:

```
Address    anAddress;

anAddress = new Address();
anAddress.name = "Bob";
System.out.println(anAddress.name.charAt(0));
```

At this point, it would be good to mention something about variable bindings. A variable is ***bound to*** *(i.e., attached to)* a value when we assign something to it using the **=** operator.

You need to understand that each time we make a new object, we get back a new *instance* of that object which is stored in a separate location in memory.

```
Car   myCar, yourCar, bobsCar;

myCar = new Car();
yourCar = new Car();
bobsCar = new Car();
```

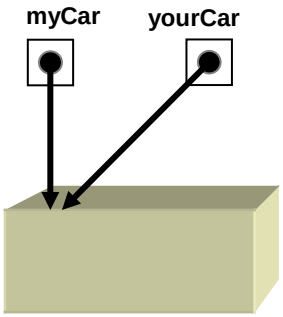


So in the above code, all three variables ***point to*** different/unique objects. In fact, it is often the case that one or more variables may point to (or refer to) the same object.

For example two people may share the same car as in this code:

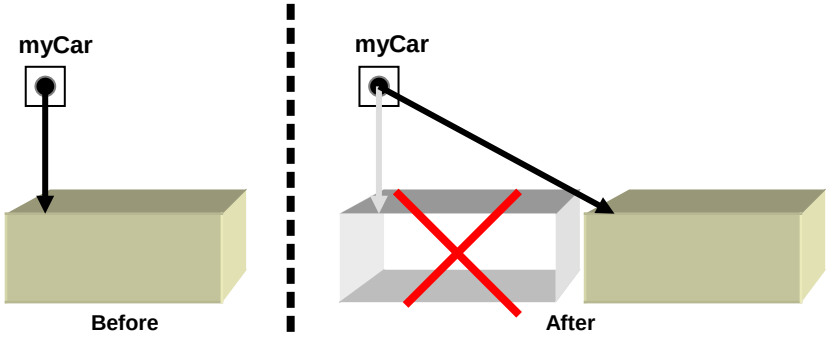
```
Car   myCar, yourCar;

myCar = new Car();
yourCar = myCar;
```

In this case, the two variables point to the exact same object.



What would happen if we re-assign a value to an object variable ?

```
Car   myCar;

myCar = new Car();
// ...
myCar = new Car();
```



In this case, the previous **Car** object that was assigned to the **myCar** variable is discarded and the variable simply points to the new **Car** object.

Supplemental Information (The Garbage Collector)

Objects that have been created and are no longer being "pointed to" by anyone are ***garbage collected***. Garbage collection usually does not happen immediately when you are finished with an object. It may happen at a later convenient time (decided upon by the Java Virtual Machine). Note that in the example above, the old car may not be garbage collected immediately, and so it may actually remain around for a while.

The ***garbage collector*** :

- is a low priority process running in the Java Virtual Machine
- is used to free up memory for unused objects
- is necessary to release resources
- runs automatically, programmer need not do anything
- can be forced to run by using **System.gc()**

Example:

Recall the **LoanApplicationProgram** from chapter 2. In that example, we asked the user to enter information as an applicant applying for a loan. We then stored the information into these four variables:

```
boolean  employed;
boolean  hasDegree;
int      age;
int      yearsWorked;
```

Since all of this information pertains to the applicant, we can actually combine all of it into a **Applicant** data type by defining an **Applicant** class as follows:

```
public class Applicant {
    boolean  employed;
    boolean  hasDegree;
    int      age;
    int      yearsWorked;
}
```

Then, in the main code, we simply create a single variable ot type **Applicant** instead of the four variables:

```
Applicant  applicant;
```

The following page shows a table with the original code on the left and the code using the **Applicant** class on the right. The code has been simplified to fit into the table and some of the code has been left out completely. Just try to understand the differences:

Using Local Variables

```
public class LoanQualificationProgram {
  public static void main(String[] arg) {
    char      charInput;

    boolean   employed, hasDegree;
    int       age, yearsWorked;

    // ... some code has been omitted ...

    System.out.print("...");
    charInput = ...;
    employed = ...;

    System.out.print("...");
    charInput = ...;
    hasDegree = ...;

    System.out.print("...");
    age = ...;

    System.out.print("...");
    yearsWorked = ...;

    if (employed) {
      if (hasDegree)
        System.out.println("...");
      else {
        if (age >= 30) {
          if (yearsWorked >= 10)
            ...;
          else
            ...;
        }
        else
          ...;
      }
    }
    else
        ...;
  }
}
```

Using the Applicant Object

```
public class LoanQualificationProgram {
  public static void main(String[] arg) {
    char       charInput;

    Applicant  applicant;
    applicant = new Applicant();

    // ... some code has been omitted ...

    System.out.print("...");
    charInput = ...;
    applicant.employed = ...;

    System.out.print("...");
    charInput = ...;
    applicant.hasDegree = ...;

    System.out.print("...");
    applicant.age = ...;

    System.out.print("...");
    applicant.yearsWorked = ...;

    if (applicant.employed) {
      if (applicant.hasDegree)
        System.out.println("...");
      else {
        if (applicant.age >= 30) {
          if (applicant.yearsWorked>=10)
            ...;
          else
            ...;
        }
        else
          ...;
      }
    }
    else
        ...;
  }
}
```

All of the applicant's information is now packaged into the single **Applicant** data structure and stored in the **applicant** variable. The code seems longer. However, the data is now set up for more abstract use.

For example, assume that we created a function to get the user's information and another to determine whether or not they qualify. Notice the simple main code:

```
public static void main(String[] arg) {
      System.out.println("Welcome ...");

      Applicant   applicant = getUserInformation();
      determineQualifications(applicant);

      System.out.println("Thank you, have a nice day ...");
}
```

Notice how the **Applicant** object is created in the **getUserInformation()** procedure, populated with information from the user, and then returned to the main algorithm:

```
public static Applicant getUserInformation() {
    Applicant  applicant = new Applicant();

    Scanner keyboard = new Scanner(System.in);

    System.out.print("...");
    char charInput = ...;
    applicant.employed = ...;
    System.out.print("...");
    charInput = ...;
    applicant.hasDegree = ...;
    System.out.print("...");
    applicant.age = ...;
    System.out.print("...");
    applicant.yearsWorked = ...;

    return applicant;
}
```

Notice how the return type of the function is **Applicant** and that we are returning an **Applicant** object. The **determineQualifications()** procedure then accepts an incoming **Applicant** object (which is labelled as **appl**) and uses it within the code for making various decisions:

```
public static void determineQualifications(Applicant appl) {
    if (appl.employed) {
        if (appl.hasDegree)
            System.out.println("Congratulations ...");
        else {
            if (appl.age >= 30) {
                if (appl.yearsWorked >= 10)
                    System.out.println("Congratulations ...");
                else
                    System.out.println("Sorry..");
            }
            else
                System.out.println("Sorry..");
        }
    }
    else
        System.out.println("Sorry..");
}
```

Within the **determineQualifications()** procedure we simply use the dot operator to get at the specific piece of applicant information that we need.

There are some advantages of using this new data structure:

1) The main algorithm is more abstract and **simpler to understand**

2) If we add additional qualification parameters (e.g., marital status, # of dependants, credit history, etc…) then the **main program remains unchanged**.

The code is thus simpler and more organized with the use of the data structure/object.

However, it is not always obvious to know what kind of information (i.e., attributes/components) should *make up* a data structure/object. That is … there is not always a "well defined" set of data that make up the object. For an **Address** data structure, it is somewhat obvious. However, what about a **Person** data structure … what should "make up" a person ?

Some possible attributes of a **Person** data structure may be **firstName**, **lastName**, **age**, **gender** and **retired**. Why would we choose these ? In reality, our choice of attributes depends on the application that we are trying to develop. For example, while the **age** and **gender** may be vital pieces of information for a program that determines players on a team sport in some league, information about whether a person is **retired** is not necessary. And for medical applications, perhaps **weight** and **height** are vital pieces of information. If it is to be an online social network application, perhaps **emailAddress** is an important piece of information that all **Persons** should have. The choice of a data structure's attributes really depends on the application.

As another example, consider defining a **Car** data structure. We should think of what characteristics we will need to store for each car (e.g., **make**, **model**, **color**, **mileage**, etc..):

Again, the choice will depend on the program/application you are making. Consider these possible applications in which a **Car** data structure may be used:

- a program for a car repair shop
- a program for a car dealership
- a program for a car rental agency
- a program for an insurance company

So, now let us examine what kind of attributes (i.e., instance variables) that we would likely need to define for a **Car** in each of these individual applications:

- **repair shop**
  make, model, year, engine size, spark plug type, air/oil filter types, air hose diameter, repair history, owner etc..
  .

- **car dealership**
  model, price, warranty, interior finish (leather/material), color, engine size, fuel efficiency rating, etc...

- **rental agency**
  sedan or coupe, make, model, license plate, price per hour, mileage, repair history, etc...

- **insurance company**
  year, make, model, owner, vehicle identification number, insurance type (e.g., fire/theft/collision/liability), color, license plate, etc...

So you can see that it is not always straight forward to identify the components of a data structure. You need to always understand **how it fits** into the application.

Example:

Recall our **FireSpreadSimulation** program which created three arrays to store the points along the fire border as follows:

```
public static int[]    borderX;       // x values along border of fire
public static int[]    borderY;       // y values along border of fire
public static float[]  borderCost;    // costs along border of fire

borderX = new int[1000];
borderY = new int[1000];
borderCost = new float[1000];
```

Instead of doing things this way, we can define a class called **FirePoint** as follows:

```
public class FirePoint {
      int   x;
      int   y;
      float cost;
}
```

Then, we can replace all three arrays with just one as follows:

```
public static FirePoint[]    border;     // points along border of fire

border = new FirePoint[1000];
```

How will this affect the code in the program ? Well, notice below how the code for starting the first three fires will change. The old code is on the left compared to the new code on the right which uses the new class.

You will notice that the array name is consistent now (i.e., only one array) and that we use the dot operator to go into each **FirePoint** to set the **x**, **y** and **cost**. It is important to create a new **FirePoint** each time, otherwise space is not reserved for the data and we would get a **NullPointerException** when trying to set the **x**, **y** & **cost** values.

```
borderX[0] = 85;
borderY[0] = 70;
borderCost[0] = 0;

borderX[1] = 120;
borderY[1] = 30;
borderCost[1] = 0;

borderX[2] = 20;
borderY[2] = 20;
borderCost[2] = 0;
```

```
border[0] = new FirePoint();
border[0].x = 85;
border[0].y = 70;
border[0].cost = 0;

border[1] = new FirePoint();
border[1].x = 120;
border[1].y = 30;
border[1].cost = 0;

border[2] = new FirePoint();
border[2].x = 20;
border[2].y = 20;
border[2].cost = 0;
```

In the main loop for spreading the fire, notice

```
int      pX = borderX[borderSize-1];
int      pY = borderY[borderSize-1];
float pCost = borderCost[borderSize-1];

terrain[pY][pX] = BURNED;

if ((pX > 0) && isBurnable(pY,pX-1)) {
      terrain[pY][pX-1] = FIRE;
      borderX[borderSize] = pX-1;
      borderY[borderSize] = pY;
      borderCost[borderSize] = pCost + 1 + (int)(Math.random()*3);
      borderSize++;
}
// etc ...
```

The new code with the single border array of **FirePoints** looks like this:

```
FirePoint fire = border[borderSize-1];

terrain[fire.y][fire.x] = BURNED;

if ((fire.x > 0) && isBurnable(fire.y,fire.x-1)) {
      terrain[fire.y][fire.x-1] = FIRE;
      border[borderSize] = new FirePoint();
      border[borderSize].x = fire.x-1;
      border[borderSize].y = fire.y;
      border[borderSize].cost = fire.cost + 1 + (int)(Math.random()*3);
      borderSize++;
}
// etc ...
```

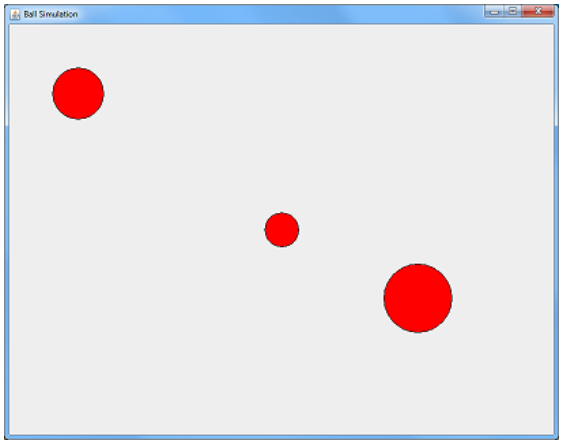
Note that we no longer need **pX**, **pY** and **pCost** variables but just the **fire** variable now. There are other similar changes to the code ... see if you can get it all working with the new **FirePoint** data structure.

Example:

Let us write a program that will simulate some balls bouncing around in a window. To begin, we will create a **Ball** class to represent a ball. Each ball will have a different size, and will have a unique location within the window and will be moving towards some direction at some specific speed. So, here is the class that we will use:

```
public class Ball {
    int     x;
    int     y;
    float   direction;
    float   speed;
    int     radius;
}
```

Here will be the code that will represent the panel that will draw the balls (do not worry about understanding it). It is similar to the **TerrainPanel** from our **FireSpreadSimulation** example:




```
import java.awt.*;
import javax.swing.*;

public class BallPanel extends JPanel {
    public static final int WIDTH = 800;
    public static final int HEIGHT = 600;

    private static Ball[] balls;

    public BallPanel(Ball[] b) {
        balls = b;
        setPreferredSize(new Dimension(WIDTH, HEIGHT));
    }

    public void paintComponent(Graphics g) {  // Display the image
        super.paintComponent(g);
        for (int i=0; i<balls.length; i++) {
            g.setColor(Color.RED);
            g.fillOval(balls[i].x-balls[i].radius,
                       balls[i].y-balls[i].radius,
                       balls[i].radius*2,
                       balls[i].radius*2);
            g.setColor(Color.BLACK);
            g.drawOval(balls[i].x-balls[i].radius,
                       balls[i].y-balls[i].radius,
                       balls[i].radius*2,
                       balls[i].radius*2);
        }
    }
}
```

The above code requires us to pass in an array of **Ball** objects to the panel so that it can display them each time we repaint the window. Consider now the code to create the window and add three balls:

```
import java.awt.*;
import javax.swing.*;

// This application simulates balls bouncing in a window
public class BallSimulation {
    public static BallPanel   ballPanel;

    // Create a window with a BallPanel, add 3 balls and start the simulation
    public static void main(String args[]) {
        Ball[]   balls = new Ball[3];
        balls[0] = new Ball();
        balls[0].x = 100;
        balls[0].y = 100;
        balls[0].direction = (float)(Math.random()*2*Math.PI);
        balls[0].speed = 15;
        balls[0].radius = 20;

        balls[1] = new Ball();
        balls[1].x = 400;
        balls[1].y = 300;
        balls[1].direction = (float)(Math.random()*2*Math.PI);
        balls[1].speed = 10;
        balls[1].radius = 30;

        balls[2] = new Ball();
        balls[2].x = 600;
        balls[2].y = 400;
        balls[2].direction = (float)(Math.random()*2*Math.PI);
        balls[2].speed = 5;
        balls[2].radius = 50;

        JFrame frame = new JFrame("Ball Simulation");
        frame.add(ballPanel = new BallPanel(balls));
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);

        startSimulation(balls);
    }
}
```

The code for creating the array of balls and adding three balls is straight forward. The only method missing is the **startSimulation**() procedure which will perform the simulation by moving the balls around properly. Here is the idea behind what we need to do:

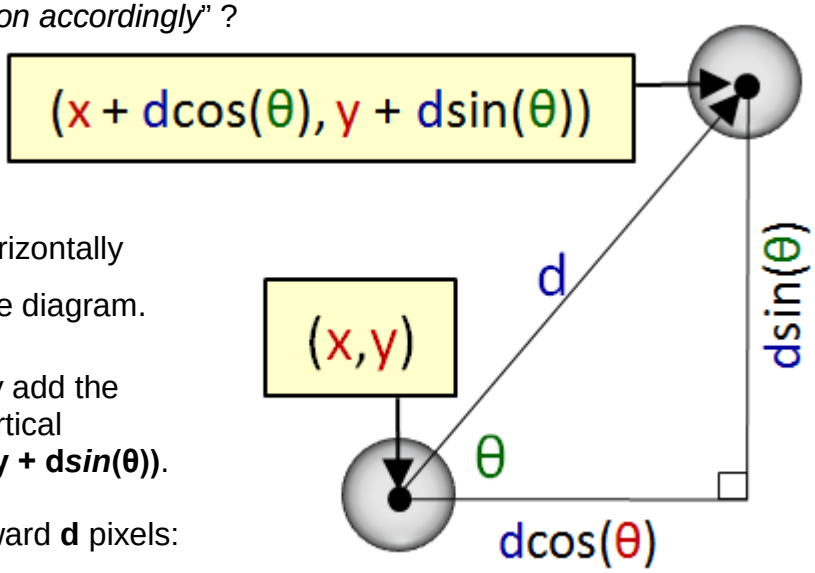
```
public static void startSimulation(Ball[] balls) {
    while(true) {
        for (int i=0; i<balls.length; i++) {
            // ... move ball forward ...
            // ... check if ball collides with borders and adjust accordingly ...
        }
        ballPanel.repaint();
        try{ Thread.sleep(10); } catch(Exception e){};  // To slow things down
    }
}
```

The code will run in an infinite loop (i.e., the program won't end on its own). Each time, it goes through the balls one at a time, updates them and then repaints the window and delays a bit.

It seems fairly straight forward, but two questions arise:

1) How do we "*move the ball forward in its current direction*" ?
2) How do we "*change the direction accordingly*" ?

The first is relatively simple, since it is just based on simple trigonometry. Given that the ball at location **(x,y)** travels distance **d** in direction **θ**, the ball moves an amount of **d•*cos*(θ)** horizontally and **d•*sin*(θ)** vertically as shown in the diagram.



So, to get the new location, we simply add the horizontal component to **x** and the vertical component to **y** to get **(x + d*cos*(θ) , y + d*sin*(θ))**.

Here is the code to move the ball forward **d** pixels:

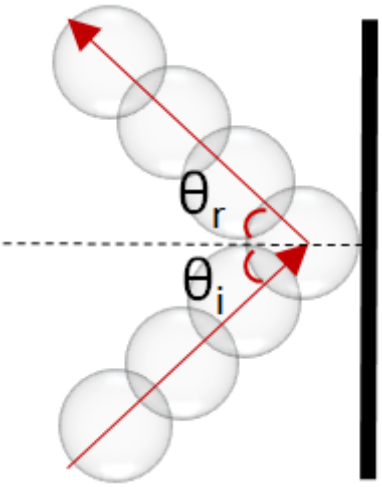
```
balls[i].x = balls[i].x + d * Math.cos(balls[i].direction);
balls[i].y = balls[i].y + d * Math.sin(balls[i].direction);
```

However, the value of **d**, will depend on how fast the ball is moving, so this is what we want:

```
balls[i].x = balls[i].x + balls[i].speed * Math.cos(balls[i].direction);
balls[i].y = balls[i].y + balls[i].speed * Math.sin(balls[i].direction);
```

Now what about changing the direction when the ball encounters a window "wall" ? Well, we would probably like to simulate a realistic collision. To do this, we must understand what happens to a real ball when it hits a wall.

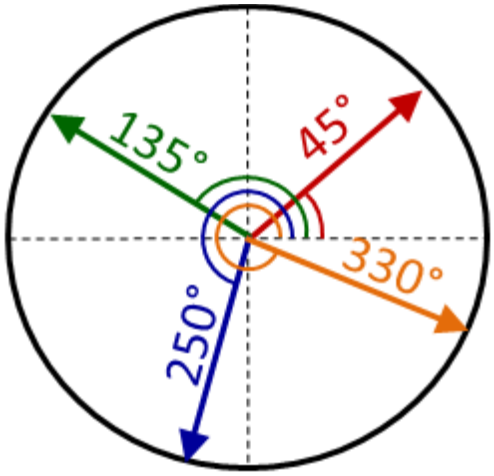
You may recall the ***law of reflection*** from science/physics class. It is often used to explain how light reflects off of a mirror. The law states that the ***angle of reflection*** is the same as the ***angle of incidence***, under ideal conditions. That is, the angle at which the ball bounces off the wall (i.e., θ_r in the diagram), will be the same as the angle at which it hit the wall (i.e., θ_i in the diagram).



However, where do we get the angle of incidence from ?

Well, we have the direction of the ball stored in its **direction** attribute. This direction will always be an angle from **0** to **360°** (or from **0** to **2π** radians).

So, our ball's direction (called **α** for the purpose of this discussion) is always defined with respect to **0°** being the horizontal vector facing to the right. **360°** is the same as **0°**. As the direction changes counter-clockwise, the angle will increase. If the direction changes clockwise, the angle decreases. It is also possible that an angle can become negative. This is ok, since **330°** is the same as **-30°**.

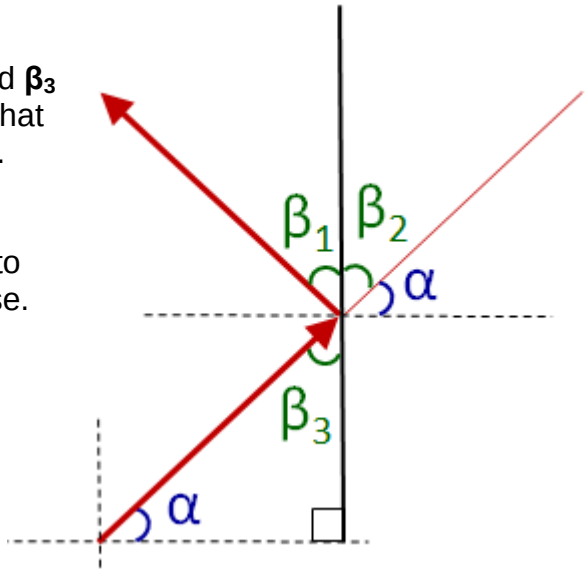


Now, if you think back to the various angle theorems that you encountered in your math courses, you may remember these two:

1) the opposite angles of two straight crossing lines are equal

2) the interior angles of a triangle add up to **180°**

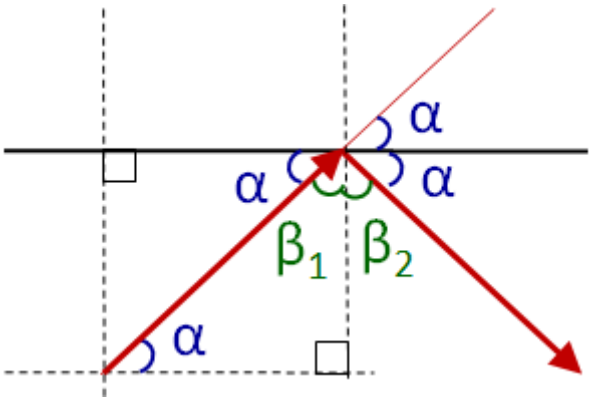
So, in the diagram on the right, for example, the 1st theorem above tells us that opposite angles β_2 and β_3 are equal. From the law of reflection, we also know that β_1 and β_3 are equal. Finally, **α** and β_3 add up to **90°**.



What does all this mean ? Well, since **α** is the ball's direction, then to reflect off the wall, we simply need to add β_1 and β_2 to rotate the direction counter-clockwise. And since β_1, β_2 and β_3 are all equal … and equal to **90° - α**, then to have the ball reflect we just need to do this:

direction = **direction** + (β_1 + β_2)
= **direction** + (90° - α + 90° - α)
= **direction** + (180° - 2 x **direction**)
= 180° - **direction**

The vertical bounce reflection is similar. In the diagram here, it is easy to see that β_1 = **90° - α**. To adjust for the collision on the top of the window, we simply need to subtract **2α** from the direction:



direction = **direction** - 2 x **direction**)
= **- direction**

To summarize then, when the ball reaches the left or right boundaries of the window, we negate the direction and add **180°**, but when it reaches the top or bottom boundaries, we just negate the direction.

Here is how we do it:

```
    if ((balls[i].x >= BallPanel.WIDTH) || (balls[i].x <= 0))
      balls[i].direction = (float)(Math.PI - balls[i].direction);

    if ((balls[i].y >= BallPanel.HEIGHT) || (balls[i].y <= 0))
      balls[i].direction = - balls[i].direction;
```

Our calculations made the assumption that the window boundaries are horizontal and vertical. Similar (yet more complex) formulas can be used for the case where the ball bounces off walls that are placed at some arbitrary angle. Also, all of our calculations assumed that the ball was a point. In reality though, the ball has a shape. If, for example, the ball was drawn as a circle centered at **(x,y)**, then it would only detect a collision when the center of the ball reached the border. How could we fix this ?

We just need to account for the ball's radius during our collision checks:

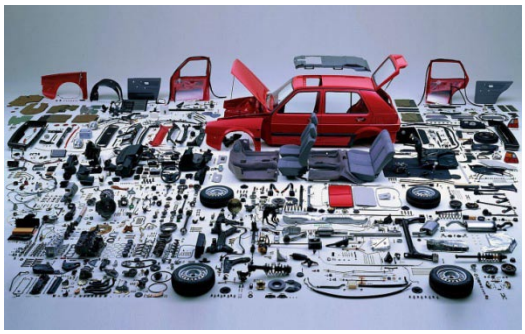
```
    if ((balls[i].x + balls[i].radius >= BallPanel.WIDTH) ||
        (balls[i].x - balls[i].radius <= 0))
      balls[i].direction = (float)(Math.PI - balls[i].direction);

    if ((balls[i].y + balls[i].radius >= BallPanel.HEIGHT) ||
        (balls[i].y - balls[i].radius <= 0))
      balls[i].direction = - balls[i].direction;
```

Here is the completed method:

```
public static void startSimulation(Ball[] balls) {
   while(true) {
      for (int i=0; i<balls.length; i++) {
         // Move the ball forward
         balls[i].x = balls[i].x + (int)(balls[i].speed*Math.cos(balls[i].direction));
         balls[i].y = balls[i].y + (int)(balls[i].speed*Math.sin(balls[i].direction));

         // Check if ball collides with borders and adjust accordingly
         if ((balls[i].x + balls[i].radius >= BallPanel.WIDTH) ||
             (balls[i].x - balls[i].radius <= 0))
                balls[i].direction = (float)(Math.PI - balls[i].direction);

         if ((balls[i].y + balls[i].radius >= BallPanel.HEIGHT) ||
             (balls[i].y - balls[i].radius <= 0))
                balls[i].direction = -balls[i].direction;
      }
      ballPanel.repaint();
      try{ Thread.sleep(10); } catch(Exception e){};
   }
}
```

The use of the **Ball** data structure helps to limit the number of variables being declared and passed around. It also allows us to hide details, making the code cleaner and more logical in the way the code reads.

7.2 Objects Within Objects

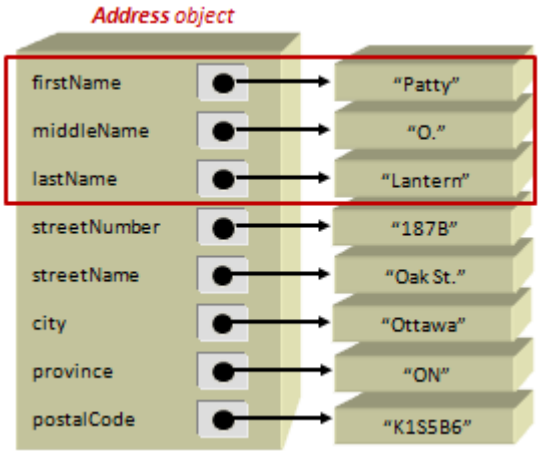


In real life, objects are contained within other objects. For example, a car is an object which contains an engine object and an engine contains other object parts ... etc... There can be many parts to a car, yet when it is all assembled we end up with a single object, called a car, which contains all the parts.

In JAVA, we can also create and store objects within other objects without difficulties. This is often done in order to design our objects similar to real life objects.

For example, consider the address's instance variable to store the **name** as a single string as follows:

```
anAddress.name = "Patty O. Lantern";
```

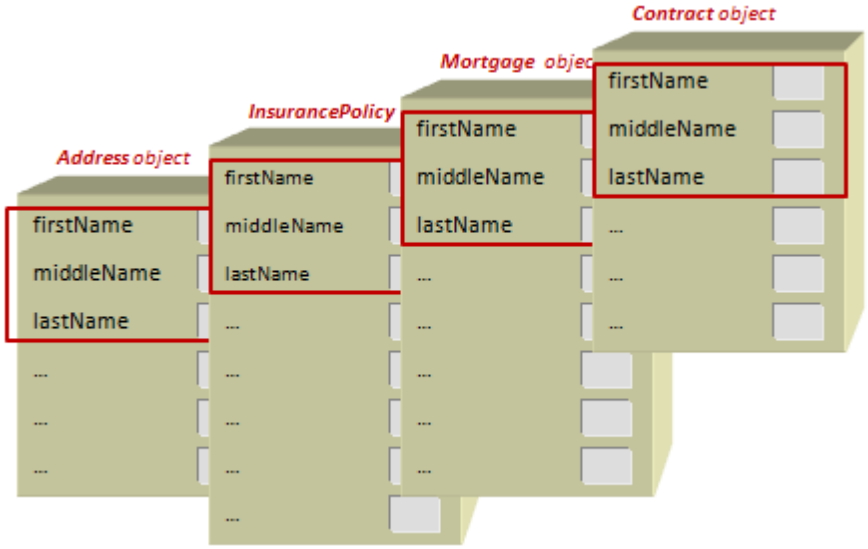


It is sometimes desirable to be able to distinguish between the first, middle and last names of a person. To do this, we would need to separate the names into different variables as follows:

```
anAddress.firstName = "Patty";
anAddress.middleName = "O.";
anAddress.lastName = "Lantern";
```

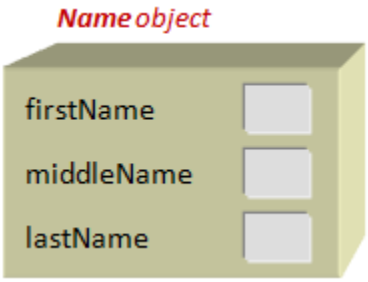
Then we can choose which portion of the person's name that we want to use at any time. A downside is that we now have to use 3 variables instead of 1. We could re-define the **Address** data type as shown in the picture. This would suffice.

However, it is possible that there are many objects for which we would like to store a first, middle and last name. For example, insurance forms, mortgage papers, business contracts, ownership papers, etc.. It is quite easy to simply duplicate all three pieces of data in each object as follows: Rather than duplicate all three pieces of data each time, it would be better to merge them into one object.

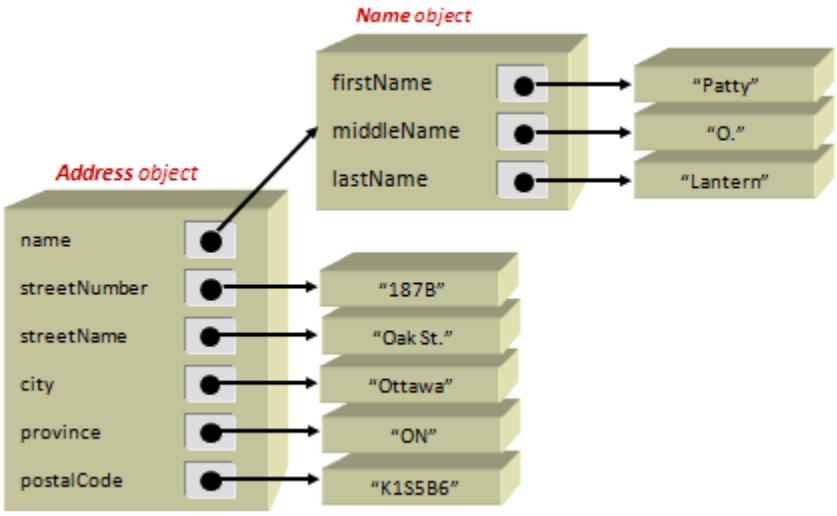


We can combine the first, middle and last names into their own unique data structure (or class) as follows:

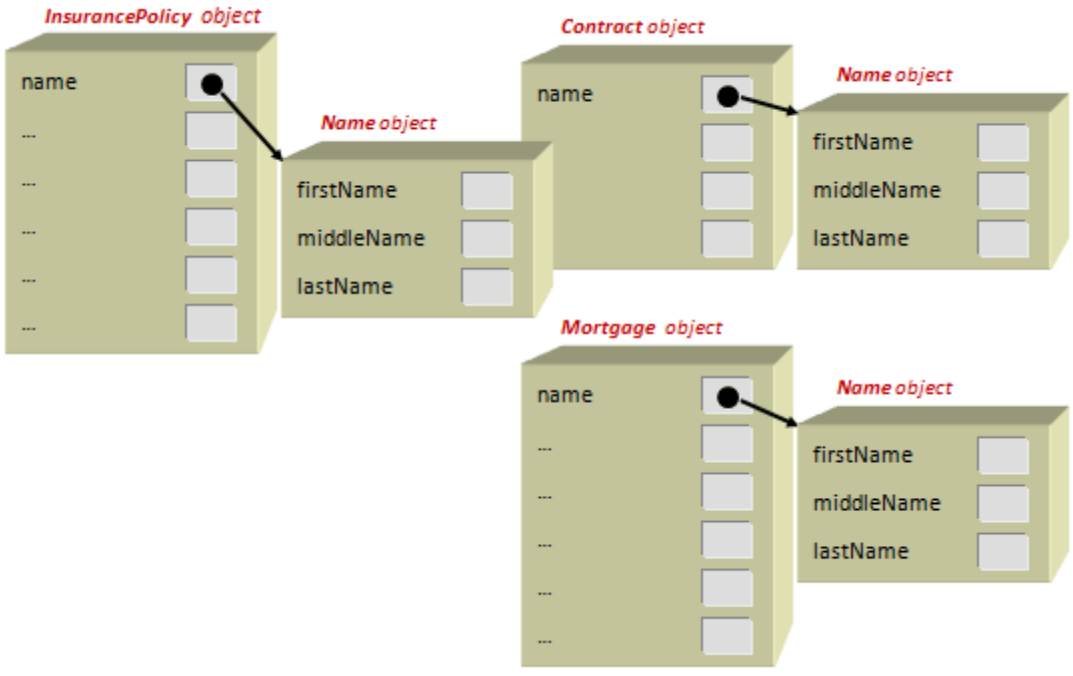
```
public class Name {
      String      firstName;
      String      middleName;
      String      lastName;
}
```



Then we could make use of this new object within our **Address** data structure as shown here.



From the picture you can see that the **name** stored in the **Address** structure is no longer a simple string of characters. Now it is a different kind of data structure (i.e., object) of type **Name**. Other objects can now make use of this **Name** class as well:

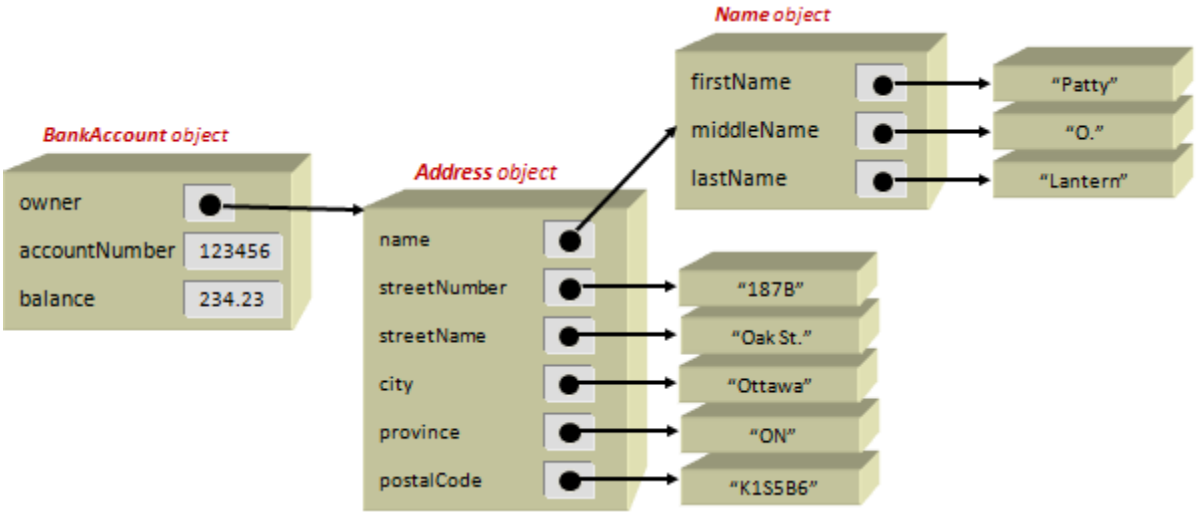


The code for setting the name of **anAddress** would be as follows:

```
Address     anAddress;

anAddress = new Address();

anAddress.name = new Name();
anAddress.name.firstName = "Patty";
anAddress.name.middleName = "O.";
anAddress.name.lastName = "Lantern";

anAddress.streetNumber = "187B";
anAddress.streetName = "Oak St.";
anAddress.city = "Ottawa";
anAddress.province = "ON";
anAddress.postalCode = "K6S8P2";
```

Notice now that the dot operator is used twice: once to go into the address to get the name, then again to get into the name to set the first, middle and last names.

The fun does not stop here. In fact, it is often the case that data structures use multiple layers of other data structures within them. For example, consider the diagram below . Can you create the class definition for the **BankAccount** object ... and then write the code that produces the diagram itself ?



If you cannot figure it out, spend some time with a TA or the instructor. It is important to understand this stuff.

7.3 Constructors

Creating an instance of a data structure that has many instance variables, may require many lines of code. For example, creating and initializing a **Ball** object from our previous example takes 6 lines of code as follows:

```
Ball b = new Ball();
b.x = 100;
b.y = 100;
b.direction = (float)(Math.random()*2*Math.PI);
b.speed = 15;
b.radius = 20;
```

Here, we supplied the initial values for our ball's instance variables. In JAVA, we can significantly reduce the amount of code that we need to write when we create and initialize objects by making use of something called a ***constructor***.

> *A **constructor** is a special function that is automatically called to initialize a new object.*

The parentheses that appear when we do **new Ball()** gives us a clue that **Ball()** is in fact a kind of procedure or function. In fact, this is actually a special kind of function known as a ***default constructor*** … which creates and returns a new fully-initialized object. That is, it reserves space for the data structure's attributes and sets all those with primitive types (e.g., **int**) to a value of "zero" and all those with object types (e.g., String) to a value of **null**.

In our above example, however, we wanted to set the **speed** value of the ball to **15** and the **radius** to **20** … we did not want zeroes. In fact, each object that we create will often have its own unique initial values.

In a way, the idea is analogous to buying a car … we would like to have control to configure our own car with our choice of various options (i.e., we often get to pick the engine size, transmission type, car color, seat style, wheel size, accessories, etc..).

In JAVA, we may write our own constructor procedures so that we can ensure that our new objects to have whatever initial values that we want. Making a constructor is almost identical to defining a function that takes a bunch of parameters (i.e., one for each of the data structure's instance variables) and then uses these parameters to set the instance variables.

A constructor must be written within the data structure's class definition as shown on the next page. You will notice that the constructor's name is identical to the class name (always starting with an uppercase letter). Also, notice in this case, how there is one parameter for each of the 5 instance variables. The types of the parameters must match the types of the instance variables. The names of the parameters (i.e., **p1**, **p2**, **p3**, **p4** and **p5**) are arbitrary, but they must be unique from one another and SHOULD NOT be the same as any instance variable names.

```
public class Ball {
     int     x;
     int     y;
     float   direction;
     float   speed;
     int     radius;

     // This is a constructor
     public Ball(int p1, int p2, float p3, float p4, int p5) {
          x = p1;
          y = p2;
          direction = p3;
          speed = p4;
          radius = p5;
     }
}
```

The code for the body of the constructor is simple. It simply sets each instance variable to have the value of its corresponding parameter.

So what does this all mean ? It means that once we define this constructor, we can then call the constructor with the parameter values that we want to have set in the instance variables. So our previous 6 lines of code shrinks down to this:

```
Ball b = new Ball(100, 100, (float)(Math.random()*2*Math.PI), 15, 20);
```

Notice, it is as if we are simply calling a function that will set all the attribute values. The code inside the constructor just takes the 5 values that we pass in and assigns them to each of the instance variables. So, we actually have the same amount of code … but the code has been moved into (i.e., hidden inside) the constructor. The real advantage becomes evident when we create multiple balls. Here is what I mean. Consider the code for making an array of three **Ball** objects both with and without the use of a constructor:

Without the constructor:

```
Ball[]   balls = new Ball[3];

balls[0] = new Ball();
balls[0].x = 100;
balls[0].y = 100;
balls[0].direction = (float)(Math.random()*2*Math.PI);
balls[0].speed = 15;
balls[0].radius = 20;

balls[1] = new Ball();
balls[1].x = 400;
balls[1].y = 300;
balls[1].direction = (float)(Math.random()*2*Math.PI);
balls[1].speed = 10;
balls[1].radius = 30;

balls[2] = new Ball();
balls[2].x = 600;
balls[2].y = 400;
balls[2].direction = (float)(Math.random()*2*Math.PI);
balls[2].speed = 5;
balls[2].radius = 50;
```

With the constructor:

```
Ball[]   balls = new Ball[3];

balls[0] = new Ball(100, 100, (float)(Math.random()*2*Math.PI), 15, 20);
balls[1] = new Ball(400, 300, (float)(Math.random()*2*Math.PI), 10, 30);
balls[2] = new Ball(600, 400, (float)(Math.random()*2*Math.PI),  5, 50);
```

Notice the significant reduction of code.

We can reduce this code even further by noticing some duplication in the code. Do you understand though, why the following is NOT a solution to reduce the code ?

```
Ball[]   balls = new Ball[3];

float     dir = (float)(Math.random()*2*Math.PI);

balls[0] = new Ball(100, 100, dir, 15, 20);
balls[1] = new Ball(400, 300, dir, 10, 30);
balls[2] = new Ball(600, 400, dir,  5, 50);
```

The code above will give all balls the same direction :(. A proper way to reduce the code is to create another constructor. JAVA allows you to actually define multiple constructors, provided that they have different parameters. A second constructor would be written below the first one. For example, consider this:

```
public class Ball {
     int      x;
     int      y;
     float    direction;
     float    speed;
     int      radius;

     // This is a constructor
     public Ball(int p1, int p2, float p3, float p4, int p5) {
          x = p1;
          y = p2;
          direction = p3;
          speed = p4;
          radius = p5;
     }
     // This is another constructor
     public Ball(int p1, int p2, float p3, int p4) {
          x = p1;
          y = p2;
          direction = (float)(Math.random()*2*Math.PI);
          speed = p3;
          radius = p4;
     }
}
```

Notice the second constructor. It takes 4 parameters now. There is no parameter for the **direction** attribute. Instead, the direction is calculated each time the constructor is called, as it is hard-coded within the constructor itself.

So, notice how we would use this constructor now:

With the 2nd constructor:

```
Ball[]   balls = new Ball[3];

balls[0] = new Ball(100, 100, 15, 20);
balls[1] = new Ball(400, 300, 10, 30);
balls[2] = new Ball(600, 400, 5, 50);
```

That sure looks simple, doesn't it ? The only thing that we would need to remember is what all the numbers mean. We need to remember that the order of the parameters is **x**, **y**, **speed**, **radius**.

In fact, if we wanted all balls to begin at the same speed and of the same size, we could do the following:

```
Ball[]   balls = new Ball[3];

balls[0] = new Ball(100, 100, 10, 30);
balls[1] = new Ball(400, 300, 10, 30);
balls[2] = new Ball(600, 400, 10, 30);
```

But this causes us to duplicate data again. Instead, we could define another constructor:

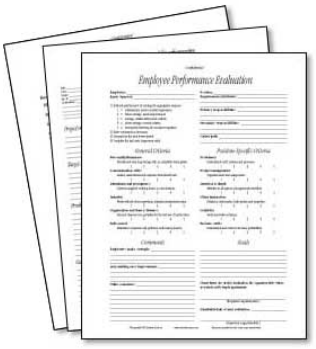
```
// This is a 3rd constructor
public Ball(int p1, int p2) {
     x = p1;
     y = p2;
     direction = (float)(Math.random()*2*Math.PI);
     speed = 10;
     radius = 30;
}
```

Now all balls created with this constructor will have a **speed** of **10** and **radius** of **30** … so our code becomes reduced now as follows:

```
Ball[]   balls = new Ball[3];

balls[0] = new Ball(100, 100);
balls[1] = new Ball(400, 300);
balls[2] = new Ball(600, 400);
```

By looking solely at this code, however, we do not know the default **direction**, **speed** and **radius**. We would have to go look at the constructor code to determine that. We could then add a nice comment in the code:

```
// Create 3 balls, each with a speed of 10, radius of 30 and random direction
Ball[]   balls = new Ball[3];

balls[0] = new Ball(100, 100);
balls[1] = new Ball(400, 300);
balls[2] = new Ball(600, 400);
```



It is a good idea to create a variety of constructors This would be analogous to the situation in real life where someone fills out a form but leaves some information blank. When information has been left out, we must decide what values to use ... so we need to choose some kind of "default" values for the blank parts (i.e., make some assumptions by filling in something appropriate).

It is also good to make a ***default constructor*** (i.e., one that has no parameters). That way, we can make an object without needing to supply any parameters ... since we might not know what we want. We will need to decide upon the default/initial values for ALL of the instance variables. Here we make random speeds and sizes as well but all balls will start at (100,100):

```
// This is a zero-parameter constructor
public Ball() {
     x = 100;
     y = 100;
     direction = (float)(Math.random()*2*Math.PI);
     speed = 5 + (int)(Math.random()*20);
     radius = 5 + (int)(Math.random()*20);
}
```

We can test it easily:

```
// Create 3 balls, each with a location of (100, 100),
// speed between 5 and 24, radius between 5 and 24
// and random direction
Ball[]   balls = new Ball[3];

balls[0] = new Ball();
balls[1] = new Ball();
balls[2] = new Ball();
```

We could even use a loop to fill up the array with a lot of random balls quite easily:

```
// Create 100 balls, each with a location of (100, 100),
// speed between 5 and 24, radius between 5 and 24
// and random direction
Ball[]   balls = new Ball[100];

for (int i=0; i<100; i++)
      balls[i] = new Ball();
```

Remember, though, that the constructors are only used to assign initial values to the objects. We can change the values at any time:

```
Ball[]   balls = new Ball[100];

for (int i=0; i<100; i++) {
      balls[i] = new Ball();
      balls[i].x = 50 + (int)(Math.random() * 500);
      balls[i].y = 50 + (int)(Math.random() * 400);
}
```

The saving in code space can be much more noticeable when multiple kinds of objects are used together. For example, consider creating constructors for **Name**, **Address** and **BankAccount** objects as follows:

```
public class Name {
    String  firstName, middleName, lastName;

    public Name(String p1, String p2, String p3) {
        firstName = p1;
        middleName = p2;
        lastName = p3;
    }
}
```

```
public class Address {
    Name      name;
    int       streetNumber;
    String    streetName, city, province, postalCode;

    public Address(Name p1,String p2, String p3, String p4, String p5, String p6){
        name = p1;
        streetNumber = p2;
        streetName = p3;
        city = p4;
        province = p5;
        postalCode = p6;
    }
}
```

```
public class BankAccount {
    Address owner;
    int     accountNumber;
    float   balance;

    public BankAccount(Address p1, int p2, float p3) {
        owner = p1;
        accountNumber = p2;
        balance = p3;
    }
}
```

Once we make such definitions, notice the significant simplification in code:

Without constructors:

```
BankAccount  b;
Address      a;
FullName     n;

n = new Name();
n.firstName = "Patty"
n.middleName = "O."
n.lastName = "Lantern"

a = new Address();
a.name = n;
a.streetNumber = "187"
a.streetName = "Oak St."
a.city = "Ottawa"
a.province = "ON"
a.postalCode = "K6S8P2"

b = new BankAccount();
b.owner = a;
b.accountNumber = 829302
b.balance = 2319.67f
```

With constructors:

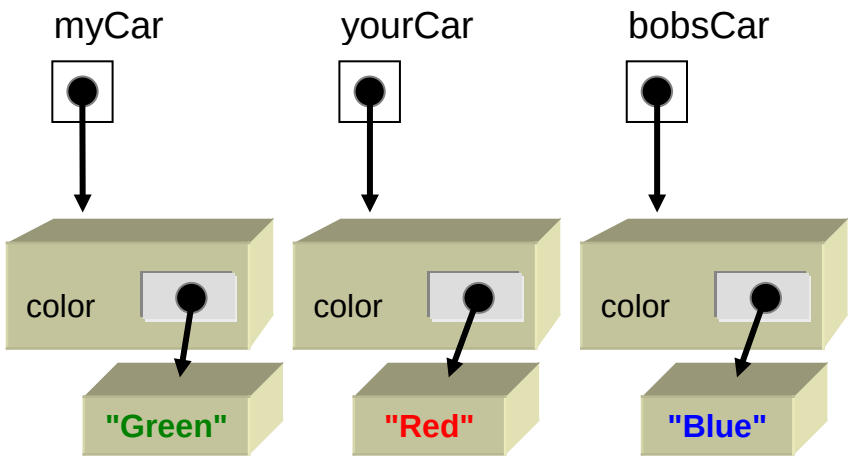
```
BankAccount  b;
Address      a;
Name         n;

n = new Name("Patty", "O.", "Lantern");
a = new Address(n, "187", "Oak St.", "Ottawa", "ON", "K6S8P2");
b = new BankAccount(a, 829302, 2319.67f);
```

So, constructors can be a significant factor in keeping your code simple. We will discuss constructors in more detail in COMP1406.

7.4 Static/Class Variables

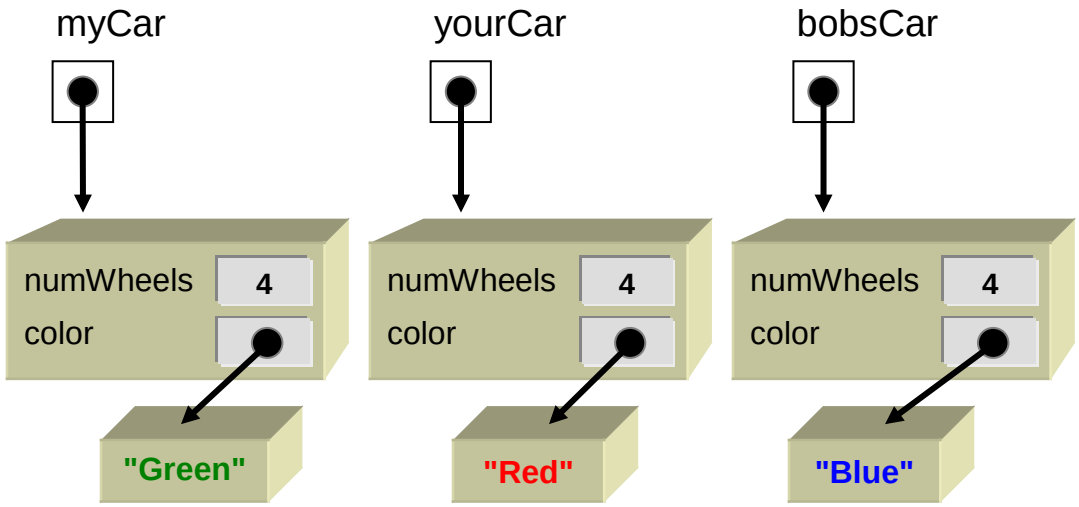
Recall that an ***instance variable*** stores an *attribute* of a particular object. You should now know that an object can have many attributes, and thus many instance variables. The values of the instance variables will vary from object to object.

For example, all **Car** objects may have a **color** attribute, but the color of different cars will likely be different:



In some situations, however, there may be an attribute for an object in which the value does **not differ** between objects of that class. That is, each object that we make of that type would have the same value for that particular attribute. For example, all **Car** objects may have **4** wheels. We could define an instance variable for that attribute (e.g., **numWheels**) and simply set all the values to **4** in the constructor as follows …

```
public class Car {
      String   color;
      int      numWheels;

      public Car() {
          color = "";
          numWheels = 4;
      }
}
```

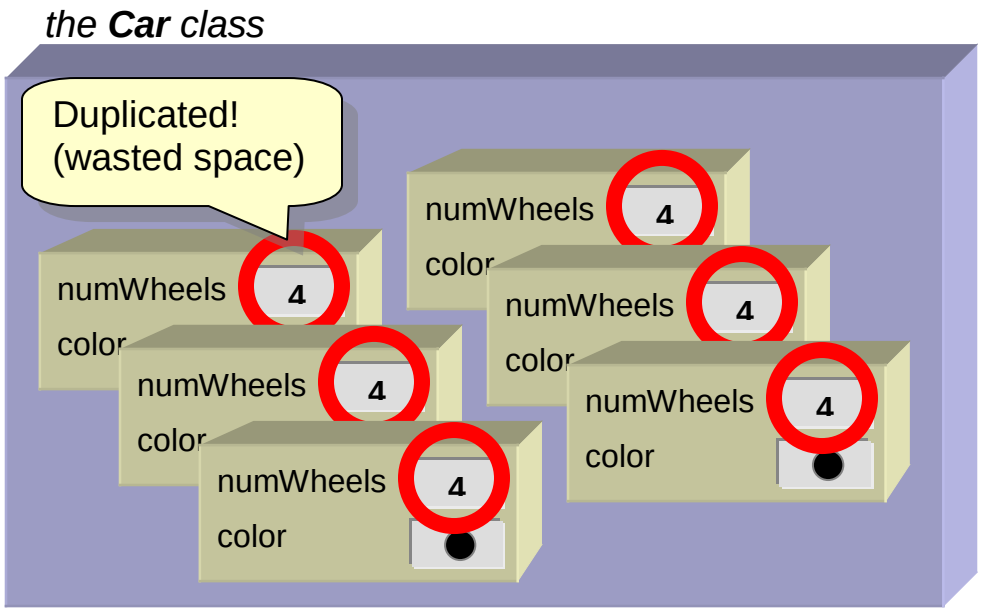


Thus, if we were to access this **numWheels** variable for any of our cars, we would get the value **4**:

```
Car   myCar, yourCar, bobsCar;

myCar = new Car();
yourCar = new Car();
bobsCar = new Car();

System.out.println(myCar.numWheels);      // displays 4
System.out.println(yourCar.numWheels);    // displays 4
System.out.println(bobsCar.numWheels);    // displays 4
```

This strategy works fine and correctly stores the proper number of wheels for each **Car** that we make. However, think about the duplication involved.

Every **Car** object that we create will store the number **4** inside of it. This takes up space in the computer's memory. It is wasteful to have the same value stored over and over again when we know already that the value is the same for all cars.



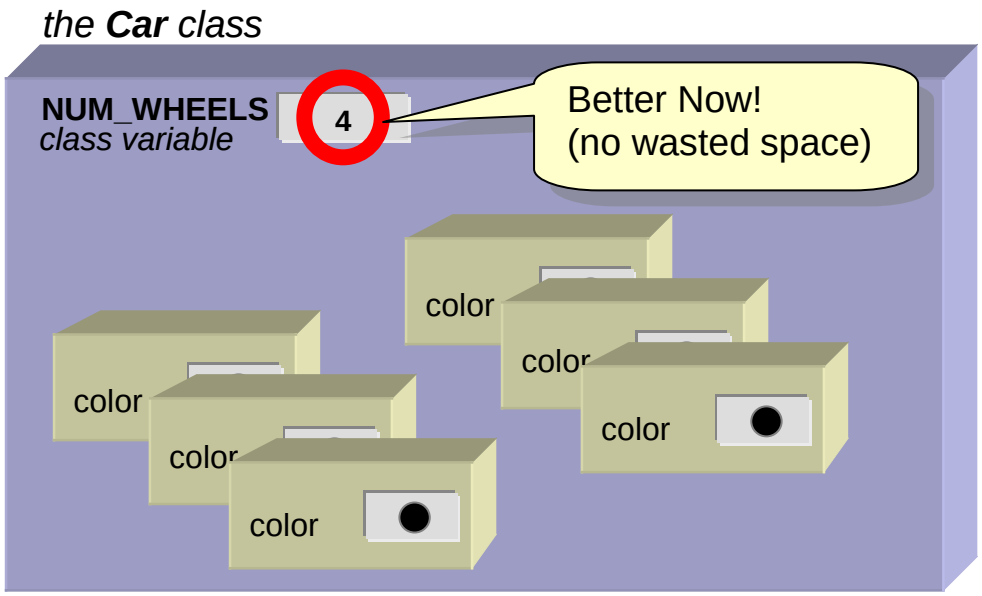
For situations like this … in which all instances of a class (i.e., all objects created of one type) will share the same attribute value, you should create what is called a ***class variable*** (also known as ***static variable***). Class variables are "a little like" instance variables in that you can access them as part of your object. However, they are actually stored in one location in memory and all objects share that location.

In this example, we could create a **class variable** called **NUM_WHEELS** to store the value **4**. (Although it is not necessary, class variables names are often chosen as uppercase characters with an underscore character _ separating the words).

To create a class variable, we write it at the top of our class definition (usually before the instance variables) and put the word **static** in front of it as follows:

```
public class Car {
      static int      NUM_WHEELS = 4;     // a class variable (static)

      String            color;            // an instance variable (not static)

      public Car() {
          color = "";
      }
}
```

Normally, for static variables, we supply an initial value for it when we create it. In this case, we assign it the value of **4** as needed. Here is a diagram showing how the storage has now changed …



Notice that the number **4** is now stored in only one location … it is not duplicated every time that a **Car** object is created. We can also perhaps imagine creating classes to store other kinds of vehicles in which we declare a similar **NUM_WHEELS** variable as follows:

```
public class Motorcycle {
      static int   NUM_WHEELS = 2;
      // ...
}
```

```
public class Unicycle {
      static int   NUM_WHEELS = 1;
      // ...
}
```

```
public class Boat {
      static int   NUM_WHEELS = 0;
      // ...
}
```

The **NUM_WHEELS** variable is a variable just like any other. We can access it at any time and change its value. You can access static/class variables anywhere in your program by preceding them by the class name that they are defined in, followed by the dot **.** operator. The following code would produce the values 4, 2, 1, 0 and 6 …

```
System.out.println("A car has " + Car.NUM_WHEELS + " wheels");
System.out.println("A motorcycle has " + Motorcycle.NUM_WHEELS + " wheels");
System.out.println("A unicycle has " + Unicycle.NUM_WHEELS + " wheels");
System.out.println("A boat has " + Boat.NUM_WHEELS + " wheels");

Car.NUM_WHEELS = 6;
System.out.println("A car now has " + Car.NUM_WHEELS + " wheels");
```

With regards to the **NUM_WHEELS** static/class variable that we defined above, it is likely that we would never change the value from 4 to 6 in a real system. Likely, the **NUM_WHEELS** variable should remain constant. In this case, we use the term **static constant** (or ***class constant***) instead of static variable … since its value will never *vary* but instead *remain the same* over time. In JAVA, whenever we want to prevent a value from being changed (i.e., to make it a *constant*), we use the **final** keyword when we declare the variable as follows:

```
public class Car {
      static final int    NUM_WHEELS = 4; // a static(class) constant

      ...
}
```

After we do this, we are no longer allowed to change the value of this variable in our program:

```
Car.NUM_WHEELS = 6;
```

Static/class variables are sometimes used to store a commonly accessed value that is shared between many objects, such as a global counter. For example, consider a **BankAccount** object, where each account is assigned a unique **accountNumber**. When creating a new **BankAccount**, it is unlikely in real life that we would be able to specify the **accountNumber**. Usually, these are assigned automatically to the customer. What **accountNumber** should a new **BankAccount** receive? It's up to us to decide (In real life however, the bank that is hiring you to write their program would specify their account numbering strategy).

Let us assume that the first created account is assigned the account number 100001, the second gets 100002, the third 100003 and so on. In this scenario, we can simply keep a counter that starts at 100001 and increases each time a new account is created.

To do this, we can create a class variable in the **BankAccount** class to represent this counter. We can call it **LAST_ACCOUNT_NUMBER** which will store the account number that was last given out. We can give this variable an initial value of 100000 as follows …

```
public class BankAccount {
      static int LAST_ACCOUNT_NUMBER = 100000;

      int         accountNumber;    // instance variable
      String      owner;            // instance variable
      float       balance;          // instance variable

      // ...
}
```

Then, when a new **BankAccount** is created, we can give it an **accountNumber** which is one more than the **LAST_ACCOUNT_NUMBER** and then increment this counter to get it ready for the next time. This counter of ours will work exactly like one of those ticket dispensers when you wait in line at a store.

This can be done by adjusting all of the **BankAccount** constructors so that they do not allow the user to "specify" the **accountNumber**. But rather set it to the next available number and then increment the counter. Here is the code that we would need to write:

```
public class BankAccount {
      static int LAST_ACCOUNT_NUMBER = 100000;

      int      accountNumber;
      String   owner;
      float    balance;

      // This is the 0-parameter constructor
      public BankAccount() {
          owner = "";
          LAST_ACCOUNT_NUMBER = LAST_ACCOUNT_NUMBER + 1;
          accountNumber = LAST_ACCOUNT_NUMBER;
          balance = 0;
      }

      // This is a 2-parameter constructor
      public BankAccount(String n, float b) {
          owner = n;
          LAST_ACCOUNT_NUMBER = LAST_ACCOUNT_NUMBER + 1;
          accountNumber = LAST_ACCOUNT_NUMBER;
          balance = b;
      }

      //...
}
```

Here is some testing code:

```
public class BankAccountTestProgram {
      public static void main(String args[]) {
            BankAccount b, m, j;

            b = new BankAccount("Bob", 250.00f);
            m = new BankAccount("Mary", 6387.27f);
            j = new BankAccount("Jay", 915.45f);

            System.out.println(b.accountNumber);
            System.out.println(m.accountNumber);
            System.out.println(j.accountNumber);
      }
}
```

The account numbers printed will be 100001, 100002 and 100003.

7.5 Displaying Objects

Do you know what happens when we display one of our objects to the System console ?

```
public class MyObjectsTestProgram {
    public static void main(String args[]) {
        System.out.println(new Name());
        System.out.println(new Address());
        System.out.println(new Applicant());
        System.out.println(new Ball());
    }
}
```

The result on the screen would be something like this:

```
Name@3b4766
Address@37601e
Applicant@12345a0
Ball@179a711
```

JAVA displays all of the objects that you make in this manner. The numbers and letters after the @ symbol will vary from program to program, as they are values that refer to the storage location of the object. JAVA happens to be converting our objects to **String** objects first and then displaying the resulting characters to the screen. In fact, every object in JAVA has, by default, a method called **toString()** which will convert the object to a **String**. Hence, the following code will take a **Address** and a **Ball** object, convert them to **String** objects and then display the resulting **String** objects:

```
Address     a;
Ball        b;
String      s1, s2;

a = new Address();
b = new Ball();
s1 = a.toString();      // s1 will be "Address@37601e"
s2 = b.toString();      // s2 will be "Ball@179a711"

System.out.println("The Address as a String is " + s1);
System.out.println("The Ball as a String is " + s2);
```

Notice that the output will be:

```
The Address as a String is Address@37601e
The Ball as a String is Ball@179a711
```

The **String** objects have the exact same characters that are displayed when we just display the objects directly using **System.out.println()**. That is because when JAVA attempts to display anything to the console, it automatically calls the **toString()** method for the object to convert it to characters before displaying. So, the following two lines of code do exactly the same thing:

```
System.out.println(b);
System.out.println(b.toString());
```

Why do we care ? Well, we can actually replace the default **toString()** behavior by writing our own **toString()** method for all of our own objects that defines exactly how to convert our object to a **String**. That is, we can control the way our object "looks" when we print it on the screen.

Consider the following **Person** class definition:

```
public class Person {
    public String   name;
    public int      age;
    public float    height;
    public char     gender;
    public boolean  retired;
}
```

Suppose that we wanted our **Person** object to display something like this:

```
Person named Bob
```

You should notice that the first two words of this String will not change but the last part will vary from person to person. We can make this to be the standard output format for all **Person** objects simply by writing the following **toString()** method in the **Person** class:

```
public class Person {
    public String   name;
    public int      age;
    public float    height;
    public char     gender;
    public boolean  retired;

    public Person(String p1, int p2, float p3, char p4, boolean p5) {
        name = p1;
        age = p2;
        height = p3;
        gender = p4;
        retired = p5;
    }

    public String toString() {
        return ("Person named " + name);
    }
}
```

Notice that the method is called **toString()** with no parameters and that it has a return type of **String**. This is important in order for the method to be used properly by JAVA. Even the spelling and upper/lower case letters must match exactly. Then, you may notice that the method returns an actual String object that is made up of the letters **"Person named "** and then followed by the value of this **Person** object's **name** attribute. What would therefore be the output of the following code ? ...

```
Person   p1, p2, p3;

p1 = new Person();      // assume name is set to "" within constructor
p2 = new Person("Holly Day", 15, 5.6. 'F', false);
p3 = new Person("Hank Urchiff", 89, 5.4, 'M', true);

System.out.println(p1);
System.out.println(p2);
System.out.println(p3);
```

Here is the output … were you correct ?

```
Person named
Person named Holly Day
Person named Hank Urchiff
```

Now what if we wanted the output to be in this format instead:

```
19 year old Person named Hank Urchif
```

To write an appropriate **toString()** method, we need to understand what is fixed in this output and what will vary. The number **19** should vary for each person as well as the **name**. Here is how we could write the code (replacing our previous **toString()** method):

```
public String toString() {
     return (age + " year old Person named " + name);
}
```

Notice that the basic idea behind creating a **toString()** method is to simply keep joining together **String** pieces to form the resulting **String**.

Now here is a harder one. Let's see if we can make it into this format:

```
19 year old non-retired person named Hank Urchif
```

Here we have the **age** and **names** being variable again but now we also have the added variance of their **retirement status** and **gender**. Here is one attempt:

```
public String toString() {
     return (age + " year old " + retired + " person named " + name);
}
```

However, this is not quite correct. This would be the format we would end up with:

```
19 year old false person named Hank Urchif
```

Notice that we cannot simply display the value of the **retired** attribute but instead need to write **"retired"** or **"non-retired"** for the **retired** status.

To do this then, we will need to use an **IF** statement. However, in JAVA, we cannot write an **IF** statement in the middle of a **return** statement. So we will need to do this using more than one line of code.

Let's make an **answer** variable to hold the result and then break down our method into logical pieces that append to this **answer**:

```
public String toString() {
    String    answer;

    answer = age + " year old ";
    answer = answer + retired;
    answer = answer + " person named " + name);

    return answer;
}
```

Now we can insert the appropriate **IF** statements as follows:

```
public String toString() {
    String    answer;

    answer = age + " year old ";

    if (retired)
        answer = answer + "retired";
    else
        answer = answer + "non-retired";
    answer = answer + " person named " + name;

    return answer;
}
```

The result is what we wanted. Note however, that we can simplify this code a little further:

```
public String toString() {
    String    answer = age + " year old ";

    if (!retired)
        answer = answer + "non-";

    return (answer + "retired person named " + name);
}
```

So, you can see that the **toString()** method may be more than one line of code but again … the main idea is to simply keep appending to the **String** as you go … building it up.

7.6 Team/League Example


Let's consider a larger example in which we create classes called **Team** and **League** ... where a **League** object will contain a bunch of **Team** objects. That is, the **League** object will have an instance variable which is an array of multiple **Team** objects within the league.

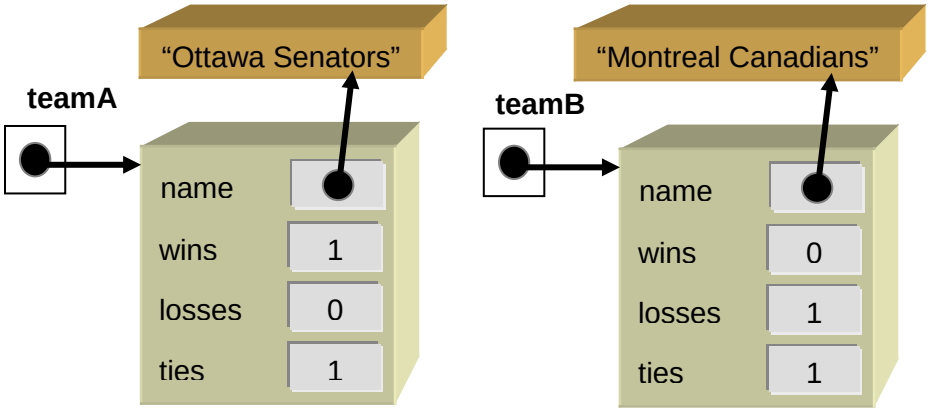
Consider first the creation of a **Team** class that will represent a single team in the league. For each team, we will maintain the team's **name** as well as the number of **wins**, **losses** and **ties** for the games that they played. Here is the basic class with attributes, a constructor and a **toString()** method:

```
public class Team {
    String  name;       // The name of the Team
    int     wins;       // The number of games that the Team won
    int     losses;     // The number of games that the Team lost
    int     ties;       // The number of games that the Team tied

    public Team(String aName) {
        name = aName;
        wins = 0;
        losses = 0;
        ties = 0;
    }

    public String toString() {
        return("The " + name + " have " + wins + " wins, " +
               losses + " losses and " + ties + " ties.");
    }
}
```

We can test out our **Team** object with the following test code, just to make sure it works:

```
public class TeamTestProgram {

    public static int totalPoints(Team t) {
        return (t.wins * 2 + t.ties);
    }

    public static int gamesPlayed(Team t) {
        return (t.wins + t.losses + t.ties);
    }

    public static void main(String args[]) {
        Team    teamA, teamB;

        teamA = new Team("Ottawa Senators");
        teamB = new Team("Montreal Canadians");

        // Simulate the playing of a game in which teamA beats teamB
        System.out.println(teamA.name + " just beat " + teamB.name);
        teamA.wins++;
        teamB.losses++;

        // Simulate the playing of another game in which they tied
        System.out.println(teamA.name + " just tied " + teamB.name);
        teamA.ties++;
        teamB.ties++;

        // Now print out some statistics
        System.out.println(teamA);
        System.out.println(teamB);
        System.out.print("The " + teamA.name + " have ");
        System.out.print(totalPoints(teamA) + " points and played ");
        System.out.println(gamesPlayed(teamA) + " games.");
        System.out.print("The " + teamB.name + " have ");
        System.out.print(totalPoints(teamB) + " points and played ");
        System.out.println(gamesPlayed(teamB) + " games.");
    }
}
```

Here is what the **Team** objects look like after playing the two games:



Here is the output from our little test program:

```
Ottawa Senators just beat Montreal Canadians
Ottawa Senators just tied Montreal Canadians
The Ottawa Senators have 1 wins, 0 losses and 1 ties.
The Montreal Canadians have 0 wins, 1 losses and 1 ties.
The Ottawa Senators have 3 points and played 2 games.
The Montreal Canadians have 1 points and played 2 games.
```

Now let's implement the **League** class. A league will also have a **name** as well as an array (called **teams**) of **Team** objects**.** Here is the basic class structure:

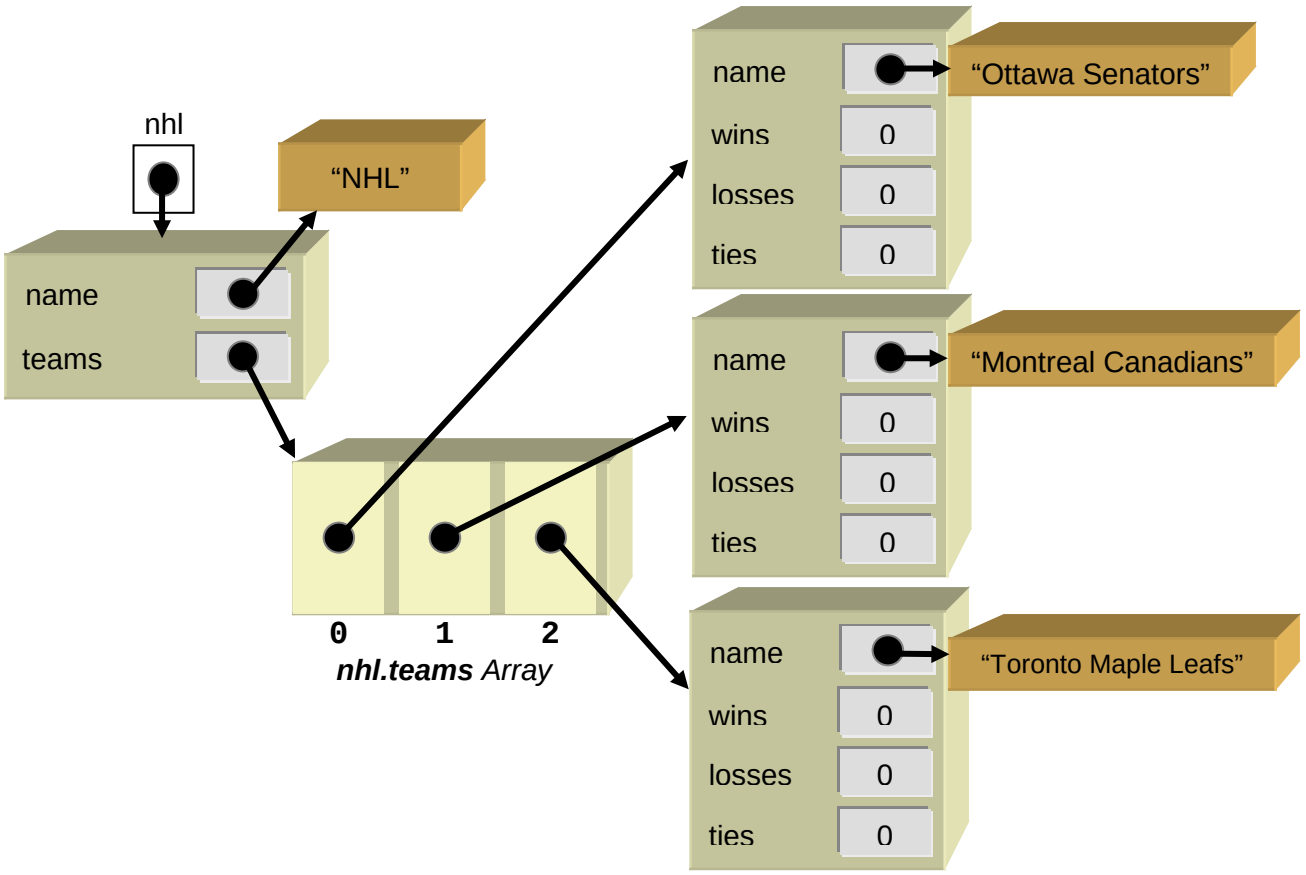
```
public class League {
    String  name;
    Team[]  teams;

    public League(String n) {
        name = n;
        teams = new Team[8]; // Doesn't make any Team objects
    }

    // This specifies the appearance of the League
    public String toString() {
        return ("The " + name + " league");
    }
}
```

Notice that the array is created within the constructor and that it is initially empty. That means, a brand new league has no teams in it, but currently has the capacity to store **8** teams in total. It is important to note also that there are no **Team** objects created at this time. That is, we just reserved space for **8** *pointers/references* to **Team** objects that will be created later.

At this point, we have defined two objects: **Team** and **League**. One thing that we will need to do is to be able to add teams to the league. Here is an example of how we can create a league with three teams in it:

```
League  nhl;

nhl = new League("NHL");
nhl.teams[0] = new Team("Ottawa Senators");
nhl.teams[1] = new Team("Montreal Canadians");
nhl.teams[2] = new Team("Toronto Maple Leafs");
```

Here is a diagram showing how the **League** object stores the 3 **Teams** …



Suppose now that we wanted to print out the teams in the league. Let's write a static method to do this. The method will need to go through each team in the **teams** array and display the particular team's information … perhaps using the **toString()** method from the **Team** class.

Hopefully, you "sense" that printing out all the teams involves repeating some code over and over again. That is, you should realize that we need a loop of some type.

```
public static void displayTeams(League aLeague) {
    for (int i=0; i<aLeague.teams.length; i++)
        System.out.println(aLeague.teams[i]);
}
```

Let us test our method out using the following test program:

```
public class LeagueTestProgram {

    // Displays a league's teams to the system console
    public static void displayTeams(League aLeague) {
        for (int i=0; i<aLeague.teams.length; i++)
            System.out.println(aLeague.teams[i]);
    }

    public static void main(String args[]) {
        League  nhl;

        nhl = new League("NHL");

        //Add a pile of teams to the league
        nhl.teams[0] = new Team("Ottawa Senators");
        nhl.teams[1] = new Team("Montreal Canadians");
        nhl.teams[2] = new Team("Toronto Maple Leafs");
        nhl.teams[3] = new Team("Vancouver Canucks");
        nhl.teams[4] = new Team("Edmonton Oilers");
        nhl.teams[5] = new Team("Washington Capitals");
        nhl.teams[6] = new Team("New Jersey Devils");
        nhl.teams[7] = new Team("Detroit Red Wings");

        //Display the teams
        System.out.println("\nHere are the teams:");
        displayTeams(nhl);
    }
}
```

Here is the output so far:

```
Here are the teams:
The Ottawa Senators have 0 wins, 0 losses and 0 ties.
The Montreal Canadians have 0 wins, 0 losses and 0 ties.
The Toronto Maple Leafs have 0 wins, 0 losses and 0 ties.
The Vancouver Canucks have 0 wins, 0 losses and 0 ties.
The Edmonton Oilers have 0 wins, 0 losses and 0 ties.
The Washington Capitals have 0 wins, 0 losses and 0 ties.
The New Jersey Devils have 0 wins, 0 losses and 0 ties.
The Detroit Red Wings have 0 wins, 0 losses and 0 ties.
```

Notice that all the teams have no recorded wins, losses or ties. Let's write a method that will record a win and a loss for two teams that play together, and another method to record a tie when the two teams play and tie.

```
public static void recordWinAndLoss(Team winner, Team loser) {
    winner.wins++;
    loser.losses++;
}

public static void recordTie(Team teamA, Team teamB) {
    teamA.ties++;
    teamB.ties++;
}
```

If we wanted to test these methods now, we could write test code like this:

```
League   nhl;

nhl = new League("NHL");
nhl.teams[0] = new Team("Ottawa Senators"));
nhl.teams[1] = new Team("Montreal Canadians"));
nhl.teams[2] = new Team("Toronto Maple Leafs"));
...
recordWinAndLoss(nhl.teams[0], nhl.teams[1]);
recordTie(nhl.teams[0], nhl.teams[1]);
recordWinAndLoss(nhl.teams[2], nhl.teams[1]);
...
```

You should now notice something tedious. We would have to remember the location of each team in the array if we want to record wins, losses and ties among them. Why ? Because the recording methods require **Team** objects ... the same **Team** objects that we added to the **League**. Perhaps a better way to record wins, losses and ties would be to do something like this:

```
League   nhl;

nhl = new League("NHL");
nhl.teams[0] = new Team("Ottawa Senators"));
nhl.teams[1] = new Team("Montreal Canadians"));
nhl.teams[2] = new Team("Toronto Maple Leafs"));
...
recordWinAndLoss(nhl, "Ottawa Senators", "Montreal Canadians");
recordTie(nhl, "Ottawa Senators", "Montreal Canadians");
recordWinAndLoss(nhl, "Toronto Maple Leafs", "Montreal Canadians");
...
```

However, we would have to make new recording methods that took Strings (i.e., the **Team** names) as parameters instead of **Team** objects. A **League** object is also required as a parameter because it contains the teams that we need to access and modify. Here are the methods that we would need to implement (notice the difference in the parameter types):

```
public static void recordWinAndLoss(League aLeague, String nameW, String nameL) {
    …
}

public static void recordTie(League aLeague, String nameA, String nameB) {
    …
}
```

To make this work, however, we still need to get into the appropriate **Team** objects and update their wins/losses/ties. Therefore, we will have to take the incoming team names and find the **Team** objects that correspond with those names. We would need to do this 4 times: once for **nameW**, once for **nameL**, once for **nameA** and once for **nameB**. Rather than repeat the code **4** times, we will make a method to do this particular sub-task of finding a team with a given name. Here is the method that we will write:

```
public static Team teamWithName(League aLeague, String nameToLookFor) {
    Team    answer;
    ...
    return answer;
}
```

Notice that it will take the team's name as a parameter and then return a **Team** object. How would we complete this method ? We can use a **for** loop to traverse through all the teams and find the one with that name as follows:

```
public static Team teamWithName(League aLeague, String nameToLookFor) {
    Team    answer = null;

    for (int i=0; i<aLeague.teams.length; i++) {
        if (aLeague.teams[i].name.equals(nameToLookFor))
            answer = aLeague.teams[i];
    }

    return answer;
}
```

Notice a few points. First, we set the answer to **null**. If we do not find a **Team** with the given name, the method returns **null** … which is the only appropriate answer. Next, notice that for each team, we compare its name with the incoming string **nameToLookFor** and if these two strings are equal, then we have found the **Team** object that we want, so we store it in the **answer** variable to be returned at the completion of the loop.

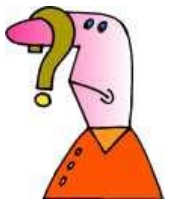
This method can be shortened as follows:

```
public static Team teamWithName(League aLeague, String nameToLookFor) {
   for (int i=0; i<aLeague.teams.length; i++) {
      if (aLeague.teams[i].name.equals(nameToLookFor))
         return aLeague.teams[i];
   }
   return null;
}
```

Now that this method has been created, we can use it in our methods for recording wins/losses and ties as follows:

```
public static void recordWinAndLoss(League aLeague, String nameW, String nameL) {
    Team    winner, loser;

    winner = teamWithName(aLeague, nameW);
    loser = teamWithName(aLeague, nameL);
    winner.wins++;
    loser.losses++;
}

public static void recordTie(League aLeague, String nameA, String nameB) {
    Team    teamA, teamB;

    teamA = teamWithName(aLeague, nameA);
    teamB = teamWithName(aLeague, nameB);
    teamA.ties++;
    teamB.ties++;
}
```

The methods work as before, but there are potential problems. What if we cannot find the **Team** object with the given names (e.g., someone spelt the name wrong) ?



In this case, perhaps **winner**, **loser**, **teamA** or **teamB** will be **null** and we will get a **NullPointerException** when we try to access the team's attributes. We can check for this with an **IF** statement as follows:

```
public static void recordWinAndLoss(League aLeague, String nameW, String nameL) {
    Team    winner, loser;

    winner = teamWithName(aLeague, nameW);
    loser = teamWithName(aLeague, nameL);
    if ((winner != null) && (loser != null)) {
        winner.wins++;
        loser.losses++;
    }
}
```

```
public static void recordTie(League aLeague, String nameA, String nameB) {
    Team    teamA, teamB;

    teamA = teamWithName(aLeague, nameA);
    teamB = teamWithName(aLeague, nameB);
    if ((teamA != null) && (teamB != null)) {
        teamA.ties++;
        teamB.ties++;
    }
}
```

Now the games are only recorded when we have successfully identified the two **Team** objects that need to be updated as a result of the played game. Interestingly though, the same problem may occur in our previous recording methods … that is … the **Team** objects passed in may be **null**. Also, in our code, we already have a method for recording the wins/losses/ties in the case where we have the **Team** objects, so we should call those methods from here. We can simply call the previous recording methods from these two new ones and move the **null**-checking in there instead as follows:

```
public static Team teamWithName(League aLeague, String nameToLookFor) {
   for (int i=0; i<aLeague.teams.length; i++) {
      if (aLeague.teams[i].name.equals(nameToLookFor))
         return aLeague.teams[i];
   }
   return null;
}
public static void recordWinAndLoss(Team winner, Team loser) {
    if ((winner != null) && (loser != null)) {
        winner.wins++;
        loser.losses++;
    }
}
public static void recordTie(Team teamA, Team teamB) {
    if ((teamA != null) && (teamB != null)) {
        teamA.ties++;
        teamB.ties++;
    }
}
public static void recordWinAndLoss(League aLeague, String nameW, String nameL) {
    Team    winner, loser;

    winner = teamWithName(aLeague, nameW);
    loser = teamWithName(aLeague, nameL);
    recordWinAndLoss(winner, loser);
}
public static void recordTie(League aLeague, String nameA, String nameB) {
    Team    teamA, teamB;

    teamA = teamWithName(aLeague, nameA);
    teamB = teamWithName(aLeague, nameB);
    recordTie(teamA, teamB);
}
```

In fact, we can even shorten the last two methods by noticing that the variables are not really necessary:

```
public static void recordWinAndLoss(League aLeague, String nameW, String nameL) {
   recordWinAndLoss(teamWithName(aLeague, nameW), teamWithName(aLeague, nameL));
}

public static void recordTie(League aLeague, String nameA, String nameB) {
   recordTie(teamWithName(aLeague, nameA), teamWithName(aLeague, nameB));
}
```

Consider a method called **totalGamesPlayed()** which is supposed to return the total number of games played in the league. All we need to do is count the number of games played by all the teams (i.e., we will need some kind of counter) and then divide by **2** (since each game was played by two teams, hence counted twice). Here is the format:

```
public static int totalGamesPlayed(League aLeague) {
    int  total = 0;
    …
    return total/2;
}
```

We will also need a **FOR** loop to go through each team:

```
public static int totalGamesPlayed(League aLeague) {
    int  total = 0;
    for (int i=0; i<aLeague.teams.length; i++) {
        …
    }
    return total/2;
}
```

To determine the answer, we can ask each team how many games they played by adding up their **wins**, **losses** and **ties**. So we can use this method that we wrote earlier:

```
public static int gamesPlayed(Team t) {
    return (t.wins + t.losses + t.ties);
}
```

Here is how to use it:

```
public static int totalGamesPlayed(League aLeague) {
    int  total = 0;

    for (int i=0; i<aLeague.teams.length; i++)
        total += gamesPlayed(aLeague.teams[i]);

    return total/2;  // Divide by two since a game is counted by both teams
}
```

Notice that the method is quite simple, as long as you break it down into simple steps like we just did. For more practice, let's find the team that is in first place (i.e., the **Team** object that has the most points).

We can make use of this method that we wrote earlier:

```
public static int totalPoints(Team t) {
    return (t.wins * 2 + t.ties);
}
```

Let's start again as follows:

```
public static Team firstPlaceTeam(League aLeague) {
    Team  teamWithMostPoints = null;

    for (int i=0; i<aLeague.teams.length; i++) {
        ...
    }
    return teamWithMostPoints;
}
```

Notice that it returns a **Team** object. Now, we can make use of our **totalPoints()** method which returns the number of points for a particular team:

```
public static Team firstPlaceTeam(League aLeague) {
    Team    teamWithMostPoints = null;

    for (int i=0; i<aLeague.teams.length; i++) {
        int points = totalPoints(aLeague.teams[i]);
    }
    return teamWithMostPoints;
}
```

But now what do we do ? The current code will simply grab each team's point values one at a time. We need to somehow compare them. Many students have trouble breaking this problem down into simple steps. The natural tendency is to say to yourself "I will compare the 1st team's points with the 2nd team's points and see which is greater". If we do this however, then what do we do with that answer ? How does the third team come into the picture ?

Hopefully, after some thinking, you would realize that as we traverse through the teams, we need to keep track of (i.e., remember) the best one so far.

Imagine for example, searching through a basket of apples to find the best one. Would you not grab an apple and hold it in your hand and then look through the other apples and compare them with the one you are holding in your hand ? If you found a better one, you would simply trade the one currently in your hand with the new better one. By the time you reach the end of the basket, you are holding the best apple.

Well we are going to do the same thing. The **teamWithMostPoints** variable will be like our good apple that we are holding. Whenever we find a team that is better (i.e., more points) than this one, then that one becomes the **teamWithMostPoints**. Here is the code:

```
public static Team firstPlaceTeam(League aLeague) {
    Team    teamWithMostPoints = null;

    for (int i=0; i<aLeague.teams.length; i++) {
        if (totalPoints(aLeague.teams[i]) > totalPoints(teamWithMostPoints))
            teamWithMostPoints = aLeague.teams[i];
    }
    return teamWithMostPoints;
}
```

Does it make sense ? There is one small issue though. Just like we need to begin our apple checking by picking up a first apple, we also need to pick a team (any **Team** object) to be the "best" one before we start the search. Currently the **teamWithMostPoints** starts off at **null** so we need to set this to a valid **Team** so start off. We can perhaps take the first **Team** in the **teams** array:

```
public static Team firstPlaceTeam(League aLeague) {
    Team    teamWithMostPoints = teams[0];

    for (int i=0; i<aLeague.teams.length; i++) {
        if (totalPoints(aLeague.teams[i]) > totalPoints(teamWithMostPoints))
            teamWithMostPoints = aLeague.teams[i];
    }
    return teamWithMostPoints;
}
```

We are not done yet! It is possible, in a weird scenario, that there are no teams in the league! In this case **teams[0]** will return **null** and we will get a **NullPointerException** again when we go to ask for the **totalPoints()**. So, we would need to add a special case to return **null** if the **teams** list is empty. Here is the new code …

```
public static Team firstPlaceTeam(League aLeague) {
    Team    teamWithMostPoints = aLeague.teams[0];

    if (aLeague.teams[0] == null)
        return null;

    for (int i=0; i<aLeague.teams.length; i++) {
        if (totalPoints(aLeague.teams[i]) > totalPoints(teamWithMostPoints))
            teamWithMostPoints = aLeague.teams[i];
    }
    return teamWithMostPoints;
}
```

What would we change in the above code if we wanted to write a method called **lastPlaceTeam()** that returned the team with the least number of points ? Try to do it.

For the purpose of a summary, here is the entire **League** class as we have defined it:

```
public class LeagueTestProgram {

   // Determine the number of games played by the given team
   public static int gamesPlayed(Team t) {
      return (t.wins + t.losses + t.ties);
   }

   // Determine the number of points that the given team has
   public static int totalPoints(Team t) {
      return (t.wins * 2 + t.ties);
   }

   // Displays a league's teams to the system console
   public static void displayTeams(League aLeague) {
      for (int i=0; i<aLeague.teams.length; i++)
         System.out.println(aLeague.teams[i]);
   }

   // Record a win and a loss between the given Teams
   public static void recordWinAndLoss(Team winner, Team loser) {
      if ((winner != null) && (loser != null)) {
         winner.wins++;
         loser.losses++;
      }
   }

   // Record a tie between the given Teams
   public static void recordTie(Team teamA, Team teamB) {
      if ((teamA != null) && (teamB != null)) {
         teamA.ties++;
         teamB.ties++;
      }
   }

   // Record a win and a loss between the teams in the League with the given names
   public static void recordWinAndLoss(League aLeague, String nmW, String nmL) {
      recordWinAndLoss(teamWithName(aLeague, nmW), teamWithName(aLeague, nmL));
   }
```

```
// Record a tie between the teams in the League with the given names
public static void recordTie(League aLeague, String nameA, String nameB) {
   recordTie(teamWithName(aLeague, nameA), teamWithName(aLeague, nameB));
}

// Find a Team with the given name in the given League
public static Team teamWithName(League aLeague, String nameToLookFor) {
   for (int i=0; i<aLeague.teams.length; i++) {
      if (aLeague.teams[i].name.equals(nameToLookFor))
         return aLeague.teams[i];
   }
   return null;
}

// Determine the total number of games played in the league
public static int totalGamesPlayed(League aLeague) {
   int  total = 0;

   for (int i=0; i<aLeague.teams.length; i++)
      total += gamesPlayed(aLeague.teams[i]);

   return total/2;  // Divide by two since a game is counted by both teams
}

// Return the team that has the most points
public static Team firstPlaceTeam(League aLeague) {
   Team    teamWithMostPoints = aLeague.teams[0];

   if (aLeague.teams[0] == null)
      return null;

   for (int i=0; i<aLeague.teams.length; i++) {
      if (totalPoints(aLeague.teams[i]) > totalPoints(teamWithMostPoints))
         teamWithMostPoints = aLeague.teams[i];
      }
      return teamWithMostPoints;
   }

// Return the team that has the least points
public static Team lastPlaceTeam(League aLeague) {
   Team    teamWithLeastPoints = aLeague.teams[0];

   if (aLeague.teams[0] == null)
      return null;

   for (int i=0; i<aLeague.teams.length; i++) {
      if (totalPoints(aLeague.teams[i]) < totalPoints(teamWithLeastPoints))
         teamWithLeastPoints = aLeague.teams[i];
   }
   return teamWithLeastPoints;
}
```

```
      public static void main(String args[]) {
         League  nhl = new League("NHL");

         //Add a pile of teams to the league
         nhl.teams[0] = new Team("Ottawa Senators");
         nhl.teams[1] = new Team("Montreal Canadians");
         nhl.teams[2] = new Team("Toronto Maple Leafs");
         nhl.teams[3] = new Team("Vancouver Canucks");
         nhl.teams[4] = new Team("Edmonton Oilers");
         nhl.teams[5] = new Team("Washington Capitals");
         nhl.teams[6] = new Team("New Jersey Devils");
         nhl.teams[7] = new Team("Detroit Red Wings");

         // Now we will record some games
         recordWinAndLoss(nhl, "Ottawa Senators", "New Jersey Devils");
         recordWinAndLoss(nhl, "Edmonton Oilers", "Montreal Canadians");
         recordTie(nhl, "Ottawa Senators", "Detroit Red Wings");
         recordWinAndLoss(nhl, "Montreal Canadians", "Washington Capitals");
         recordWinAndLoss(nhl, "Ottawa Senators", "Edmonton Oilers");
         recordTie(nhl, "Washington Capitals", "Edmonton Oilers");
         recordTie(nhl, "Detroit Red Wings", "New Jersey Devils");
         recordWinAndLoss(nhl, "Vancouver Canucks", "Toronto Maple Leafs");
         recordWinAndLoss(nhl, "Toronto Maple Leafs", "Edmonton Oilers");
         recordWinAndLoss(nhl, "New Jersey Devils", "Detroit Red Wings");

         // This one will not work
         recordWinAndLoss(nhl, "Mark's Team", "Detroit Red Wings");

         // Now display the teams again and some statistics
         System.out.println("\nHere are the teams after recording the " +
                            "wins, losses and ties:\n");
         displayTeams(nhl);

         System.out.println("\nThe total number of games played is " +
                             totalGamesPlayed(nhl));
         System.out.println("The first place team is " + firstPlaceTeam(nhl));
         System.out.println("The last place team is " + lastPlaceTeam(nhl));
   }
}
```

Here would be the output (make sure that it makes sense to you) …

```
Here are the teams after recording the wins, losses and ties:

The Ottawa Senators have 2 wins, 0 losses and 1 ties.
The Montreal Canadians have 1 wins, 1 losses and 0 ties.
The Toronto Maple Leafs have 1 wins, 1 losses and 0 ties.
The Vancouver Canucks have 1 wins, 0 losses and 0 ties.
The Edmonton Oilers have 1 wins, 2 losses and 1 ties.
The Washington Capitals have 0 wins, 1 losses and 1 ties.
The New Jersey Devils have 1 wins, 1 losses and 1 ties.
The Detroit Red Wings have 0 wins, 1 losses and 2 ties.

The total number of games played is 10
The first place team is The Ottawa Senators have 2 wins, 0 losses and 1 ties.
The last place team is The Washington Capitals have 0 wins, 1 losses and 1 ties.
```