

(MORE)OBJECT ORIENTED PROGRAMMING

(*PARENT*) CLASS

this is a class of types of *Birds*

The class is like a template/blueprint of properties common to all of the objects contained in the class. '**Template**' means that it is blank, the actual values are found in each object, and may vary from object to object.

Member Variables(Properties)

wingspan	
color	
weight	
miles per hour it flies	
owner	

OBJECT

This is an object of a cardinal.

Member Variables(Properties) values

Specific for Cardinal

wingspan

12"

color

red

weight

100g

miles per hour it flies

30 mph

owner

zoo

METHODS(func/proc)

flying

feeding

OBJECT

This is an object of a bluejay.

Member Variables(Properties)

wingspan

14"

color

blue

weight

120g

miles per hour it flies

25 mph

owner

Mr. Smith

METHODS(proc)

flying

feeding

(SUB)CLASS

The *Roadrunner* class

(new) MemberProperty

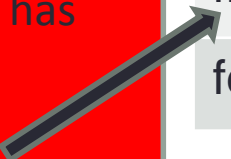
miles per hour it runs	55 mph
---------------------------	--------

(overridden) Method

flying	
--------	--

LOCAL VARIABLE

The 'flying' method requires a new variable unique to the flying method called "hopping height". Because "hopping height" is only used in this one method, and its value dies once the method has been executed, then it is considered a local variable.



OBJECT

This is an object of a roadrunner.

Member Variables(Properties)

wingspan	24"
----------	-----

color	brown
-------	-------

weight	400g
--------	------

miles per hour it flies	10 mph
-------------------------	--------

owner	no one
-------	--------

METHODS(proc)

flying ***	
------------	--

feeding	
---------	--

Roadrunner Subclass Cont'd

- ❑ Notice that because roadrunners prefer to run than fly, they are treated differently.
- ❑ The miles per hour it runs is a property unique to roadrunners.
- ❑ Also, since its flying is made up of hops and flaps, then *its flying method must override the typical flying method of other bird objects in the bird class.*

SUBCLASS (derived class, child class or descendant)

PARENT CLASS (base class, superclass, ancestor)

- ❑ Roadrunner is a subclass of its parent class, Birds. The Roadrunner class is a blueprint that describes a hypothetical roadrunner.
- ❑ **It is understood that a subclass will *inherit* all the properties(member data) and methods of its parent class.**
- ❑ Additional properties unique to the subclass may be added if needed (i.e. miles per hour it runs).

Subclasses cont'd

- ❑ A subclass can also override its parent's methods by simply using the same name, but redefining the method (i.e. the flying method).
- ❑ The process of creating subclasses is called '*designing by difference*' and is useful when you have large libraries.
- ❑ Designing by difference allows you to create a new class by simply defining the parts of it that are different from the parent class.

- When a message is sent to an object to invoke a method, **the object finds the most recent ancestor class that implements the method and invokes that method.**
- In other words - sending the Roadrunner instance a flying message will use the overridden method found in the roadrunner class, and not the flying method of the bird class.

MEMBER DATA ('properties, variables')

- ❑ Member data refers to the data stored within an object or class.
- ❑ Member data within an instantiated object is called instance variables.
- ❑ You define which member variables are part of a class in the class definition and new instances of these variables are created every time a new object is created from the class. Every instance of a class will have the same types of variables contained within it, but what's stored in those variables will be different from one object to another.

LOCAL VARIABLES

If a variable is found within a method and nowhere else, such as the ***hoppingHeight*** variable of the Roadrunner class, then it is a local variable whose value dies once the method has been executed.

METHODS and 'CALLING' A METHOD

- ❑ Calling a method is also referred to as *sending a message to the object that contains the method.*
- ❑ Calling the flying method of the cardinal object is referred to as sending a flying message to that targeted object.
- ❑ When a method is invoked (or 'called'), any reference to a member variable in the methods code will actually deal **with the instance variable contained in the object of which the method is a member.**

ENCAPSULATION

Member variables can be hidden from code in other objects or classes.

For example, *number of eggs laid* variable could be specified as hidden or private to the cardinal object. This process is called **encapsulation**.

ABSTRACTION

Abstraction is the process of identifying a **general-purpose class** that contains common properties and methods that you want to use as the parent class for a number of related classes.

```
class Box{  
    int height;  
    int width;  
    int repetitions;  
  
    public Box( )  
    {  
        height=10;  
        width=10;  
        repetitions=3;  
    }  
}
```

member (global)
variables

If there is nothing in
the brackets then
this is the “default
constructor”.

```
public Box(int h, int w)
{
    height=h;
    width=w;
    repetitions=3;
}
```

This constructor allows the client to pass in their own values for height and width


```
public Box(int h, int w, int r)
{
    height=h;
    width=w;
    repetitions=r;
}
```

This constructor allows the client to pass in their own values for height and width AND repetitions.

OVERRIDING VARIABLES

When writing up your driver class that defines your objects, it really is undesirable to override a variable by:

objectname.variablename = new Value;

Example:

redCat.colour=Color.red;

redCat.x=70;

That is because you would have to know the variable name found in the template class.

- ❑ This is considered an *invasion of privacy*, and allows for no protection of variables because they can then be overridden.
- ❑ Best practice: create a method in your template that allows you to *pass* in a value that can then be used later on in the template when the corresponding method is called.

Bad Approach

Driver class....

object created called redCat

redCat.x = 120;

Template class....

x=100; //default value

Desired Approach

Driver class...

redCat.setXValue(120);

Template class....

x=100; //default value

***public void setXValue(int newX) {
x=newX;
}***

- ❑ *"mutator" method can modify an object, in this case, assigning a new value to one of the global variables ("fields").*
- ❑ *Often such methods start with the word "set".*
- ❑ *"accessors" are methods that accesses the contents of an object without changing it, such as returning the value of one of the variables.*
- ❑ *Often these method names start with the word "get".*

SETTING THIS AND THAT

Quite often you will see methods that start with the word "SET". The methods are seemingly short, often containing just 1 line.

Why do we use these "set" such-and-such methods?

So we can avoid having to pass variables into every single method that needs them and to protect template classes from having too many methods and variables tampered with.

"THIS" EXPLAINED

The keyword "this" has 4 different uses:

1. as a constructor: **this**()
2. as a variable: **this**.variable name
3. as a method: **this**.methodname()
4. as an argument:

anyMethod(**this**, other arguments too if needed)

1. CONSTRUCTOR

If a class has 2 or more constructors.

"This" means use the current constructor .

```
public class Robot{  
    int speed;  
    public Robot()    {  
        int unitsmove=1;  
        String direction="right";  
    }  
}
```



```
public Robot(int howFast) {  
    this();  
    speed=howFast;  
}
```

"this()" means that you substitute the above default constructor
(i.e. *int unitsmove=1;*
and *String direction="right"*) etc.

2. VARIABLE

When using more than one class, you may want to use the same variable name in both classes. For example, we are so used to using the variable "c" for the Console class, that to change the name would only pose confusion.

Similarly, the terms "x" and "y" are so frequently used to describe axis and co-ordinates that to use "p" and "q" would only pose confusion.

- ❑ So, when passing a variable from one class into another class, the word "this" is used to distinguish the one variable from another with the same name.
- ❑ "This" is usually used to describe the variable in the parent class, the current class, whereas NO "this" is used for the variable *passed* in from another class.

No "this" used

different variable names are used (c & cons)

Driver class

main method

```
Console c=new Console();  
Another d=new Another();  
d.methodname(c);
```

passed in

Another class

```
Console cons;
```

```
...
```

```
methodname(Console c){  
    cons=c;
```

"this" is used since both classes have the same variable name for the Console class: **c**

Driver class

main method

```
Console c=new Console();  
Another d=new Another();  
d.methodname(c);
```

passed in

Another class

```
Console c;
```

...

```
methodname(Console c){  
    this.c =c;
```

3. METHOD

- ❑ Similar to above, you may have two classes that have the same method name, although the 2 methods are entirely different.
- ❑ **And so "*this*" would mean use the method from the current class**, and not from another class (i.e. the parent class).

4. PARAMETER

- ❑ When "*this*" is used as an argument of a method, then it means that it is passing a reference to the current object. (Where there may be more than one argument in any given method, the other arguments would remain the same.)
- ❑ For example, the program on the next slide has a method of the Graphics class that assumes its sole parameter is a reference to the current object (which would access the parent class)

with "*this*"

```
class Robot extends Applet{
```

```
    public void paint(Graphics g){  
        g.methodname(this);  
    }
```


without "*this*"

```
class Robot extends Applet{
```

```
    Applet apple = new Applet();
```

```
    public void paint(Graphics g){  
        g.methodname(apple);  
    }
```

STATIC STUFF

Static, (like public), is a modifier word that can be placed in front of a class, method, or variable.

Static means 'doesn't change'

1. STATIC AND NON-STATIC CLASSES

Static classes are not used for outer top-level classes. They are only used in nested top-level classes often as a convenient way to group related classes without creating a new package. (We won't be using this application).

2. STATIC AND NON-STATIC VARIABLES

Non-static variables are also called *object* or *instance* variables because they are only accessed from another class by way of an object:

```
Cat kitty=new Cat();
```

```
kitty.height=7;
```

Static variables are also called *class variables*, because they don't need an object, just the name of the class to access them from another class.

i.e. *Classname.variablename=2;*

You can also access a class variable the same way as an object variable

i.e. *objectname.variablename=2;*

(but this method is discouraged since it isn't clear as to whether it is a static or non static variable.)

1. When a class is first loaded, static variables are created only once and have only one copy available which is shared by the entire class. This means that if you create several objects that all access this static variable, then there is only 1 variable capable of being changed.

2. This is useful when you want to have a counter variable made static so that each method can add on to it if something happens (Like points chalked up in a game if you hit a target or stay alive for a certain length of time etc.)

Each copy made (although seemingly the same variable) can end up with different values when acted upon because each copy is independent of each other.

(Because of the copies made, non- static variables should be avoided unless necessary because of the room taken up in memory.)

STATIC

Cat class

```
{ ...  
    static int lives;
```

Driver class

```
{ ...
```

```
Cat.lives=0;
```

```
...survived_crash_method(){  
    Cat.lives++;  
}
```

```
...survived_fall_method(){  
    Cat.lives++;  
}
```

```
...
```

If survived_crash_method is called

Cat.lives=1

if then survived_fall_method is called

Cat.lives=2

STATIC (same behaviour as the previous example... don't use this approach as it's confusing)

Cat class

```
{ ...  
    static int lives;
```

Driver class

```
{ ...  
    Cat kitty1=new Cat();  
    Cat kitty2=new Cat();  
  
    kitty1.lives=0;  
    kitty2.lives=0; // any other value than 0  
    //would over-ride the above statement
```

```
...survived_crash_method(){  
    kitty1.lives++;  
}  
...survived_fall_method(){  
    kitty2.lives++;  
}  
...
```

If survived_crash_method is called
 kitty1.lives=1
if then survived_fall_method is called
 kitty2.lives=2
(since same variable as above)

NON-STATIC

```
Cat class
{ ...
    int lives; //non-static

Driver class
{ ...
    Cat kitty1=new Cat();
    Cat kitty2=new Cat();

    kitty1.lives=0;
    kitty2.lives=0;

...survived_crash_method(){
    kitty1.lives++;
}
...survived_fall_method(){
    kitty2.lives++;
}
...
}
```

If survived_crash_method is called
kitty1.lives=1
if then survived_fall_method is called
kitty2.lives=1
(since separate variable as above)

3. STATIC AND NON-STATIC METHODS

Non-static methods are also called *object* or *instance methods* because they are only accessed from another class by way of an object i.e. *Cat kitty=new Cat();*
kitty.purr();

All of the Console classes methods are non-static, and so they must be first accessed by an object:

Console c=new Console();
c.println();

Static methods are also called *class methods*, because they don't need an object, just the name of the class to access them from another class.

i.e. `Classname.methodname();`

Very few of Java's library classes have static methods in them, but the `Math` class has mostly static methods since calculations like square root or rounding never change

i.e. `int num=Math.sqrt(9);`

- ❑ Static methods can access static variables or static methods, within the class,
- ❑ Accessing a non- static variable or method within the class causes an error message, (since your static method may be executed when no objects of the class have been created, and therefore no instance variables exist).

That is why the main class, which is static, must have other static methods within the class if it plans on calling them into action.

On the other hand, non-static methods can legally access static variables and methods, in addition to non-static ones.

Static methods are shared by all instances of the class, (just like static variables).

Unlike non-static variables, which have copies made every time an object is created, there is only one copy of a *non- static method* stored in memory but it is shared by all objects of the class.

This is done to save memory, since methods take up more room than variables.

Also, *static methods* (like `main`) cannot access global('member') variables like `c` unless the variable is static

— hence *static* `Console c=new Console();`

So, When Do I Use Static Methods VS Non-Static methods?

If a class doesn't need to have objects created from it, and so doesn't need anything different derived from it, then make the class have *static* methods in it. If you don't want to instantiate a class, then all of its methods must be *static*.

So, When Do I Use Static Methods VS Non-Static methods?

You use *static* methods when you want to have methods that can be called without having to instantiate an object.

The most common reason for this is that you might want a method that creates an object of the target class and returns it.

So, When Do I Use Static Methods VS Non-Static methods?

If methods don't need data other than parameters passed to them, then instantiating them would be unnecessary.

If these methods were useful to several classes, then copying and pasting them would be wasteful, so therefore, it is wise to make them *static* methods in a "utility" class.

So, When Do I Use Static Methods VS Non-Static methods?

Static methods are a way to provide functionality associated with an entire class, instead of a specific object. Use a *static* method when the method does nothing that affects the individual object of the class.

Why is the main method static?

The main method is always declared as static because the class in which it resides in (i.e. Driver class) never needs to be instantiated with the new command.

This makes sense, since the Driver class with the main method in it might access things, but never the reverse (*nothing ever accesses the Driver class*).

Also, before execution, no objects exist, so in order to start execution, you need a method that is executable, even though there are no objects.