
Chapter 4

Data Structures

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 later course, we will further develop the notion of an Object.



4.1 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 the same country, 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 has all of that information. In a sense, the individual pieces of information *make up* (or **define the structure of**) the address.

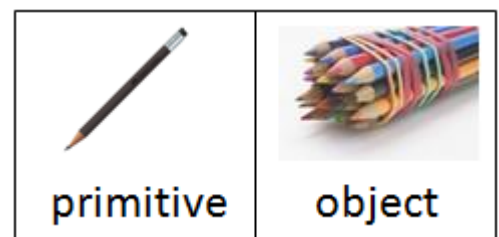
We can create such “more abstract” types of data (e.g., like an *address*) simply by combining or structuring the more primitives (i.e., simpler) pieces of data together in meaningful ways:

*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 Processing and 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 Processing and JAVA), strings are represented by placing double quotes around a set of characters like this:

name ← "Patty O. Lantern"

A string is not a *primitive* data type because it is made up of characters...which themselves are the 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:

name ← "Patty O. Lantern"
streetNumber ← 187
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 ← ... ? ...

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

address ← "Patty O. Lantern, 187 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** and sometimes **defining a class**.

When defining such a structure, we 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.

We will use the following notation to define a structure in our pseudocode (below), as well as visually (shown to the right):

```
define Address to be made of {
    .name
    .streetNumber
    .streetName
    .city
    .province
    .postalCode
}
```

anAddress

.name	...
.streetNumber	...
.streetName	...
.city	...
.province	...
.postalCode	...

Notice that we capitalized the data type **Address**. This is proper coding style and any definition of data structure, data type or object should always be capitalized.

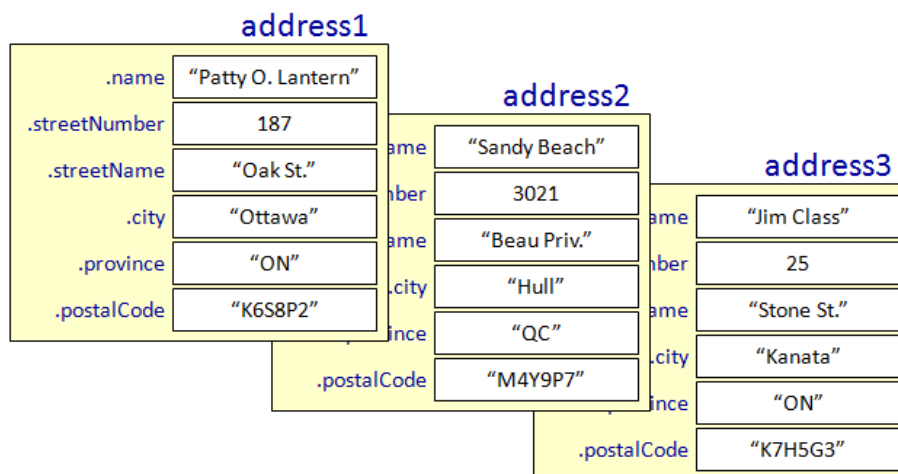
The above notation shows that an address is made up of 6 pieces of data with the given name labels. It is as if the **Address** data type is a “blank form” onto which we can fill in appropriate values. It defines a kind of *template* for creating data of this type.



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:

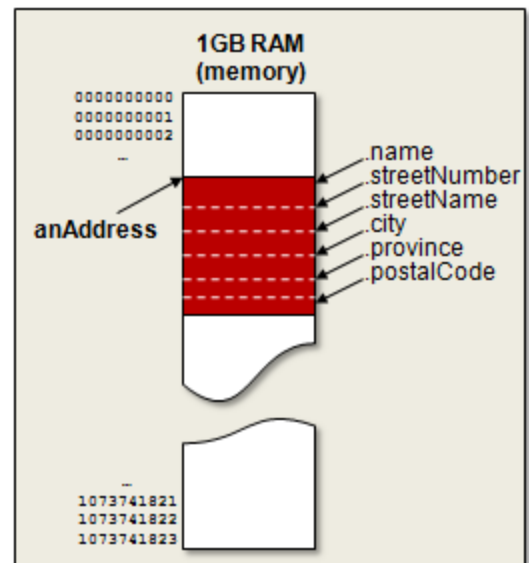


Recall that when we created variables, we had to reserve space in the computer's memory that would allow us to store information in the variable. Similarly, in order to make a new variable that can hold a data type, we have to also declare the variable in order to allocate sufficient memory.

When we declare a variable that belongs to the new data type, however, it is as if we are declaring space for all of its pieces. In fact, the pieces of a particular instance of a data type are actually called **instance variables**, since they are just regular variables ... that happen to be grouped together to form a particular instance.

In some programming languages, you have to allocate/reserve space (i.e., memory) for the instance of the data structure yourself. The popular languages **C**, for example, requires you to call a function that will allocate the memory for you (e.g., **malloc()**). Once you are done with the variable, it is your responsibility to free up that memory space once again by calling another function (e.g., **free()**).

The allocating and freeing up of memory is known as **dynamic memory allocation & deallocation** and is quite tedious and unpleasant. It can also be a source of many problems/bugs/errors in a program. For example, if memory is allocated many times, but never freed, the program will fill up the computer's memory and the program will crash. These are called **memory leaks**. Also, you have to make sure that you allocate enough memory to hold everything that you need and that you don't free memory that you still want to use ... otherwise your program will likely crash.



In order to make programmers live happier lives, some language designers have decided that it would be convenient and safer to perform this **memory management** for you. That is, they provide a means of allowing you to automatically reserve memory when you want to make a new instance of a data type (i.e., when you want to use a new object in your program). Object-oriented languages such as C++, C#, JAVA and Processing have built-in memory management.










In order to allocate enough memory for a new instance of a data type and start using it, the **new** keyword is used. For example, here is how we will indicate that we want a new instance of the **Address** data type in our pseudocode:

```
anAddress ← new Address
```

This will allocate 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
--------------------------	--------------------	---------------------------------

		
The Person Class	new Person	Instance of type Person
		
The House Class	new House	Instance of type House
		

We will then assign values to the individual components (i.e., instance variables) of **anAddress** by using the dot operator as follows:

```

anAddress ← new Address
anAddress.name ← "Patty O. Lantern"
anAddress.streetNumber ← 187
anAddress.streetName ← "Oak St."
anAddress.city ← "Ottawa"
anAddress.province ← "ON"
anAddress.postalCode ← "K6S8P2"

```

The dot operator indicates that we are going 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 that part of the address's memory.

Sometimes, however, 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. Nevertheless, when data is missing, we would want to make sure that we don't assume potentially wrong data. Our data structure, may therefore be missing data.

What is the value of the instance variables in this case, since we did not supply any values? It actually depends what is in the computer's memory at the location where the city, province and postal code is stored. It could be "garbage" data that was from a data type whose memory was previously freed up. With memory-managed languages, however, these values are usually set to **0** (when the variable's type is a number) or **null** (when the variable's type is a data type).

anAddress

.name	"Patty O. Lantern"
.streetNumber	187
.streetName	"Oak St."
.city	?
.province	?
.postalCode	?

Null is a word that represents an undefined value. If a variable has a value of null, it means that it does not yet have a value.

Therefore, the following code logic is flawed:

```
anAddress ← new Address
print (anAddress.name)
```

The code does not make sense because we are trying to print out the address's name before we have assigned a value to it. Depending on the language used, the result may be **null** or perhaps even random "garbage" data.

We can actually go deeper into each piece of data as well, making them more abstract. For example, we created 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 name, last name and middle name(s) of the person. To do this, we would need to separate the names into different variables as follows:

```
anAddress.firstName ← "Patty"
anAddress.middleName ← "Ohh"
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 here →

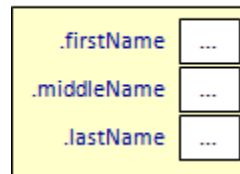
anAddress

.firstName	...
.middleName	...
.lastName	...
.streetNumber	...
.streetName	...
.city	...
.province	...
.postalCode	...

This would suffice. However, the address seems more complicated. We could perform additional data abstraction by combining the first, middle and last names into their own unique data structure:

```
define FullName to be made of {
  .firstName
  .middleName
  .lastName
}
```

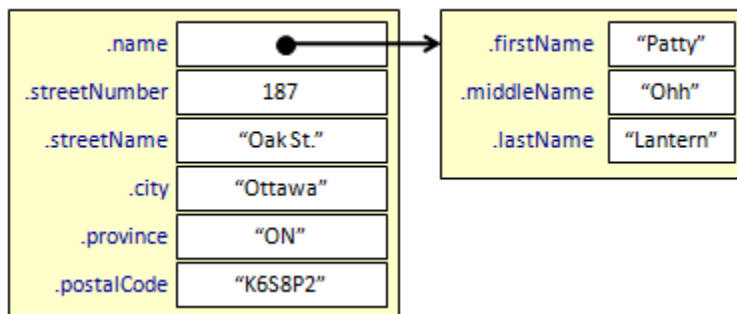
aFullName



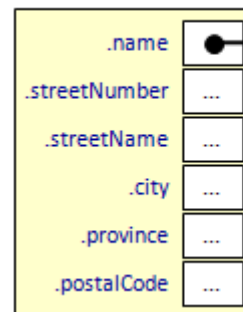
Then we could make use of this within our **Address** data structure as shown here. From the picture → you can see that the **name** stored in the **Address** structure is no longer a simple string of characters. Now it is a different kind of data structure (i.e., object) of type **FullName**.

Here is what our example would look like now:

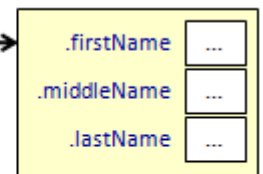
anAddress



anAddress



aFullName



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

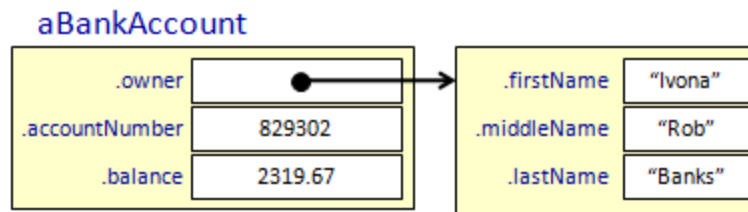
```
anAddress ← new Address
anAddress.name ← new FullName
anAddress.name.firstName ← "Patty"
anAddress.name.middleName ← "Ohh"
anAddress.name.lastName ← "Lantern"
```

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

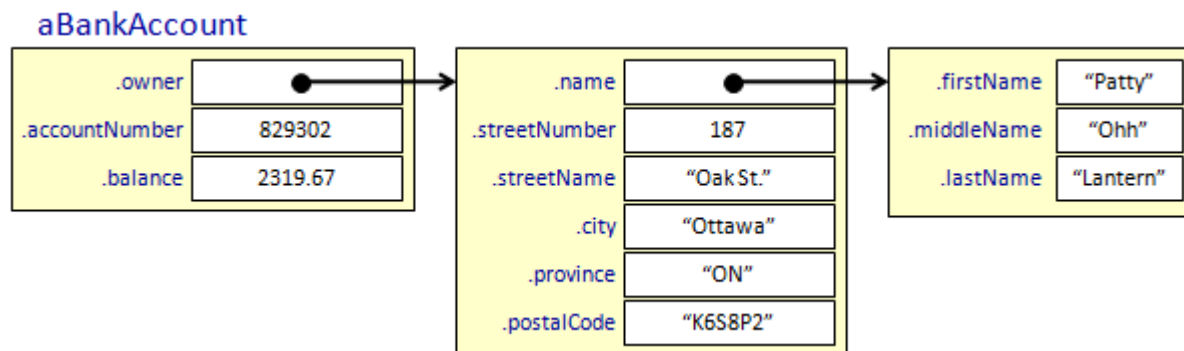
An additional advantage of creating the separate **FullName** data structure is that we can use it in other applications. Consider, a **BankAccount** data structure that is defined as follows:

```
define BankAccount to be made of {
  .owner
  .accountNumber
  .balance
}
```


We could then set the owner to be a **FullName** data type as well:



The fun does not stop here. In fact, it is often the case that data structures use multiple kinds of data structures within them. For example, could you determine the code that would produce this image ?



Now we should look at examples of using a data structure.

Example:

Recall the "Loan Qualification Kiosk" program that we implemented earlier which was to allow a people to enter personal information and then inform them as to whether or not they qualify for the loan.

```

1.  print welcome message
2.  employed ← ask user if he/she is currently employed
3.  hasDegree ← ask user if he received a univ. degree within past 6 months
4.  age ← ask user for his/her age
5.  yearsWorked ← ask user for # years worked at full time status

6.  if (employed is true) then {
7.      if (hasDegree is true) then print "Congratulations..."
8.      otherwise {
9.          if (age >= 30) then {
10.             if (yearsWorked >= 10) then print "Congratulations, ..."
11.             otherwise print "Sorry, ..."
12.         }
13.     }
14. }
15. otherwise print "Sorry, ..."

```

How can we adjust the code to combine all of the employee information into one data structure ? Notice what the code looks like now when an **Employee** data structure is used:

```
define Employee to be made of {
    .employed
    .hasDegree
    .age
    .yearsWorked
}
```



Algorithm: LoanQualificationKiosk

```
1.  print welcome message
2.  employee ← new Employee
3.  employee.employed ← ask user if he/she is currently employed
4.  employee.hasDegree ← ask user if he received a univ. degree within past 6 months
5.  employee.age ← ask user for his/her age
6.  employee.yearsWorked ← ask user for # years worked at full time status

7.  if (employee.employed is true) then {
8.      if (employee.hasDegree is true) then print “Congratulations...”
9.      otherwise {
10.         if (employee.age >= 30) then {
11.             if (employee.yearsWorked >= 10) then print “Congratulations, ...”
12.             otherwise print “Sorry, ...”
13.         }
14.         otherwise print “Sorry, ....”
15.     }
16. }
17. otherwise print “Sorry, ...”
```

All of the employee’s information is packaged into the single **Employee** data structure and stored in the **employee** 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 (i.e., lines 1 to 4):

Algorithm: LoanQualificationKiosk2

```
1.  print welcome message
2.  employee ← getUserInformation()
3.  determineQualifications(employee)
4.  print “Thank you, have a nice day.”
```

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

```

getUserInformation() {
5.     print welcome message
6.     employee ← new Employee
7.     employee.employed ← ask if currently employed
8.     employee.hasDegree ← ask if received a univ. degree within past 6 months
9.     employee.age ← ask for age
10.    employee.yearsWorked ← ask for # years worked at full time status
11.    return employee
}

```

The **determineQualifications()** procedure then accepts an incoming **Employee** object (which is labelled as **emp**) and uses it for various computations and decisions:

```

determineQualifications(emp) {
12.    if (emp.employed is true) then {
13.        if (emp.hasDegree is true) then
14.            print "Congratulations..."
15.        otherwise {
16.            if (emp.age >= 30) then {
17.                if (emp.yearsWorked >= 10) then
18.                    print "Congratulations, ..."
19.                otherwise print "Sorry, ..."
20.            }
21.        }
22.    }
23.    otherwise print "Sorry, ..."
}

```

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

There are some advantages of using the 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** (lines 1 through 4) **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., 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, doesn't our choice of attributes depend 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 components 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.):



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, insurance type (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 the algorithm that accelerates a car across the window:

```

1.  speed ← 0
2.  for x locations from 0 to windowHeight by speed {
3.      draw car at position (x, windowHeight - 100)
4.      if x < (windowWidth / 2) then
5.          speed ← speed + 0.10
6.      otherwise
7.          speed ← speed - 0.10
    }
```

We can create a **Car** data structure and use it within the program. What should make up the data structure? What components can be grouped together to represent the car?

```

define Car to be made of {
    .x
    .y
    .speed
}
```

```

1.  myCar ← new Car
2.  myCar.x ← 0
3.  myCar.y ← windowHeight - 100
4.  myCar.speed ← 0
5.
6.  for x locations from 0 to windowHeight by myCar.speed {
7.      myCar.x ← x
8.      draw(myCar)
9.      if myCar.x < (windowWidth / 2) then
10.         myCar.speed ← myCar.speed + 0.10
11.     otherwise
12.         myCar.speed ← myCar.speed - 0.10
    }
```

Now all of the car's parameters (i.e., location and speed) are kept together. Notice as well that the **draw()** procedure now simply takes one parameter, representing the car. It can then extract the **x** and **y** locations easily with the dot operator.

This may not seem like an advantage in this simple example, but as more and more car-related functions or procedures are added, this will greatly reduce the number of parameters being passed around. Also, if we add additional instance variables to the car's structure (e.g., color, scale) then this information will be readily available in the **draw()** procedure and we will not have to change line 8 of our program!! Once again, the code is cleaner and more organized.

4.2 Using Objects in Processing/JAVA

In Processing (and Java), data structures are called **objects**. To define a new data structure, we define a **class** using the **class** keyword and simply list the variables one after another (along with their types) between braces. Here are a few examples:

Pseudocode	Processing code
<pre>define FullName to be made of { .firstName .middleName .lastName }</pre>	<pre>class FullName { String firstName; String middleName; String lastName; }</pre>
<pre>define Address to be made of { .name .streetNumber .streetName .city .province .postalCode }</pre>	<pre>class Address { FullName name; int streetNumber; String streetName; String city; String province; String postalCode; }</pre>
<pre>define BankAccount to be made of { .owner .accountNumber .balance }</pre>	<pre>class BankAccount { Address owner; int accountNumber; float balance; }</pre>
<pre>define Employee to be made of { .employed .hasDegree .age .yearsWorked }</pre>	<pre>class Employee { boolean employed; boolean hasDegree; int age; int yearsWorked; }</pre>
<pre>define Car to be made of { .x .y .speed }</pre>	<pre>class Car { int x; int y; float speed; }</pre>

Notice how the types must now all be specified. Also, notice that when one data structure is contained within another (e.g., **FullName** inside **Address**), we must indicate the name of the data structure (i.e., class) as the variable's type.

The above class definitions cannot be done within the **setup()** or **draw()** procedures, nor in any other procedures or functions. They are defined on their own, separately, just like the **setup()** and **draw()** procedures. It is a good idea to gather all your class definitions near the top or bottom of your program. Here is an example template showing **Car** and **House** data structures defined just above the **setup()** and **draw()** procedures:

```
int  aVariable;
float anotherVariable;

class Car {
    ...
}

class House {
    ...
}

void setup() {
    ...
}

void draw() {
    ...
}
```

Later, in COMP1406, when we start doing Java programming, we will define each class in its own separate file.

Now, to create a new *instance* of one of these classes, we usually need to first create a variable to store it in and then we simply use the **new** keyword followed by the class name and parenthesis. Then we assign values to the instance variables using the = operator. Here are some examples of how this is done:

Pseudocode	Processing code
<pre>myCar ← new Car myCar.x ← 0 myCar.y ← windowHeight - 100 myCar.speed ← 0</pre>	<pre>Car myCar; myCar = new Car(); myCar.x = 0; myCar.y = height - 100; myCar.speed = 0;</pre>
<pre>aFullName ← new FullName aFullName.firstName ← "Patty" aFullName.middleName ← "Ohh" aFullName.lastName ← "Lantern"</pre>	<pre>FullName aFullName; aFullName = new FullName(); aFullName.firstName = "Patty"; aFullName.middleName = "Ohh"; aFullName.lastName = "Lantern";</pre>

<pre> addr ← new Address addr.name ← new FullName addr.name.firstName ← "Patty" addr.name.middleName ← "Ohh" addr.name.lastName ← "Lantern" addr.streetNumber ← 187 addr.streetName ← "Oak St." addr.city ← "Ottawa" addr.province ← "ON" addr.postalCode ← "K6S8P2" </pre>	<pre> Address addr; addr = new Address(); addr.name = new FullName(); addr.name.firstName = "Patty" addr.name.middleName = "Ohh" addr.name.lastName = "Lantern" addr.streetNumber = 187 addr.streetName = "Oak St." addr.city = "Ottawa" addr.province = "ON" addr.postalCode = "K6S8P2" </pre>
<pre> b ← new BankAccount b.owner ← new Address b.owner.name ← new FullName b.owner.name.firstName ← "Patty" b.owner.name.middleName ← "Ohh" b.owner.name.lastName ← "Lantern" b.owner.streetNumber ← 187 b.owner.streetName ← "Oak St." b.owner.city ← "Ottawa" b.owner.province ← "ON" b.owner.postalCode ← "K6S8P2" b.accountNumber ← 829302 b.postalCode ← 2319.67 </pre>	<pre> BankAccount b; Address addr; addr = new Address(); addr.name = new FullName(); addr.name.firstName = "Patty" addr.name.middleName = "Ohh" addr.name.lastName = "Lantern" addr.streetNumber = 187 addr.streetName = "Oak St." addr.city = "Ottawa" addr.province = "ON" addr.postalCode = "K6S8P2" b = new BankAccount(); b.owner = addr; b.accountNumber = 829302 b.postalCode = 2319.67 </pre>
<pre> emp ← new Employee emp.employed ← ... some code ... emp.hasDegree ← ... some code ... emp.age ← ... some code ... emp.yearsWorked ← ... some code ... </pre>	<pre> Employee emp = new Employee(); emp.employed = /* some code */; emp.hasDegree = /* some code */; emp.age = /* some code */; emp.yearsWorked = /* some code */; </pre>

Example:

Recall our example in which we drew 5 houses of various sizes at adjacent locations on the window →



Here is the code that we used:

```
void setup() {
    size(500,150);

    drawHouse(0, 150, 1);
    drawHouse(100, 150, 0.8);
    drawHouse(180, 150, 0.6);
    drawHouse(240, 150, 0.4);
    drawHouse(280, 150, 0.2);
}

void drawHouse(int x, int y, float s) {
    rect(x, y-100*s,100*s,100*s);
    triangle(x,y-100*s,(x+50*s),y-150*s,(x+100*s),y-100*s);
    rect((x+35*s),y-40*s,30*s,40*s);
    point((x+55*s),y-20*s);
}
```

How could we adjust this code to make use of a House data structure ? What information in this example represents the attributes of the Houses ?

The x, y location are attributes of the houses as well as their scale factor. So, we could define a house data structure (i.e., object) as follows:

```
class House {
    int x;
    int y;
    float s;
}
```

Then we can make use of this new **House** object in our code:

```
void setup() {
    House h1, h2, h3, h4, h5;

    size(500,150);

    h1 = new House();
    h1.x = 0;
    h1.y = 150;
    h1.s = 1.0;

    h2 = new House();
    h2.x = 100;
    h2.y = 150;
    h2.s = 0.8;

    h3 = new House();
    h3.x = 180;
```

```
h3.y = 150;
h3.s = 0.6;

h4 = new House();
h4.x = 240;
h4.y = 150;
h4.s = 0.4;

h5 = new House();
h5.x = 280;
h5.y = 150;
h5.s = 0.2;

drawHouse(h1);
drawHouse(h2);
drawHouse(h3);
drawHouse(h4);
drawHouse(h5);
}

void drawHouse(House h) {
    rect(h.x, h.y-100*h.s,100*h.s,100* h.s);
    triangle(h.x, h.y-100*h.s,(h.x+50*h.s), h.y-150*h.s,
            (h.x+100*h.s), h.y-100*h.s);
    rect((h.x+35*h.s), h.y-40*h.s,30* h.s,40*h.s);
    point((h.x+55*h.s), h.y-20*h.s);
}
```

Notice how the houses are each stored in variables **h1** through **h5**. That is, the **x**, **y** and **s** data is all kept together for each individual house and stored in their own unique House variable.

Notice as well how the **House** objects are passed as parameters into the **drawHouse** procedure. Within the procedure, each house is referred to as **h**, as indicated by the parameter name. Then, inside the procedure, we simply access the **x**, **y** or **s** component of the house by using **h.x**, **h.y** or **h.s**, respectively.

Example:

Here is how we could use a **Car** data structure in a modification of our accelerating car example that uses two cars. Notice how the use of the data structure simplifies the code when multiple cars are used.

```

Car    myCar, yourCar;    // The two cars to move around

class Car {                // Definition of what a "Car" actually is
    int    x, y;
    float  speed;
    int    direction;
}

void setup() {             // Initialize the cars by setting the values of their components
    size(600,300);
    myCar = new Car();
    myCar.x = 0;
    myCar.y = 300;
    myCar.speed = 0;
    myCar.direction = 1;

    yourCar = new Car();
    yourCar.x = 300;
    yourCar.y = 300;
    yourCar.speed = 0;
    yourCar.direction = -1;
}

void draw() {              // Repeatedly draw and move each car
    background(255,255,255);
    drawCar(myCar);
    drawCar(yourCar);

    moveCar(myCar);
    moveCar(yourCar);
}

// Draw the car that is passed in as a parameter
void drawCar(Car aCar) {
    fill(150,150,150);
    rect(aCar.x, aCar.y-30, 100, 20);
    quad(aCar.x+20,aCar.y-30, aCar.x+30,aCar.y-45, aCar.x+55,aCar.y-45, aCar.x+70,aCar.y-30);

    fill(0,0,0); // black
    ellipse(aCar.x+20, aCar.y-10, 20, 20);
    ellipse(aCar.x+75, aCar.y-10, 20, 20);
    fill(255,255,255); // white
    ellipse(aCar.x+20, aCar.y-10, 10, 10);
    ellipse(aCar.x+75, aCar.y-10, 10, 10);
}

// Move the car that is passed in as a parameter
void moveCar(Car aCar) {
    aCar.x = int(aCar.x + aCar.speed*aCar.direction);

    if (mousePressed)
        aCar.speed = min(100, aCar.speed + 0.10);
    else
        aCar.speed = max(0, aCar.speed - 0.10);

    if (abs(aCar.x+50 - mouseX) <= aCar.speed)
        aCar.speed = 0;

    if (mouseX < aCar.x+50)
        aCar.direction = -1;
    else
        aCar.direction = 1;
}

```

Creating an instance of a data structure that has many instance variables, may require many lines of code. For example, creating and initializing **myCar**, in our previous example required 5 lines of code as follows:

```
myCar = new Car();
myCar.x = 0;
myCar.y = 300;
myCar.speed = 0;
myCar.direction = 1;
```

In Processing (and 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 procedure that is automatically called to initialize a new object.*

In fact, the parentheses that appear when we do **new Car()** indicate that **Car()** is in fact a kind of procedure or function. This is actually a special kind of function known as the **default constructor**. What it actually does is that it creates and returns a new fully-initialized object. That is, it reserves space for the data structure's components and sets them all to a value of "zero".

In our above example, however, we wanted to set the **y** value of the car to **300** and the **direction** to **1** ... we did not want zeros. In the **yourCar** variable, we further set the **x** value to **300** and **direction** to **-1**. The point is ... each object that we create will often have their own unique initial values. In a way, the idea is analogous to building our own computer ... we would like to have control to configure our own system with our choice of internal components (i.e., we pick the hard drive, the video card, the motherboard, the monitor, etc..).

We are allowed to write our own constructor procedures so that we can supply our own initial values for our objects. 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 variable. The constructor must be written within the data structure's class definition as follows:

```
class Car {                                     // Definition of what a "Car" actually is
  int    x, y;
  float  speed;
  int    direction;

  Car(int p1, int p2, float p3, int p4) {      // a constructor
    x = p1;
    y = p2;
    speed = p3;
    direction = p4;
  }
}
```

Notice that the constructor's name is identical to the class name (always uppercase letter to start). Also, notice how there is one parameter for each of the 4 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** and **p4**) are arbitrary, but they must be unique from one another and **MUST NOT** be the same as any instance variable names.

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. Below shows the previous **setup()** code before we had the constructor ... along with the new code that makes use of the constructor:

Without the constructor:	With the constructor:
<pre> Car myCar, yourCar; class Car { int x, y; float speed; int direction; } void setup() { size(600,300); myCar = new Car(); myCar.x = 0; myCar.y = 300; myCar.speed = 0; myCar.direction = 1; yourCar = new Car(); yourCar.x = 300; yourCar.y = 300; yourCar.speed = 0; yourCar.direction = -1; } </pre>	<pre> Car myCar, yourCar; class Car { int x, y; float speed; int direction; Car(int p1, int p2, float p3, int p4) { x = p1; y = p2; speed = p3; direction = p4; } } void setup() { size(600,300); myCar = new Car(0, 300, 0, 1); yourCar = new Car(300, 300, 0, -1); } </pre>

Notice the drastic reduction of code within the **setup()** procedure. The saving in code space can be much more drastic when multiple kinds of objects are used together. For example, consider that we created constructors for the **FullName**, **Address** and **BankAccount** objects:

<pre> class FullName { String firstName, middleName, lastName; FullName(String p1, String p2, String p3) { firstName = p1; middleName = p2; lastName = p3; } } </pre>

```

class Address {
    FullName name;
    int streetNumber;
    String streetName, city, province, postalCode;

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

```

```

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

    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 the constructor:	With the constructor:
<pre> BankAccount b; Address a; FullName n; n = new FullName(); n.firstName = "Patty" n.middleName = "Ohh" 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.postalCode = 2319.67 </pre>	<pre> BankAccount b; Address a; FullName n; n = new FullName("Patty", "Ohh", "Lantern"); a = new Address(n, 187, "Oak St.", "Ottawa", "ON", "K6S8P2"); b = new BankAccount(a, 829302, 2319.67); </pre>

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

Example:

Recall our example that allowed us to throw a ball around in the window:

```

int      x, y;           // location of the ball at any time
float    direction;     // direction of the ball at any time
boolean  grabbed;      // true if the ball is being held
float    speed;         // the ball's speed

final int  RADIUS = 40;           // the ball's radius
final float ACCELERATION = 0.10; // acceleration/deceleration amount

void setup() {
  size(600,600);
  x = width/2;
  y = height/2;
  direction = random(TWO_PI);
  grabbed = false;
  speed = 10;
}

void draw() {
  background(0,0,0);
  ellipse(x, y, 2*RADIUS,2*RADIUS);

  // move the ball forward if not being held
  if (!grabbed) {
    x = x + int(speed*cos(direction));
    y = y + int(speed*sin(direction));
  }
  else {
    x = mouseX;
    y = mouseY;
  }

  speed = max(0, speed - ACCELERATION);

  if ((x+RADIUS >= width) || (x-RADIUS <= 0))
    direction = PI - direction;
  if ((y+RADIUS >= height) || (y-RADIUS <= 0))
    direction = -direction;
}

void mousePressed() {
  if (dist(x,y,mouseX,mouseY) < RADIUS)
    grabbed = true;
}

void mouseReleased() {
  if (grabbed) {
    direction = atan2(mouseY - pmouseY, mouseX - pmouseX);
    speed = int(dist(mouseX, mouseY, pmouseX, pmouseY));
  }
  grabbed = false;
}

```

How could we adjust the code to use a **Ball** data structure ? What information in this program, will make up the attributes of the ball ?

It is easy to see that 4 of the 5 variables actually represent the ball's state ... that is, its (x, y) position, direction, and speed. Whether the ball has been grabbed or not is not necessarily part of the Ball's attributes. That is, a ball does not need to know whether or not it has been grabbed. For example, we may have an application in which the balls are moving around without the ability to be grabbed. So, here is what our **Ball** data structure would look like along with its constructor:

```
class Ball {
    int      x, y;          // location of the ball at any time
    float    direction;    // direction of the ball at any time
    float    speed;        // the ball's speed

    Ball(int p1, int p2, float p3, float p4) {
        x = p1;
        y = p2;
        direction = p3;
        speed = p4;
    }
}
```

Assuming that we define this class in our program, the remainder of the program can now be adjusted as follows. Notice the difference between the code before the Ball class is used and after:

Without Ball Class Definition	With Ball Class Definition
<pre>final int RADIUS = 40; final float ACCELERATION = 0.10;</pre>	<pre>class Ball { // as defined earlier ... }</pre> <pre>final int RADIUS = 40; final float ACCELERATION = 0.10;</pre>
<pre>int x, y; float direction; float speed;</pre>	<pre>Ball b; // the Ball</pre>
<pre>boolean grabbed;</pre> <pre>void setup() { size(600,600);</pre>	<pre>boolean grabbed;</pre> <pre>void setup() { size(600,600);</pre>
<pre>x = width/2; y = height/2; direction = random(TWO_PI); speed = 10;</pre>	<pre>b = new Ball(width/2, height/2, random(TWO_PI), 10);</pre>
<pre>grabbed = false; }</pre> <pre>void draw() { background(0,0,0); ellipse(x, y, 2*RADIUS,2*RADIUS);</pre>	<pre>grabbed = false; }</pre> <pre>void draw() { background(0,0,0); ellipse(b.x, b.y, 2*RADIUS,2*RADIUS);</pre>


```

if (!grabbed) {
    x = x + int(speed*cos(direction));
    y = y + int(speed*sin(direction));
}
else {
    x = mouseX;
    y = mouseY;
}

speed = max(0, speed - ACCELERATION);

if ((x+RADIUS>=width)|| (x-RADIUS<=0))
    direction = PI - direction;
if ((y+RADIUS>=height)|| (y-RADIUS<=0))
    direction = -direction;
}

void mousePressed() {
    if (dist(x,y,mouseX,mouseY) < RADIUS)
        grabbed = true;
}

void mouseReleased() {
    if (grabbed) {
        direction = atan2(mouseY - pmouseY,
                          mouseX - pmouseX);
        speed = int(dist(mouseX, mouseY,
                          pmouseX, pmouseY));
    }
    grabbed = false;
}

```

```

if (!grabbed) {
    b.x=b.x+int(b.speed*cos(b.direction));
    b.y=b.y+int(b.speed*sin(b.direction));
}
else {
    b.x = mouseX;
    b.y = mouseY;
}

b.speed = max(0, b.speed-ACCELERATION);

if ((b.x+RADIUS>=width)|| (b.x-RADIUS<=0))
    b.direction = PI - b.direction;
if ((b.y+RADIUS>=height)|| (b.y-RADIUS<=0))
    b.direction = -b.direction;
}

void mousePressed() {
    if (dist(b.x, b.y,mouseX,mouseY)<RADIUS)
        grabbed = true;
}

void mouseReleased() {
    if (grabbed) {
        b.direction = atan2(mouseY - pmouseY,
                          mouseX - pmouseX);
        b.speed = int(dist(mouseX, mouseY,
                          pmouseX, pmouseY));
    }
    grabbed = false;
}

```

Notice how the ball's attributes are all now defined inside the **Ball** object so that less variables are needed in the main program. Also, the remainder of the code simply requires **b.** in front of these attributes in order to go into the object to get their values.

While it seems as though we are writing a little bit more code now, the advantage of creating this **Ball** data structure will be more clear later.