

Part 08 - Classes

Part 08 - Classes

Definition: Class

A cohesive package that consists of a particular kind of compile-time metadata. A class describes the rules by which objects behave. A class specifies the structure of data which each instance contains as well as the methods (functions) which manipulate the data of the object.

Definition: Object

An instance of a class

Defining a Class

Classes are important because they allow you to split up your code into simpler, logical parts. They also allow for better organization and data manipulation.

declaring a function

```
class Cat:  
    pass  
  
fluffy = Cat()
```

This declares a blank `class` called "Cat". It can't do anything at all, because there's nothing to do with it. `fluffy`

Recommendation

Name all your `classes` using PascalCase.
That is, Capitalize every word and don't use spaces.
If it includes an acronym, like "URL", call it "Url".

Fields and Properties

Definition: Field

An element in a `class` that contains a specific term of information.

Definition: Property

A syntax nicety to use instead of getter/setter functions.

Simply, `fields` hold information and `properties` are accessors to that information.

property example

```
class Cat:
    [Property(Name)]
    _name as string

fluffy = Cat()
fluffy.Name = 'Fluffy'
```

1. `class Cat:` declares the start of a class.
 - a. `[Property(Name)]` declares a property around `_name`. You named the property "Name".
 - b. `_name as string` declares a field of `Cat` that is a string called `_name`.
2. `fluffy = Cat()` declares an instance of `Cat`.
3. `fluffy.Name = 'Fluffy'` accesses the property `Name` of `Cat` and sets its value to `'Fluffy'`. This will cause `Name` to set `_name` to `'Fluffy'`.

Fields are not set directly because of security.

Recommendation

Name all your `properties` using `PascalCase`, just like `classes`.

Name all your `fields` using `_underscoresCamelCase`, which is similar to `PascalCase`, only it is prefixed with an underscore and the first letter is lowercase.

There are two other types of `properties`, a `getter` and a `setter`. Technically, a regular `property` is just the combination of the two.

getter/setter example

```
class Cat:
    [Getter(Name)]
    _name = 'Meowster'

    [Setter(FavoriteFood)]
    _favoriteFood as string

fluffy = Cat()
print fluffy.Name
fluffy.FavoriteFood = 'Broccoli'
```

Output

Meowster

If you were to try to assign a value to `fluffy.Name` or retrieve a value from `fluffy.FavoriteFood`, an error would have occurred, because the code just does not exist for you to do that.

Using the attributes `Property`, `Getter`, and `Setter` are very handy, but it's actually Boo's shortened version of what is really happening. Here's an example of the full code.

explicit property example

```
class Cat:
    Name as string:
        get:
            return _name
        set:
            _name = value

    _name as string

fluffy = Cat()
fluffy.Name = 'Fluffy'
```

Because `fields` are visible inside their own class, you can see that `Name` is just a wrapper around `_name`. Using this expanded syntax is handy if you want to do extra verification or not have it wrap exactly around its `field`, maybe by trimming whitespace or something like that first. `value` is a special keyword for the `setter` statement, that contains the value to be assigned.

✔ Property Pre-condition

It is also possible to define a precondition that must be met before setting a value directly through the Property shorthand.

property example

```
class Cat:
    [Property(Name, Name is not null)]
    _name as string

fluffy = Cat()
fluffy.Name = null # will raise an ArgumentException
```

Class Modifiers

Modifier	Description
<code>public</code>	Creates a normal, public class, fully accessible to all other types.
<code>protected</code>	Creates a class that is only accessible by its containing class (the class this was declared in) and any inheriting classes.
<code>internal</code>	A class only accessible by the assembly it was declared in.
<code>protected internal</code>	Combination of protected and internal.
<code>private</code>	Creates a class that is only accessible by its containing class (the class this was declared in.)
<code>abstract</code>	Creates a class that cannot be instantiated. This is designed to be a base class for others.
<code>final</code>	Creates a class that cannot be inherited from.

✓ Recommendation

Never use the `public` Class Modifier. It is assumed to be `public` if you specify no modifier.

class modifier example

```
abstract class Cat:
    [Property(Name)]
    _name as string
```

The `abstract` keyword is the Class Modifier.

Inheritance

i Definition: Inheritance

A way to form new classes (instances of which will be objects) using pre-defined objects or classes where new ones simply take over old ones's implementations and characteristics. It is intended to help reuse of existing code with little or no modification.

Inheritance is very simple in Boo.

inheritance example

```
class Cat(Feline):
    [Property(Name)]
    _name as string

class Feline:
    [Property(Weight)]
    _weight as single //In Kilograms
```

This causes `Cat` to inherit from `Feline`. This gives the members `Weight` and `_weight` to `Cat`, even though they were not declared in `Cat` itself.

You can also have more than one `class` inherit from the same `class`, which promotes code reuse.

More about inheritance is covered in [Part 10 - Polymorphism, or Inherited Methods](#)

Classes can inherit from one or zero other `classes` and any number of `interfaces`.

To inherit from more than one interface, you would use the notation `class Child(IBaseOne, IBaseTwo, IBaseThree)`:

Interfaces

Definition: Interface

An interface defines a list of methods that enables a class to implement the interface itself.

`Interfaces` allow you to set up an API (Application Programming Interface) for `classes` to base themselves off of.

No implementation of code is put inside `interfaces`, that is up to the `classes`.

`Interfaces` can inherit from any number of other `interfaces`. They cannot inherit from any `classes`.

interface example

```
interface IFeline:  
    def Roar()
```

Name:

get

set

This defines IFeline having one method, Roar, and one property, Name. Properties must be explicitly declared in interfaces. Methods are explained in [Part 09 - Methods](#).

✔ Recommendation

Name your interfaces using PascalCase prefixed with the letter I, such as IFeline.

Difference between Value and Reference Types

There are two types in the Boo/.NET world: Value and Reference types. All classes form Reference types. Numbers and such as was discussed in [Part 02 - Variables#List of Value Types](#) are value types.

i Definition: null

A keyword used to specify an undefined value for reference variables.

Value types can never be set to `null`, they will always have a default value. Numbers default value will generally be 0.

Exercises

1. Create a `class` that inherits from more than one `interface`.
2. See what happens if you try to inherit from more than one `class`.

Go on to [Part 09 - Methods](#)