

---

## Chapter 2

# Variables and Objects

---

### What is in This Chapter ?

In this chapter we will consider the notion of **variables** as a means of storing information. Variables are fundamental building blocks of object-oriented programming since in order to fully understand object-oriented programming, you must know *where* your data is at all times as well as *how to get it* and *how to change it*. We will look at examples of how to use variables to store primitive information as well as object information. You will learn how to define your *own* objects that contain their own variables called **instance variables**. We will then look at how to create **constructors** that allow us to provide initial values to all of our object's instance variables. Finally, we will examine shared data in the form of **class/static variables**.



## 2.1 Variables

Consider a piece of code in which we are given some numbers and we would like to compute some calculations with those numbers such as adding, multiplying and averaging. We could write the following code:

```
System.out.println("Given numbers 34, 89 and 17: ");
System.out.println("The sum is " + (34 + 89 + 17));
System.out.println("The product is " + (34 * 89 * 17));
System.out.println("The average is " + ((34 + 89 + 17)/3));
```

The code is straight forward. JAVA will perform the calculation and then print the results. If we wanted to change one of the numbers (e.g., change 34 to 15) then we would have to make this change in 4 places:

```
System.out.println("Given numbers 15, 89 and 17: ");
System.out.println("The sum is " + (15 + 89 + 17));
System.out.println("The product is " + (15 * 89 * 17));
System.out.println("The average is " + ((15 + 89 + 17)/3));
```

This is not a desirable situation. In this particular code it is a simple change, however in larger programs, this may require you to go through hundreds of lines of code to make the simple change.

Another undesirable characteristic of the program is that the sum (15 + 89 + 17) is computed twice:

```
System.out.println("Given numbers 15, 89 and 17: ");
System.out.println("The sum is " + (15 + 89 + 17));
System.out.println("The product is " + (15 * 89 * 17));
System.out.println("The average is " + ((15 + 89 + 17)/3));
```

Again, this is not so serious in a small program, but in a larger program we would not want to duplicate portions of our code in many places because:

- it wastes space (i.e., computer memory)
- it requires more programming time to write the same code over and over again
- if a change needs to be made, we must remember every place that the duplicated code appears and then go make the change multiple times.

In order to avoid these problems, we can use the notion of a **variable**.

Simply put, a variable is a **placeholder for a piece of information**. That is, a variable is used to “hold on to” or remember a data value for later use.

In real life, we often “jot down” something on a piece of paper to remember it later (e.g., a phone number, an address, a price etc...). That way, whenever we want to recall the value that we wrote down, we just get that piece of paper and read the value that we wrote there.

So when programming, we use a variable to store information for later use (i.e., for reference later).

But what if you have written down a few phone numbers? How do you tell which one is which?

We need something that will let us distinguish between the different phone numbers. It should be something unique ... shouldn't it?

In real life, we would probably also write the name of each person along with the phone number. As long as the names are unique, we will know exactly which phone number belongs to each person.

When programming, variables also have a name. This allows us to refer to the variable's value at any time in our program.

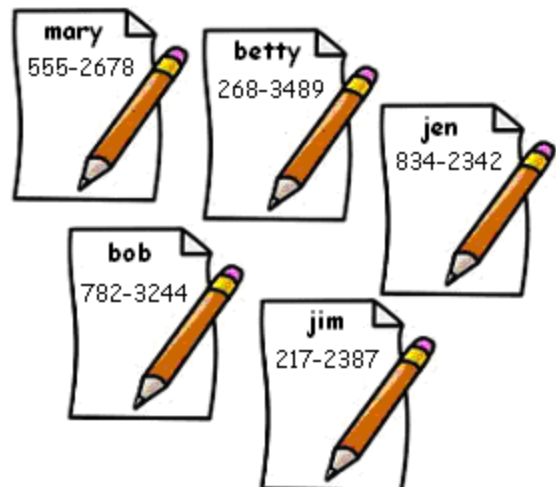
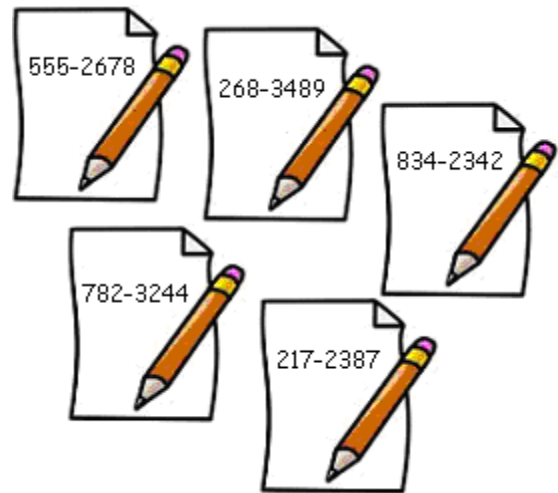
Thus, when we use the **variable name** in our program, we are actually referring to the **value** stored in the variable with that name.

We know that a phone number is a sequence of digit characters with a '-' character. We can thus represent a phone number using a string as follows: **“555-2678”**. In this case, we would say that phone numbers can be stored in variables that are of **type String**. Hence, **String** would be the **data type** for our individual phone numbers.

So, in JAVA, to make a variable that holds (i.e., stores, keeps, remembers, etc.) a phone number, we need to specify both the **type** and **name** (must be unique) of the variable. Here is how we would do this for our 5 phone numbers above:

```
String    mary;
String    betty;
String    jen;
String    bob;
String    jim;
```

Notice that we specify the **type** on the left and the **name** on the right (separated by one or more spaces or tab characters) and then conclude with a **‘;’** character. It should be noted



however, that the variable definitions above do not specify the value for the variables, it just “reserves space” for the variable (i.e., it just makes the “sheet of paper” with the label on it, but does not yet have a value).

Make sure to pick **meaningful** variable names that are not too long !! The name must be unique and it is case-sensitive (i.e., **Hello** and **hello** would not be considered the same).

Variable names may contain only letters, digits and the ‘\_’ character (i.e., no spaces in the name). As standard convention, multiple word names should have every word capitalized (except the first). Here are some good examples of variable names:

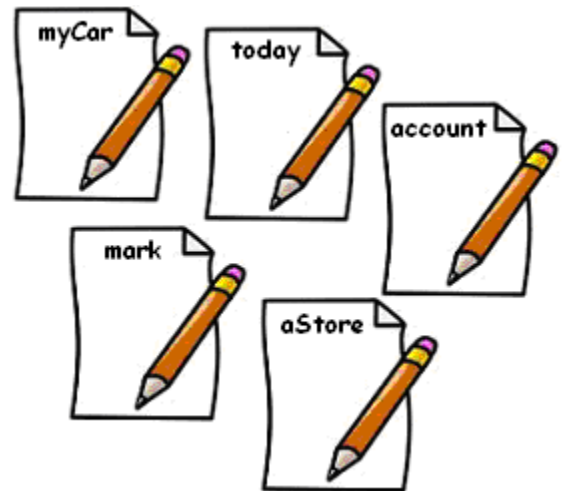
- count
- average
- insuranceRate
- timeOfDay
- poundsPerSquareInch
- aString
- latestAccountNumber
- weightInKilograms

Here are some more examples of how we can declare some additional variables for various kinds of objects:

```
Car           myCar;
Date          today;
Person        mark;
Store         aStore;
BankAccount   account;
```

Note that we can also declare variables whose type is one of the 8 data types. For example:

```
int           days;
float         age;
double        weight;
char          sex;
boolean       hungry;
```













Now that we know how to **declare** (i.e., *define*) a variable, we need to know how to use them in our programs. We use the term **assign** to represent the idea of “giving a value” to a variable. In JAVA, the *assignment operator* is the = sign. So, we use = to put a value into a variable. Here are a few examples of how we can do this with some of the variables that we declared earlier:

```
mary = "555-2678";
bob = "782-3244";
myCar = new Car();
mark = new Person();

days = 15;
age = 21.3f;
weight = 165.23;
sex = 'M';
hungry = true;
```

Something VERY important to remember when learning to program is that the value of the variable **must be the same type** of object (or primitive) **as the variable's type** that was specified when you declared it earlier. So for example, in the following table, make sure that you understand why the examples on the left are wrong, while the examples of the right are correct:

 String mary; mary = 23;	 String mary; mary = "555-2678";
 Car myCar; myCar = "Porsche";	 Car myCar; myCar = <b>new</b> Car();
 int days; days = 10.2789;	 int days; days = 10;
 boolean hungry; hungry = 'y';	 boolean hungry; hungry = <b>false</b> ;
 char sex; sex = "F";	 char sex; sex = <b>'F'</b> ;

To help cut down the number of lines of code in our program, JAVA actually allows us to both **declare** and **assign** a value to our variables all on one line. So, from our earlier examples, we can do the following:

```
String    mary = "555-2678";
String    bob  = "782-3244";
Car       myCar = new Car();
Person    mark = new Person();
int      days = 15;
float    age  = 21.3f;
double   weight = 165.23;
char     sex  = 'M';
boolean  hungry = true;
```

It is interesting that we use the term **variable** to store our information. The word “variable” is derived from the word “**vary**”, which means that the value may change (or vary) as time goes on. For example, if our friend changes their phone number, we can simply erase the old number from our piece of paper and write in the new number. Thus, the **value** of the variable will have changed.

So, even though a variable may be **declared** only once in the program, we may **assign** a value to it multiple times. Can you determine the output of this piece of code:

```
int    days;

System.out.println(days);
days = 43;
System.out.println(days);
days = 15;
System.out.println(days);
```

There are some interesting aspects to take note of in this code:

- the code prints out numbers 0, 43 and 15 (notice that **days** starts off with a value of **0**)
- the code prints out the **days** variable simply by using the variable's name
- even though the code displays the same **days** variable 3 times, its value is different because it is re-assigned a value 2 times during the program.

Variables can be **re-assigned** a value, but cannot be **declared** again. Therefore, the following code will not compile:



```
int days = 365;
System.out.println(days);
int days = 7;           // cannot declare days again
System.out.println(days);
```

Here are some more pieces of code. Do you know what the output is ?

```
int x;
int y;
x = 34;
y = 23;
System.out.println(x + y);
```

Here is a similar example. Notice in JAVA that we are allowed to declare multiple variables of the same type on the same line, each separated by a **','**:

```
int x, y;
x = 34;
y = x;
System.out.println(x + y);
```

Here is another one:

```
int x, y, z;

x = 3*2*1;
y = x + x;
z = x;
System.out.println(z);
```

Note that even though we use **x** a few times, it does not change its value. Here is one that is a little more interesting:

```
int    total;
float  average;

total = 12 + 25 + 36 + 15;
average = total / 4;
System.out.println("The average is " + average);
```

Notice that we can append the variable to a String using the '+' before printing it. JAVA simply takes the value of the variable and appends it to the end of the String.

The result is **22.0**, not just **22**. That is because average is declared as type **float**. If we would have declared it as type **int**, the answer would have been **22**.

Here is an example using **char** variables. Can you guess the output ?

```
char a='M', b='A', c='R', d='K';

System.out.println("My name is: " + a + b + c + d);
System.out.println("My name backwards is: " + d + c + b + a);

b = 'E';
c = 'I';

System.out.println("The mystery name is: " + a + c + d + b);
```

Here is the output:

```
My name is: MARK
My name backwards is: KRAM
The mystery name is: MIKE
```

Don't forget ... variables are used to hold whole objects and they too are used exactly the same way as variables that hold primitives:

```
Date today = new java.util.Date();
System.out.println("Date of Birth = " + today);
```

Now let us consider a more complete example that uses the **Scanner** object to get three numbers from the user and then computes the sum, product and average. You will notice that code uses variables named **a**, **b**, **c** and **sum** which all hold integer values. Notice also how a single **Scanner** object is used now since we can store it in a variable:

```
import java.util.Scanner;

class SuperCalculatorProgram {
    public static void main(String args[]) {
        int      a, b, c, sum;
        Scanner  keyboard = new Scanner(System.in);

        System.out.println("Enter three numbers:");
        a = keyboard.nextInt();
        b = keyboard.nextInt();
        c = keyboard.nextInt();

        sum = a + b + c;    //compute the sum and store it
        System.out.println("Given numbers " + a + ", " + b +
            " and " + c + ":");
        System.out.println("The sum is " + sum);
        System.out.println("The product is " + (a * b * c));
        System.out.println("The average is " + (sum / 3));
    }
}
```

Here is the output (making sure to disable the "Capture Output" option in JCreator):

```
Enter three numbers:
34
88
16
Given numbers 34, 88 and 16:
The sum is 138
The product is 47872
The average is 46
```

Notice now that by simply changing the first integer to 64, we get different output:

```
Enter three numbers:
64
88
16
Given numbers 64, 88 and 16:
The sum is 168
The product is 90112
The average is 56
```

At this point, you should have a good understanding of what a variable is and how to use them. The only question you may have is "When do I need to use a variable?".



In summary, we need to use a variable:

- to store intermediate results in an extensive computation
- whenever we need to use a value more than once in a computation
- to simplify steps in a piece of code.
- to reduce code duplication

Other than that, there is not much more to say about variables. You just need to start getting practice using them.

## 2.2 Instance Variables / Object Attributes

We have just discussed the use of simple variables in our programs. Recall that all variables have a **name** and a **type** and that the type must either be one of the 8 primitives or an object of our choosing.

Remember too that an object is really just a bunch of data grouped together. The data (i.e., information) itself defines the particular **attributes** of the object. Just as we use variables to “remember” data in our programs, an object must “remember” its data at all times. These attributes define the **state** of the object as time goes on.



The state of the object is stored (i.e., remembered) by using variables. Therefore, at the most basic level, an object is simply made up of a group of variables. Since an object is an **instance** of some class, we call these particular kinds of variables **instance variables**.

Consider a variable that stores a person’s name:

```
String    name;           // declare a variable to hold the name

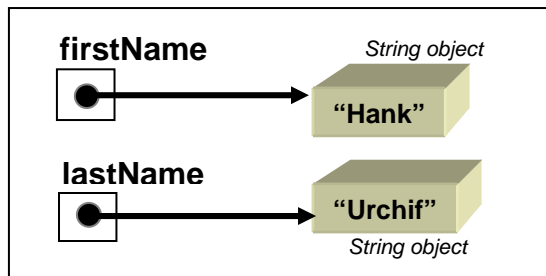
name = "Hank Urchif";
System.out.println(name);
```

The code above displays “**Hank Urchif**” to the console window. Very simple. Suppose that we wanted the option of displaying the name as either “**Hank Urchif**” or “**Urchif, Hank**”. To do this, we would need to distinguish between the first and last name by using two variables as follows:

```
String    firstName, lastName; // declare 2 variables to hold the name

firstName = "Hank";
lastName = "Urchif";
System.out.println(firstName + " " + lastName);
System.out.println(lastName + ", " + firstName);
```

The code above displays both “**Hank Urchif**” and “**Urchif, Hank**” to the console window. Here is a corresponding diagram showing that each of the two **String** variables are pointing to (i.e., holding on to) their own unique **String** objects:



What would we do though if we had three person's names? Likely, we would do something like this:

```
String    firstName1, lastName1; // variables to hold the 1st person's name
String    firstName2, lastName2; // variables to hold the 2nd person's name
String    firstName3, lastName3; // variables to hold the 3rd person's name

firstName1 = "Hank";
lastName1  = "Urchif";
firstName2 = "Holly";
lastName2  = "Day";
firstName3 = "Bobby";
lastName3  = "Socks";
System.out.println(lastName1 + ", " + firstName1);
System.out.println(lastName2 + ", " + firstName2);
System.out.println(lastName3 + ", " + firstName3);
```

This code would produce the following output:

```
Urchif, Hank
Day, Holly
Socks, Bobby
```

You can see however, that with just a few more people, the number of variables that you need can increase quickly. A more convenient way of writing this code would be to group the **firstName** and **lastName** into one object (since they always belong together anyway).

We can group these two variables together by defining a new object (i.e., remember that an object is just variables with an elastic around them). What would be a good name for the object? It makes sense to call it something like **Person**. Here is how we do this ...

```

class Person {
    String    firstName;    // instance var to hold a person's first name
    String    lastName;    // instance var to hold a person's last name
}

```

Notice that the **Person** class defines one single person, not multiple people (that is why we called the class **Person** and not **Persons**). Notice as well that all we did is move the two variables inside this object. Once we do this and compile our **Person** class, JAVA now knows that all **Person** objects are made up of a first name and a last name. That is, whenever we make a new **Person** object, they will each have their own **firstName** and **lastName** as part of them. Recall, that these two variables are now called the *instance variables* (or *attributes*) of the **Person** class.

Assuming that we have defined, saved and compiled the **Person** class above, now we can simplify our program to use the new **Person** object as shown below. Note that the original code is included as a comment so that you can compare them side-by-side:

```

Person    aPerson;    // declare the variable that will hold the person

aPerson = new Person();
aPerson.firstName = "Hank";
aPerson.lastName = "Urchif";
System.out.println(aPerson.firstName + " " + aPerson.lastName);
System.out.println(aPerson.lastName + ", " + aPerson.firstName);

/* Here was the original code:
String    firstName, lastName;

    firstName = "Hank";
    lastName = "Urchif";
    System.out.println(firstName + " " + lastName);
    System.out.println(lastName + ", " + firstName);*/

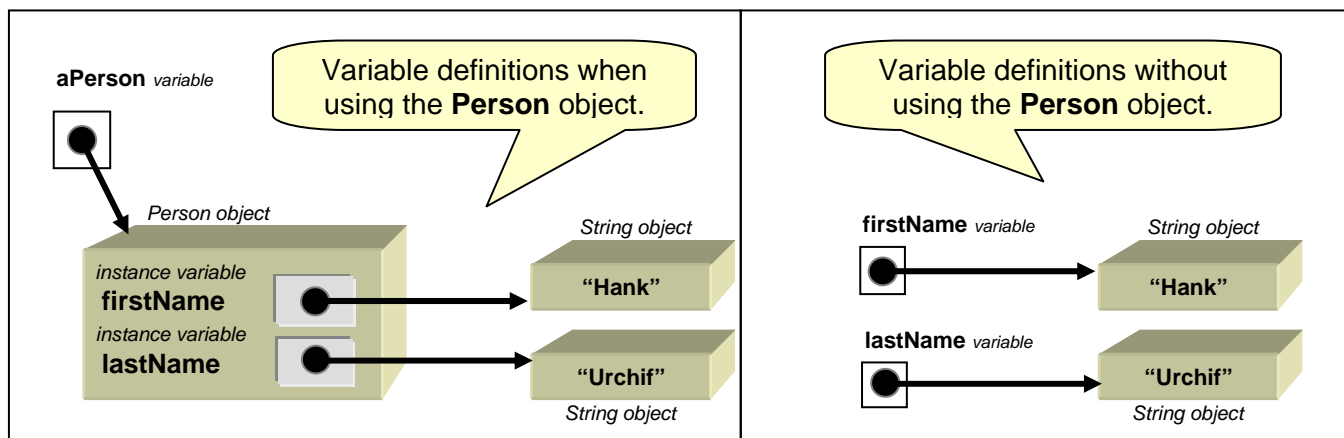
```

Notice that we now have one variable (called **aPerson**) which is of type **Person** now instead of type **String**. That is, the **aPerson** variable will now hold a whole **Person** object (which includes the **firstName** and **lastName** within it).

Now although the type of the variable MUST be **Person**, the actual name of the variable is unimportant. We could have used any variable name such as **p**, **x**, **hank**, **friend**, **man**, **customer**, etc...

When we define the variable on the first line, it does not create any **Person** objects. It simply **declares** that we will be using (and storing) a **Person** object in the program. So, we actually need to create a new **Person** object on our own and store it in the variable (see the 2<sup>nd</sup> line).

Here is a diagram showing what has now happened. Notice that the **firstName** and **lastName** variables hold **String** objects. That is, the names themselves are unique objects (hence drawn as separate objects here). We therefore draw the arrows to indicate that the **firstName** and **lastName** variables are just *pointing to* (or *referring to*) the two **String** objects. Notice in the diagram below, that the two String variables are still there, they are just inside the **Person** object now:



Notice as well in our program that we use the dot `.` character after the `aPerson` variable name in order to get inside of it. After the dot we specify which of the two variables we are trying to refer to (i.e., either **firstName** or **lastName**). Otherwise, the code is the same as before. That is, we use the instance variables inside the object just as if they were regular ordinary **String** variables.

You should realize that all instance variables are allowed to be changed at any time (i.e., they can **vary**). For example, assume that Hank watches a "Simpsons" episode on TV and then decides to change his name to "Max Power".

```
Person    aPerson;

// Make Hank with his original name
aPerson = new Person();
aPerson.firstName = "Hank";
aPerson.lastName = "Urchif";

// Assume that Hank watches The Simpsons and decides to make the change

aPerson.firstName = "Max";           // Change the first name to "Max"
aPerson.lastName = "Power";         // Change the last name to "Power"
System.out.println(aPerson.firstName + " " + aPerson.lastName);
```

The output will show the new name "Max Power". It is important that you do not forget that instance variables are just regular variables ... but they are inside the object now.

Now what about our example that used 3 names ? We can make 3 separate **Person** objects as follows:

```

Person    p1, p2, p3;    // declare the three variables

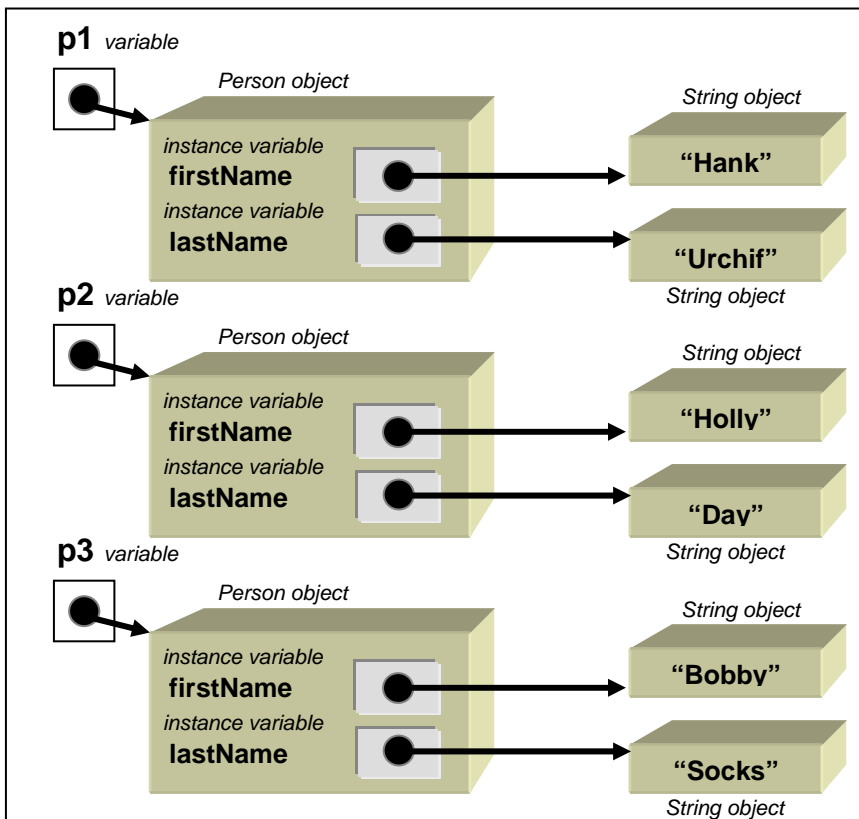
// Create the 3 people (i.e., make 3 instances of the Person class)
p1 = new Person();
p2 = new Person();
p3 = new Person();

// Set all of their names (i.e., give values to their instance variables)
p1.firstName = "Hank";
p1.lastName = "Urchif";
p2.firstName = "Holly";
p2.lastName = "Day";
p3.firstName = "Bobby";
p3.lastName = "Socks";

// Display the names to confirm that the names were set correctly
System.out.println(p1.lastName + ", " + p1.firstName);
System.out.println(p2.lastName + ", " + p2.firstName);
System.out.println(p3.lastName + ", " + p3.firstName);

```

Notice that the diagram would appear as follows:



So as you can see, objects are used to hold a group of data together so that your program code stays more organized and relates better to real world objects.

Suppose that we want to add some more interesting attributes to the **Person** object to keep track of a person's age, gender and whether or not they are retired. We can do this by adding 3 more instance variables as follows:

```
class Person {
    // These are the instance variables that define a Person object
    String    firstName;    // person's first name
    String    lastName;    // person's last name
    int       age;         // person's age
    char      gender;      // person's gender (i.e., 'M' or 'F')
    boolean   retired;     // person's retirement status
}
```

Note that the order of the instance variables does not matter. It is wise though to keep them all together like this. Here is how we can set them and then print them out to test if it works:

```
Person    p1, p2;    // declare the two variables

// Create the first person and set his instance variables
p1 = new Person();
p1.firstName = "Hank";
p1.lastName = "Urchif";
p1.age = 19;
p1.gender = 'M';
p1.retired = false;

// Create the second person and set her instance variables
p2 = new Person();
p2.firstName = "Holly";
p2.lastName = "Day";
p2.age = 67;
p2.gender = 'F';
p2.retired = true;

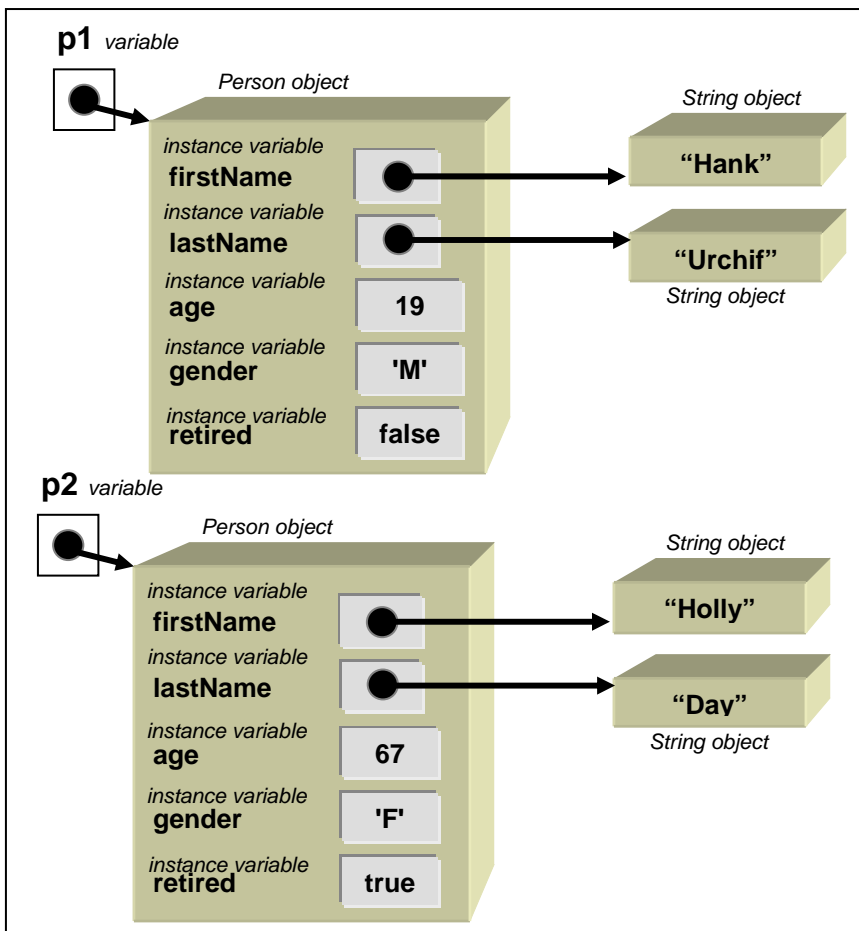
// Display the instance variables for these people
System.out.println(p1.lastName + "," + p1.firstName + "," +
    p1.age + "," + p1.gender + "," + p1.retired);
System.out.println(p2.lastName + "," + p2.firstName + "," +
    p2.age + "," + p2.gender + "," + p2.retired);
```

Here is the output that you should expect to see:

```
Urchif,Hank,19,M,false
Day,Holly,67,F,true
```

Do you now understand why we use the term “**instance variable**”? Recall that every time we use **new** to get a new object (e.g., **new Person()**), we get back a new **instance** of that class. Thus, the data that defines the object (e.g., first name, last name, age, etc..) will **vary** from object to object, that is, from instance to instance.

Here is the diagram for the above example. Notice that the **age**, **gender** and **retired** are *primitive* types and so they each store their literal value directly, whereas **firstName** and **lastName** just stored “*pointers*” to the actual **String** objects that they store:



## Supplemental Information (Choosing Instance Variables)

In the above example, we chose **firstName**, **lastName**, **age**, **gender** and **retired** as attributes for our **Person** object. Why did we choose these? We chose them as an example of how to build an object. In reality however, the application that we are trying to develop may require us to keep information for a **Person** that is different from the attributes we chose here. For example, we might want to keep their **emailAddress**, **weight**, **height**, **favoriteColor**, etc.. The choice depends on the application and on what we need to store for this object.

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



The choice will depend on the program/application you are making. Consider these possible applications in which a **Car** object 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 lets 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, insurance type (fire/theft/collision/liability), color, license plate, etc...

So you can see that it is not always straight forward to identify the state for an object. You need to always understand **how it fits** into the application.



So far, we have looked at storing just **String** and primitive values in the variables within our object. We should realize however, that since variables can store any kind of object, then our instance variables too can store arbitrary objects. That is, the objects that we make may themselves be made up of other objects.

For example assume that we want to store an address as well for our person. We can define the following **Address** object to maintain all the appropriate information for an address:

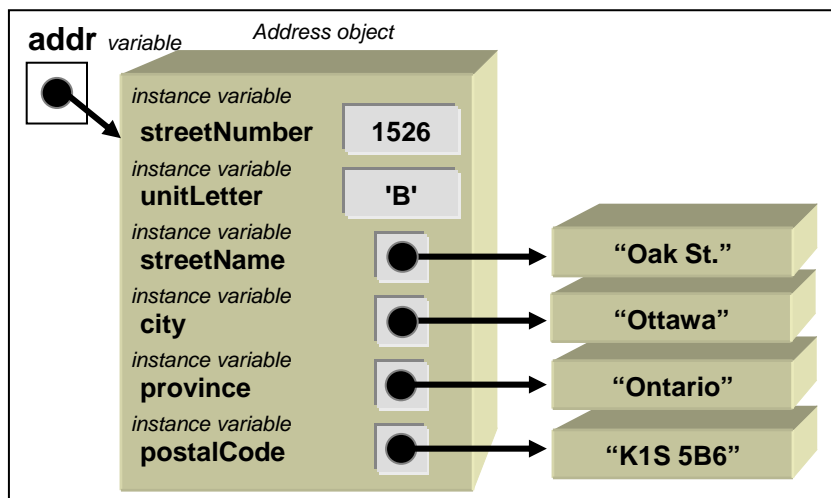
```
class Address {
    // These are the instance variables that define an Address object
    int    streetNumber;    // address's street number
    char   unitLetter;     // address's unit letter (e.g., 'A');
    String streetName;     // address's street name
    String city;           // address's city
    String province;      // address's province
    String postalCode;    // address's postal code
}
```

Assume that this class was compiled and saved to a local directory, we could create a new **Address** as follows:

```
Address    addr;        // declare the variable that will hold the address

// create an Address object and then set its values
addr = new Address();
addr.streetNumber = 1526;
addr.unitLetter = 'B';
addr.streetName = "Oak St.";
addr.city = "Ottawa";
addr.province = "Ontario";
addr.postalCode = "K1S 5B6";
```

Notice the diagram that would represent this object and its composite instance variable values:



Now that the **Address** object has been defined, we can add an instance variable to our **Person** class that keeps track of a person's address by using this new **Address** object as follows:

```
class Person {
    // These are the instance variables that define a Person object
    String    firstName;    // person's first name
    String    lastName;    // person's last name
    int       age;         // person's age
    char      gender;      // person's gender (i.e., 'M' or 'F')
    boolean   retired;     // person's retirement status
    Address   address;     // person's address
}
```

Here is how we can try this out to see if it all works:

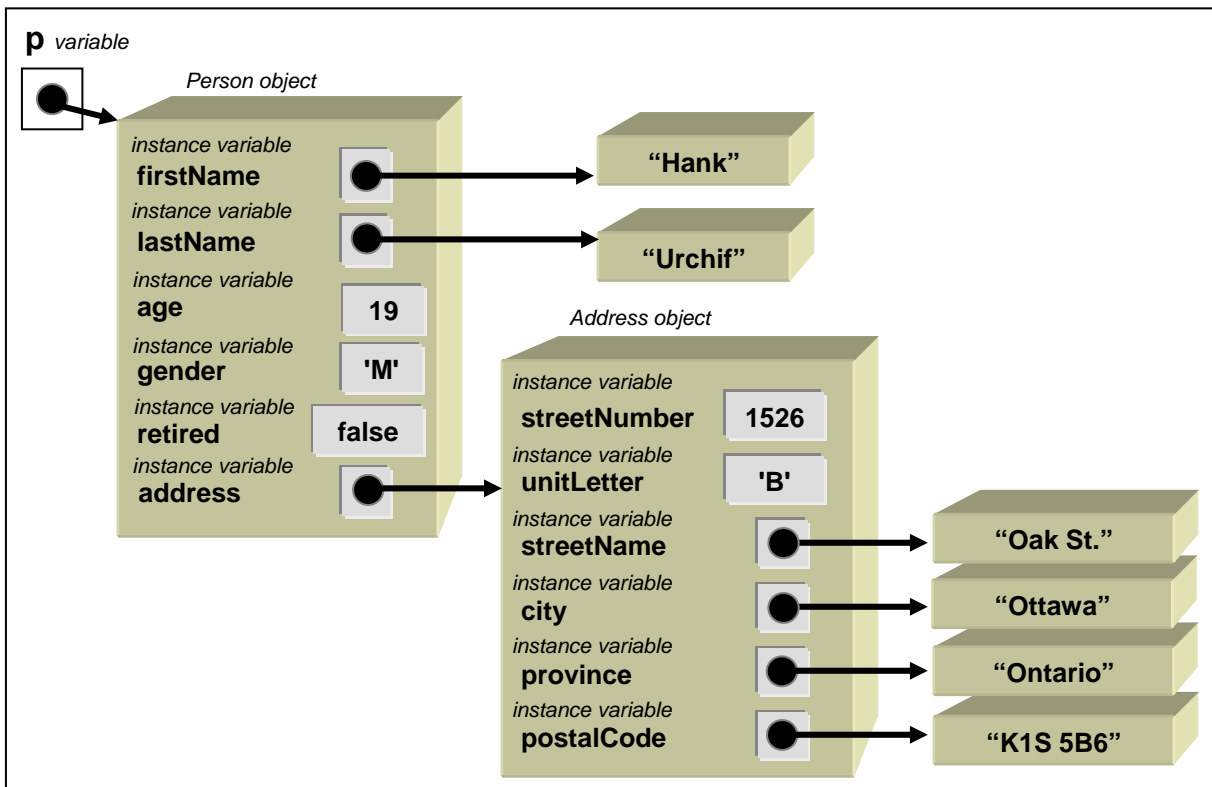
```
Person    p;    // declare a Person variable

// Create the person and set his instance variables
p = new Person();
p.firstName = "Hank";
p.lastName = "Urchif";
p.age = 19;
p.gender = 'M';
p.retired = false;

p.address = new Address();    // we need to create an address object
p.address.streetNumber = 1526;
p.address.unitLetter = 'B';
p.address.streetName = "Oak St.";
p.address.city = "Ottawa";
p.address.province = "Ontario";
p.address.postalCode = "K1S 5B6";

// Display some instance variables from the address
System.out.println(p.address.streetNumber);    // displays 1526
System.out.println(p.address.city);    // displays Ottawa
```

You will notice that the person's **address** variable stores a new **Address** object (which we had to create). Then, we simply set the address attributes by using an extra dot **.** character to get inside the address object to change its internal values. Here is a diagram of what is happening ...



Lets go one more level of objects. Suppose that we have a **BankAccount** object that keeps track of a person's bank account information. We can define it as follows:

```
class BankAccount {
    // These are the instance variables that define a BankAccount object
    Person    owner;           // person who owns the account
    int       accountNumber;   // the account number
    float     balance;        // amount of money currently in the account
}
```

Notice that the owner is of type **Person**, which will contain all the owner's information (i.e., name, address, phone number, age, etc...). We can test it as follows:

```
BankAccount    account;    // declare a BankAccount variable

// Create the bank account and set its instance variables
account = new BankAccount();
account.accountNumber = 178193;    // arbitrarily chosen
account.balance = 100;            // new account with $100.00

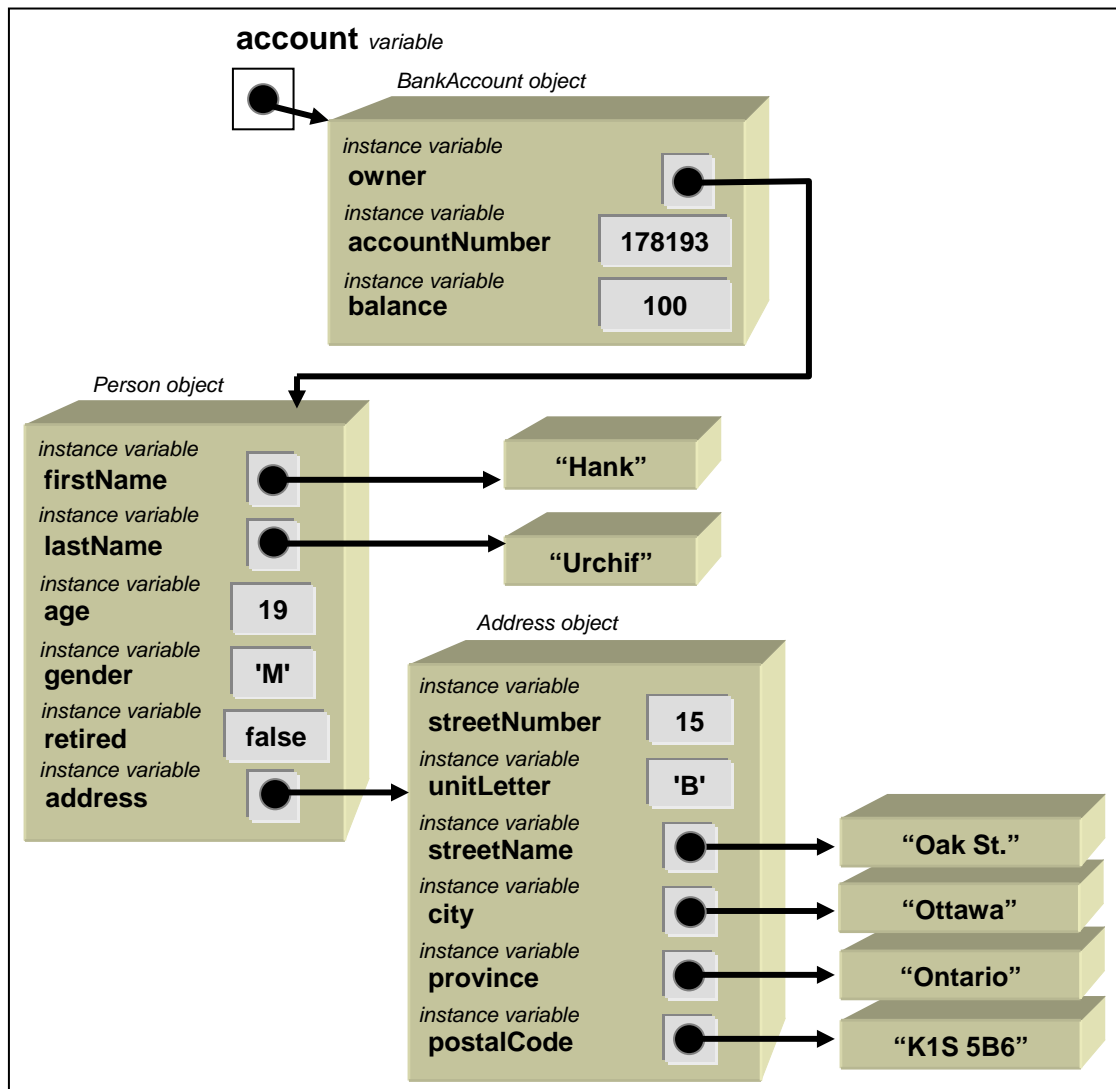
account.owner = new Person();    // must create a new Person object
account.owner.firstName = "Hank";
account.owner.lastName = "Urchif";
account.owner.age = 19;
account.owner.gender = 'M';
account.owner.retired = false;
```

```

account.owner.address = new Address(); // must create an Address object too
account.owner.address.streetNumber = 1526;
account.owner.address.unitLetter = 'B';
account.owner.address.streetName = "Oak St.";
account.owner.address.city = "Ottawa";
account.owner.address.province = "Ontario";
account.owner.address.postalCode = "K1S 5B6";

```

Here is the diagram showing how all of the data is stored. Make sure that it makes sense to you because a fundamental part of programming in an object-oriented language involves “*understanding where the data is*” and “*how to get to it*”:



Also, in regards to this example, we can actually simplify things a little by using intermediate variables. Note that the following code does the same thing, but seems a bit simpler:

```

BankAccount    account;    // declare a BankAccount variable
Person         p;          // temporary variable that refers to the Person
Address        adr;        // temporary variable that refers to the Address

// Create the bank account and set its instance variables
account = new BankAccount();
account.accountNumber = 178193;
account.balance = 100;

// Store the new Person object in a temporary variable so that we can refer
// to the person using just p in the code below instead of account.owner
p = new Person();
account.owner = p;
p.firstName = "Hank";
p.lastName = "Urchif";
p.age = 19;
p.gender = 'M';
p.retired = false;

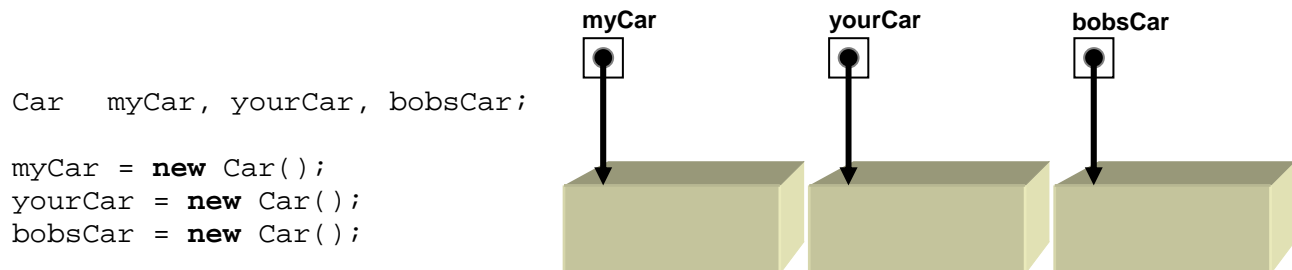
// Store the new Address object in a temporary variable so that we can refer
// to the address using just adr in the code below instead of p.address
adr = new Address();
p.address = adr;
adr.streetNumber = 1526;
adr.unitLetter = 'B';
adr.streetName = "Oak St.";
adr.city = "Ottawa";
adr.province = "Ontario";
adr.postalCode = "K1S 5B6";

```

## Variable Bindings

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.



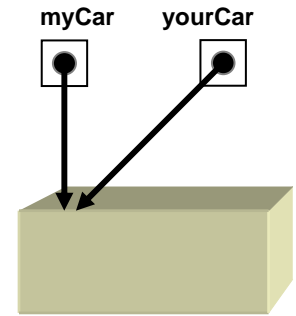
So in the above code, all three variables **point to** different/unique objects.

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;
```

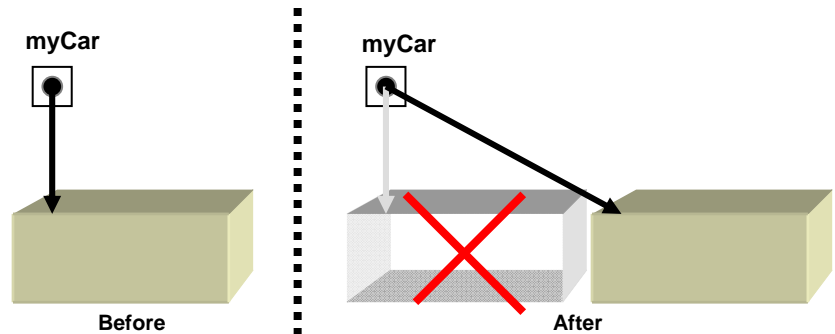


In this case, the two variables point to the exact same object.

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.

There is one more thing that you should understand with respect to variables that store objects. When we declare a variable of some object type, but do not assign it a value, the JAVA compiler will usually complain. For example, consider the following code:

```
Car myCar;
System.out.println(myCar);
```

This code will generate a compile error saying:

**variable myCar might not have been initialized**

What does this mean ? Well, when we declare a variable, it simply reserves space, but no object is actually created. Objects that have not been through the construction process are considered to be "undefined". The construction process occurs only when we use the **new** keyword (e.g., `new Car()`).

Sometimes we do not want to give a value right away to our variable. We are allowed to assign the value of **null** to any reference variable. The following code will satisfy the compiler:

```
Car myCar = null;
System.out.println(myCar);
```

The above example does not generate an error, and prints out **null** to the console. **Null** represents an **undefined object** and you should remember that if you declare a variable of some type of object but do not give it a value, then its value will always be **null**.

## Supplemental Information (The Garbage Collector)

Objects that have been created and are no longer being "pointed to" (or "referred to" from anywhere in your code) 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()`



## Object Summary

So in summary:

- objects must be **defined before we can use** them
- objects are each defined as their **own class**
- objects contain **instance variables** (or **attributes**) which are just regular "run-of-the-mill" variables that happen to store information about a particular object.
- objects may **contain other objects**
- we **access the "insides"** of an object by using the dot `.` operator followed by the name of the information/attribute/variable that we are trying to get
- we can **modify the "insides"** of an object by setting its instance variables to new values using the `=` operator.

## 2.3 Initializing Our Objects By Using Constructors

When testing our newly created object examples in the previous section (i.e., `Person`, `Address`, `BankAccount`), we explicitly assigned values to each instance variable and then printed them out to test whether or not it worked. Now, we will see an easier way to put some initial values into our objects using the notion of a **constructor**.

A *constructor* is a special chunk of code that we can write in our object classes that will allow us to **hide the ugliness** of setting all of the initial values for our objects each time we use them. The main advantage of making a constructor is that it will **allow us to reduce the amount of code that we need to write each time we make a new object**.

Consider the simple **Person** object that we defined previously:

```
class Person {
    String    firstName;
    String    lastName;
}
```

Recall the code that we wrote to try out our new object:

```
Person    p1, p2;

// Create the first person and set his instance variables
p1 = new Person();
p1.firstName = "Hank";
p1.lastName = "Urchif";

// Create the second person and set her instance variables
p2 = new Person();
p2.firstName = "Holly";
p2.lastName = "Day";
```

Notice that for each **Person**, we must write out the names of the instance variables each time so that we can assign them some initial values. This is tedious and annoying.

Recall that the code `new Car()` is responsible for *making* (or *constructing*) a **Car** object. The **Car()** portion of the code is actually called a **constructor**. We explained this earlier as if we were asking the **Car factory** to create (or construct) a **Car** object for us. Similarly, `new Person()` calls the **Person** class constructor that creates a new **Person** object that we can use in our program.

In real life however, when ordering something from a factory we usually specify some of the features that we would like our object to have (e.g., the car's make, model, color, other options,



etc...). In JAVA, we can do the same thing. Whenever we use a constructor in our code, we can supply some additional information pertaining to some of the **features** that we would like our new object to have.

In JAVA, whenever we use round brackets **()**, we are often allowed to supply some additional information in between the brackets. This additional information is known as **parameters**.

So, what we would **like to do** in our program is something like this:

```
Person    p1, p2;

p1 = new Person("Hank", "Urchif");
p2 = new Person("Holly", "Day");
```

This would be “wonderful” because it greatly reduces the amount of code that we would need to write. However, the code above will not compile because we first need to tell JAVA that we will be passing all of this new information as parameters to the constructor. That is, we need to specify exactly what information is being “passed in” (or supplied) as well as what to do with the information.

To make this work, we therefore need to go back to our **Person** class and write our *own* constructor. Normally we do this just below the list of instance variables. Below is the constructor that we need for our **Person** class:

```
class Person {
    String    firstName;
    String    lastName;

    // This is a constructor
    Person(String s1, String s2) {
        this.firstName = s1;
        this.lastName = s2;
    }
}
```

Notice that the constructor has the *same name* as the class itself (i.e., **Person**). Also notice that when *defining* a constructor we DO NOT use the **new** word anywhere. That is, **new** is only used to **call** (i.e., to use) a constructor.

Notice that some code appears between the round brackets **()**. We will look at this in a moment. You should also notice that there is an opening brace character **{** and later ... aligned underneath the constructor name is the closing brace **}** character. These braces define what is called the **body** of the constructor. The body represents all the code that is to be evaluated by JAVA whenever we call the constructor.

Notice the code inside the body. It is code that assigns values to the instance variables **firstName** and **lastName**. This code looks similar in format to what we were doing in our test code. Look at it side by side:

### TEST CODE:

```
p1.firstName = "Hank";  
p1.lastName = "Urchif";
```

### CONSTRUCTOR CODE:

```
this.firstName = s1;  
this.lastName = s2;
```

Let us examine this a bit more carefully. Notice that in both cases, we are setting the values for the **firstName** and **lastName** instance variables. Take note of the object that appears in front of the dot operator in each case. In the test code, **p1** represents the person whose name that we were trying to set. The word **this** in the constructor must therefore also represent that same person, otherwise the code would not work.

In fact, the word **this** is a special word in JAVA that is used in constructors (and in other places as we will see later) to represent the object that we are constructing. So, when we say:

```
p1 = new Person("Hank", "Urchif");
```

JAVA goes into our constructor to evaluate our code and at that time, the word **this** will represent (or refer to) the person **p1** that we constructed. Then when we say:

```
p2 = new Person("Holly", "Day");
```

JAVA goes into the constructor again, making a new object again and then the word **this** will represent (or refer to) the second person **p2** that we constructed.

So the word **this** is just a temporary “*nickname*” for the particular object that is being created and used in the constructor.

Notice what else differed from the test case and our constructor. The **firstName** and **lastName** are set to **s1** and **s2**, respectively. That means, **s1** must refer to “Hank” and **s2** must refer to “Urchif” in order for our code to work. This is indeed what is happening. To understand, we must look in-between the round brackets now.

Notice that the code between the brackets looks like two variable declarations separated by a comma. These are called the **parameters** of the constructor. A parameter actually **is** a variable ... a temporary one ... that can be used only within the constructor but not outside of it.

Just as the word **this** referred to the particular person that we were trying to set the names for, the strings **s1** and **s2** refer to the particular values that we want the **firstName** and **lastName** instance variables to be set to. Hence, when we say:

```
p1 = new Person("Hank", "Urchif");
```

then **s1** becomes a “*nickname*” for the incoming value “Hank” and **s2** becomes a “*nickname*” for the incoming value “Urchif”.

That is, in our constructor, whenever we refer to **s1**, we are actually referring to the string “Hank” that was passed in as a parameter. The actual name of these parameters is unimportant. We could have chosen any names. For example, we could have written the constructor in any of the following ways:


```
Person(String a, String b) {
    this.firstName = a;
    this.lastName = b;
}
```

```
Person(String name1, String name2) {
    this.firstName = name1;
    this.lastName = name2;
}
```


```
Person(String initialFirstName, String initialLastName) {
    this.firstName = initialFirstName;
    this.lastName = initialLastName;
}
```

It is important that the type of **s1** matches the type of **firstName** and that the type of **s2** matches the type of **lastName**. That is why we had to specify the type of **s1** and **s2** to be **String** parameters within the round brackets. So the following constructors would NOT be valid:

```
Person(int s1, int s2) { // WRONG TYPES!!!
    this.firstName = s1;
    this.lastName = s2;
}
```



```
Person(Address s1, Person s2) { // WRONG TYPES!!!
    this.firstName = s1;
    this.lastName = s2;
}
```



In general, a constructor should ALWAYS set ALL of the instance variables to some initial value. For example, consider the **Person** class being defined as follows ...

```

class Person {
    String    firstName;
    String    lastName;
    int       age;
    char      gender;
    boolean   retired;
}

```

We could thus write a constructor as follows:

```

Person(String fn, String ln, int a, char g, boolean r) {
    this.firstName = fn;
    this.lastName = ln;
    this.age = a;
    this.gender = g;
    this.retired = r;
}

```

And here is how we would call the constructor (as it pertains to an earlier example):

```

Person    p1, p2;

p1 = new Person("Hank", "Urchif", 19, 'M', false);
p2 = new Person("Holly", "Day", 67, 'F', true);

```

Of course, we can create constructors for all of our classes. Consider the larger example that used a **Person** object, an **Address** object and a **BankAccount** object:

```

BankAccount    account;

account = new BankAccount();
account.accountNumber = 178193;
account.balance = 100;

account.owner = new Person();
account.owner.firstName = "Hank";
account.owner.lastName = "Urchif";
account.owner.age = 19;
account.owner.gender = 'M';
account.owner.retired = false;

account.owner.address = new Address();
account.owner.address.streetNumber = 1526;
account.owner.address.unitLetter = 'B';
account.owner.address.streetName = "Oak St.";
account.owner.address.city = "Ottawa";
account.owner.address.province = "Ontario";
account.owner.address.postalCode = "K1S 5B6";

```

Here is what the simplified code would look like if we created constructors as well for **Address** and **BankAccount**:

```
BankAccount    account;
Person         per;
Address        adr;

adr = new Address(1526, 'B', "Oak St.", "Ottawa", "Ontario", "K1S 5B6");
per = new Person("Hank", "Urchif", 19, 'M', false, adr);
account = new BankAccount(178193, 100, per);
```

We could even simplify this into one big line ... but this looks a little uglier:

```
BankAccount    account;

account = new BankAccount(178193, 100,
                          new Person("Hank", "Urchif", 19, 'M', false,
                                      new Address(1526, 'B', "Oak St.",
                                                  "Ottawa", "Ontario",
                                                  "K1S 5B6")));
```

Of course, for this to all work, we would need to define the following constructors in the **Person**, **Address** and **BankAccount** classes, respectively:

```
Address(int n, char ul, String sn, String ci, String pr, String pc) {
    this.streetNumber = n;
    this.unitLetter = ul;
    this.streetName = sn;
    this.city = ci;
    this.province = pr;
    this.postalCode = pc;
}
```

```
Person(String fn, String ln, int a, char g, boolean r, Address adr) {
    this.firstName = fn;
    this.lastName = ln;
    this.age = a;
    this.gender = g;
    this.retired = r;
    this.address = adr;
}
```

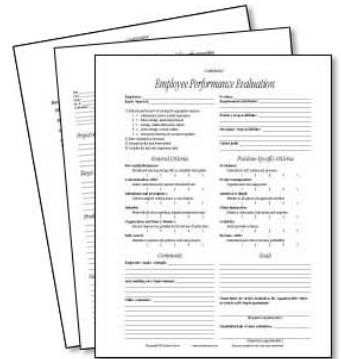
```

BankAccount(int acc, float bal, Person per) {
    this.accountNumber = acc;
    this.balance = bal;
    this.owner = per;
}

```

Certainly, you can see that the constructor allows us to greatly simplify our code when we need to create objects in our program.

Suppose though, that we do not know the initial value to use. That is, suppose that we want to create an object but are unsure as to what parameters to use. This would be analogous to the situation in real life where someone fills out a form but leaves some information blank. What do we do when the person leaves out information? We have two possible choices. Either **(1)** do not let them leave out any information, or **(2)** choose some kind of “default” values for the blank parts (i.e., make some assumptions by filling in something appropriate).



At this point in our program, we have chosen to force the user of our objects to supply parameters for ALL of the instance variables when they use (i.e., call) our constructor. So, we have taken approach number (1) above. However, in JAVA, we are allowed to create more than one constructor as long as the constructors each have a unique list of parameter types.

Consider the constructor for the **Person** class that we just defined:

```

Person(String fn, String ln, int a, char g, boolean r, Address adr) {
    this.firstName = fn;
    this.lastName = ln;
    this.age = a;
    this.gender = g;
    this.retired = r;
    this.address = adr;
}

```

What if, for example, we did not know the person’s age, nor their address.

```

Person    p1, p2;
p1 = new Person("Hank", "Urchif", ?, 'M', false, ?);
p2 = new Person("Holly", "Day", ?, 'F', true, ?);

```

In this situation, we are not allowed to pass in *nothing* as our parameters. But what we ARE allowed to do is to make another constructor that leaves these two parameters out:

```
Person(String fn, String ln, char g, boolean r) {
    this.firstName = fn;
    this.lastName = ln;
    this.gender = g;
    this.retired = r;
    this.age = 0;           // No age given, choose default value
    this.address = null;  // No address given, choose default value
}
```

Notice that there are two less parameters now (i.e., no age and no address). However, you will notice that we still set the age and address to some default values of our choosing. What is a good default **age** and **address**? Well, we used **0** and **null**. The word **null** is a special JAVA keyword that represents an *undefined object*. That is, since we do not have an **Address** object to store in the address instance variable, we leave it undefined by setting it to **null**. Alternatively, we could have created a “dummy” **Address** object with some kind of values that would be recognizable as invalid such as:

```
this.address = new Address(-1, '?', "Unknown", "Unknown", "Unknown", "Unknown");
```

It is entirely up to you to decide what the default values should be. Make sure not to pick something that may be mistaken for valid data. For example, some bad default values for **firstName** and **lastName** would be “John” and “Doe” because there may indeed be a real person called “John Doe”.

Here is one more constructor that takes no parameters. It has a special name and is known as the *zero-parameter constructor*, the *zero-argument constructor* or the *default constructor*. This time there are no parameters at all, so we need to pick default values for all the instance variables:

```
Person() {
    this.firstName = "UNKNOWN";
    this.lastName = "UNKNOWN";
    this.gender = '?';
    this.retired = false;
    this.age = 0;
    this.address = null;
}
```

Remember, that we can make many constructors. We just write them all one after another in our class definition and the user can decide which one to use at any time. Here is our resulting **Person** class definition showing the three constructors that we just defined ...

```

class Person {
    String    firstName;
    String    lastName;
    int       age;
    char      gender;
    boolean   retired;
    Address   address;

    // This is the zero-parameter constructor
    Person() {
        this.firstName = "UNKNOWN";
        this.lastName = "UNKNOWN";
        this.gender = '?';
        this.retired = false;
        this.age = 0;
        this.address = null;
    }

    // This is a 4-parameter constructor
    Person(String fn, String ln, char g, boolean r) {
        this.firstName = fn;
        this.lastName = ln;
        this.gender = g;
        this.retired = r;
        this.age = 0;
        this.address = null;
    }

    // This is a 6-parameter constructor
    Person(String fn, String ln, int a, char g, boolean r, Address adr) {
        this.firstName = fn;
        this.lastName = ln;
        this.age = a;
        this.gender = g;
        this.retired = r;
        this.address = adr;
    }
}

```

At any time we can use any of these constructors:

```

Person    p1, p2, p3,

p1 = new Person();
p2 = new Person("Holly", "Day", 'F', true);
p3 = new Person("Hank", "Urchif", 19, 'M', false,
               new Address(1526, 'B', "Oak St.", "Ottawa",
                           "Ontario", "K1S 5B6"));

```



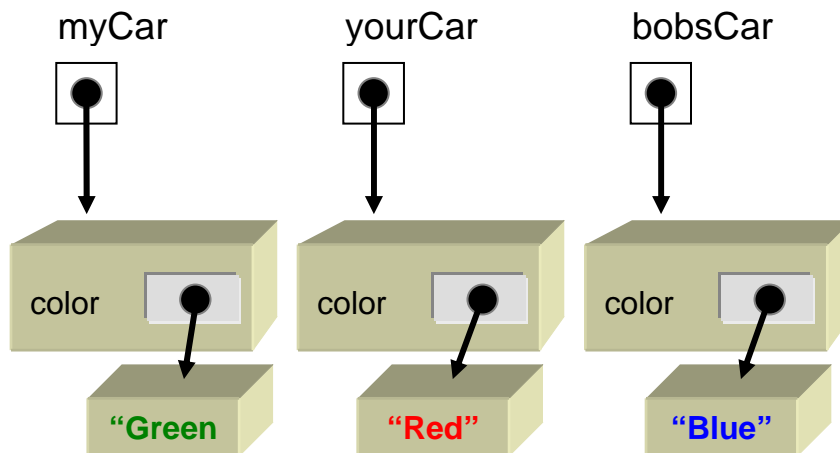
Note that it is always a good idea to ensure that you have a zero-parameter constructor. As it turns out, if you do not write any constructors, JAVA provides a zero-parameter constructor for free. That is, we can always say `new Car()`, `new Person()`, `new Address()`, `new BankAccount()` without even writing those constructors. However, once you write a constructor that has parameters, the free zero-parameter constructor is no longer available. That is, for example, if you write constructors in your `Person` class that all take one or more parameters, then you will no longer be able to use `new Person()`. JAVA will generate an error saying:

```
cannot find symbol constructor Person()
```

In general, you should always make your own zero-parameter constructor along with any others that you might like to use because others who use your class may expect there to be a zero-parameter constructor available.

## 2.4 Shared Data: 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:



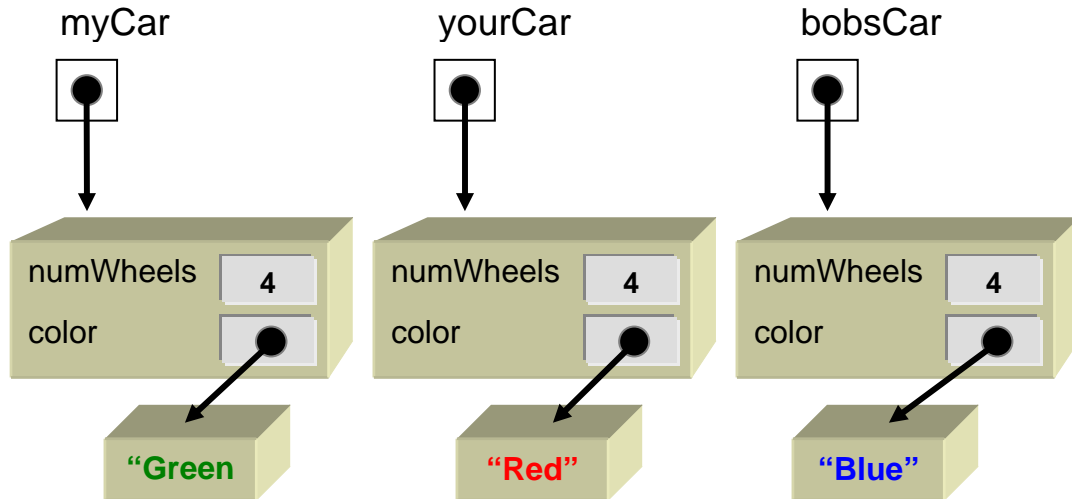
In some situations, however, there may be an attribute for an object in which the value does **not differ** between objects of that class. That is, each object that we make of that type would have the same value for that particular attribute. For example, all `Car` objects may have `4` wheels. We could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

```

class Car {
    String    color;
    int      numWheels;

    Car() {
        this.color = "";
        this.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;

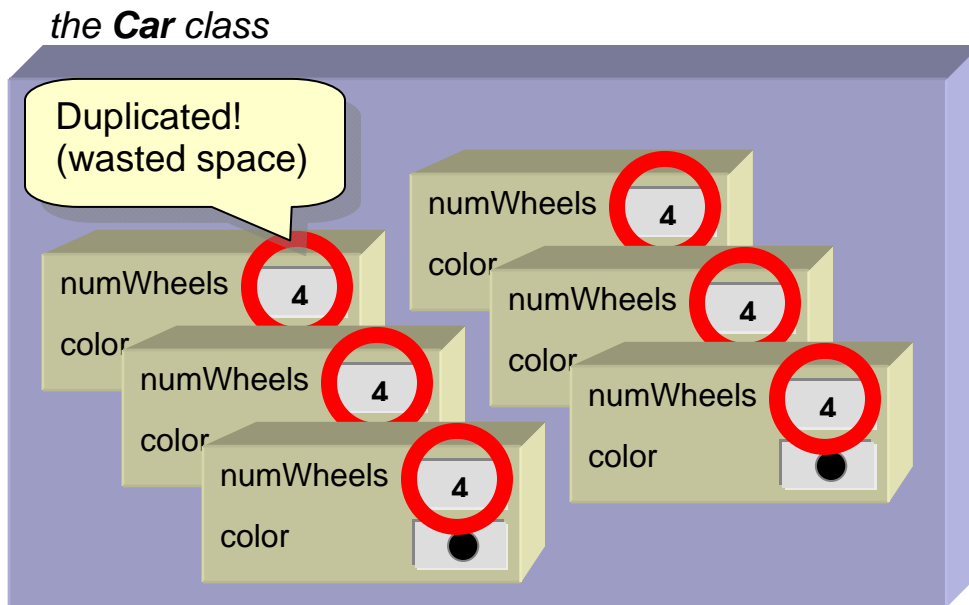
myCar = new Car();
yourCar = new Car();
bobsCar = new Car();

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.



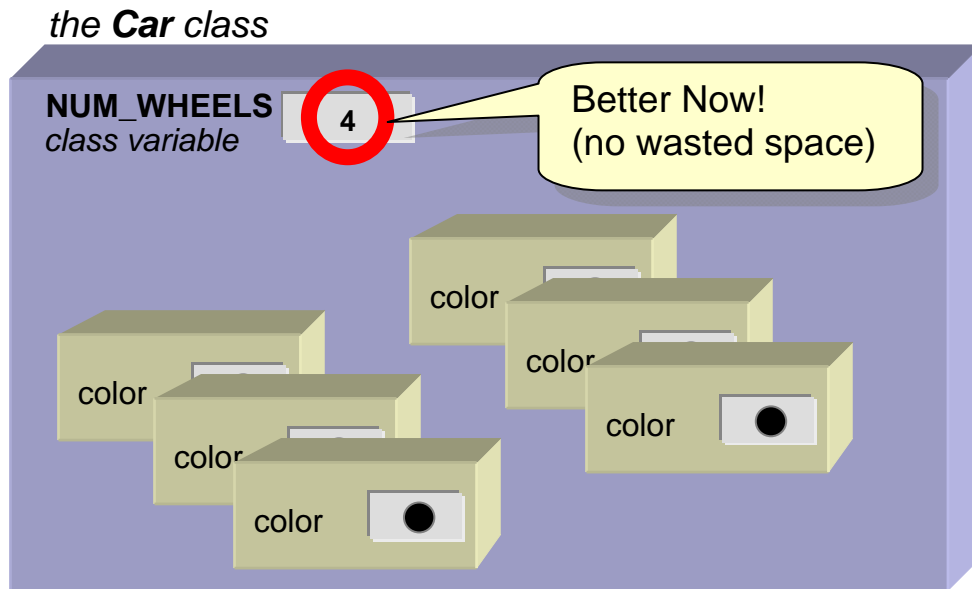
For situations like this ... in which all instances of a class (i.e., all objects created of one type) will share the same attribute value, you should create what is called a **class variable** (also known as a **static variable**). Class variables are "a little like" instance variables in that you can 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:

```
class Car {
    static int    NUM_WHEELS = 4; // a class variable (static)
    String       color;         // an instance variable (not static)

    Car() {
        this.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:

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

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

```
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. However, the way in which we access the value is different. 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:

```

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 is up to us to decide (In real life however, the Bank who 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 ...

```

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:



```

class BankAccount {
    static int LAST_ACCOUNT_NUMBER = 100000;

    int     accountNumber;
    String  owner;
    float   balance;

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

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

    //...
}

```

Here is some testing code:

```
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.