RB120

- ▼ Object Oriented Programming

  Definition

  Paradigm

  Core concepts

  Abstraction

  Simplifying, Essence

  Polymorphism

  Different Input, Same Interface/Methods

  Encapsulation

  Hiding Implementation

  Inheritance

  Relationship Modeling

  Vs Procedural

  Compartmentalizing
  Independant Sub-Programs
  Easier to maintain & update
  Reusability of Code
  Concrete way to model problem
  Abstracting the problem to nouns, verbs

  For the programmer/Design Level Starter

  Objects can be described as nouns

  Methods can be described as verbs

- ▼ Classes & Objects

  keywords: Outline, molds, behaviors, attributes, instance (object)

  Objects have value

  Non-objects don't

Class - Object Relationship

- class: Mold
- object: Instance of Mold
- The class is core to creating an object
- The object is what OOP revolves around
    - Messaging
    - State
    - Behaviors

Class as template for objects

- State
    - Instance variables keep track of state
    - Individual objects can contain different information
- Behavior
    - Instance methods

Instantiation

- Creating an instance of a class
- How do you do it

Instantiation with `initalize`

- Private, constructor, relation to `::new`
- Arguments
    - If needed
    - Can trigger certain code
- When it's required
    - Relation to instance variables
- Triggers when we call `::new`
    - Instantiation of new objects

▼ Encapsulation

- ▼ Definition

  Encapsulation is a concept where we seek to hide the implementation details from the user or other parts of our program.

Hiding implementation details

Benefits

Data Protection

Limiting what is accessible in the public interface

New layer of abstraction

Practical Examples

Variable

Object

Method

Class

Private, Protected and Private

Encapsulation Vs Abstraction

Encapsulation hides

Abstraction Exposes

- ▼ Abstraction

Exposing the essence

Design Level

Design Process

- ▼ Inheritance

Definition

Creating relationship that model hierarchical domains

Benefit

DRY code

Class based inheritance "is-a"

Define a class by taking components of a general class and making it more specialized

1 Superclass, Subclass using the `<` symbol

Class inherits from a class, objects do not inherit

Interface inheritance "has-a"

Supports multiple inheritance

The interface portion is inherited by the class

No limit to how many modules

Module

Cannot instantiate objects from module

Used for namspacing and grouping common methods together

Every class inherits from `Object` and the module `Kernel`

Implementation

A subclass can override a superclass's method

The last included module within a class will be the second place Ruby looks for when invoking a method, if it isn't found first in the object's class

`super`

`super` calls the method of the same name up the inheritance hierarchy and invokes it

Ruby will look through the **method lookup path** to find the method and invoke it before returning to the current method invocation (to quickly identify the method lookup path, call `::ancestors` on the class)

`super` vs `super()`

- if `super` is invoked without specifying argument, and the method it calls up the inheritance hierarchy takes an argument, the arguments from the current method invocation will automatically be forwarded

- we can decide which arguments get passes to the method invocation `super` calls by either sending them with `super` (ex: `super(arg1, arg2)`) or informing that we're not sending any arguments with `super()`.

▼ Polymorphism

Purpose

Concept of different objects responding to the same interface (method invocation).

the method may have different implementation based on the object, but the method name is the same.

Via Inheritance

class can inherit from supercalss

method overriding by defining custom implementation

module

module can allow behaviors to be shared among different classes

different objects can invoke the same method

Duck typing / Public Interface

Ruby concerns itself with what an object responds to more than what the object is (client code)

A common method can be implemented in different classes, the objects can be treated as generic

Does not have inheritance relationship

▼ Instance

Variables

can hold any value, data structure, and even custom objects

Purpose

Hold data for the object (hold state for an object's attribute)

Initialization

`initialize`

return value of method on calling object

technical comm: A `Person` object's `name` attribute can be saved into a `@name` instance variable as a string.

Scope: Object Level

- Exist as long as the object exists
- Exists where the object exists
  - Including instance methods
- returns `nil` if uninitialized

Inheritance

- Can be inherited from superclass
  - Must be initialized, otherwise, returns `nil`
- Can be initialized via modules
  - The method must return an instance variable for it to be assigned to the object

Methods

In relation to objects

- all objects have the same behaviors even if their state differ
- is defined within a class
- have access to instance variables (the object scope has access to it's instance variables at all times)

Accessor Methods/ Getter and Setter

Makes the value of the instance variable available in the public interface (exposing information)

General:

`def get_name @name end`

`def set_name(name) @name = name end`

Conventions:

`def name @name end`

`def name=(name) @name = name end`

called as `object.name = 'name'` (syntactic sugar)

Attr*

To improve security, wrap instance variables in an `attr*` method instead of using the previous forms. From here, instead of referencing the instance variable directly, you simply call the method for the attribute.

The `attr_accessor` method takes a symbol as an argument and create the method name for the getter and setter methods.

The `attr_reader` creates a getter method for the attributes it has as a symbol arguments.

The `attr_writer` creates a setter method for the attributes it has as a symbol arguments.

> Special requirement when using an `attr_writer` method. In order to let Ruby know you are updating the value of the object's instance variable, you must specify `self` before the method name (ex: `self.name`). This ensures we aren't creating a new local variable, but instead, updating the value the instance variable references.

Private, Protected, Public

By using the `private` or `protected` method invocation

By default, the methods defined in a class are considered **public**, available to the user and other methods defined within the class, as long as they know what method to call and which arguments to pass in.

> All methods are by default public, except for `#initialize`, which is always private

A method that is set to **private** is only available within the class. These methods cannot be called outside the class. You need a `self` with a private setter method.

A **protected** method is not available outside the class. It could always be called with an explicit receiver. Protected methods are useful when we want to access another object's state or behavior within the class, and it's not available outside the class.

`to_s`

`puts` calls this method automatically. If it's not overriden in our class, `puts` will call `Object#to_s` which returns a string encoding of the object.

string interpolation automatically calls `to_s` on the object output format with an array

it will print on separate lines the return value of calling `to_s` on each element of the array

`Object#to_s`

Availability - in all objects, since it's defined in `Object` and is part of all the custom classes inheritance chain.

▼ Class

A class is an object

Variables

Syntax: `@@`

All objects of the class (and sub-classes) share the 1 copy of the class variable

Used for class-level attributes

Scope: Class level

Accessible via class and instance methods

Purpose: Data that pertains to the class and not instances of the class

Inheritance

Sub-class can override class variables defined in the super-class.

If the super-class class variable is reassigned or mutated, all other sub-classes with see this change reflected.

Methods

Defined at the class level

Available to the class and must be called on the class object itself

Not available to instances of the class

Syntax: prepend method with `self`

Used for methods that don't pertain to individual objects

▼ Constants

Purpose: used for data that we don't intend to change. Constants can be mutated or reassigned, and Ruby will give you a warning if you do so. This should be avoided in practice.

Definition: uppercase letter at the start of the constant's name, convention say to make it all upcased.

Scope

Lexical scope means where it's defined determines where it's available. They are not evaluated at runtime, meaning where they are initialized and scoped is important to consider.

Inner scope can access outer scope.

Inheritance

If it's not found in the current scope, Ruby will look in the inheritance hierarchy.

From

superclass: Ruby will find it thanks to the inheritance hierarchy.

seperate classes: We can tell Ruby where to find the constant with the namespace resolution operator (`::`)

Invoked from a module:

When a method from a module references a constant, it must be within the module for the method to access it.

The inheritance hierarchy doesn't help when it comes to constants since constants have lexical scope (where they are initialized defines where it's available).

We can rely on the namespace resolution operator (`::`) to reference the constant in question when it's not in scope.

▼ `self`

is a keyword

the object the current method is attached to (refers to the calling object)

in a module, it would be the module object itself rather than the class it's included in used to be explicit about our intentions in terms of what object we are referring to in the current scope

class method

instance method

> `self.weight=` is the same as `sparky.weight=`

▼ Module

Purpose

A module is a **container** for different objects and methods in Ruby. It helps in DRYing up our code. A convention for naming modules that group a common type of behavior is to end the module name with 'able', like Swimmable.

Interface inheritance

It helps support multiple inheritance, through interface inheritance. This is seen as a "has-a" type of ability inheritance.

> This is done by including the module in a class via the `include` invocation.
>
> A module has no limit to how many classes it can be mixed into. Whereas a class can only inherit from 1 super-class.

Modules are also used for **namespacing**

> Namespacing is grouping objects and methods together in a container (the module).
>
> This helps in avoiding filenames to collide.

A module **cannot instantiate** objects. A class within a module can.

The **namespace resolution operator** (`::`) helps in retrieving specific objects from the module and invoking methods from the module directly.

Why Avoid using `self` ?

> `self` refers to the module if prepended to a method definition within a module
>
> when included in a class, it will still reference the module rather than the class itself

▼ Method lookup path

When we invoke a method on an object, Ruby looks for the method in the object's class. If it doesn't find it there, it goes the modules included within the class, starting with the most recent one. If none of the modules have the method in question, Ruby goes up to the super-class and repeats this process until it either finds the method or returns a `NoMethodError`.

The list of places that are always part of this path include `Object`, `Kernel`, `BasicObject` as these are the classes and module that all classes inherit from in Ruby.

If we wish to quickly identify the method lookup path for a certain class, we can call `::ancestors` on the class and it will be returned.

▼ Fake Operator

A fake operator is actually a method. Most of the time, it will have a form of syntactic sugar to hide it's implementation details as well as looking like a regular operator. All methods can be overwritten to implement custom logic within a class of our creation.

Consider the return object if we piggyback on a fake operator already defined in Ruby.

Consider the context it's used in regularly and the interface it'll provide for the user.

▼ Equivalence

Two main elements come into play when we talk about comparison/equality:

1. **Object Value**
2. **Object ID**
3. *** Same Value and Class, Different Object (less used)**

`BasicObject#==`: *At the Object level,* `#==` *returns* `true` *only if obj and other are the same object.*

For most objects, the `==` operator compares the **values** of the **objects**, and is frequently used this way.

When you define `==` in your class, you will also get a `!=` for free.

Syntactic sugar version (`str1 == str2`) vs explicit (`str1.==(str2)`)

`===`

Used by `case` statements more often.

Relies on the implementation of the object's class to provide a meaningful comparison.

Generally used for pattern matching (does this object belong to this group ?)

`BasicObject#equal?`

Used to check if the two variables are pointing to the **same object in memory**.

Should not be overwritten

`Object#object_id`

call on a variable and it will return an integer representation of the object it's referencing location in memory

▼ Collaborator Objects

Objects that are stored as state within another object are also called "collaborator objects". The object stored in the instance variable acts like state.

A Library has-a Book. "has-a" relationship

▼ We've set `bob`'s `@pet` instance variable to `bud`, which is a `Bulldog` object. This means that when we call `bob.pet`, it is returning a `Bulldog` object. **`bob` has a collaborator object stored in the `@pet` instance variable.**

```
class Person
  attr_accessor :name, :pet

  def initialize(name)
    @name = name
  end
end

bob = Person.new("Robert")
bud = Bulldog.new             # assume Bulldog class from previous assignment

bob.pet = bud
```