

Essential Ruby

BALA PARANJ



Table of Contents

1. [Introduction](#)
2. [Object Oriented Programming](#)
 1. [What is a Class?](#)
 2. [What is an Object?](#)
 3. [Creating an Object](#)
 4. [State and Behavior](#)
 5. [Hidden Instance Variables](#)
 6. [Sending a Message to a Receiver](#)
 7. [Message Passing](#)
 8. [Inheritance](#)
 9. [Module](#)
3. [Essential Ruby](#)
 1. [Symbol](#)
 2. [The yield Keyword](#)
 3. [Everything is Not an Object](#)
 4. [Top Level Context](#)
 5. [Code Execution](#)
 6. [Binding](#)
 7. [Pseudo Variables](#)
 8. [The Default Receiver](#)
 9. [Message Sending Expression](#)
 10. [The Self at the Top Level](#)
 11. [The Dynamic Nature of Self](#)
 12. [When does self Change?](#)
 13. [The main Object](#)
 14. [Message Sender at the Top Level](#)
 15. [Top Level Methods](#)
 16. [Same Sender and Receiver](#)
 17. [Private Methods](#)
 18. [Scope of Variables](#)
 19. [Scope of Variables Redux](#)
 20. [Every Object Has a Class](#)
 21. [Instance Methods and Instance Variables](#)
 22. [Block Object](#)

- 23. [Closures](#)
- 24. [Focus on Messages](#)
- 25. [Self and Scope](#)
- 26. [Retry Library](#)
- 4. [Basics for Ruby Object Model](#)
 - 1. [introduction](#)
 - 2. [Class Methods](#)
 - 3. [Singleton Methods](#)
 - 4. [Objects and Inheritance Hierarchy](#)
 - 5. [Class, Object and Module Hierarchy](#)
 - 6. [Hierarchy of Class Methods](#)
 - 7. [The Method Lookup](#)
- 5. [Object Oriented Programming Revisited](#)
 - 1. [Modeling the Real World](#)
 - 2. [Resources](#)
- 6. [Key Takeaways](#)
- 7. [Essential Book Series](#)

Essential Ruby

This book covers the most essential concepts in Ruby. The goal is to provide a solid foundation to build upon. This book distills my Ruby programming knowledge into a concise and easy-to-read format. Repetition is key to learning Ruby. We will visit the concepts from different angles. You will get the most benefit out of the book if you work through every example as you read through the book.

There are four sections in this book. Section 1 introduces the basic Object Oriented Programming (OOP). Section 2 covers the required concepts to understand Ruby. Section 3 provides the basics needed to learn the Ruby Object Model. Section 4 revisits OOP concepts required to clarify the reader's questions.

Each early chapters are as small as possible and focuses on explaining one concept at a time. Gradually the later chapters increase in complexity and introduce readers to subtle concepts. These subtle concepts are not discussed in any of the current books on Ruby. This book uses Ruby 2.3.0.

Intended Audience

This book is for experienced programmers of other languages as well as new programmers. Experienced programmers will learn how Ruby differs from other Object Oriented languages. New programmers will build a solid foundation for learning Ruby. Programmers familiar with Ruby will learn about some of the common misconceptions.

Programmers familiar with Ruby can jump into the following chapters:

1. Message Passing
2. Message Sender at the Top Level
3. Same Sender and Receiver
4. Private Methods
5. Focus on Messages
6. Modeling the Real World

Technical Reviewers

[Jesus Castello](#), [Michael Heinrich](#), [Maciej Mensfeld](#), [Csaba Nagy](#), [Marko Čilimković](#),
[Philip Hallstrom](#) and [Gaurab Paul](#)

About the Author



Bala Paranj has a Master's degree in Electrical Engineering from Wichita State University. He began working in the IT industry in 1996. He started his career as a Technical Support Engineer and then became a Web Developer using Perl, Java and Ruby.

He is available for freelance work. Please contact him at support@zepho.com or via [Ruby Plus](#). He is also working on screencasts based on this book. If you want notification about the release, please contact him.

Image Credits [Pixabay](#)

Object Oriented Programming

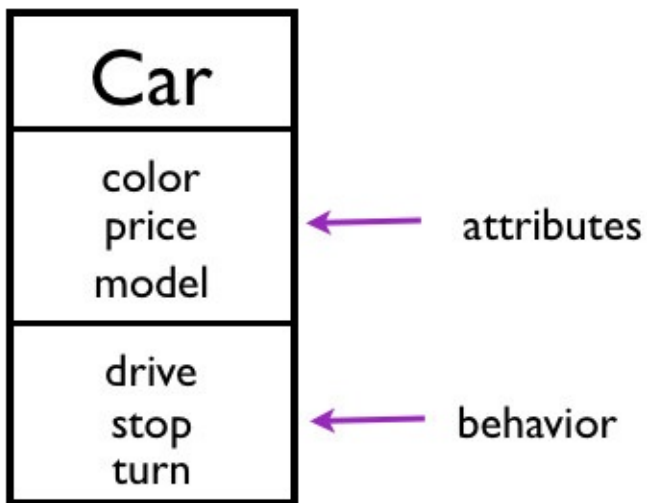
This section covers the basics of Object Oriented Programming. Even if you are familiar with Object Oriented Programming, I recommend you to read through all chapters to get the most out of this book.

Class

In this chapter, you will learn the basics of a class and how to define them in a program. We will look at the concept of car and how to represent a car in a program.

Concept of Car

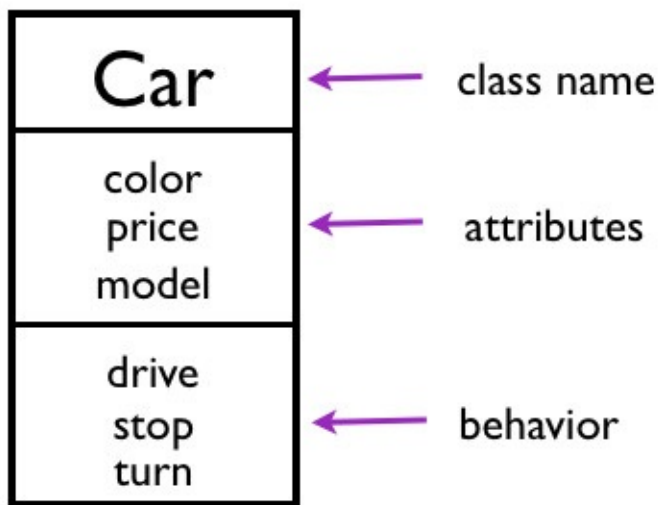
A car has attributes such as color, price, model and so on. You can exercise certain behavior such as drive, stop, turn, etc on a car.



Concept of Car

Representing a Car

We can represent the concept of a car using a class.



Class Car

A class has a class name. In this example, the class name is Car.

Defining a Class

Let's define a Car class.

```
class Car  
end
```

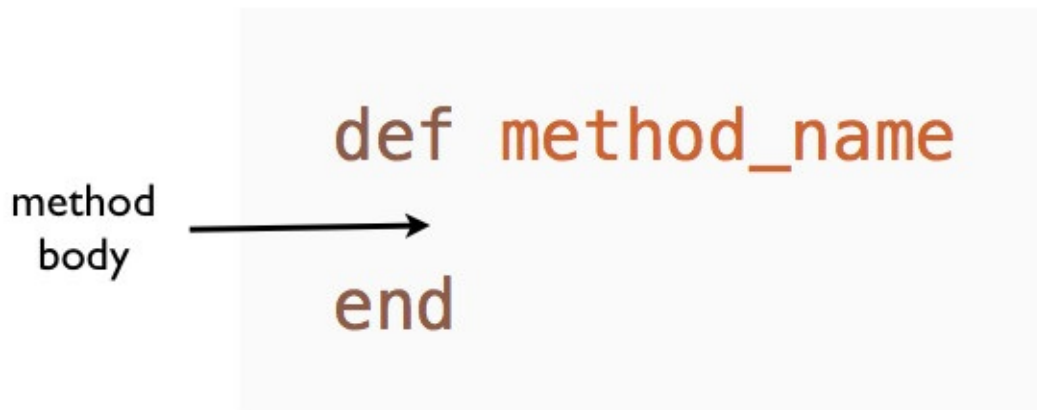


We use the **class** keyword followed by the class name to define a class. In this case, Car is the class name.

Representing the Behavior

We can represent the drive behavior of a car by defining a **drive()** method for the Car class.

```
class Car
  def drive()
    return 'driving'
  end
end
```

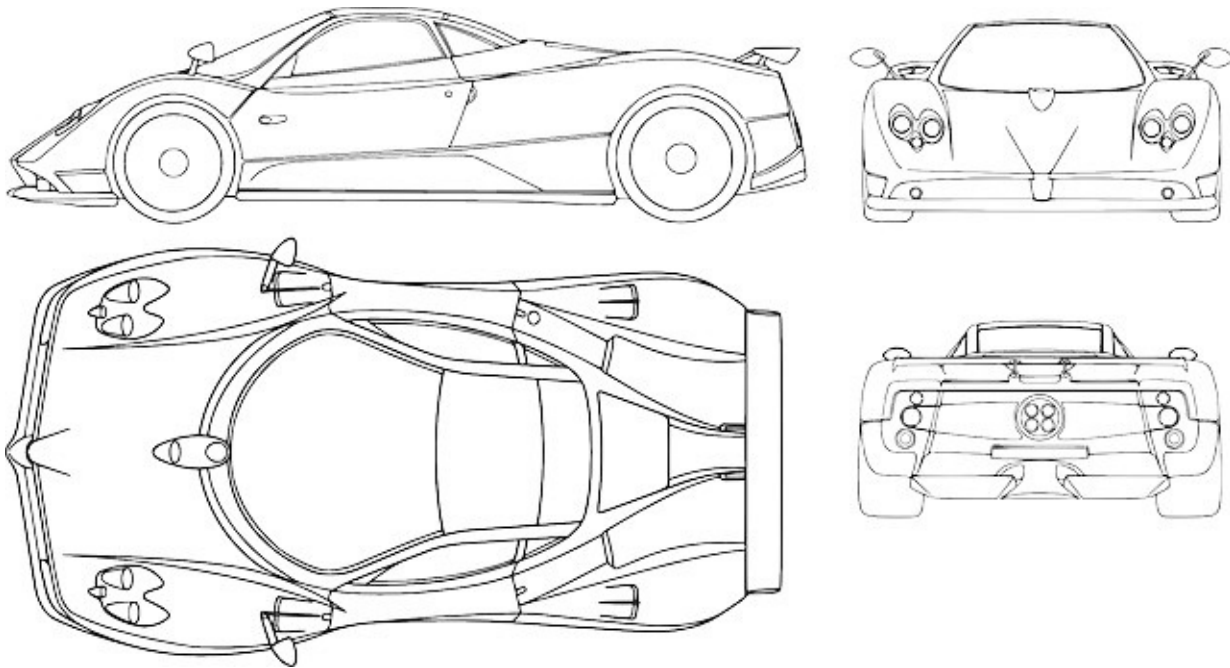


Defining a Method

The methods provide the behavior for the objects. We use the **def** keyword followed by the method name to define a method. The body of the method becomes the implementation.

Car Class Analogy

A Car class is like a car blueprint.



A car blueprint is used to manufacture many cars.

Key Takeaways

- A car class acts as a template used to create cars.
- A car class describes the behavior and attributes of a car.



Define a method in the Car class to stop a car.

Summary

In this chapter, you learned the concept of class. A class describes the behavior and attributes of a certain concept.

Object

In this chapter, you will the learn the basics of an object and how to create them.

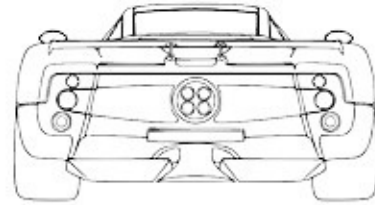
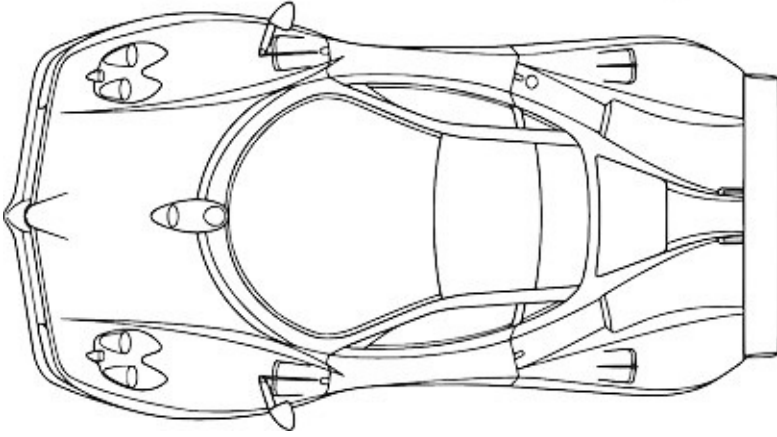
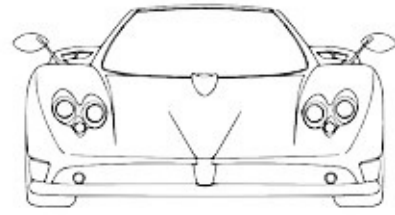
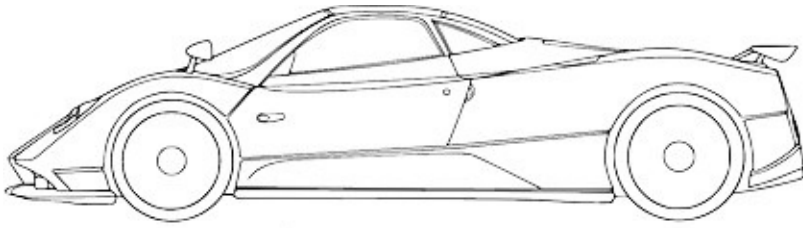
Car Class

In the previous chapter, we defined a Car class. It has **drive** method that provides the behavior for the Car class.

```
class Car
  def drive
    return 'driving'
  end
end
```

Car Class Analogy

Car class is like a car blueprint.



Car Object Analogy

Car object is like a car made using the car blueprint.



We cannot drive a car blueprint. We need a real car to exercise the drive behavior. How do we create a car that we can drive in a Ruby program?

Car Object

We use the blueprint to create an object. The process of creating an object from a class is called instantiation. In Ruby the **new** method is used for instantiating an object. Let's create an instance of the Car class.

```
class Car
  def drive
    return 'driving'
  end
end

car = Car.new
```

We now have a car object. How do we invoke the drive behavior of the car object?

Sending the Message

We invoke the drive behavior by sending a message to the car object. Let's send the **drive** message to the car object.

```
p car.drive
```

The dot notation is used to send a message to any object. This prints:

```
driving
```

From now on, you will hear *calling a method* and *invoking a method*. These terms mean the same thing as sending a message.

Return Value of Method

In Ruby, the last statement executed is the return value of a method. We can simplify the program by removing the return statement in the **drive** method.

```
class Car
  def drive
    'driving'
  end
end

car = Car.new('red')
p car.drive
```

This will still print:

```
driving
```

Key Takeaway

- Every object is an instance of a class.

Summary

In this chapter, we created a car object by creating an instance of the Car class. We invoked the instance method by sending a drive message.

Creating an Object

In this chapter, you will learn how to create an object and the initialization process.

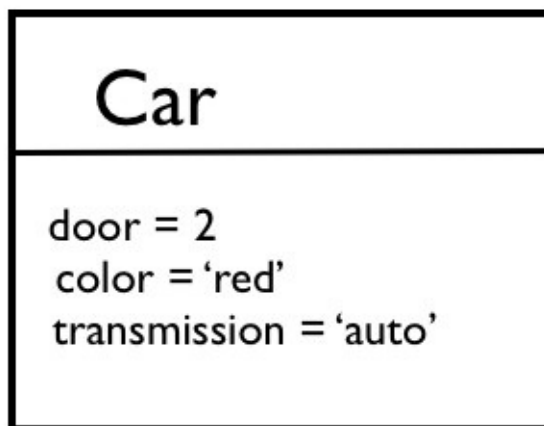
Initialization

Initialization is the process of preparing an instance of a class. This process involves setting an initial value for each instance variable on that instance. Any required setup is also performed before the new instance is ready for use. For the car example, it will answer questions such as:

- How many doors?
- What is the color?
- Auto or Manual transmission?

We can say:

```
door = 2  
color = 'red'  
transmission = 'auto'
```



Initialization of Car Object


The code answers those questions by initializing the variables. How do we initialize instance variables? Where do we initialize them? Let's discuss about them now.

Instance Variable

An instance variable has a name beginning with `@`. The instance variables comprise the state unique to an object. We can have two car objects with different colors such as red and black. Let's say that the instances of the Car class have an instance variable called **color**. We can initialize the color instance variable like this:


```
class Car
  def initialize(color)
    @color = color
  end
end
```

```
class Car
  def initialize(color)
    @color = color
  end
end
```



Instance Variable

```
class Car
  def initialize(color)
    @color = color
  end
end
```



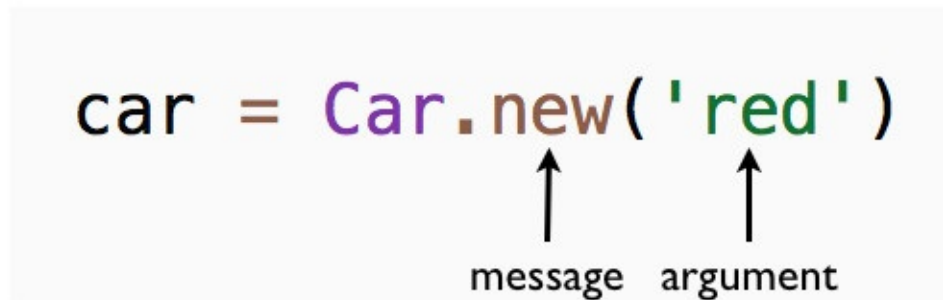
State Unique to an Object

The parameters to the initialize method is used to assign initial values to the attributes of an object. Instance variables spring into existence when they are initialized for the first

time. In this example, we initialize the color instance variable `@color` in the **initialize** method. We can now create an instance of a car object with a specific color.

```
car = Car.new('red')
```

We send the **new()** message to the Car class with the color **red** as the argument.



```
car = Car.new('red')
```

The diagram shows the code `car = Car.new('red')` with two arrows pointing upwards. The first arrow points to the `new` method and is labeled "message". The second arrow points to the string `'red'` and is labeled "argument".

Creating an Instance of a Car

Ruby calls the **initialize** method with **red** as the argument. In the **initialize** method, we store the value of color, red, in the instance variable `@color`.

Fabio Asks

What happens if you call the initialize method?

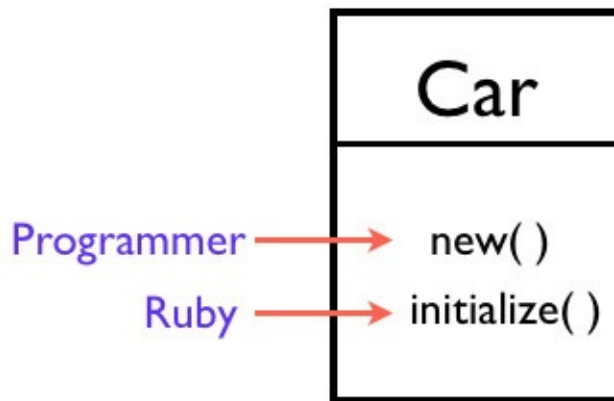
Let's call the initialize method.

```
Car.initialize('black')
```

You will get the error:

```
NoMethodError: private method 'initialize' called for Car:Class
```

We cannot call the **initialize** method. Only Ruby can call it. To initialize an instance of any class, always use the **new** method.



Creating a Car Object

Uninitialized Instance Variables

Instance variables have nil value if they are not initialized. The nil represents the concept of nothing in Ruby. Let's look at an example.

```
class Car
  def initialize(color)
    @color = color
  end

  def price
    @price
  end

  def drive
    return 'driving'
  end
end

c = Car.new('red')
p c.price
```

This prints nil. The Car class has a new method called **price** that returns the price instance variable of the car. Since it was not initialized anywhere in the Car class, it has nil value.

Rhonda Asks

Why is it called an instance variable?

Because the variable is unique to a specific instance of a class.

Summary

In this chapter, you learned about the initialization of an object. Initialization process instantiates a specific object with certain attributes.

State and Behavior

Instance variables represent the state of an object. The instance methods defined in the class provide the behavior for an object. Every object has its own unique state. But they share the same instance method behavior defined in the class. We can have a black car that has its own color that shares the drive method.

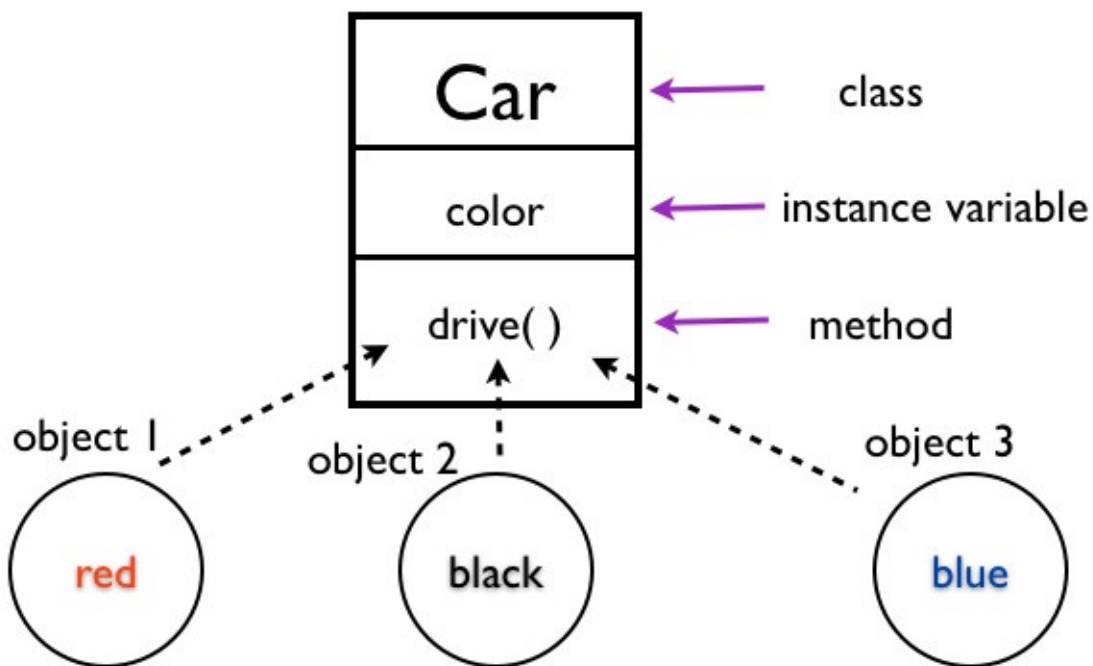
```
class Car
  def initialize(color)
    @color = color
  end

  def drive
    'driving'
  end
end

black_car = Car.new('black')
p black_car.drive
```

This prints:

```
driving
```



Multiple Objects Share Single Method

Rhonda Asks

Why is the method defined in the Car class called as an instance method?

Because, we need an instance of the Car class to call the method.

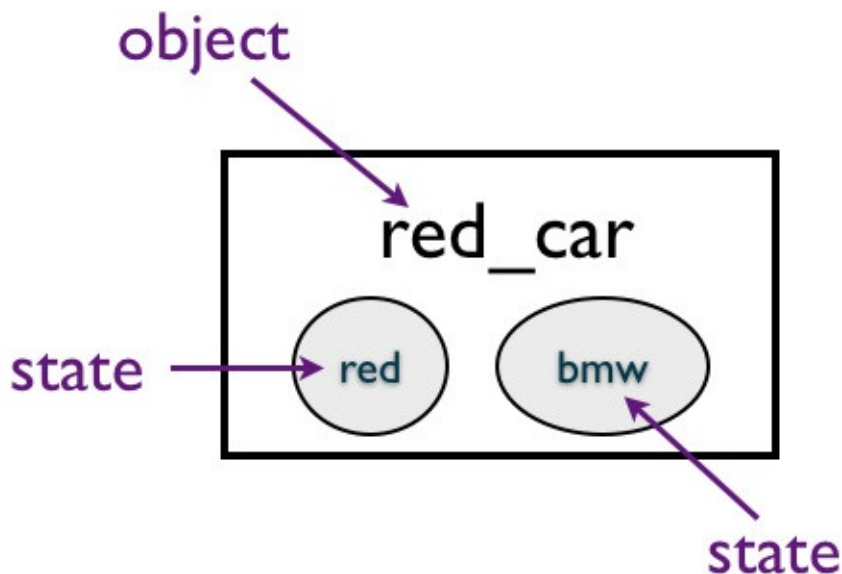
Instance Variables Live in the Object

We can ask Ruby to list all the instance variables for car object.

```
p car.instance_variables
```

This prints:

```
[:@color]
```



Instance Variables Live in Object

The illustration shows that you can define many instance variables such as color, model of the car etc. for a car. Let's print the instance variables in the Car class.

```
p Car.instance_variables
```

This prints:

```
[]
```

The result array is empty. There are no instance variables in the Car class.

Instance Method Lives in the Class

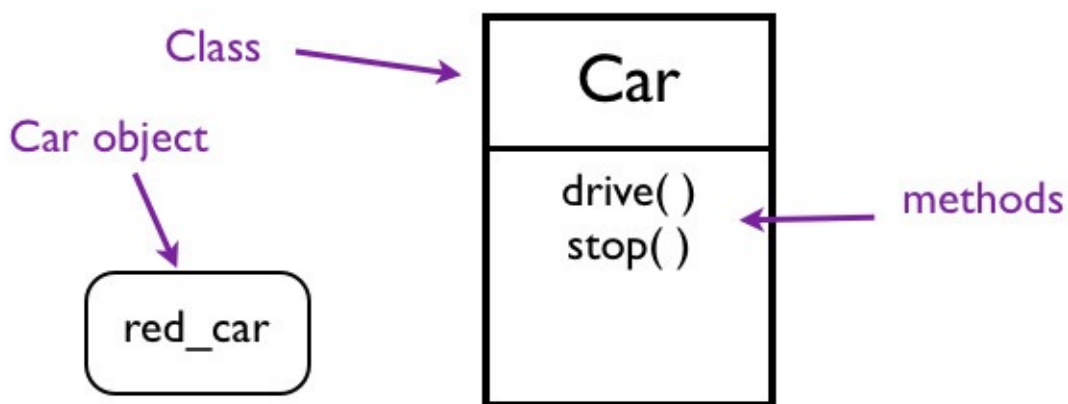
We can ask Ruby for the list of instance methods in the Car class.

```
p Car.instance_methods(false)
```

This prints:

```
[:drive]
```

The Car class defines the **drive** instance method, so it shows up in the output. We pass **false** to the `instance_methods` to print instance methods found in the Car class only.



Instance Methods Live in Car Class

The illustration shows that you can define many instance methods in the Car class. If you print the instance methods for the car object:

```
p car.instance_methods(false)
```

You will get an error:

```
NoMethodError: undefined method 'instance_methods' for #<Car:0x00ca0 @color="red">
```

Key Takeaways

- Instance methods live in the class.
- Instance variables live in the object.
- Instance variables are unique to each object.

Summary

In this chapter, you learned about the state and behavior and where the instance methods and instance variables live.

Hidden Instance Variables

In this chapter, you will learn about inspecting an object, customizing the inspect message and visibility of instance variables.

Inspecting an Object

We can print the car object to inspect it.

```
class Car
  def initialize(color)
    @color = color
  end

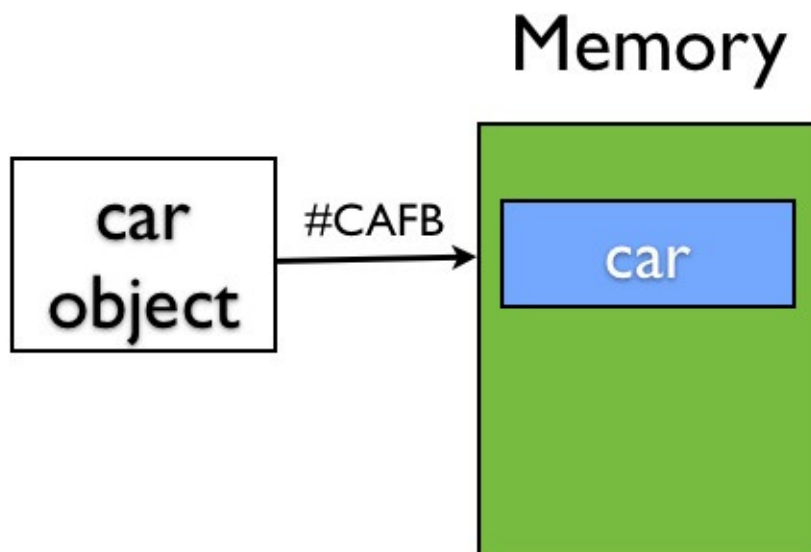
  def drive
    'driving'
  end
end

car = Car.new('red')
p car
```

This prints:

```
#<Car:0xcafb @color="red">
```

This shows that the car instance has the color instance variable with the value red and the memory location #CAFB of the car object.



Car Object in Memory

Customize Inspecting an Object

We can provide a custom implementation of `to_s` method.

```
class Car
  def initialize(color)
    @color = color
  end

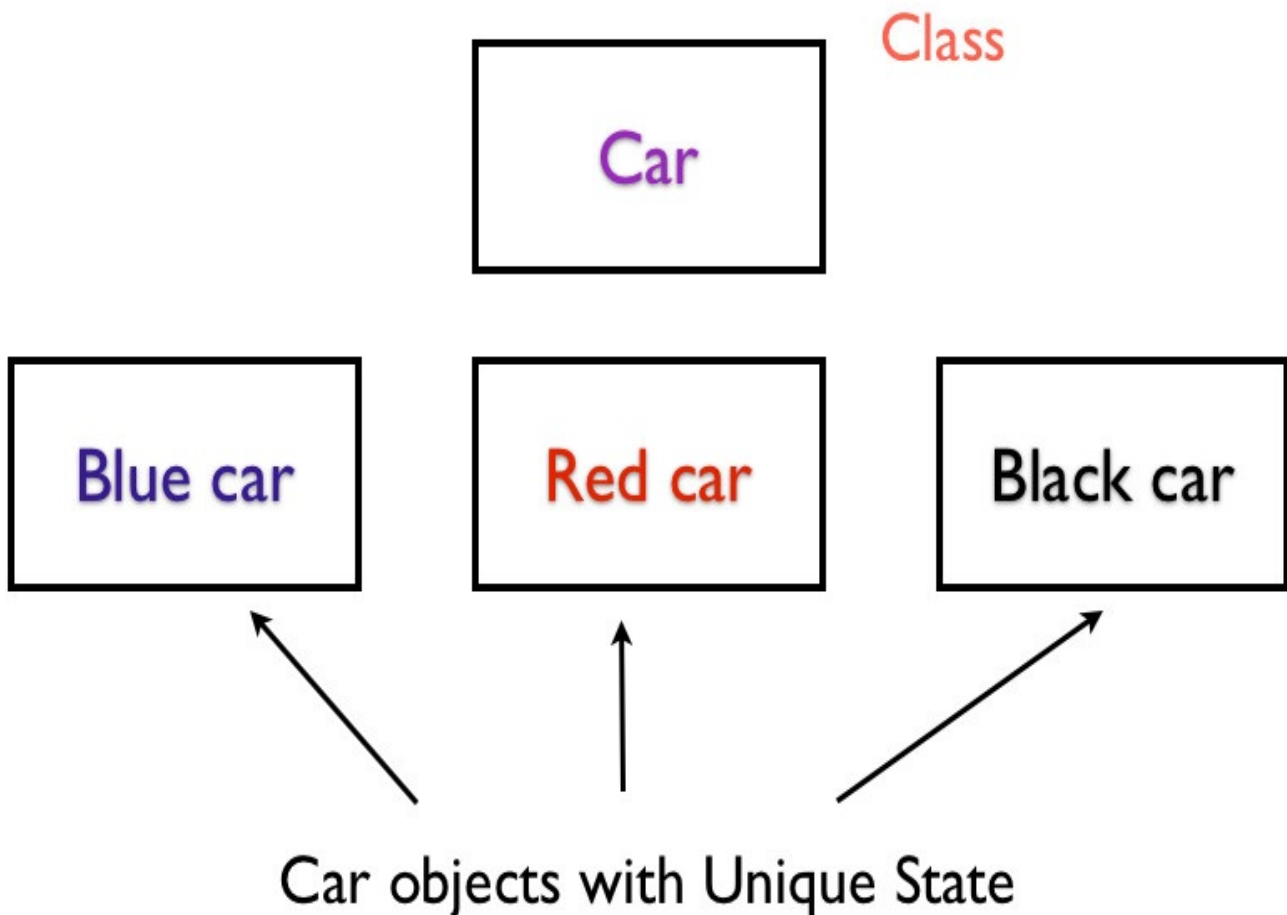
  def drive
    'driving'
  end

  def to_s
    "I am a #{@color} car"
  end
end

car = Car.new('red')
puts car
```

The `to_s` provides string representation of the car object. This prints:

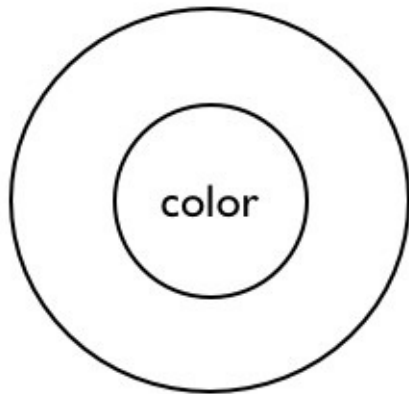
```
I am a red car
```



Visibility of Instance Variable

The instance variables are not visible from the outside of an object.

Car Class



Hidden Attribute

Let's see what happens when we access the color instance variable.

```
class Car
  def initialize(color)
    @color = color
  end

  def drive
    'driving'
  end
end

car = Car.new('red')
p car.color
```

This gives an error:

```
NoMethodError: undefined method 'color' for #<Car:0x007fbc0 @color="red">
```

We can define a color method that returns the color instance variable.

```
class Car
  def initialize(color)
    @color = color
  end

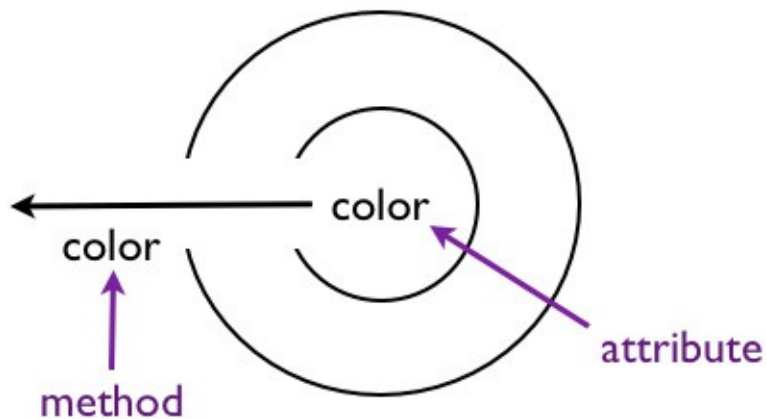
  def color
    @color
  end
end

car = Car.new('red')
p car.color
```

This prints:

red

Car Class



Expose Hidden Attribute

Ruby has accessors that expose variables to the outside. So, instead of defining our own method, we can use **attr_reader**.

```
class Car
  attr_reader :color

  def initialize(color)
    @color = color
  end
end

car = Car.new('red')
p car.color
```

This prints:

red

Key Takeaways

- By default, instance variables are hidden from the outside.
- We can expose an instance variable via an accessor.
- You can over-ride the **to_s** method to customize the inspect message.

Summary

In this chapter, we inspected the color instance variable of the car object. We also discussed the concept of accessing the instance variables via accessors.

Sending a Message to a Receiver

In this chapter, you will learn about the receiver object and sending messages to a receiver.

Hail Taxi

Let's write a simple program to hail a taxi.

```
3.times do  
  puts 'Taxi'  
end
```

This prints:

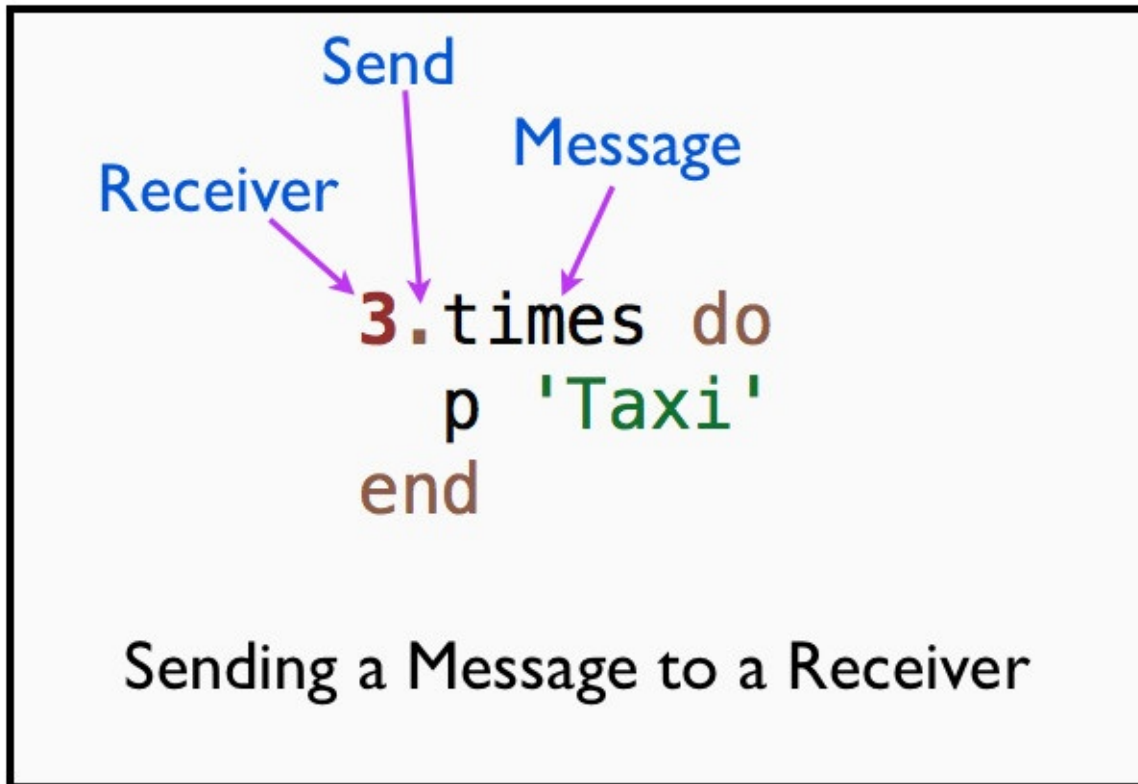
```
Taxi  
Taxi  
Taxi
```



Hail Taxi

Receiver and Message

Let's identify the receiver and the message in this simple program. The number **3** is the receiver. The **times()** method is the message. The dot represents **sending** the message to the receiver.



The dot notation is used when sending a message is explicit.

Receiver Object

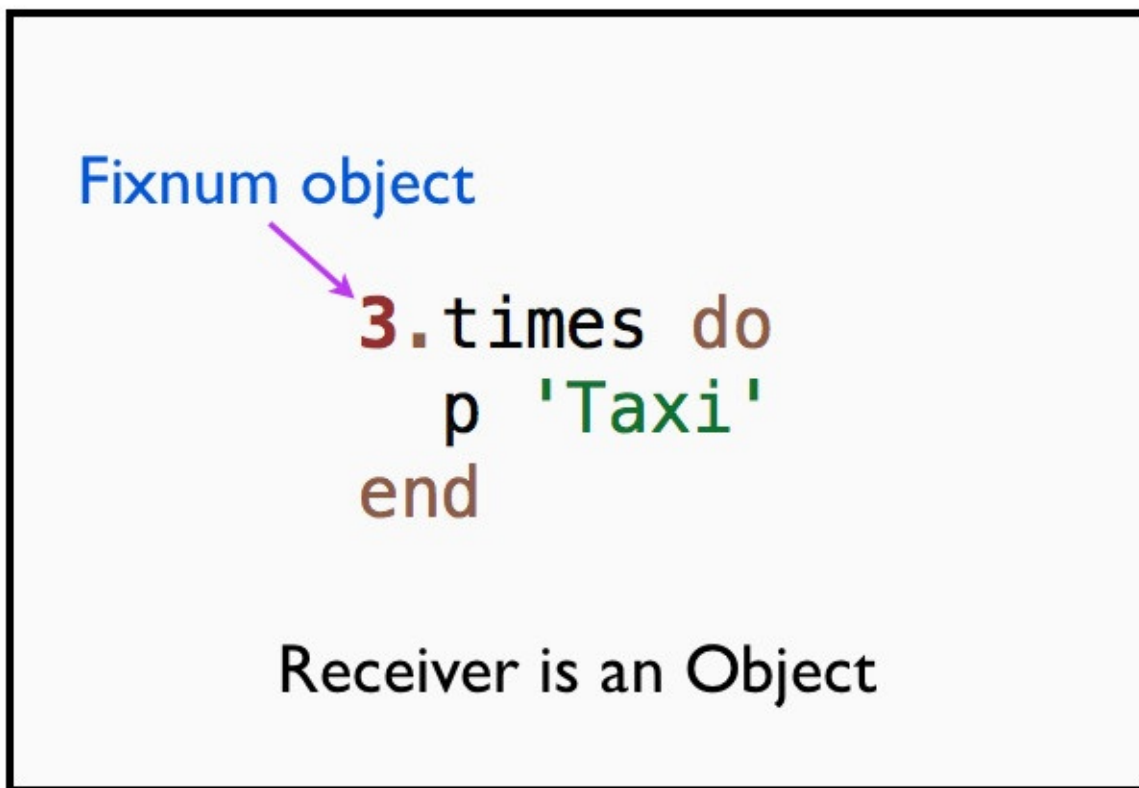
The receiver is an object. Let's find out the class of the receiver object for our simple program.

```
puts 3.class
```

This prints:

```
Fixnum
```

This demonstrates that **Fixnum** object 3 is the receiver object. It receives the **times()** message.



Fabio Asks

Why is it called the receiver?

Because this object would have received the message that caused the method to execute.

Rhonda Asks

Why do we need a receiver?

In a OO language like Ruby, everything happens by sending messages to an object. This object is the receiver. In the next chapter, we will discuss more about message passing.



Identify the receiver for the car program we wrote in the previous chapter.

Summary

In this chapter, we saw:

1. What is a receiver?
2. What is a message?

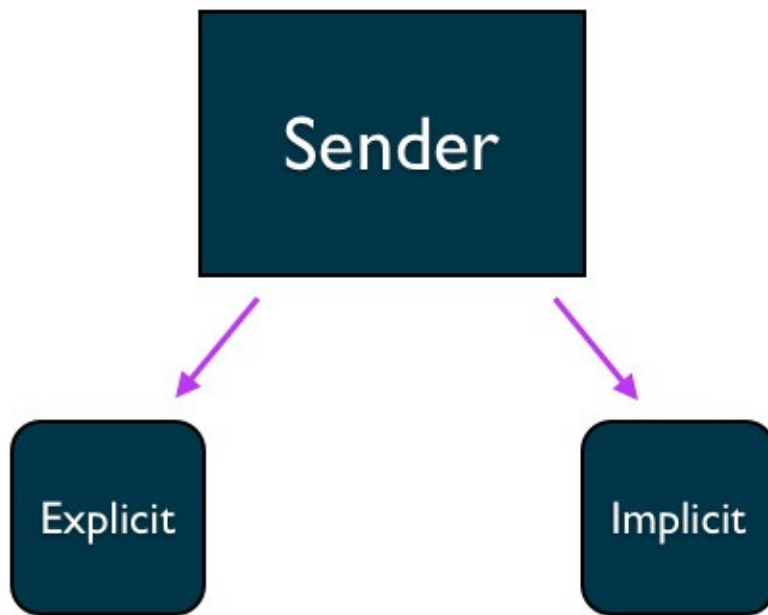
In the next chapter we will answer questions such as, what is a sender? Who is sending this message?

Message Passing

In this chapter, you will learn the basics of message passing. We will discuss about message sender and the difference between intent and implementation. We will also see an example for an explicit sender.

Explicit and Implicit Sender

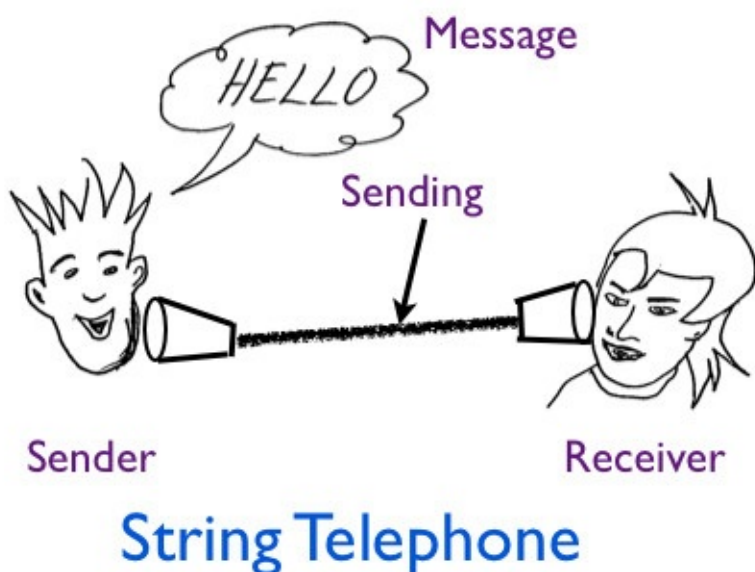
This chapter will examine a program with an explicit sender. In a later chapter we will revisit our hail taxi program to identify an implicit sender.



Explicit and Implicit Sender

Sender Sends a Message to a Receiver

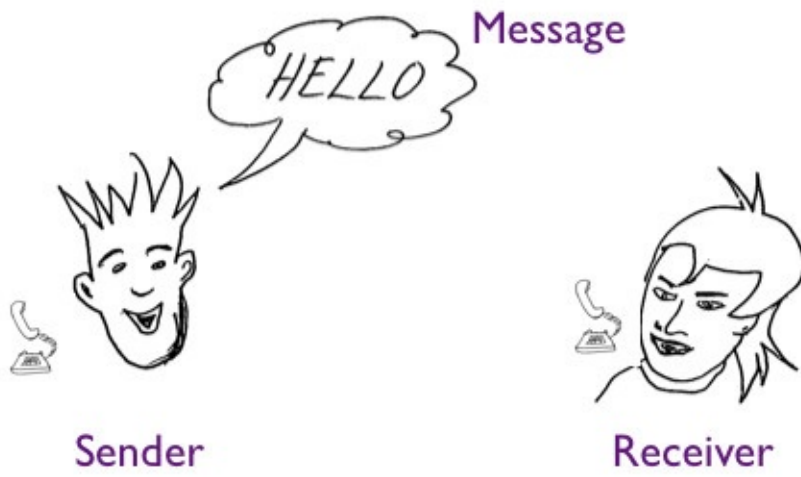
In Ruby, the message-sending metaphor is the basis for object interaction.



Why is message-sending metaphor used? Because it provides modularity.



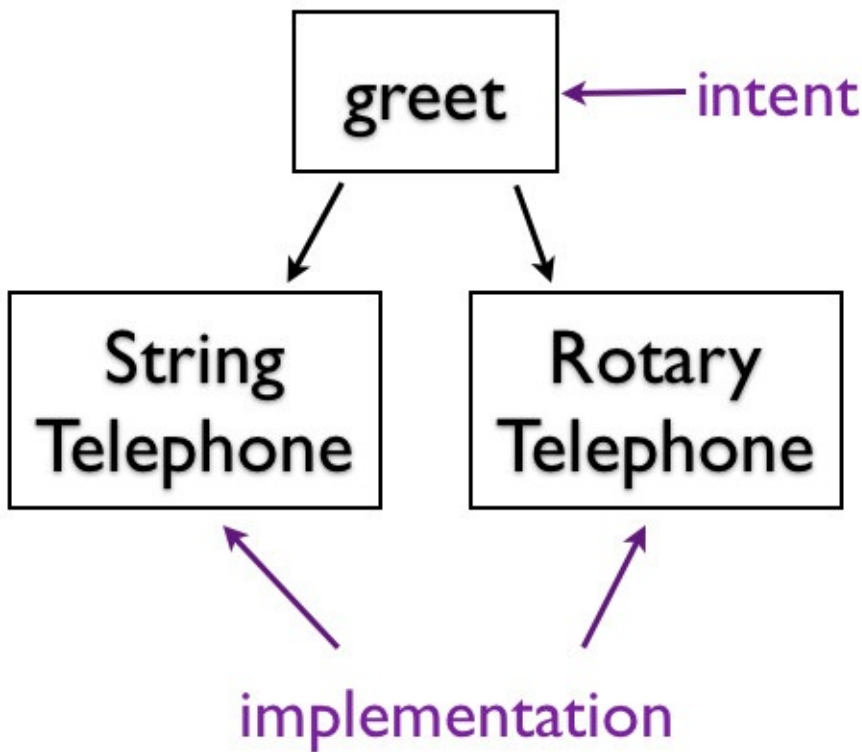
The metaphor decouples the intent of a message from the implementation details of the method. This allows us to vary the implementation without impacting the objects that send the messages.



Rotary Telephone

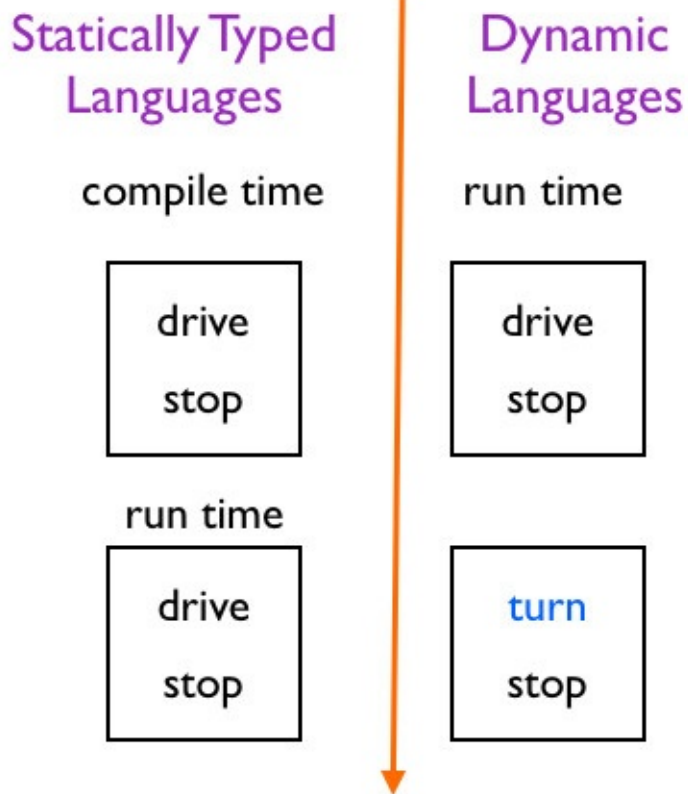
Intent vs Implementation

The messages are more important than the objects involved in the interaction. We express intent by means of the messages we send. The name of the message embodies the intent of a message. The implementation details are hidden behind the method. In this example, the intent is to say hello. The implementation is using either a string telephone or a rotary telephone.



Responding to Messages

In statically typed languages, you know at compile time the set of messages an object can handle. In Ruby, the objects can gain the ability to respond to new messages at run-time. It can also lose the ability to respond to messages at run-time.

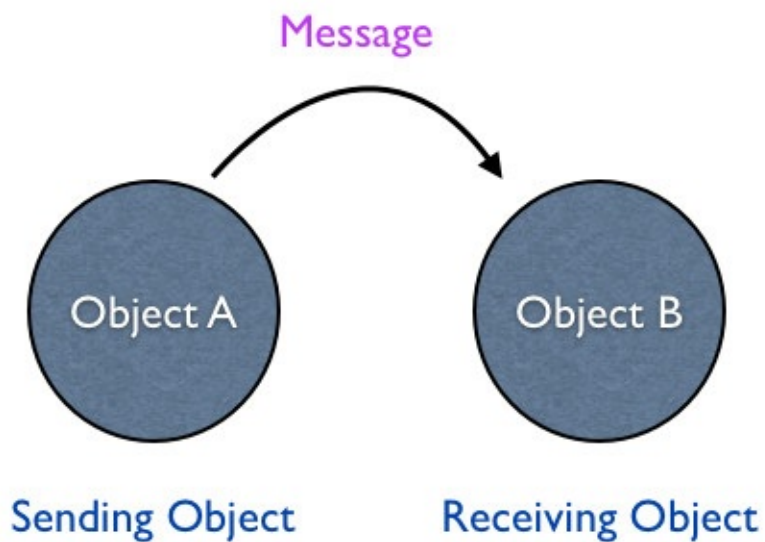


Static vs Dynamic Languages

Collaborating Objects

In the book, Object-Oriented Programming and Java by Poo, Danny, Kiong et al, the authors say:

Objects communicate with one another by sending messages. A message is a method call from a message-sending object to a message-receiving object. A message-sending object is a sender while a message-receiving object is a receiver.



Message Passing

Sender Object

The sender is the object that owns the scope where the message originates.

Explicit Sender



Let's now look at an example where the sender is explicit in the code.

```
class Teacher
  def initialize(student)
    @student = student
  end

  def ask_student_name
    @student.tell_name
  end
end

class Student
  def initialize(name)
    @name = name
  end

  def tell_name
```

```

    @name
  end
end

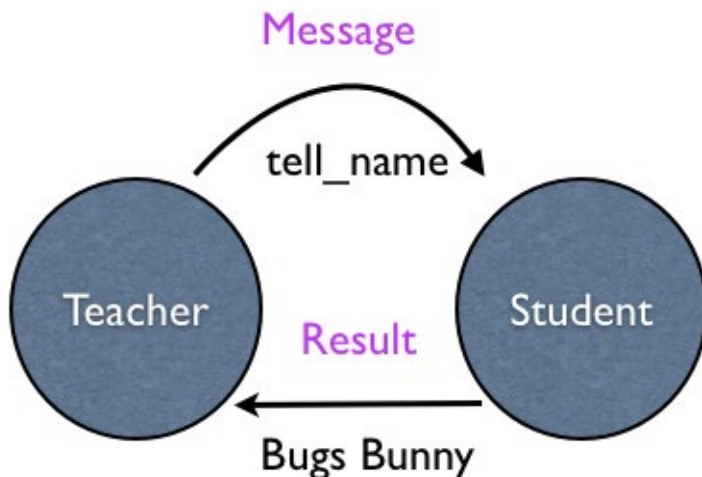
student = Student.new('Bugs Bunny')
teacher = Teacher.new(student)

p teacher.ask_student_name

```

This prints:

```
Bugs Bunny
```



Teacher and Student Objects

We create an instance of a Student class with the name *Bugs Bunny*. We then create an instance of a Teacher class and pass in the student object to its constructor. We send *ask_student_name* message to the teacher object that plays the role of a receiver.

Then the **ask_student_name** method runs. Inside this method, we can print the value of the sender object.

```

def ask_student_name
  p "The sender object is : #{self}"
  @student.tell_name
end

```

This prints:

```
The sender object is : #<Teacher:0x007fc3b40>
Bugs Bunny
```

We can make this concept explicit by printing the name of the class.

```

def ask_student_name
  puts "The sender object is : #{self.class}"
  @student.tell_name
end

```

This prints:

```
The sender object is : Teacher  
Bugs Bunny
```

Fabio Asks

How did you figure out what and where to print the value of the sender object?

Ask yourself the following questions.

1. Where did the message originate?
2. Which object owns the scope at that instant?

The answer for the question one, the line before:

```
@student.tell_name
```

The answer for the question two, the value of **self** owns the scope at that instant.

Key Takeaways

- Every sender and receiver in a message passing interaction is an object.
- Sender can be either explicit or implicit.
- Sender is the owner of the scope where the message originates.

Summary

In this chapter, you learned the basics of message passing. You learned about the message sender and the difference between intent and implementation. We also saw an example for an explicit sender.

Inheritance Basics

In this chapter you will learn about inheritance and how the method look-up works in an inheritance hierarchy.



Scope

We will not discuss the **why** and **when** inheritance must be used. It is beyond the scope of this book. Please checkout [Nothing is Something](#) presentation by Sandi Metz or [Essential Object Oriented Analysis](#) for a good discussion on this topic.

Inheritance

Inheritance represents **is-a** relationship between classes. We can say, “Car is a Vehicle”.

Car is a Vehicle

The is-a Relationship between Classes

We can define Car as the sub class of Vehicle.

```
class Vehicle
  def drive
    'drive method'
  end
end

class Car < Vehicle
end

car = Car.new
p car.drive
```

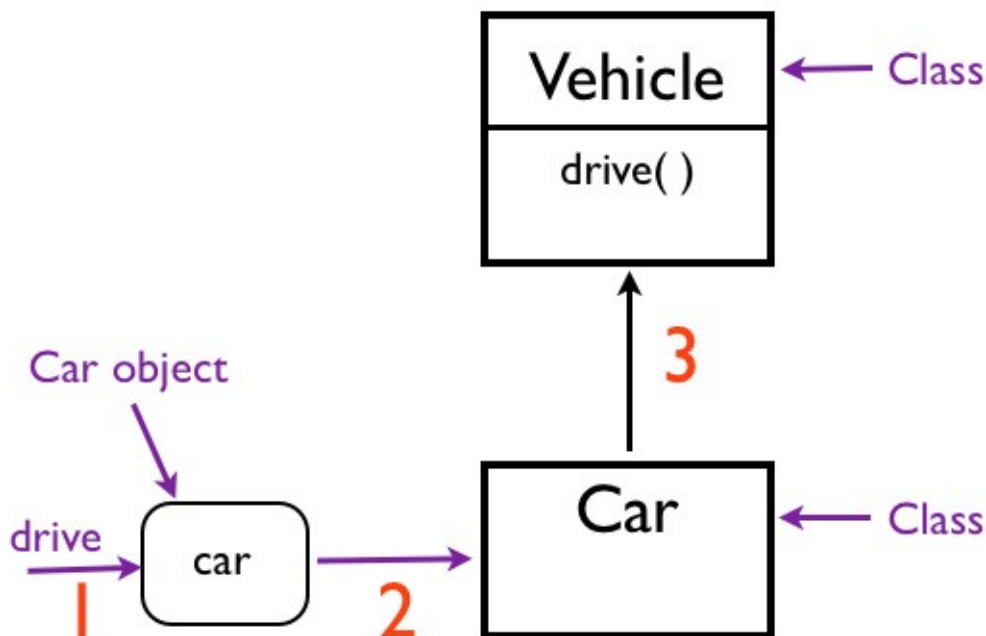
The less than symbol, <, is the syntax for inheritance. This prints:

```
drive method
```

In this example, the Car class will inherit the behavior of its parent class, Vehicle.

Method Lookup

In the above example, when Ruby encounters the **drive** message sent to a car object, it looks for the **drive** method in the Car class. It does not find it there. It goes to the super class, Vehicle of the Car class. It finds the **drive** method in Vehicle. It executes that method.



Method Lookup

What happens when you call a method that does not exist in the parent?

```
class Vehicle
  def drive
    p 'driving'
  end
end

class Car < Vehicle
end

car = Car.new
car.stop
```

This raises a NoMethodException error. In general, Ruby looks in the class of the object receiving the message for the method. If it does not find the method, it goes to its parent and looks for the method. It keeps searching through the inheritance hierarchy until it finds the method. What happens when Ruby reaches the root of inheritance hierarchy and still does not find the method? In that case, it raises NoMethodException.

Fabio Asks

Why does Ruby look for the method in the Car class?

The instance methods live in the class. We discussed this concept in the *What is an Object?* chapter.

Single Inheritance

Explicit Inheritance

In Ruby, a class can have only one parent, so, there is no multiple inheritance.



Let's check the super class of the Car class.

```
class Vehicle
  def drive
    'drive method'
  end
end

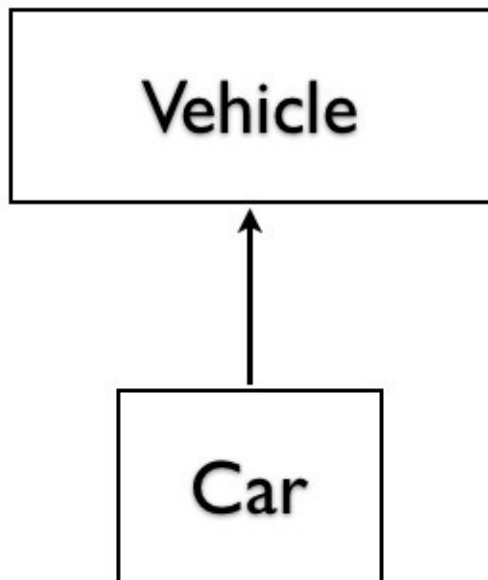
class Car < Vehicle
end

p Car.superclass
```

This prints:

```
Vehicle
```

We see that there is one value for the super-class.



Vehicle is Super Class of Car

The arrow pointing to the Vehicle class is the notation for inheritance. In this example the inheritance is explicit because we can see it in the code.

Implicit Inheritance

Even if you don't have an explicit superclass in the code, any class you define will have one superclass. Let's look at an example.

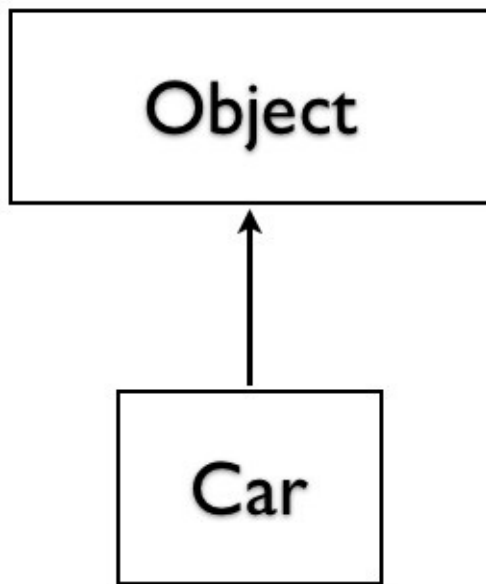
```
class Car
end

p Car.superclass
```

This prints:

```
Object
```

The Car class has the Ruby built-in Object as its superclass.



Object is Super Class of Car

Summary

In this chapter, you learned about inheritance and that Ruby is a single inheritance language. You learned that any user defined class has either an explicit or an implicit parent. We briefly saw how the method look-up occurs in an inheritance hierarchy.

Module Basics

In this chapter you will learn about the concept of module and how to overcome single inheritance limitation.

Single Inheritance

In the previous chapter you learned that Ruby is a single inheritance language. We can have only one parent. In this chapter, we will see how Ruby overcomes this limitation by using module.

What is a Module?

Module provides a way to share behavior. We can define methods in the module and instead of using inheritance, we can use the module to re-use code.

```
module Driveable
  def drive
    p 'driving'
  end
end

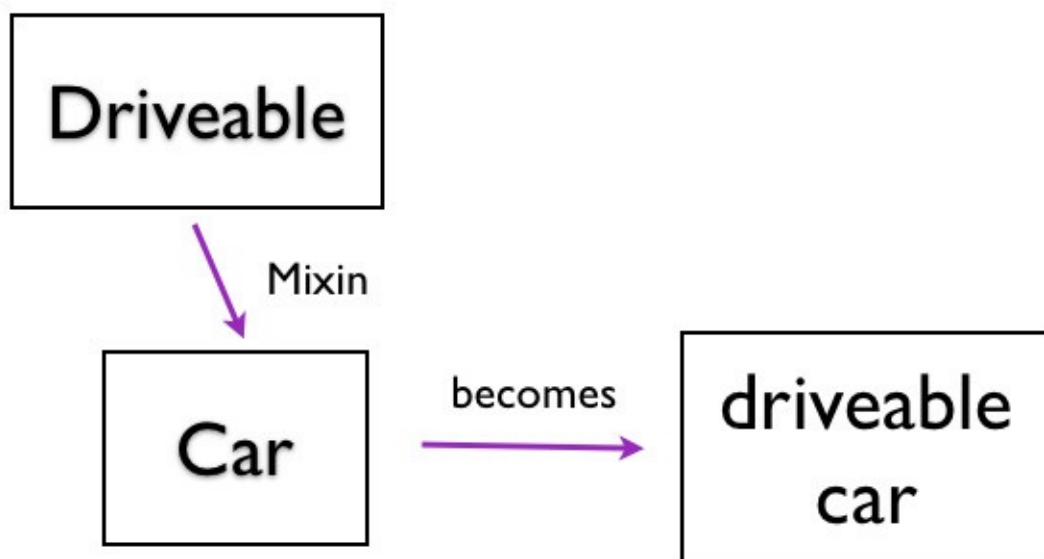
class Car
  include Driveable
end

car = Car.new
car.drive
```

We define a Driveable module with **drive** method. The **include** statement inside the Car class mixes in the Driveable module into the Car class. We can instantiate the car object and send the **drive** message to it. This prints:

```
driving
```

This is the mixin concept.



Mixin Driveable Module into a Car class

Mixin Multiple Modules

We can mixin multiple modules to our Car class.

```
module Driveable
  def drive
    p 'driving'
  end
end

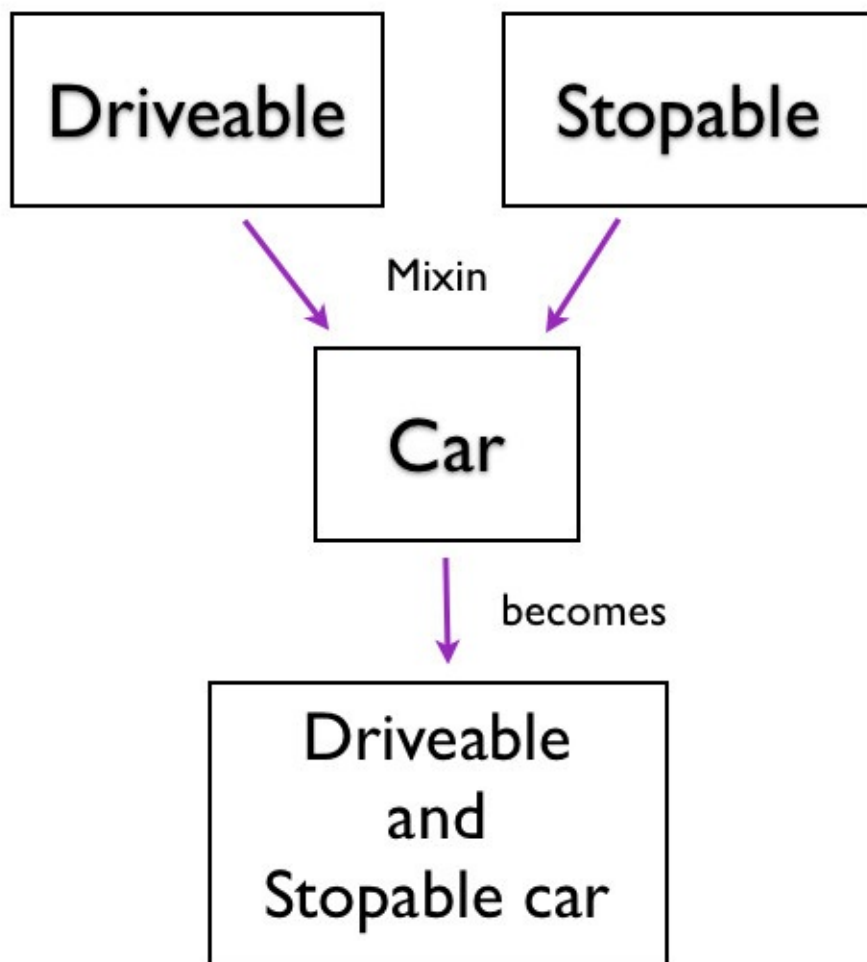
module Stopable
  def stop
    p 'brake failure, cannot stop'
  end
end

class Car
  include Driveable
  include Stopable
end

car = Car.new
car.drive
car.stop
```

We create a car object and we can call any of the methods that is available in the mixed-in modules. This prints:

```
driving
brake failure, cannot stop
```



Mixin Multiple Modules into a Car class

Key Takeaway

The **include** statement is used to mixin methods defined in a module into a class. The mixed-in methods become instance methods in the class.

Extending a Module

What if we want a class method? We can use the **extend** statement to pull in the methods defined in a module as class methods in a class.

```
module Turnable
  def turn
    p 'turning'
  end
end

class Car
  extend Turnable
end

Car.turn
```

This prints:

```
turning
```

The **turn** method is a class method because we don't need an instance of Car class to call the **turn** method. We use the Car class to call the **turn** method.

Summary

In this chapter, you learned how to overcome the single inheritance limitation. You can define as many modules as you want and mixin the behavior into a class by using the include statement. We also saw how to use extend statement to re-use class methods.

Essential Ruby

This section will cover essential Ruby concepts. You must already be familiar with the content discussed in Section 1.

Symbol

In this chapter you learn the concept of symbol and when it is used in Ruby programs.

Symbol Analogy



The dove is a symbol. It represents peace. There is a one-to-one association between the symbol and what it represents.

Ruby Symbol

The symbol identifier begins with a colon. In the IRB console, we can represent dove symbol like this:

```
> :dove  
=> :dove
```

It is unique, as there is only one object corresponding to the dove symbol. We can verify it.

```
> :dove.object_id  
=> 1175068  
> :dove.object_id  
=> 1175068  
> :dove.object_id  
=> 1175068  
> :dove.object_id  
=> 1175068
```

Regardless of how many times you call the `object_id`, the memory location of the object is the same.

Summary

In this chapter, you learned about symbol and how it is unique in a running program. Symbols are used as the arguments to methods and as name of methods.

The yield Keyword

In this chapter, you will learn the basics of the yield keyword. We will implement our own version of the Ruby built-in times method to get a better understanding of how it works.



Yield Example

Let's look at an example that uses `yield` keyword to execute code.

```
class Actor
  def act
    yield
  end
end

snowy = Actor.new
snowy.act { p 'wag the tail' }
```

The **act** method in Actor class has the **yield** keyword. The `yield` keyword yields flow of control to the block that calls the **act** method. In our example, the code between the curly braces is the block that gets executed. This prints:

```
wag the tail
```

Code Block

Our simple hail taxi program uses a block with do-end keywords.

```
3.times do  
  p 'Taxi'  
end
```

The code block is part of the syntax of the method call. The **times** method is called with the block that prints *Taxi*.

Implementing times Method

Let's write our version of **times** method to get a better understanding of the Ruby built-in **times** method.

```
class Fixnum
  def my_times
    for i in 1..self
      yield
    end
  end
end

3.my_times do
  p 'Taxi'
end
```

We re-opened the Fixnum class to define our **my_times** method. Inside the **for** loop, we call the block using the **yield** keyword. The block is a nameless function and has no name, so **yield** is the only way to call it. You can think of **yield** keyword as yielding control of flow to the block. We then call **my_times** method with a block as an implicit argument. This produces the same output.

Fabio Asks

I am getting:

```
LocalJumpError: no block given (yield)
```

when I run the code without a block.

```
class Fixnum
  def my_times
    for i in 1..self
      yield
    end
  end
end
```

```
3.my_times
```

You can prevent this error by checking if the block is given.

```
class Fixnum
  def my_times
    for i in 1..self
      yield if block_given?
    end
  end
end
```

```
3.my_times
```

Summary

In this chapter, you learned the basics of yield keyword. We implemented our own version of the Ruby built-in times method to get a better understanding of how it works.

Everything is Not an Object

In this chapter you will learn that every sender and receiver in a message passing interaction is an object.



Number is an Object

Let's ask Ruby for the class of the number 1.

```
1.class
```

This prints:

```
Fixnum
```

Fixnum is the class used to create an instance of the number one. The number is an object. You can send messages to them.

```
1.odd?
```

This prints:

```
true
```

String is an Object

String is a sequence of characters strung together. Let's ask Ruby for the class of a string object.

```
'hello'.class
```

This prints:

```
String
```

We can send messages to the string object 'hello'.

```
'hello'.reverse
```

This reverses the string to print:

```
olleh
```

Array is an Object

Let's ask Ruby for the class of an array.

```
[1,2,3,4].class
```

This prints:

```
Array
```

We can send messages to the array object.

```
[1,2,3,4].reverse
```

This prints:

```
[4, 3, 2, 1]
```

Hash is an Object

Let's ask Ruby the class of a hash.

```
{a: 1, b: 2}.class
```

This prints:

```
Hash
```

We can send messages to the hash object.

```
{a: 1, b: 2}.keys
```

This prints:

```
[:a, :b]
```

Messages are Not Objects

The messages we send to an object is not an object. But, we can convert them to an object. Let's convert the **keys()** message that we sent to a hash to a Method object.

```
keys_method = {a: 1, b: 2}.method(:keys)
=> #<Method: Hash#keys>
> keys_method.class
=> Method
```

We can convert a message to a method object by sending **method** message to a given object with the argument of the method name as the symbol. In this example, it is of the following format.

```
hash_object.method(:method_name)
```

The `method_name` argument is a symbol. We can now use the `keys_method` object to invoke the **keys()** like this:

```
keys_method.call
```

This prints:

```
[:a, :b]
```


Conditionals and Loops are Not Objects

Control structures do not have special syntax in Smalltalk. They are instead implemented as messages sent to objects. For example, Smalltalk implements if-else by sending a message to a Boolean object. In Ruby, control structures such as if-else, for, while etc. are not objects. For an in-depth discussion on this topic, please read [Flexing Your Message Centric Muscles](#).

Rhonda Asks

Why are we comparing Smalltalk with Ruby?

Ruby was influenced by many languages, Smalltalk is one of them.

Fabio Asks

Are blocks objects in Ruby?

The short answer is No. Blocks are not objects in Ruby. We need to use Proc, lambda or literal constructor `->`, to convert blocks into objects. In Smalltalk, blocks are objects. The statement: Everything is an object is true for Smalltalk but not for Ruby. We will discuss this topic in detail in the **Closures** chapter.

Summary

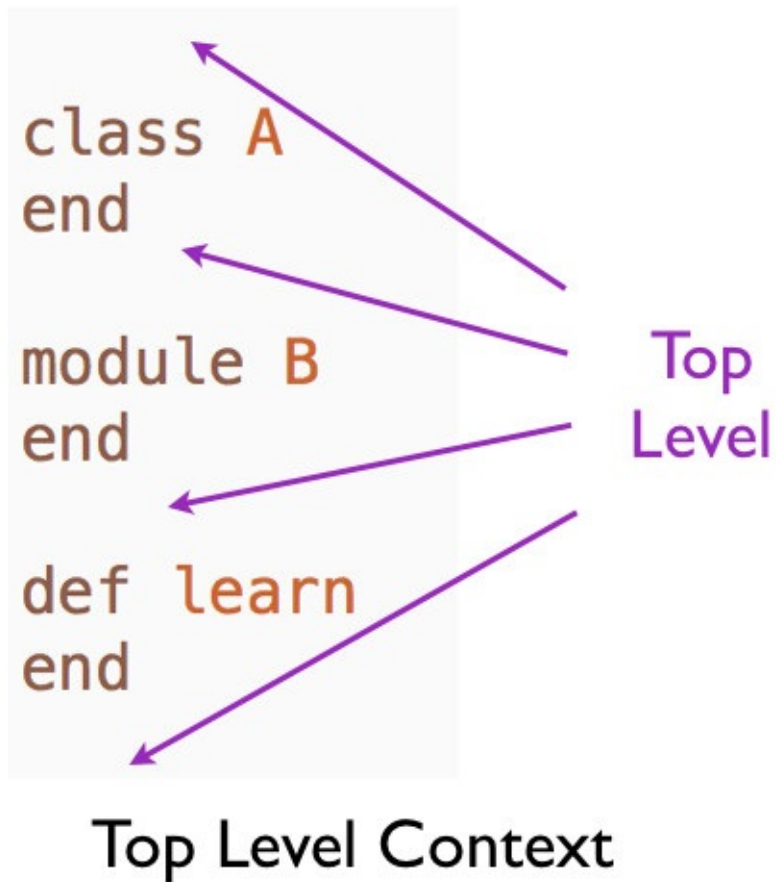
In this chapter, you learned that everything is not an object in Ruby. It is much more accurate to say: Every sender and receiver in a message passing interaction is an object. You can explore more objects in Ruby such as Symbols, Integer, Float etc. You can browse the documentation and experiment.

The Top Level Context

In this chapter we will discuss about the top level context in Ruby program. We will experiment sending messages with and without explicit receiver.

Top Level

What is top level? You are in top level when you have not entered into a class, module or method definition. Or exited from all class, module or method definitions.



Hello without Receiver

Let's write a simple program that prints hello at the top level.

```
puts 'hello'
```

As you would expect, this prints:

```
hello
```

The IO Class

Can we use an explicit receiver? Let's ask Ruby for the public instance methods of IO class.

```
puts IO.public_instance_methods(false).grep(/put/)
```

The **grep** searches for methods that has **put** in its name. The result shows that the **puts** is a public instance method of IO class.

```
putc  
puts
```

The false argument to the method filters out the methods from the super-class of IO class.

Hello with Receiver

The **puts** is an instance method in IO class. Let's call the public instance method **puts** in the **IO** class.

```
io = IO.new(1)
io.puts 'hello'
```

The argument to IO constructor, 1, tells Ruby to print to the standard output. This prints **hello** to the terminal.

Standard Output Global Variable

It's the same as doing:

```
$stdout.puts 'hello'
```

Here the **\$stdout** is the Ruby built-in global variable for standard output. Global variables can be accessed from anywhere.

Summary

In this chapter we discussed about the top level context in Ruby program. We called the puts method using an explicit receiver as well as without providing an explicit receiver. In *The Default Receiver* chapter, we will see what happens when we call puts without an explicit receiver.

Code Execution

In this chapter we will answer the question, when does Ruby execute code as it encounters the code?

At the Top Level

Open the editor and print 'hi' to the standard output.

```
puts 'hi'
```

This prints:

```
hi
```

Ruby encountered the **puts()** method and it executed the instruction. You see the output in the terminal.

Inside a Class

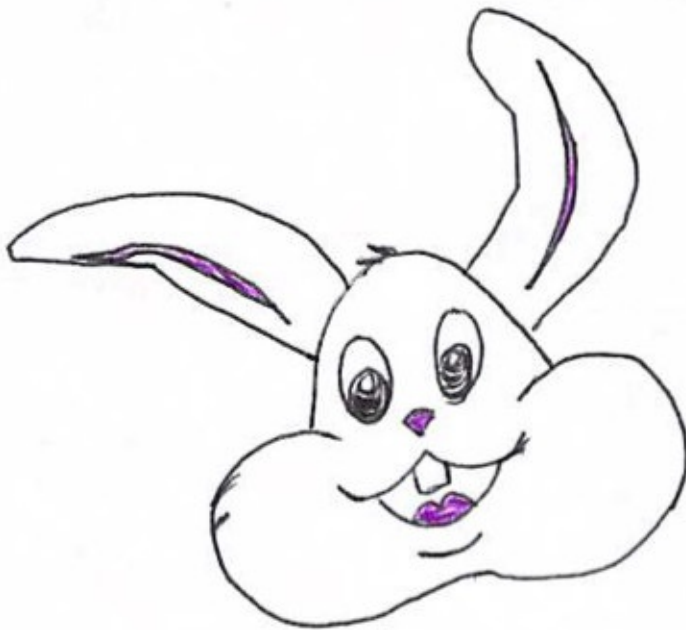
Define a class and print something to the standard output within the class.

```
class Rabbit
  puts "I am within Rabbit class"
end
```

Running this program prints:

```
I am within Rabbit class
```

This dynamic nature of Ruby surprises developers who are familiar with other languages.



Inside a Module

Define a module and print something to the standard output within the module.

```
module Rabbit
  puts "I am within Rabbit module"
end
```

Running this program prints:

```
I am within Rabbit module
```

Inside a Method in a Class

Let's now add a method to the Rabbit class:

```
class Rabbit
  def speak
    puts "My name is Bugs Bunny"
  end
end
```

Running this program does not print anything to the standard output.

Invoking the Instance Method

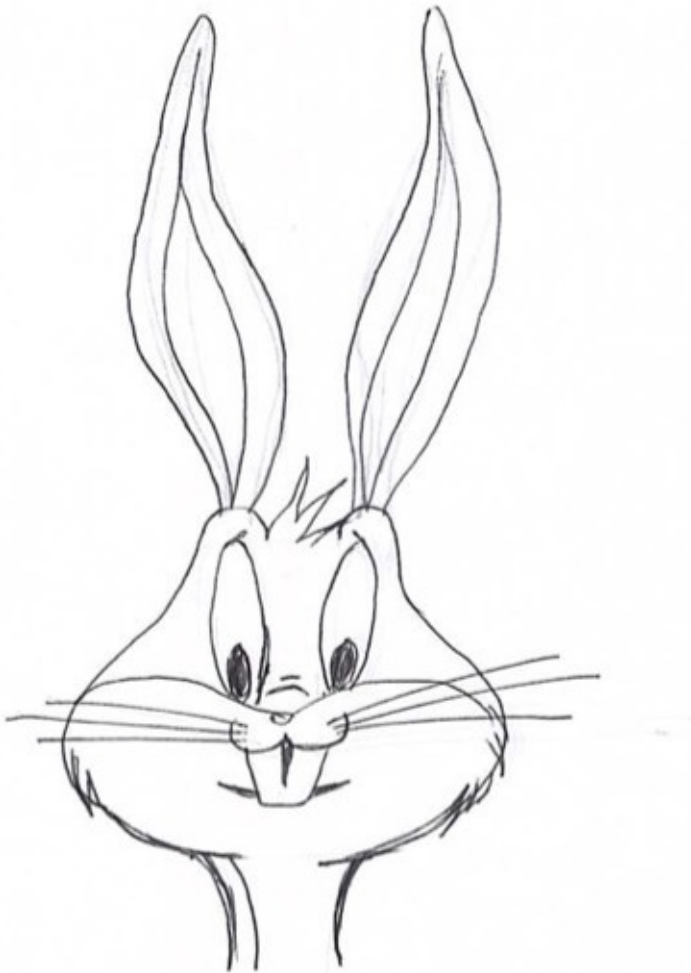
Why? Because, we need an instance of Rabbit to send the **speak()** message to it.

```
class Rabbit
  def speak
    puts "My name is Bugs Bunny"
  end
end

bugs = Rabbit.new
bugs.speak
```

Running this program prints:

```
My name is Bugs Bunny
```



Inside a Method in a Module

What happens when we define a method in a module?

```
module Rabbit
  def speak
    puts "My name is Bugs Bunny"
  end
end
```

Running this program prints nothing to the standard output.

Mixin the Module

We can mixin the Rabbit module to the top level and invoke the **speak()** method.

```
module Rabbit
  def speak
    puts "My name is Bugs Bunny"
  end
end

include Rabbit
speak
```

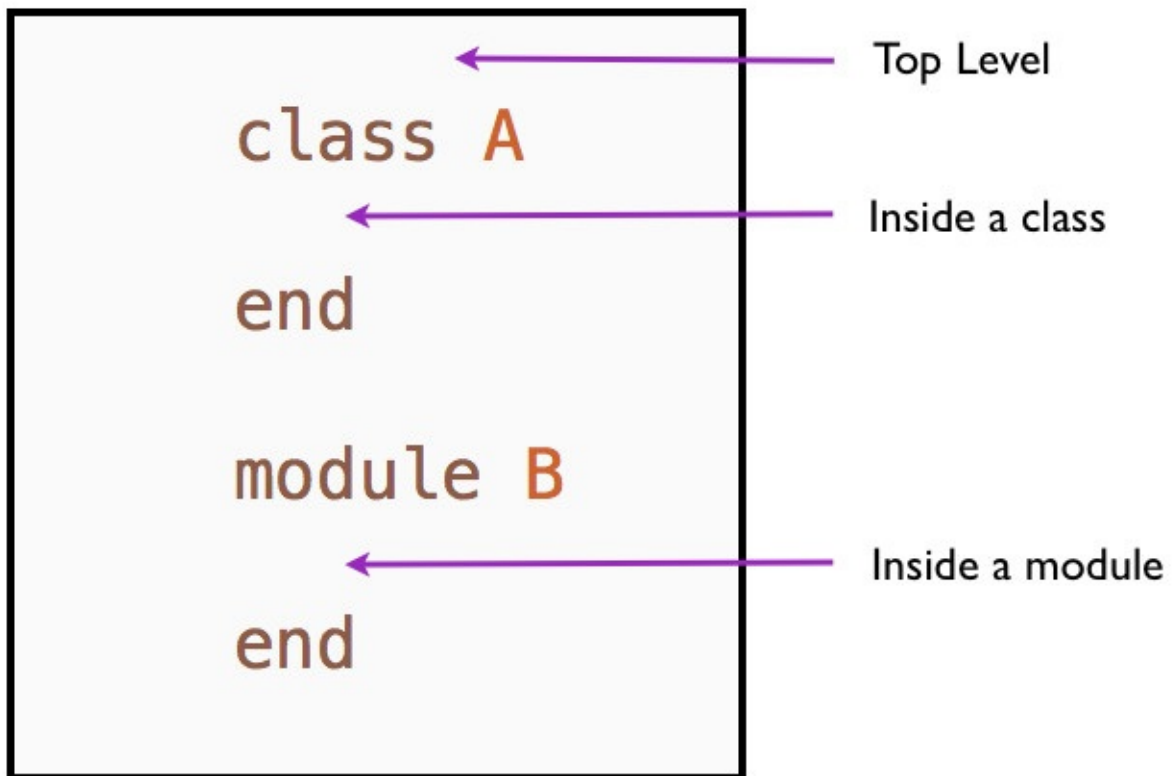
Running this program prints:

```
My name is Bugs Bunny
```

Summary

In this chapter, we learned that Ruby executes code as it encounters code:

1. At the top level.
2. Inside a class.
3. Inside a module.



Where Does Code Execute Immediately?

But, it does not execute the code inside the instance method defined in a class or a method defined inside a module as it encounters it. We need an object to execute an instance method defined in a class or mixin the method defined in a module and then call the method.

Binding

In this chapter you will learn the basics of binding and how we can execute code in different execution contexts.

Background

We have already covered the basic concepts of variables, methods and self. This chapter will combine all those concepts into one.

variables

+

methods

+

self

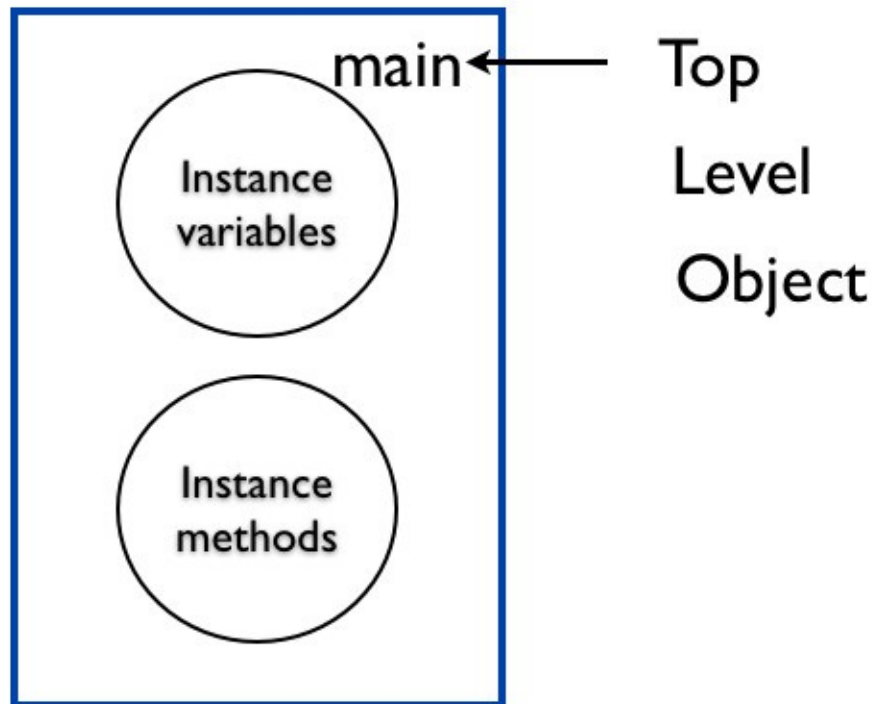
Binding

What is Binding?

A Binding object encapsulates the execution context at a particular place in the program. The execution context consists of the variables, methods and value of self. This context can be later accessed via the built-in function **binding**. We can create the binding object by using `Kernel#binding` method. The `Kernel#eval` method takes binding object as the second argument. Thus, the binding object can establish an environment for code evaluation.

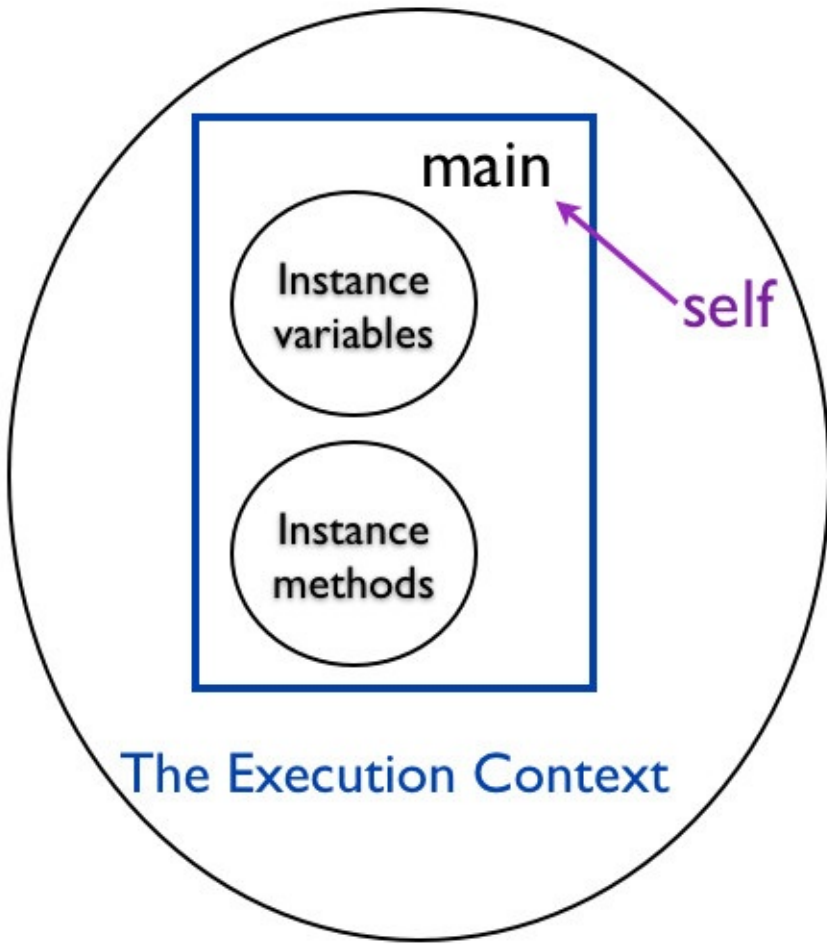
The Execution Context

In *The main Object* chapter, you saw this diagram:



The main bound instance methods and instance variables

The particular place in the program in this example is the top level. The value of self is main. We also saw how the instance variables and the instance methods are bound to the main object. The above diagram is the execution context for the top level. The diagram can be redrawn to make the binding concept clear.



The Execution Context

The Binding Object

Fabio Asks

Why do we need Binding?

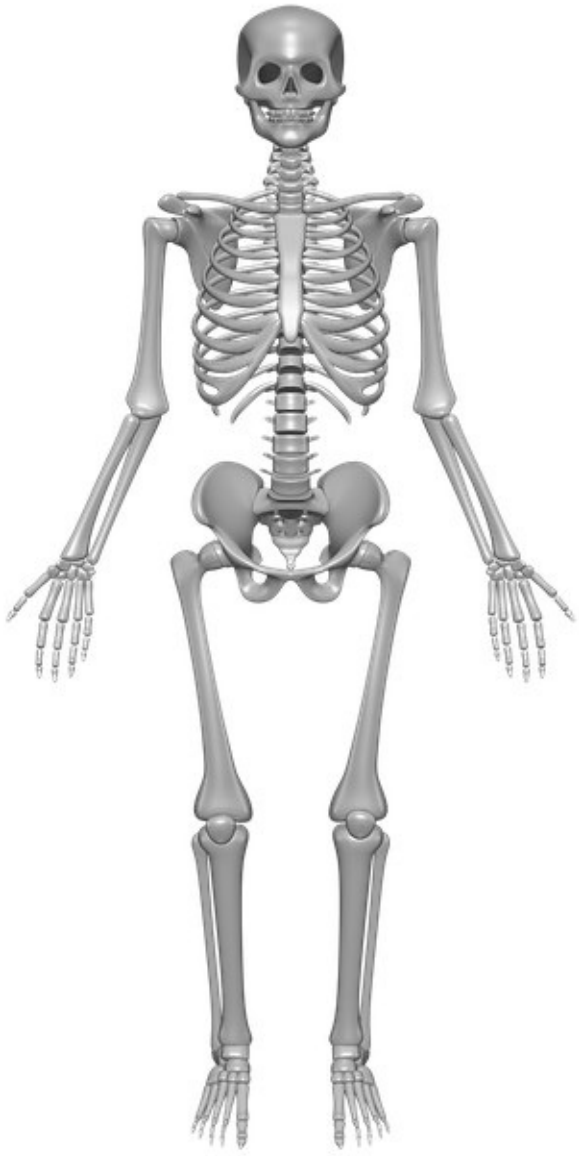
Code does not run in a vacuum. Code combined with an execution context becomes a running program.



Code, Execution Context and Running Program

Skeleton Analogy

Code is like a skeleton.



Execution Context is like the human flesh and skin.



Just like the human flesh and skin on a skeleton forms the human body. Running program is the combination of code and execution context.

Self at the Top Level

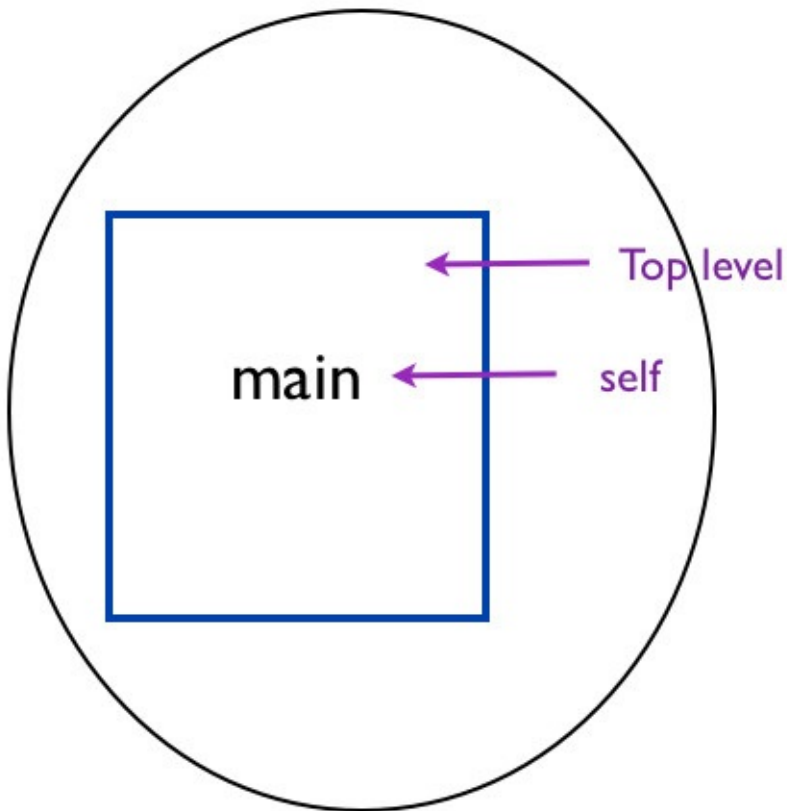
We can verify that the binding object at the top level context has main as the value of self.

```
p TOPLEVEL_BINDING.receiver
```

This prints:

```
main
```

The **TOPLEVEL_BINDING** is a Ruby built-in constant that captures the binding at the top level.



The Binding Object

Local Variables at the Top Level

Let's check for local variables defined at the top level.

```
p TOPLEVEL_BINDING.local_variables
```

This prints an empty array.

```
[]
```

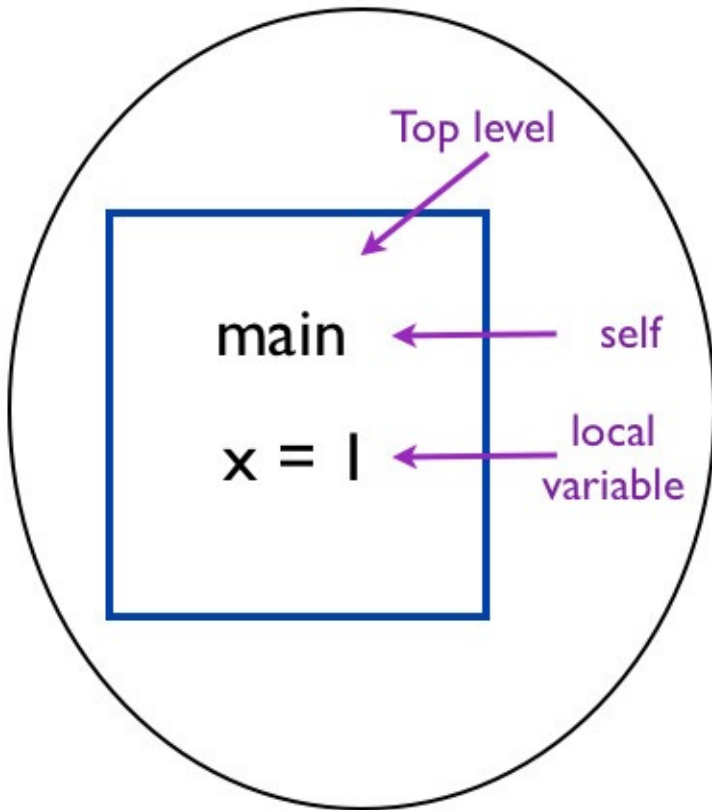
Let's define a local variable at the top level.

```
p TOPLEVEL_BINDING.local_variables  
x = 1
```

This prints :

```
[ :x ]
```

Ruby read all the statements at the top level, so it was able to print the value of x that comes even after the print statement.



The Binding Object

Instance Variable at the Top Level

How can we inspect the instance variable at the top level in the binding object?

```
@y = 0
p TOPLEVEL_BINDING.instance_variables
```

This prints an empty array.

```
[]
```

However, if we set the instance variable dynamically, we can print it.

```
TOPLEVEL_BINDING.instance_variable_set('@y', 0)
p TOPLEVEL_BINDING.instance_variables
```

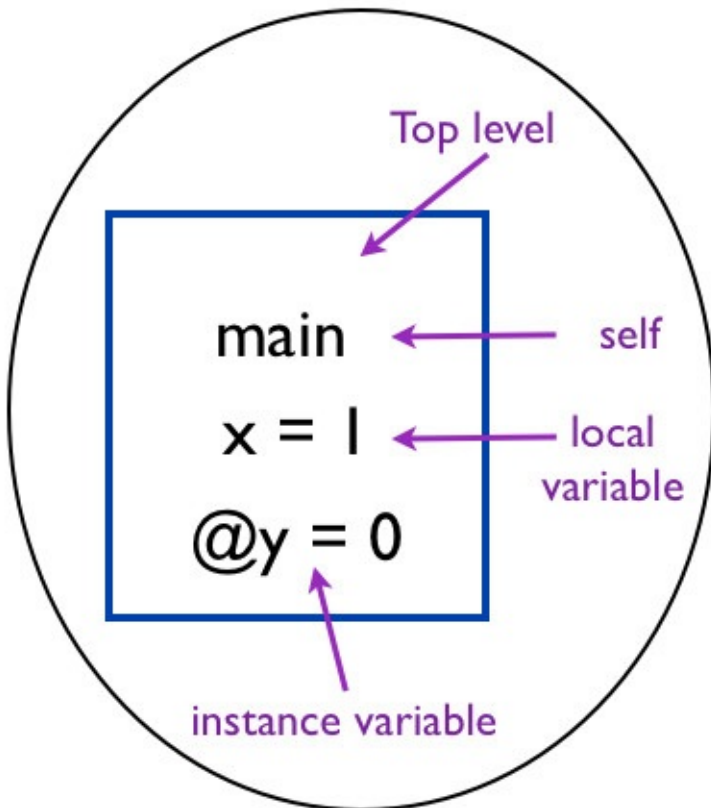
This prints:

```
[:@y]
```

We can also read the value of the instance variable at the top level.

```
TOPLEVEL_BINDING.instance_variable_set('@y', 0)
p TOPLEVEL_BINDING.instance_variable_get('@y')
```

This prints:



The Binding Object

Accessing the Local Variable at the Top Level

Let's use **eval** to access the local variable at the top level.

```
binding_before_x = binding
p "Before defining x : #{eval("x", binding_before_x)}"

x = 1

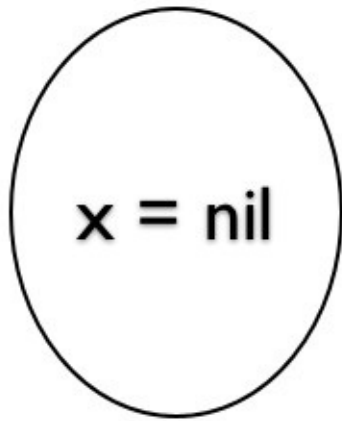
binding_after_x = binding
p "After defining x : #{eval("x", binding_after_x)}"
```

This prints:

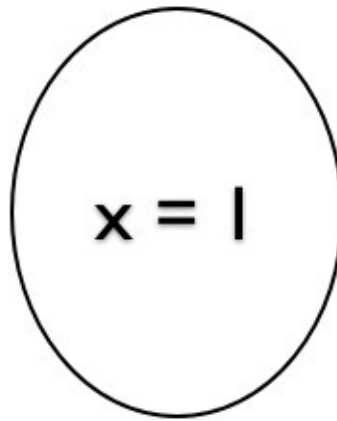
```
Before defining x :
After defining x : 1
```

The local variable `x` did not have any value before the assignment statement.

Before



After



The Execution Context

You can see the difference in this program from the previous section *Instance Variable at the Top Level*. The **eval** evaluates the value of `x` at the top level at the point at which it encounters. We then print the value before and after the local variable is initialized.

Object Context

Finding the self using Binding Object

Here is the example we saw in *Same Sender and Receiver* chapter.

```
class Car
  def drive
    p "self is : #{self}"
    self.start
  end

  private
  def start
    p 'starting..'
  end
end

c = Car.new
p "receiver is : #{c}"
c.drive
```

Let's rewrite the above example to use the binding object to find the receiver.

```
class Car
  def drive
    p "self is : #{self}"
    p "receiver is : #{binding.receiver}"
    self.start
  end

  private
  def start
    p 'starting..'
  end
end

c = Car.new
c.drive
```

This example uses:

```
binding.receiver
```

to print the receiver object. The output is:

```
self is : #<Car:0x007fe373864dc8>
receiver is : #<Car:0x007fe373864dc8>
```

The self and the receiver is the same car object.

Fabio Asks

Can I find the sender using the binding object?

No, binding object does not have a **sender** method that can give us the sender object.

Accessing the Instance Variable

In *What is an Object* chapter we could not access the color instance variable of the car object.

```
class Car
  def initialize(color)
    @color = color
  end

  def drive
    'driving'
  end
end

car = Car.new('red')
p car.color
```

This example gave the error:

```
NoMethodError: undefined method 'color' for #<Car:0x007f9be4025bc0 @color="red">
```

How can we access the color instance variable in the car object using binding? We know **eval** method takes the code as the first argument and binding as the second argument. Let's print the result of **eval** that takes the color instance variable and binding of the car object.

```
class Car
  def initialize(color)
    @color = color
  end

  def drive
    'driving'
  end
end

car = Car.new('red')
p eval("@color", car.binding)
```

This results in the error:

```
NoMethodError: private method 'binding' called for #<Car:0x007ffb639adbc8 @color="red">
```

Kernel module defines the **binding** method. Thus, it is available as a private method in the Object. Let's define a **my_binding** method that will provide us access to the execution context.

```
class Car
  def initialize(color)
```

```
@color = color
end

def drive
  'driving'
end

def my_binding
  binding
end

car = Car.new('red')
p eval("@color", car.my_binding)
```

This prints:

```
red
```

We are able to take a peek at the instance variable in the binding object. The binding object captures the value of self inside the **my_binding** method. We know that the value of self inside the **my_binding** method is a car object. The above example is the same as the following example:

```
class Car
  def initialize(color)
    @color = color
  end

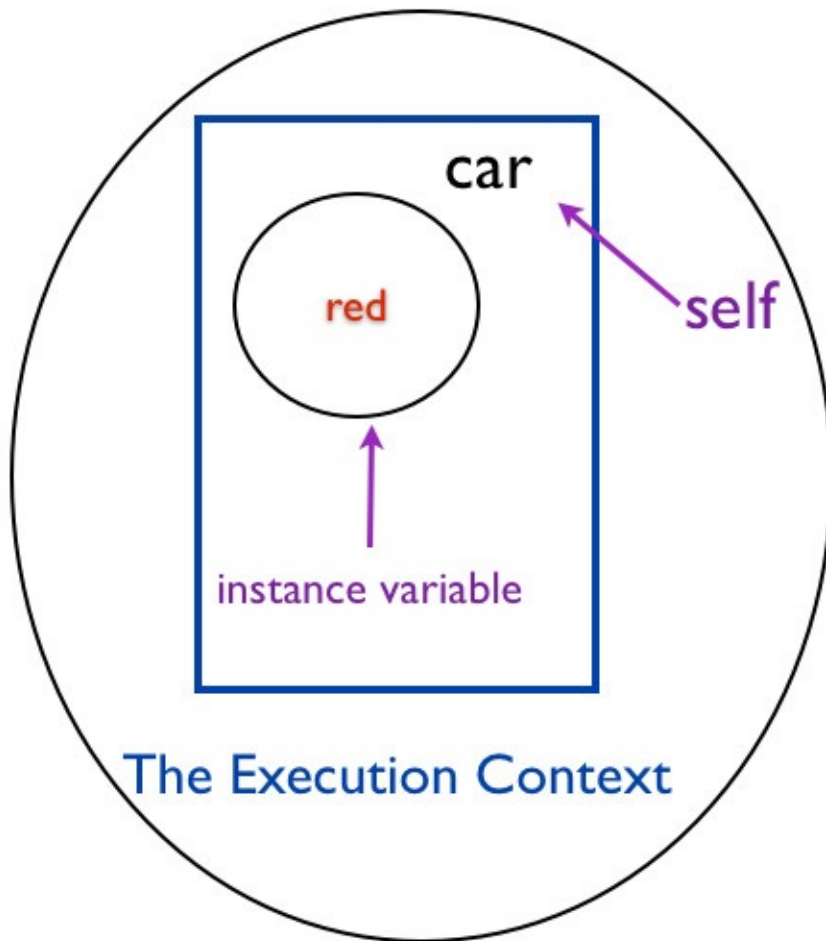
  def drive
    'driving'
  end

  def my_binding
    puts @color
  end
end

car = Car.new('red')
car.my_binding
```

This also prints:

```
red
```



The Binding Object

We can verify the value of self by running the following example.

```
class Car
  def initialize(color)
    @color = color
  end

  def drive
    return 'driving'
  end

  def my_binding
    puts self
    puts @color
  end
end

car = Car.new('red')
car.my_binding
```

This prints the car object memory location and the color.

```
#<Car:0x007fa132018e90>
red
```

Execution Context Analogy

It's like using a probe to send some piece of code to execute in a different context.



Executing Code in Different Execution Contexts

The power of binding is in the ability to run the same code in different contexts. Let's take a look at an example.

```
class Car
  def initialize(color)
    @color = color
  end

  def my_binding
    binding
  end
end

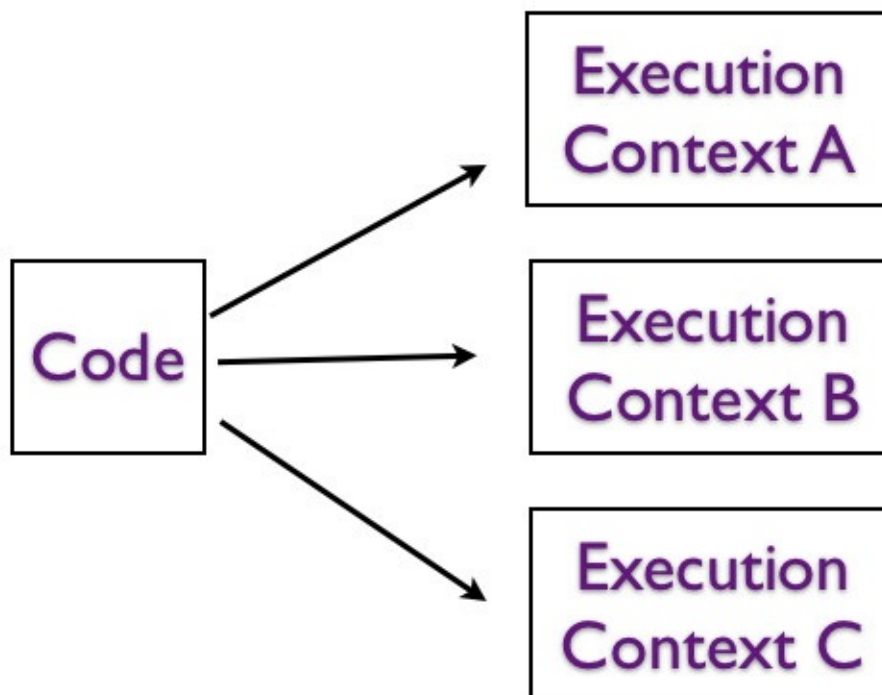
red_car = Car.new('red')
black_car = Car.new('black')

code = "@color"

p eval(code, red_car.my_binding)
p eval(code, black_car.my_binding)
```

This prints:

```
red
black
```



Executing Code in Different Execution Contexts

There is no restriction on which object should provide the binding. We can have another

class, let's say dog, that provides an execution context for the same code.

```
class Dog
  def initialize(color)
    @color = color
  end

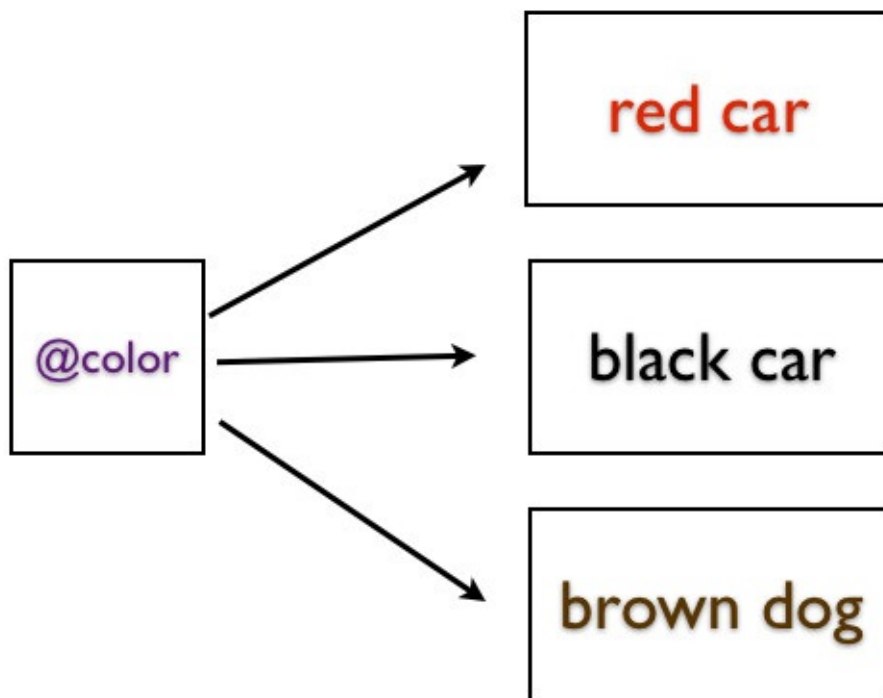
  def my_binding
    binding
  end
end

dog = Dog.new('Brown')
code = "@color"
p eval(code, dog.my_binding)
```

This prints:

Brown

The **my_binding** method is like a probe. Code is put inside of this method.



Executing Code in Different Execution Contexts

Rhonda Asks

Why do we need the ability to run the code in different execution contexts?

Ruby is dynamic in nature. This allows us to reuse code in scenarios that we might not have imagined when we wrote the code.

Executing Code in Different Scope

Execute Code in an Object Scope from Top Level Scope

Let's look at an example that executes code in the rabbit object scope from the top level scope.

```
class Rabbit
  def context
    @first = 'Bugs'
    last = 'Bunny'

    binding
  end
end

binding = Rabbit.new.context
# Scope here has changed because this is top level scope.
# But we are executing the following code in the rabbit object scope by using eval.
p eval("self", binding)
p eval("last.size", binding)
p eval('@first', binding)

# This uses binding only, no eval is used to get the local variable
p binding.local_variable_get('last')
```

This prints:

```
#<Rabbit:0x007fc2a406c318 @first="Bugs">
5
Bugs
Bunny
```

We were able to take a look at the local variable, instance variable, value of self and execute methods on the local variable. It is as if we were inside the context method like this:

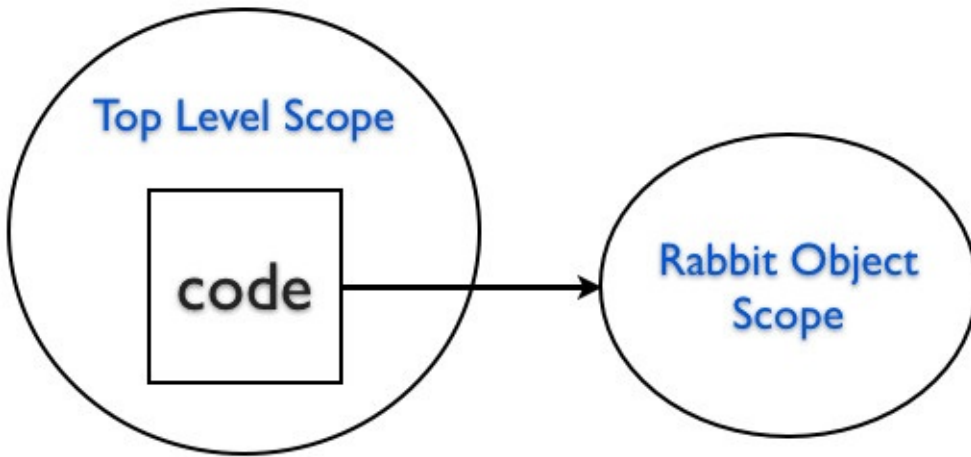
```
class Rabbit
  def context
    @first = 'Bugs'
    last = 'Bunny'

    puts self
    puts @first
    puts last
    puts last.size
  end
end

Rabbit.new.context
```

This prints:

```
#<Rabbit:0x007fe3f3881038>
Bugs
Bunny
5
```



Executing Code in Different Scope

Execute Code in Top Level Scope from an Object Scope

Let's now see an example where we execute code in the top level scope when we are in an object scope.

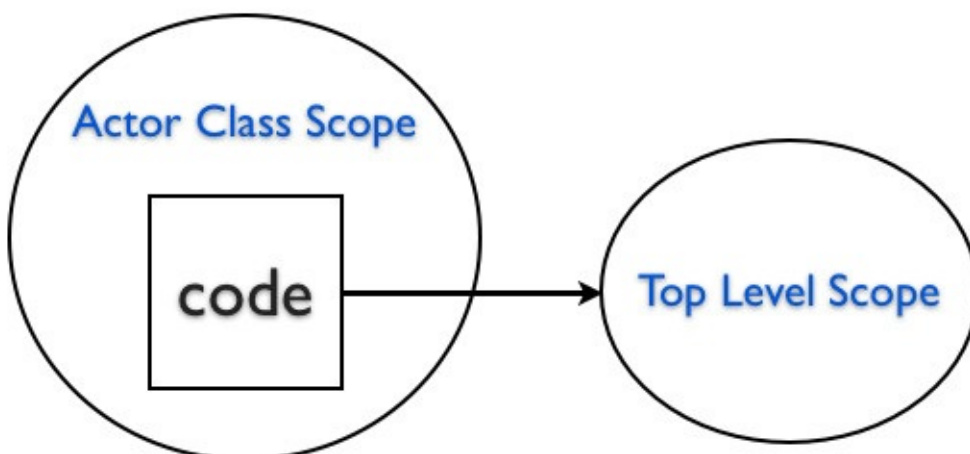
```
@actor = 'Daffy'

class Actor
  def self.act
    eval("@actor", TOPLEVEL_BINDING)
  end
end

p Actor.act
```

This prints:

Daffy



Executing Code in Different Scope

If you access the top level instance variable inside the method from the Actor class scope, you will not be able to access the value.

```
@actor = 'Daffy'  
class Actor  
  def self.act  
    @actor  
  end  
end  
  
p Actor.act
```

This will print nil, because, the `@actor` inside the `act` method is in a different scope and it is not initialized. They are two different variables that happens to have the same name.

Practical Example

Read the article [Generate Documents using Templates](#). This example shows how we can apply what we have learned in this chapter to a real problem.

Summary

In this chapter, we learned that we can package up the execution environment for later use via the binding. We looked at the value of self, local variable and the instance variable inside a binding object.

Pseudo Variables

In this chapter, you will learn about the pseudo-variables true, false, nil, self and super.

What are Pseudo variables?

The nil, true, false, self and super are pseudo-variables. Why are they called pseudo-variables? Because they are predefined and we cannot assign values to them.

The true, false, and nil are constants. The self and super vary dynamically as the code executes.

The boolean classes

The true and false are unique instances of the boolean classes TrueClass and FalseClass.

```
> true.class
=> TrueClass
> false.class
=> FalseClass
```

The Concept of Nothing is an Object

The nil is an unique instance of NilClass.

```
> nil.class
=> NilClass
```

The nil is a false value that represents 'no value' or 'unknown'. It expresses nothing.

```
> p 'false' unless false
"false"
=> "false"

> p 'false' unless nil
"false"
=> "false"
```

The unless is equal to negation with if statement:

```
p 'false' if !nil
"false"
=> "false"
```

This shows that nil is a false value.

The self and super

The self always refers to the receiver of the current executing method. The super also refers to the receiver of the current method. But when you send a message to super, the method look-up changes. The method look-up starts from the super-class of the class containing the method that uses super.

```
class Vehicle
  def initialize(wheels)
    @wheels = wheels
  end
end
```

```
class Car < Vehicle
  attr_reader :wheels

  def initialize(color)
    super(4)
    @color = color
  end
end

car = Car.new('red')
p car.wheels
```

In this example, the super is in the Car class. The method look-up starts from the superclass of Car, Vehicle. This prints:

Assigning Values

You can see that we get an error when we assign a value to true, false or nil.

```
> nil = 1
SyntaxError: (irb):1: Can't assign to nil
nil = 1
  ^
  from /Users/bparanj/.rvm/rubies/ruby-2.3.0/bin/irb:11:in `<main>'
> true = 1
SyntaxError: (irb):2: Can't assign to true
true = 1
  ^
  from /Users/bparanj/.rvm/rubies/ruby-2.3.0/bin/irb:11:in `<main>'
> false = 1
SyntaxError: (irb):3: Can't assign to false
false = 1
  ^
  from /Users/bparanj/.rvm/rubies/ruby-2.3.0/bin/irb:11:in `<main>'
```

You can see that we get an error when we try to change the value of self.

```
> self = 1
SyntaxError: (irb):7: Can't change the value of self
self = 1
  ^
  from /Users/bparanj/.rvm/rubies/ruby-2.3.0/bin/irb:11:in `<main>'
```

We get a syntax error when we try to change the value of super.

```
> super = 1
SyntaxError: (irb):12: syntax error, unexpected '='
super = 1
  ^
  from /Users/bparanj/.rvm/rubies/ruby-2.3.0/bin/irb:11:in `<main>'
```

Summary

In this chapter, you learned about the pseudo-variables `true`, `false`, `nil`, `self` and `super`. We cannot assign any values to them. The `true`, `false` and `nil` are predefined. The `self` and `super` varies dynamically.

The Default Receiver

In this chapter, you will learn that when you don't provide an explicit receiver in the code, Ruby uses **self** as the default receiver object.

The self Within Car

Let's define a Car class and print the value of self inside the Car class.

```
class Car
  puts self
end
```

This prints:

Car



The drive Class Method

Let's define a **drive()** class method in the Car class.

```
class Car
  def self.drive
    p 'driving'
  end
end
```

We know that the value of self is Car. This is the same as:

```
class Car
  def Car.drive
    p 'driving'
  end
end
```

Call the Class Method

Explicit Receiver

We can call the **drive()** class method inside the Car class.

```
class Car
  def self.drive
    p 'driving'
  end

  Car.drive
end
```

This prints:

```
driving
```

We can also use self instead of Car to call the **drive()** class method.

```
class Car
  def self.drive
    p 'driving'
  end

  self.drive
end
```

This also prints:

```
driving
```

Call the Class Method

No Receiver

We know the value of self inside the Car class is Car. We can omit the self from the above example.

```
class Car
  def self.drive
    p 'driving'
  end

  drive
end
```

This prints:

```
driving
```

We don't have the dot notation that sends a message or the receiver object. The message is sent to the current value of self whenever you call a method with no receiver. Since the current value of self is Car, we are able to call the drive class method. In this example, Car is the default receiver object.

Fabio Asks

Why is it called default receiver?

Because, when you don't provide an explicit receiver, the current value of self defaults as the receiver.

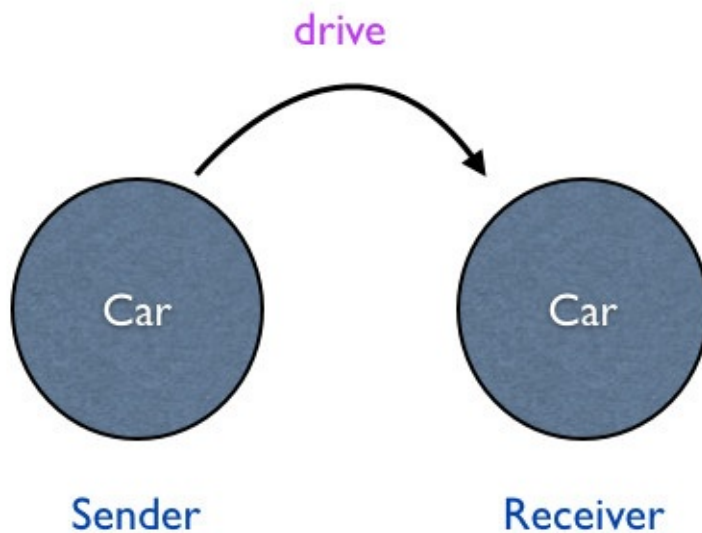
Rhonda Asks

Why do we need a receiver to send a message?

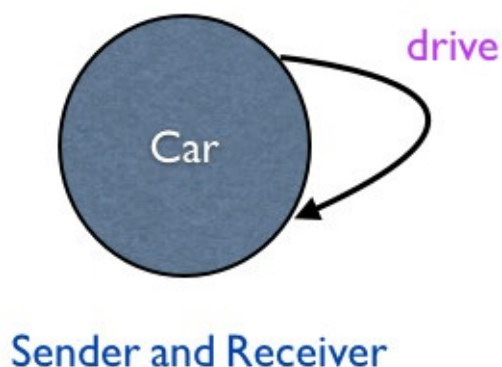
In a OO language like Ruby, all messages are sent to some object.

Insight

There is another way to think about this example. You can say that whenever the sender object and the receiver object is the same, you can omit the receiver.



Same Sender and Receiver



We will revisit this idea in an upcoming chapter.

Subclass Calling Class Method

The example may seem trivial, but this allows us to write code like this:

```
class Car
  def self.drive
    p 'driving'
  end
end

class Beetle < Car
  drive
end
```

This example prints:

```
driving
```

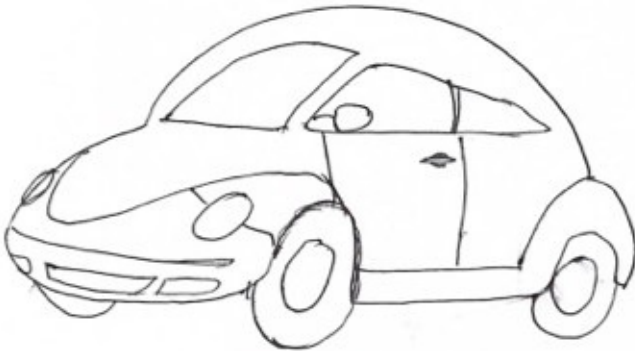
Rails Example

This is like ActiveRecord library in Rails. Imagine that ActiveRecord **find()** implementation is like this:

```
class ActiveRecord
  def self.find(id)
    p "Find record with id : #{id}"
  end
end
```

In our web application we have a Beetle subclass of ActiveRecord as the model.

```
class Beetle < ActiveRecord
end
```



In the controller, we use the find method:

```
Beetle.find(1)
```

This prints:

```
"Find record with id : 1"
```

The dot notation makes sending messages explicit. If the receiver and the sender is the same, you can omit the receiver and the dot. In this case, Ruby will use value of self as the receiver. Thus, self is the default receiver.

Summary

In this chapter, you learned that the message is sent to the current value of self when you call a method with no receiver. This object is called the default receiver.

Message Sending Expression

In this chapter, we will identify the sender, receiver, message and arguments in a message sending expression.

Background

By now, you already know:

- There is always a receiver.
- There is always a sender.
- There is always a message that passes between the sender and the receiver.

Let's see an example that ties all these concepts together.

Messaging

Let's write the simplest program to illustrate the three key takeaways of this book so far.

```
x = 1 + 2
```

```
p x
```

This prints 3.

Explicit Message

The above program can be rewritten in an equal way.

```
x = 1.+ 2
```

```
p x
```

This also prints 3. This example makes sending a message explicit by using the dot notation.

Explicit Argument

The above program can be rewritten in an another way.

```
x = 1.+(2)
```

```
p x
```

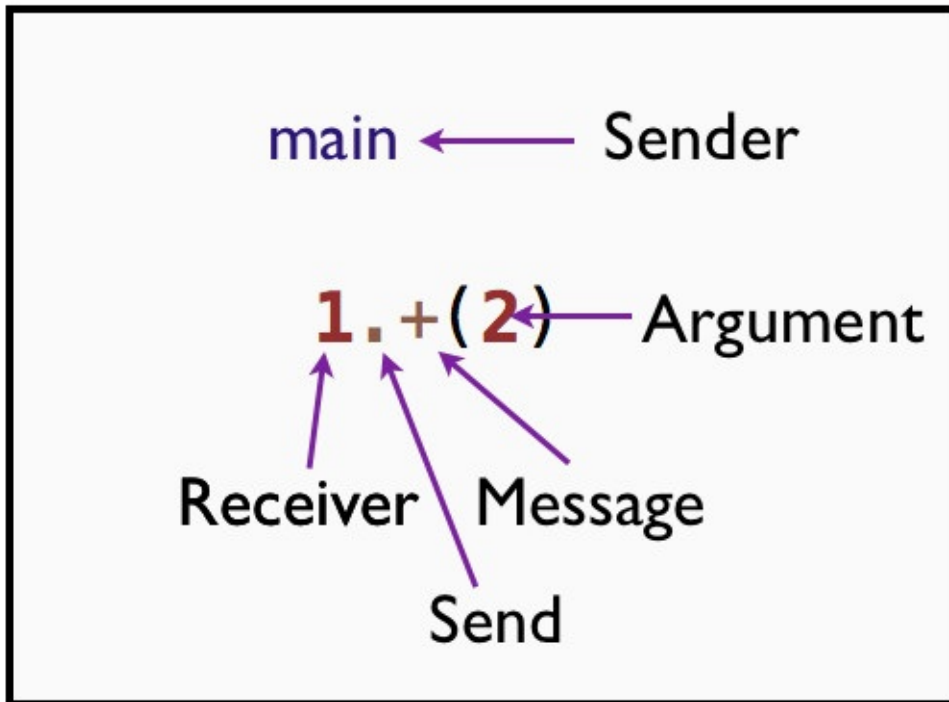
This also prints 3. This example makes the argument, 2 to the message + explicit.

Identify Receiver

It's now clear that 1 is the receiver. We already know that this object is Fixnum.

Identify Sender

The sender is implicit because we have written this code at the top level context. We know that main object is the sender.



Sending a Message

The `+` is not an operator in Ruby. The `+` is a message that passes between the sender and the receiver. Let's verify it.

```
p 1.respond_to?(:+)
```

The **`respond_to?`** method takes a symbol of the method name as the argument and returns either true or false depending on whether it can respond to it or not. This prints:

```
true
```

This proves that Fixnum object responds to the `+` message. We can explicitly send the message to Fixnum object.

```
p 1.send(:+, 2)
```

The **`send`** method is defined in Kernel module and is mixed into the Object. So, it's available everywhere. The first argument to the send method is the symbol of the method name and the second argument is the argument for that method. This prints 3.

Convert Message to an Object

We can convert the + message sent to the Fixnum object into an object and call it.

```
> addition = 1.method(:+)
=> #<Method: Fixnum#+>
> addition.call(2)
```

The method called **method** is defined in Kernel module. Since the Kernel module is mixed into the Object, it's available everywhere. This returns a Method object. We send the call message with 2 as the argument to add 1 and 2. This prints 3.

Fabio Asks

Are there other messages in Ruby that looks like an operator?

Yes, you can experiment in the IRB and get a list of those messages:

```
> 1.public_methods(false)
=> [:%, :&, :*, :+, :-, :/, :<, :>, :^, :|, :~, :-@, :**, :<=>, :<<, :>>, :<=, :>=, :==, :===, :[],...]
```

The false argument is used to list only the methods found in Fixnum class. We can also send the instance_methods message to the Fixnum class.

```
> Fixnum.instance_methods(false)
```

This prints:

```
[:-@, :+, :-, :*, :/, :%, :**, :==, :===, :<=>, :>, :>=, :<, :<=, :~, :&, :|, :^, :[], :<<, :>>, ...]
```

Key Takeaway

You must read `1 + 2` as, the object 1 is sent the message `+`, with the object 2 as the argument.

Summary

In this chapter, you learned that in Ruby, adding two numbers is a first-class message sending expression. We identified the sender, receiver, message and the argument in the message sending expression.

The self at the Top Level

In this chapter, you will learn about the current object, **self** at the top level.

What is self?

In Ruby, there is always one object that plays the role of current object at any given instant. The current object provides an execution context for the code. This current object is the default receiver. When the receiver is not provided, the message is sent to the default receiver. This is the **self**.

Current Object

=

Default Receiver

=

self

What is self ?

Self at the Top Level

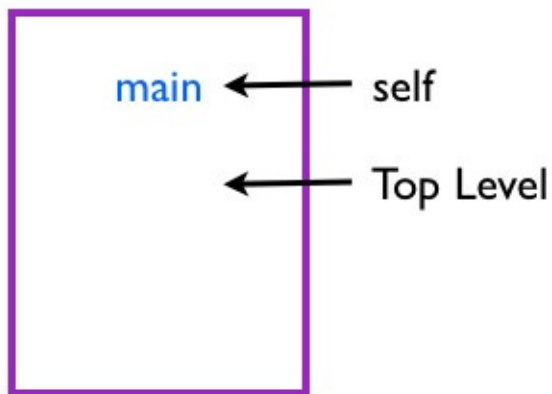
Let's see the value of `self` at the top level.

```
puts self
```

This prints:

```
main
```

This tells us that Ruby has created an object called **main** for us at the top level. And all the code we write at the top level will use **main** as the receiver in method calls.



The self at the Top Level

The main Object

If **main** is an object, it must be instance of some class. We can ask Ruby to tell us which class **main** is an instance of:

```
puts self.class
```

This prints:

```
Object
```

We now know, Ruby did something like this:

```
main = Object.new
```

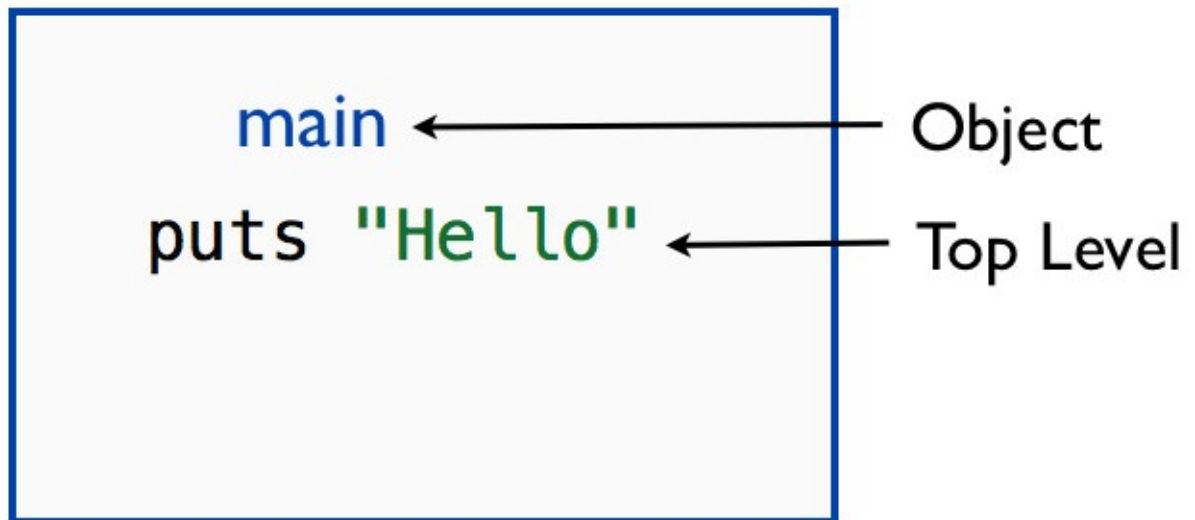
This provides us an object context to execute our code at the top level. Thus, providing us the default receiver **main** at the top level. The main is an instance of the Object class.

Hello at the Top Level

Let's print hello to the standard output.

```
puts 'hello'
```

As expected, this prints **hello**.



Top Level and the main Object

Is main a Receiver Object?

Can we use **main**, the instance of Object, to call puts?

```
main.puts 'hi'
```

We get:

```
NameError: undefined local variable or method 'main' for main:Object
```

Ruby prints **main** as the current object at the top level. But, there is no such variable called **main**.

Human Visible main Object

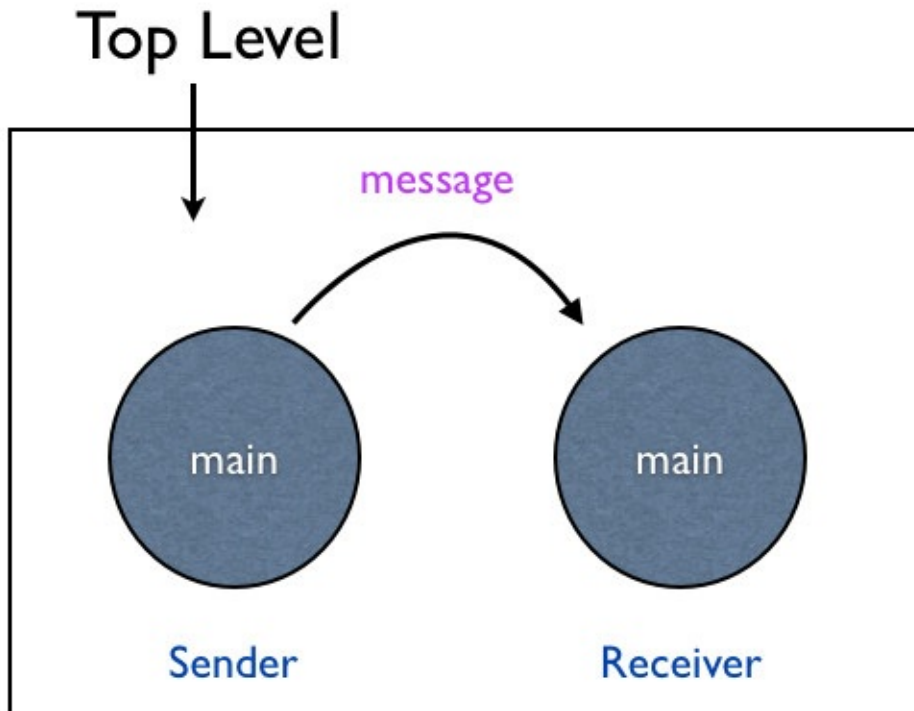
The **main** is the human visible representation of the current object. Let's see this in action in the IRB console.

```
> self
=> main
> self.inspect
=> "main"
> self.to_s
=> "main"
```

Fabio Asks

Why does Ruby create main object at the top level?

In a OO language like Ruby, there is always a sender and receiver involved in a message send. We did not explicitly create a sender object. Thus, Ruby created a sender object for us. It is implicit, because it is not visible in the code.



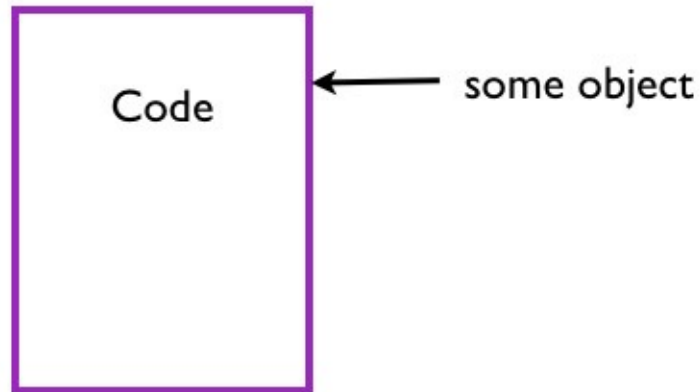
Same Sender and Receiver

In an upcoming chapter, we will see that both sender and receiver are main at the top level.

Rhonda Asks

If I don't create a receiver object, does Ruby create a receiver object?

No. Ruby does not create a receiver object; it uses the existing value of **self** as the default receiver.



Code Executes in Current Object

Summary

In this chapter, we discussed the default receiver `self` at the top level context. You learned that we cannot specify an explicit receiver for methods in the top level like **`puts()`**. In such cases, the receiver is implicit and is not specified. We will see why in an upcoming chapter.

The Dynamic Nature of self

In this chapter you will learn how the value of self changes as the program executes.

Car with Drive Method

Let's define a Car class with a **drive()** method.

```
class Car
  def drive
    'driving'
  end
end
```

We can create an instance of the car class and send a message to the car object.

```
car = Car.new
car.drive
```

This prints:

```
driving
```

The self Before Sending a Message

Let's see the value of self before we send the drive message to car object.

```
class Car
  def drive
    'driving'
  end
end

car = Car.new

p "Before sending the message, self is : #{self}"
car.drive
```

This prints:

```
Before sending the message, self is : main
```

The self After Sending a Message

Let's see the value of self after we send the drive message to car object.

```
class Car
  def drive
    'driving'
  end
end

car = Car.new

car.drive
p "After sending the message, self is : #{self}"
```

This prints:

```
After sending the message, self is : main
```

The self During Message Send

Let's check the value of self during the message send.

```
class Car
  def drive
    'driving'
  end
end

car = Car.new
car.tap { |x| p "The value of self is : #{x}"}.drive
```

This prints:

```
The value of self is : #<Car:0x007fc6bb912d08>
```

The value of self is the same as the car object.

Rhonda Asks

Where is the tap method defined?

Let's find out.

```
p Object.method(:tap)
```

This prints:

```
#<Method: Class(Kernel)#tap>
```

The **tap** method is in the Kernel module.

```
> Kernel.methods.grep(/tap/)
=> [:tap]
```

Kernel is mixed in to Object.

```
> Object.methods.grep(/tap/)
=> [:tap]
```

Thus, tap method is available on any object. It executes a code block, yielding the receiver to the block and returns the receiver.

Fabio Asks

Why does the following code print main, when I change code ?

```
car.tap { |x| p "The value of self is : #{self}" }.drive
```

The self printed inside tap is the same value of self as in the line above the tap call. This is due to closure in Ruby. You will learn about closures in a later chapter. You must print the block variable, if you want to peek inside the car object. We can verify it.

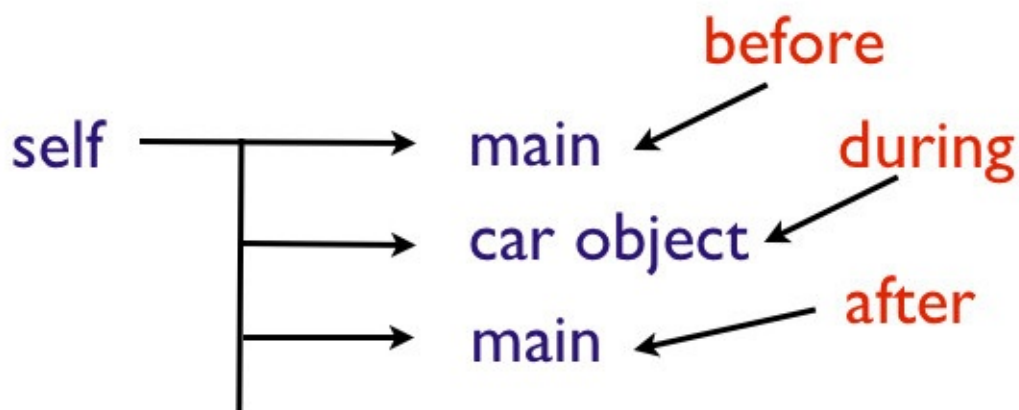
```
class Car
  def drive
    'driving'
  end
end

car = Car.new
car.tap { |x| p "The value of self is : #{x}" }.drive
p car
```

This prints:

```
"The value of self is : #<Car:0x007f854b141ea0>"
#<Car:0x007f854b141ea0>
```

The memory address of self and the car object is the same. Thus, they are the same object.



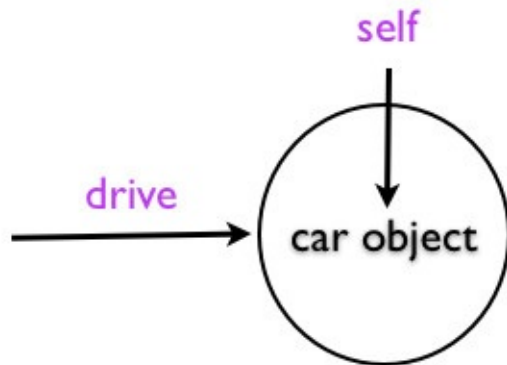
The Dynamic Nature of Self

The diagram illustrates the value of self before, during and after executing the drive method.

Rhonda Asks

What is self ?

The self refers to the object inside which the current method is executing. It is the receiver of the current executing method.



The self is Car Object

Summary

In this chapter, you learned that Ruby changes the value of `self` as it executes the program. There is always a `self`. The value of `self` changes to the current executing object and Ruby sends a message to that object. Once the method completes the execution, the value of `self` changes again. You know that `self` cannot be assigned any value to it. But you can use language constructs such as class, module and method declarations to make Ruby change the value of `self`. We will discuss this in the next chapter.

When Does self Change?

In this chapter, you will learn that the value of self changes whenever Ruby encounters the class, module or def keyword.

Self at Top Level

Open your code editor and print the value of self.

```
p self
```

This prints:

```
main
```

Self Inside a Class

Define a class and print the value of self.

```
class A
  p self
end
```

This prints:

```
A
```

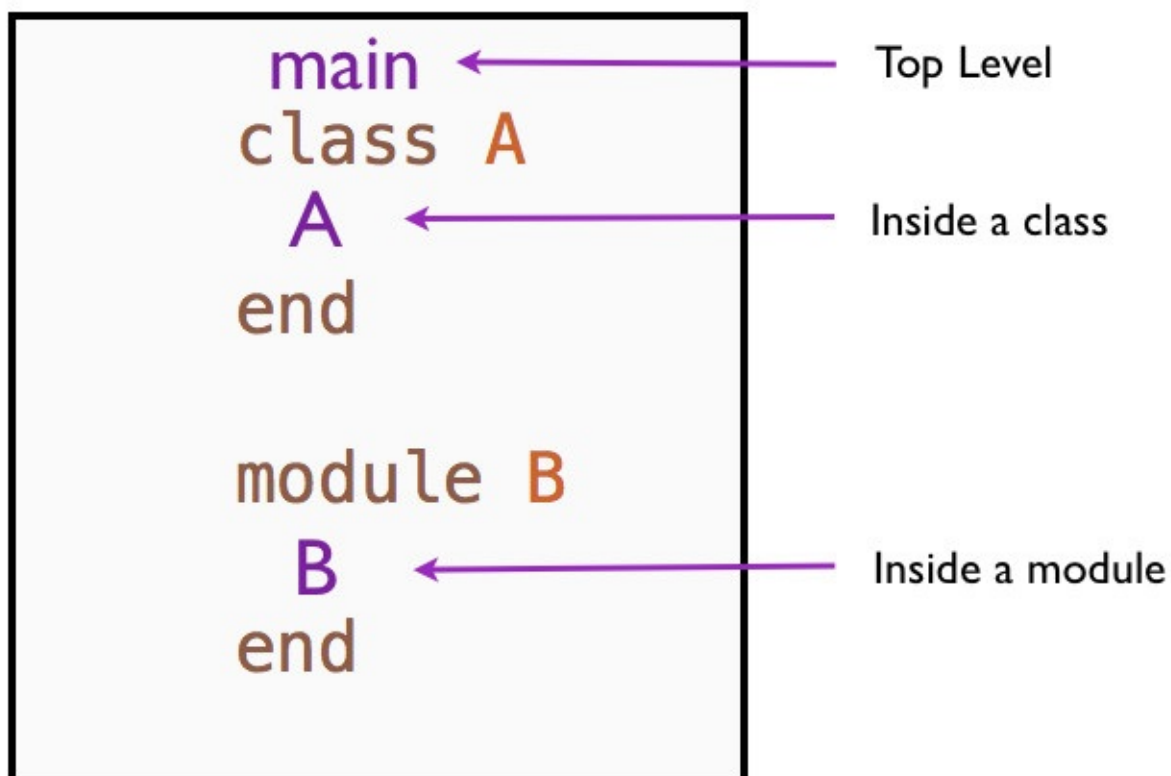
Self Inside a Module

Define a module and print the value of self.

```
module B  
  p self  
end
```

This prints:

B



The value of self

Self Inside a Method in a Class

Define a method inside the class A and print the value of self.

```
class A
  def m
    p self
  end
end
```

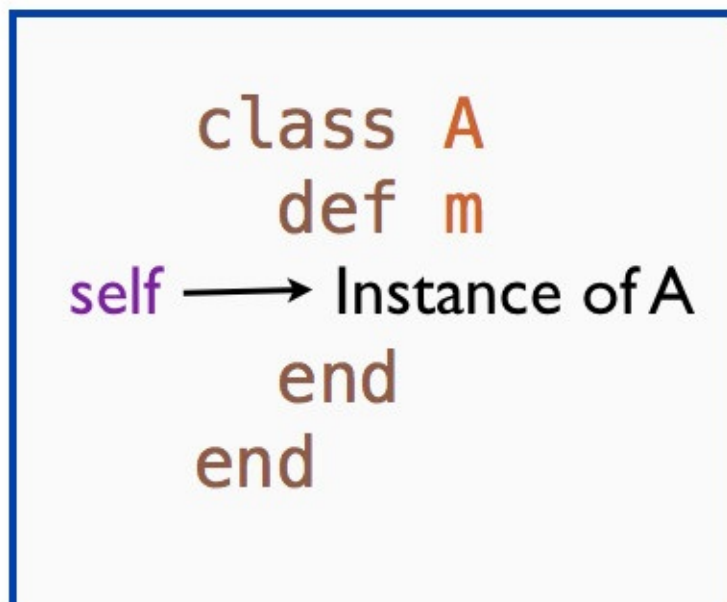
This prints nothing. Until now Ruby executed code as it read the code. In this case, it does not execute the method as it reads it. It makes the method available for instances of class A to call it. Let's create an instance of A to call the method **m()**.

```
class A
  def m
    p self
  end
end

a = A.new
a.m
```

This prints the instance of A:

```
#<A:0x007fec6b827210>
```



The value of self inside a method in a class

Self Inside a Method in a Module

Define a method in module B and print the value of self inside the method.

```
module B
  def mw
    p self
  end
end
```

This prints nothing. We can mixin the module B in a class. Create an instance of that class and call the method **mw()**.

```
module B
  def mw
    p self
  end
end

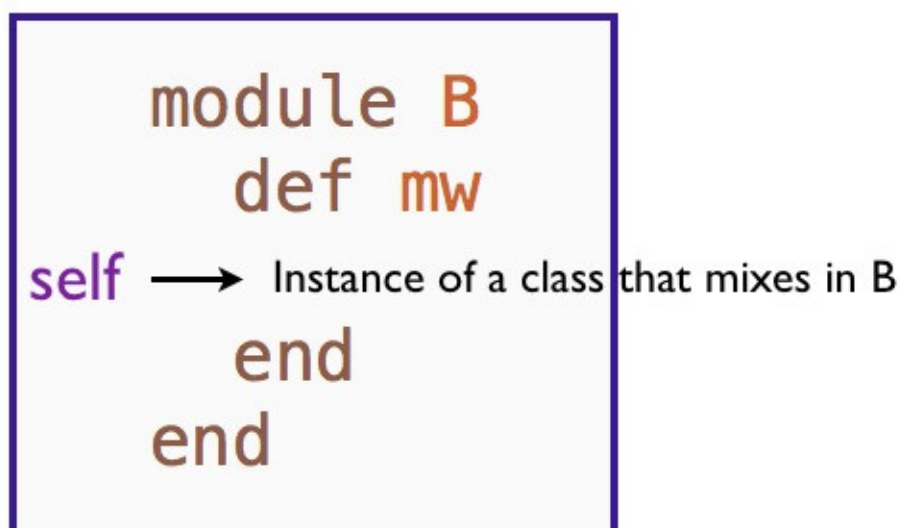
class Tester
  include B
end

t = Tester.new
t.mw
```

This prints:

```
#<Tester:0x007fdf3c805ef8>
```

The value of self inside the method defined in a module is the instance of Tester class.



The value of self inside a module

Self When Extending a Module

We can extend the module and call the class method `mw()`.

```
module B
  def mw
    p self
  end
end

class Tester
  extend B
end

Tester.mw
```

This prints:

```
Tester
```

Self Inside a Module Method

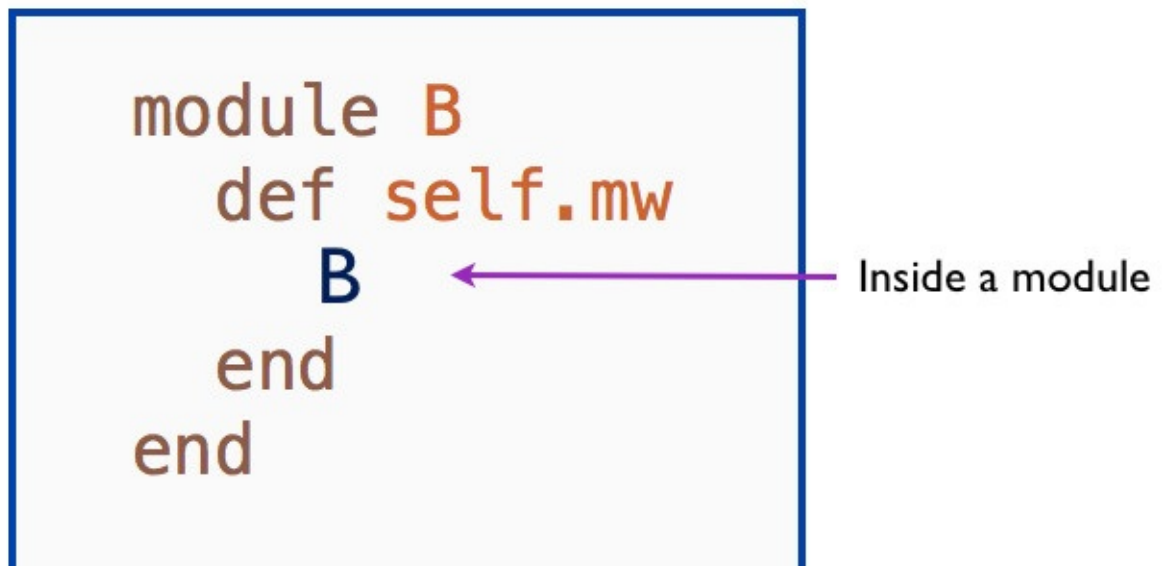
Define a class method in a module to print the value of self.

```
module B
  def self.mw
    p self
  end
end
```

```
B.mw
```

This prints:

```
B
```



Self Inside a Class Method in Module

Self Inside a Class Method

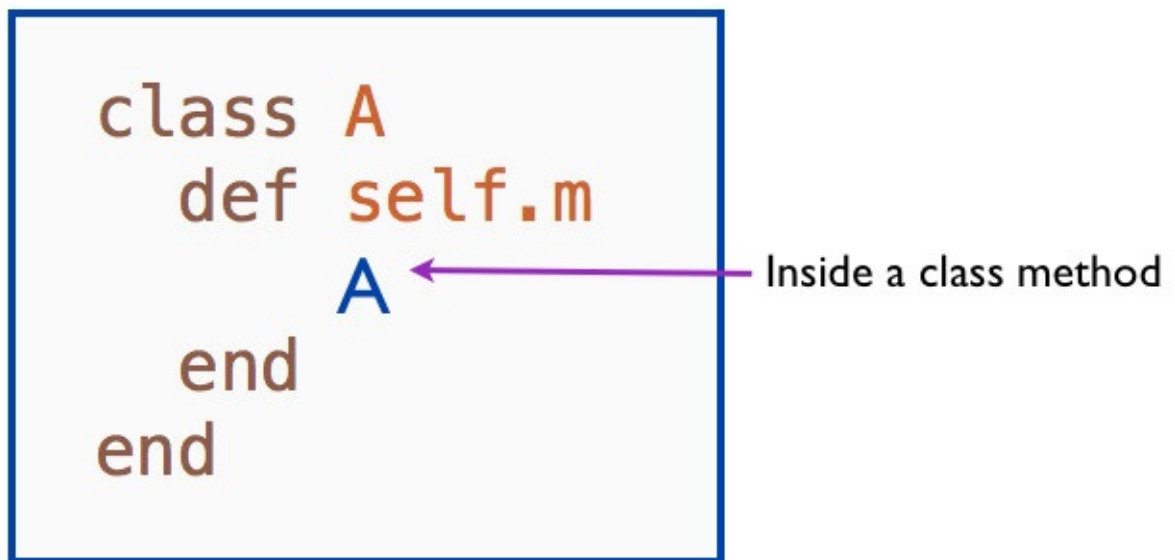
Define a class method in a class to print the value of self.

```
class A
  def self.m
    p self
  end
end

A.m
```

This prints:

```
A
```



Self Inside a Class Method in Class

Summary

The following list summarizes what you learned in this chapter.

Self Location	Self Value
Top Level	main
Inside a Class	Class name
Inside a Module	Module name
Inside a Method in a Class	Instance of the class
Inside a Method in a Module	Instance of the class that mixes in the module
Inside a Method in a Module	Class name that extends the module
Inside a class Method in a Module	Module name
Inside a class Method in a Class	Class name

The main Object

In this chapter, you will learn about the main object and that the instance variables at the top level is bound to the main object.

In Ruby, everything is executed in the context of some object. The methods are bound to the value of self. Whenever self points to main, the methods are bound to main.

What is a main Object?

The **main** is the object at the top level. It is an instance of Object. Any methods defined in **main** become instance methods of Object. This makes them available everywhere, meaning that we can call the method without a receiver.

Memory Location of main Object

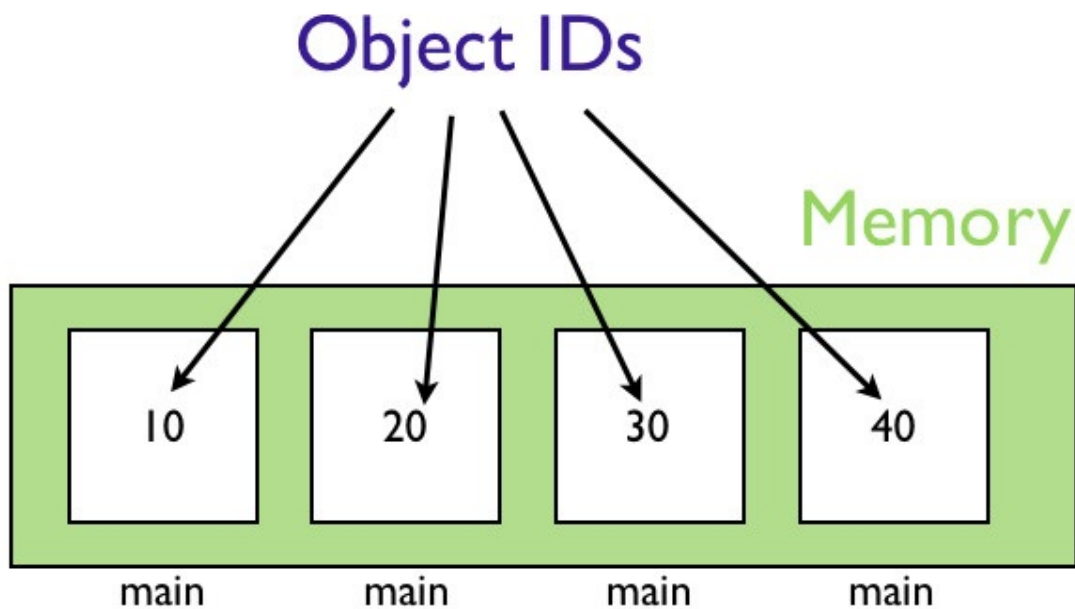
Let's look at an example.

```
puts object_id
```

This will print the `object_id`. On my computer, it is:

```
10
```

The `object_id` will change every time you run the program. Because, it represents the integer identifier for the main object. Ruby creates a new **main** object every time you run the program.



The object ids differ for different main objects

The above code is same as:

```
puts self.object_id
```

Memory Location of Instance Variable

Let's print the instance variable at the top level.

```
@age = 22  
puts @age
```

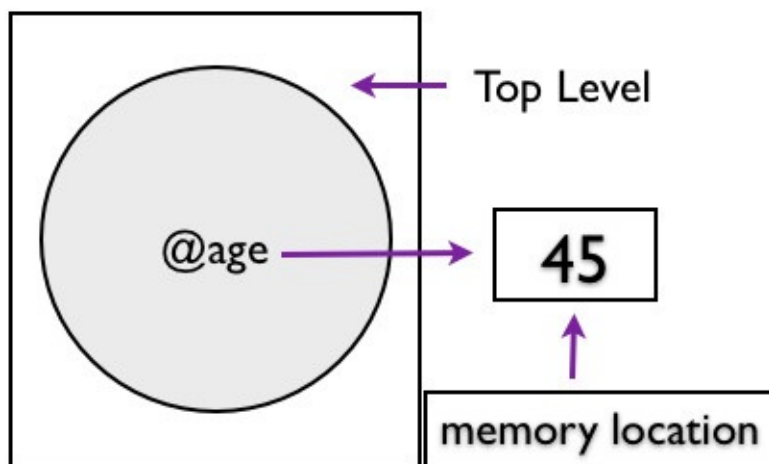
This prints:

```
22
```

The age will be the same every time you run the program. The `object_id` of the number **22** will be the same. We can verify this:

```
@age = 22  
puts @age.object_id
```

This always prints **45** on my laptop. The reason is that Ruby reuses some objects of built-in classes for optimization.



Object ID of Instance Variable

Instance Variables at the Top Level

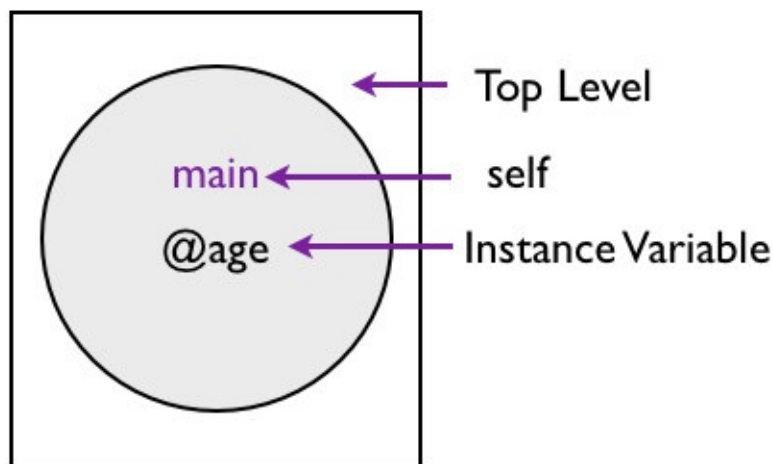
Instance variables defined in the top level context are also bound to the main object. Let's look at an example.

```
@age = 22  
p self.instance_variables
```

This prints:

```
[:@age]
```

We know that the value of **self** is **main** at the top level. The output shows that age instance variable is bound to the main object.



Instance Variable at the Top Level

Methods at the Top Level

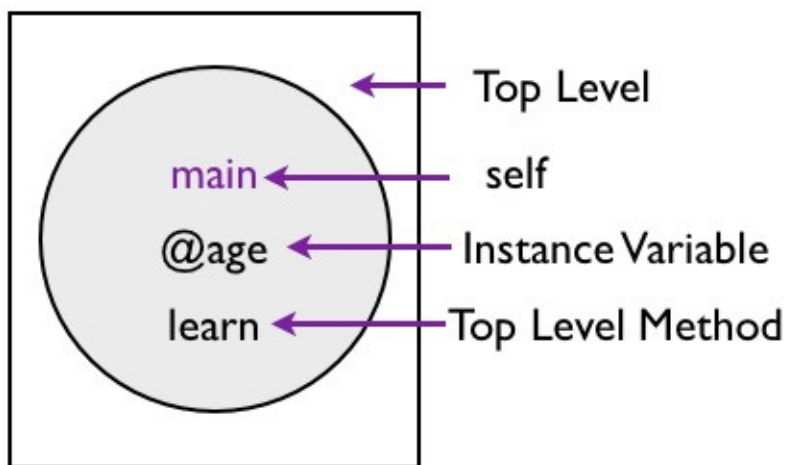
Let's define a method at the top level.

```
def learn
  p 'learning'
end
```

```
learn
```

This prints:

```
learning
```



Method at the Top Level

as the program executes.

Let's trace the value of `main`

```
p self
def learn
  p self
  p 'learning'
end
p self
learn
```

This prints:

```
main
main
main
learning
```

The value of `self` is `main` throughout the program.

```
main
def learn
  main
end
main
```

The self Remains the main

Fabio Asks

Can we use self to call this method?

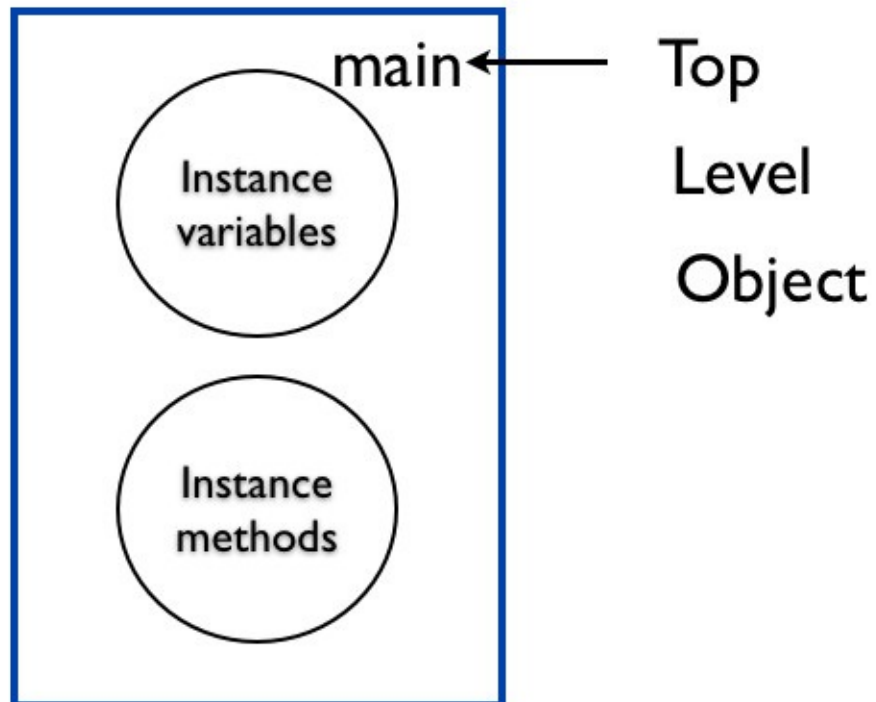
```
def learn
  p 'learning'
end

self.learn
```

This results in an error.

```
NoMethodError: private method 'learn' called for main:Object
```

The methods defined at the top level becomes private method in the Object. This shows that these methods are bound to the main object.



The main bound instance methods and instance variables

Rhonda Asks

Why does Ruby add the top level methods as the private methods to the main object?

The reason is that if it adds it as a public method, sub classes will inherit those methods. This will pollute the sub classes with unnecessary methods in the public interface.

Summary

In this chapter, you learned about the main object. You learned how the instance variables and methods defined at the top level is bound to the main object.

Message Sender at the Top Level

In this chapter you will learn that the **main** is the message sender when we call methods in the top level context.

Main the Message Sender

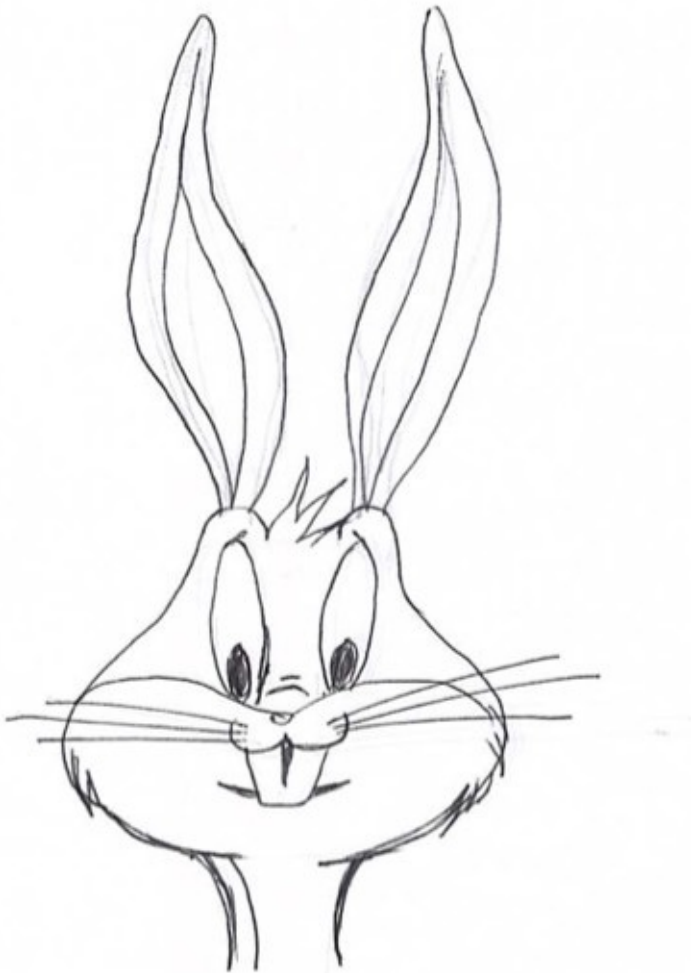
Let's write a simple program to illustrate that **main** is the message sender at the top level.

```
class Rabbit
  def funny?
    true
  end
end

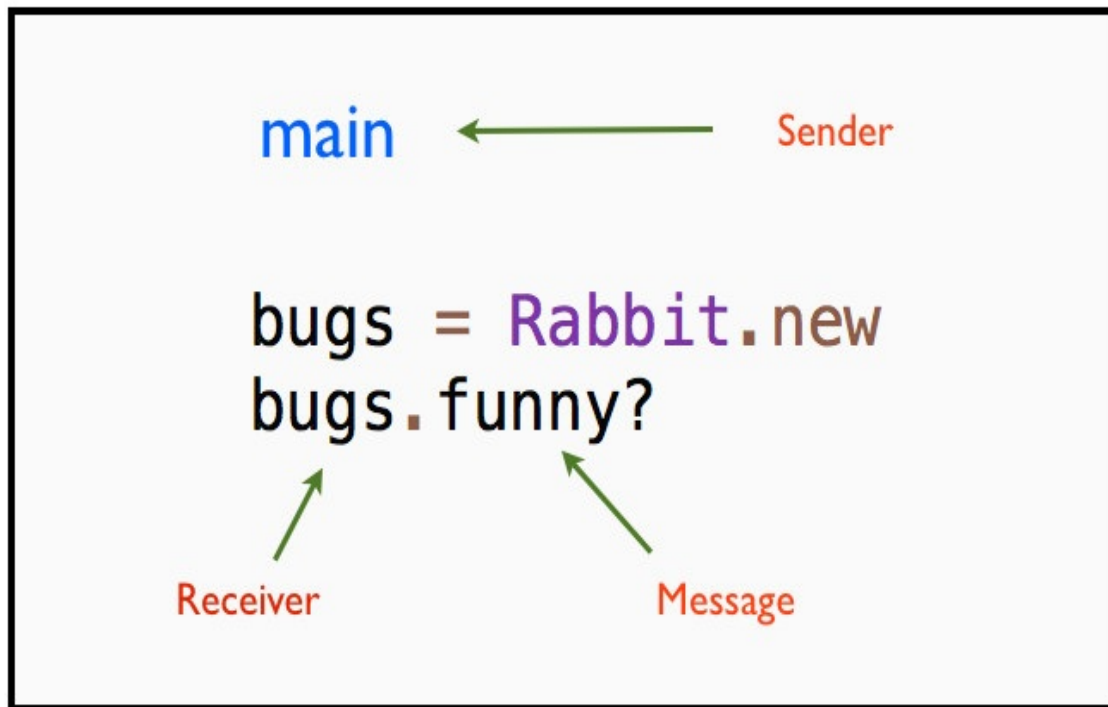
bugs = Rabbit.new
p bugs.funny?
```

This prints:

```
true
```



The message sender is the value of **self**, which is **main** in the top level context. The variable `bugs` is the message receiver and the message sent is **funny?**



Sender, Receiver and Message at Top Level Context

Implicit Sender for Hail Taxi

Let's now find out the implicit sender for our simple hail taxi program. Where did the message originate? Message originates at:

```
3.times
```

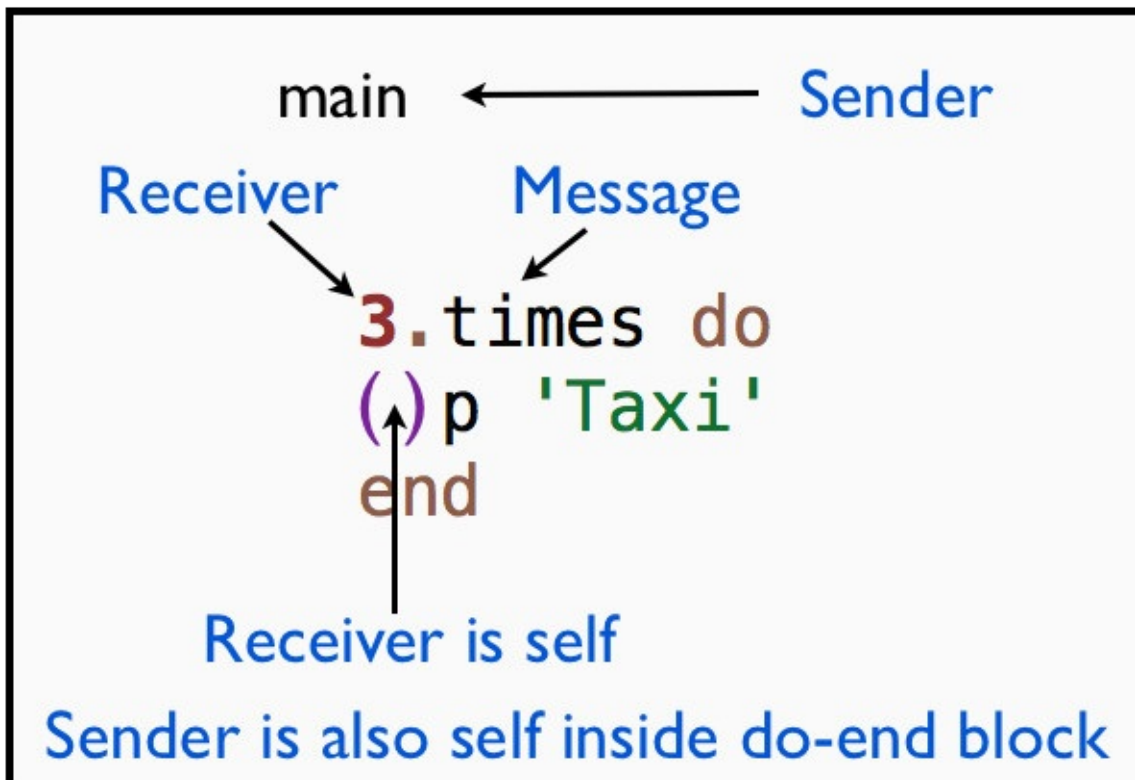
So, let's print the value of self before and after this line of code.

```
puts "outside loop self is #{self}"
3.times do
  puts "inside loop self is #{self}"
  p 'Taxi'
end
```

This prints:

```
outside loop self is main
inside loop self is main
"Taxi"
inside loop self is main
"Taxi"
inside loop self is main
"Taxi"
```

This demonstrates that **main** is the object that is sending the message. Thus, **main** is the sender. The self is acting as the default sender. The sender is implicit in the code, so it is invisible. But it is there in the context of executing the example hail taxi program. The Fixnum object 3 is the receiver and **times** is the message.



Inside the do-end block, the sender and receiver are the same. Because the receiver of the

p() and **puts()** methods is also main. This is subtle. If you understand this concept, you will have a solid foundation to become a Ruby expert.

Implicit Sender for Teacher Program

In the *Message Passing* chapter, we saw the explicit sender for the teacher program. Let's take a look at that example so that we can identify the implicit sender.

```
class Teacher
  def initialize(student)
    @student = student
  end

  def ask_student_name
    @student.ask_name
  end
end

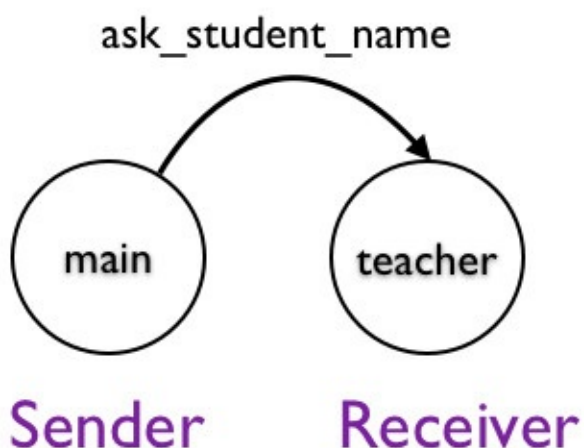
class Student
  def initialize(name)
    @name = name
  end

  def ask_name
    @name
  end
end

student = Student.new('Bugs Bunny')
teacher = Teacher.new(student)

p teacher.ask_student_name
```

The implicit sender in this program is the main object. Look at the last line in the example. You can see that the teacher object is the receiver. The main sends the message `ask_student_name` to the teacher object. The reason is that at the top level, **main** is the sender object.



Implicit Sender

Fabio Asks

How do I identify the message sender?

Ask yourself the questions:

1. Where did the message originate?
2. Who is the owner of the scope where the message originated?

Rhonda Asks

What is the message receiver in the teacher example?

You can ask Ruby:

```
def ask_student_name
  puts "The sender object is : #{self.class}"
  @student.tap do|x|
    p "Before invoking ask_name, sender object is : #{self.class}"
    p "Before invoking ask_name, receiver object is : #{x.class}"
  end.ask_name
end
```

The message receiver is student object.

Summary

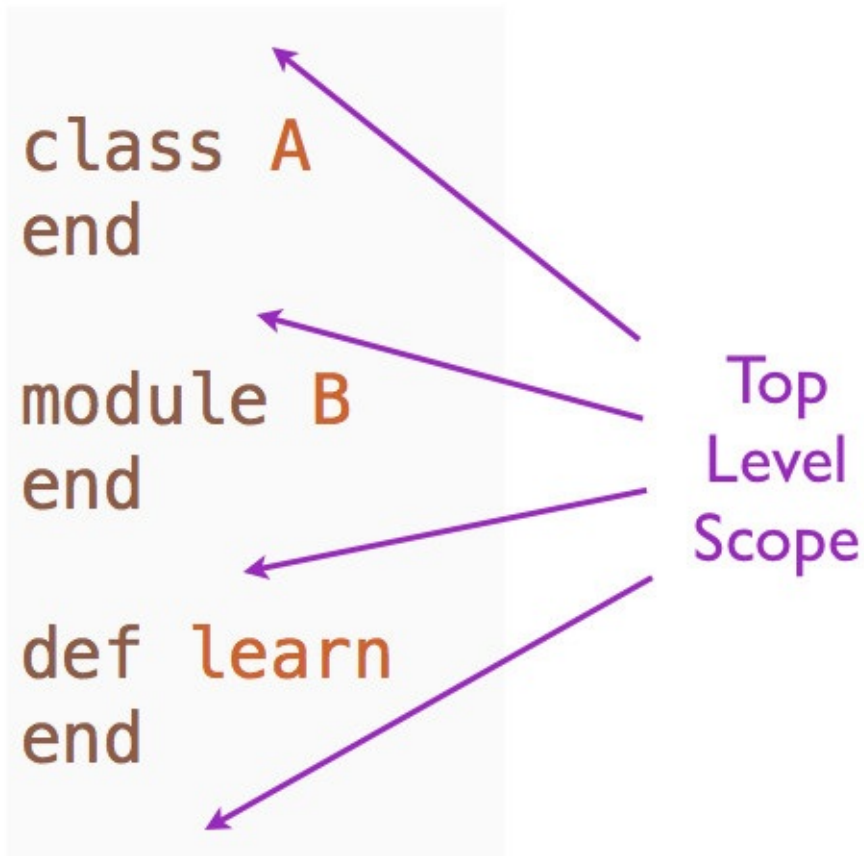
We saw an example for implicit sender where the sender object was implicit and hidden in the code. We also saw that implicit sender and implicit receiver are the same inside the do-end block for the hail taxi program. What is the use of knowing that the sender and receiver are the same? We will discuss this in the upcoming chapters on *Same Sender and Receiver* and *Private Methods*.

Top Level Methods

In this chapter, you will learn that the methods defined in the top level context are bound to the main object.

What is Top Level Method?

Top level methods are methods defined in the top level scope. They are not inside a class or module.



Top Level Methods

Top Level Method

Let's define a method at the top level.

```
def speak  
  p 'speaking'  
end
```

Call Top Level Method

We can call this method like this:

```
    speak
```

This prints:

```
    speaking
```

Private Method

Ruby programs bind methods defined in the top level scope to main as private methods. We can verify this:

```
p self.private_methods.include?(:speak)
```

This prints:

```
true
```

Inaccessible in the Self

Ruby programs do not make the top level methods available in the self object.

```
def speak
  p 'I am speaking'
end

p self.public_methods.include?(:speak)
```

This prints:

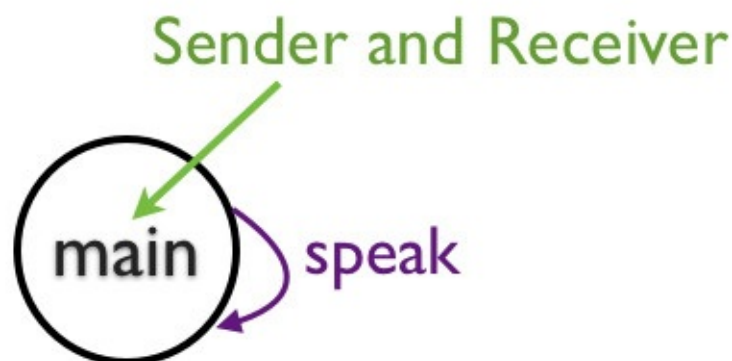
```
false
```

This means, if you call speak method on self:

```
self.speak
```

You will get the error:

```
NoMethodError: private method 'speak' called for main:Object
```



Sending a Message to a Private Method

We will discuss about private methods and how the sender and receiver is the same in the next chapter.



Sending Message to Self

The IRB Convenience

The IRB binds methods in the top level scope to main as public methods for convenience. We can verify it like this:

```
$irb
> def speak
>   p 'speaking'
> end
=> :speak
> speak
"speaking"
=> "speaking"
> self.public_methods.include?(:speak)
=> true
```

Summary

In this chapter, you learned that the methods defined in the top level context are bound to the main object.

Same Sender and Receiver

In this chapter, you will learn that when the sender and receiver are the same, we cannot use an explicit receiver to call a private method.

Functional Form

Let's define a private method **start()** in **Car**. And call it within **drive()** method in functional form.

```
class Car
  def drive
    start
  end

  private

  def start
    p 'starting..'
  end
end

c = Car.new
c.drive
```

This prints:

```
starting..
```

Explicit Receiver

Let's use an explicit receiver to call the `start()` private method.

```
class Car
  def drive
    self.start
  end

  private

  def start
    p 'starting...'
  end
end

c = Car.new
c.drive
```

This prints:

```
NoMethodError: private method 'start' called for <Car:0x007fc5d303ee68>
```

This results in the same output as the following:

```
class Car
  private

  def start
    p 'starting...'
  end
end

c = Car.new
c.start
```

The self and the Receiver

Let's check the value of `self` inside the `drive()` method and the receiver of the `drive()` method.

```
class Car
  def drive
    p "self is : #{self}"
    self.start
  end

  private

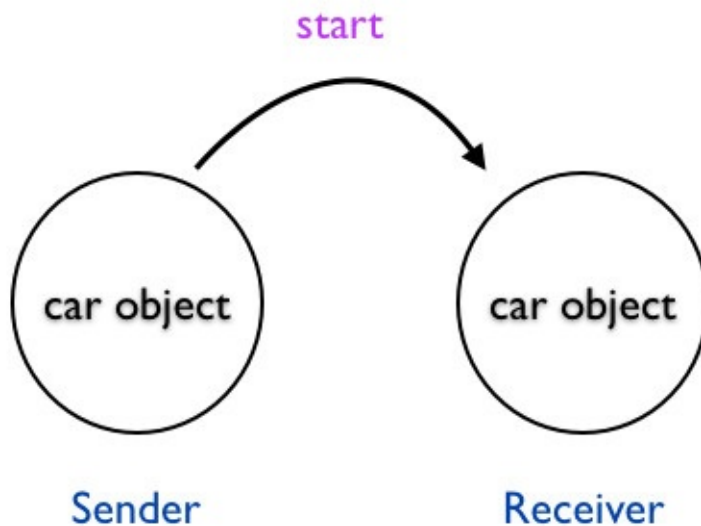
  def start
    p 'starting...'
  end
end

c = Car.new
p "receiver is : #{c}"
c.drive
```

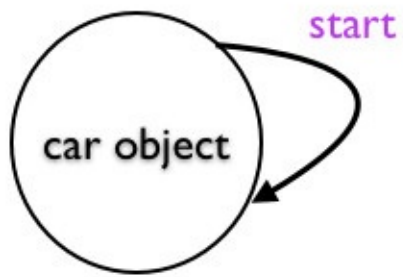
This prints:

```
receiver is : #<Car:0x007fdd8c1c5900>
self is : #<Car:0x007fdd8c1c5900>
NoMethodError: private method 'start' called for #<Car:0x007fdd8c1c5900>
```

Inside the `drive` method the sender and receiver are the same. The sender and the receiver objects are the same instance of `Car` class. In such cases, Ruby does not allow providing an explicit receiver when you want to call a private method.



Same Sender and Receiver



Sender and Receiver

Same Sender and Receiver

The Public Method

If you make the **start()** method public, it will work.

```
class Car
  def drive
    self.start
  end

  def start
    p 'starting..'
  end
end

c = Car.new
c.drive
```

As expected, this prints:

```
driving...
```

The Protected Method

Let's change the `start()` method to protected.

```
class Car
  def drive
    self.start
  end

  protected

  def start
    p 'starting...'
  end
end

c = Car.new
c.drive
```

It still works.

Summary

You learned what happens when the receiver and the sender objects are the same when calling a private method. In such cases:

- You cannot provide an explicit receiver to call a private method.
- There is no receiver and dot symbol to send a message.
- You have to call the private method in functional form.

Top Level Private Methods

In this chapter, you will learn how the top level context and private methods work together.

Implicit Receiver

Let's print hello in the standard output.

```
puts 'hello'
```

As expected, this prints:

```
hello
```

Explicit Receiver



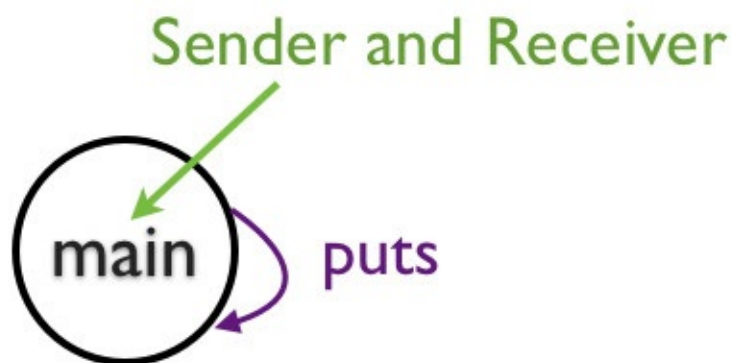
If you use an explicit receiver, the **self** to call the puts:

```
self.puts 'hi'
```

We get:

```
NoMethodError: private method 'puts' called for main:Object
```

This is because **puts** is a private method in Object. In Ruby you cannot have an explicit receiver to call a private method. Why? Because, when the sender and receiver are the same, you cannot use an explicit receiver to send a message.



Sending a Message to a Private Method

The Kernel Module

Where does the private method **puts()** live? Let's search for methods that begin with *put*.

```
Kernel.methods.grep(/put/)
```

This prints:

```
[:putc, :puts]
```

It lives in Kernel module. Ruby mixes in the Kernel module into the Object class. That's how it is available as a private method on the Object.

Private Method and Explicit Receiver

How do we grab the **main**, the top level default object? Let's grab the value of the current object from the **self** and assign it to our own variable **m**.

```
m = self
m.puts 'hi'
```

We get the same error message as we did in step 2.

```
NoMethodError: private method 'puts' called for main:Object
```

Why do we get this error? You can think of the code that gives error to be equal to this:

```
class Object
  private

  def puts(arg)
    # Implementation for printing to standard output
  end
end
```

We cannot call the **puts** private method with an explicit receiver.

Forcing an Explicit Receiver

We can use `send()` method to send the `puts()` message to the `main` object:

```
m = self  
m.send(:puts, 'hi')
```

This will work. This is the same as:

```
self.send(:puts, 'hi')
```

But, this breaks encapsulation and is generally not a good idea unless you have a good reason to do so.

Using Method Object

We can also do this:

```
m = self.method(:puts)
=> #<Method: Object(Kernel)#puts>
> m.call('hello')
hello
=> nil
```

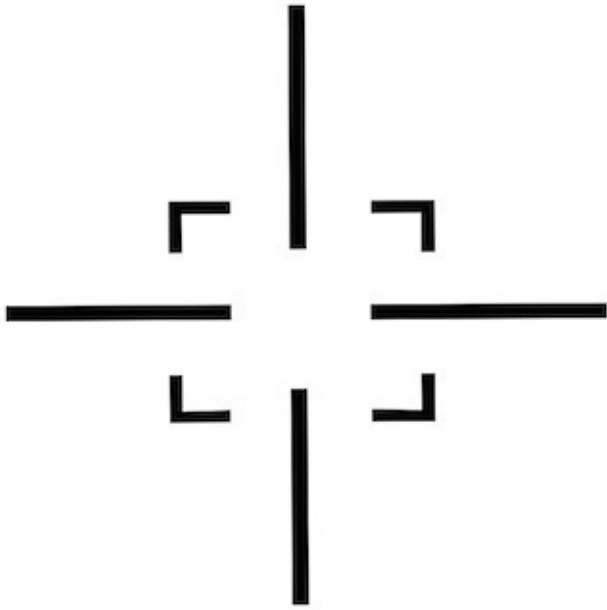
You can see how using the method object displays the relationship between the Kernel module and Object class. The **puts()** method is available by mixing in the Kernel module into the Object class.

Summary

In this chapter we saw how the private methods and the top level context work together. The concept you learned in the previous chapter is a generalized concept of the concept in this chapter.

Scope of Local Variables

In this chapter, you will learn about the visibility of local variables.



Scope

The focus is on using language constructs such as `def`, `class` and `module` and how it affects the visibility.

At the Top Level

Visibility of Local Variable

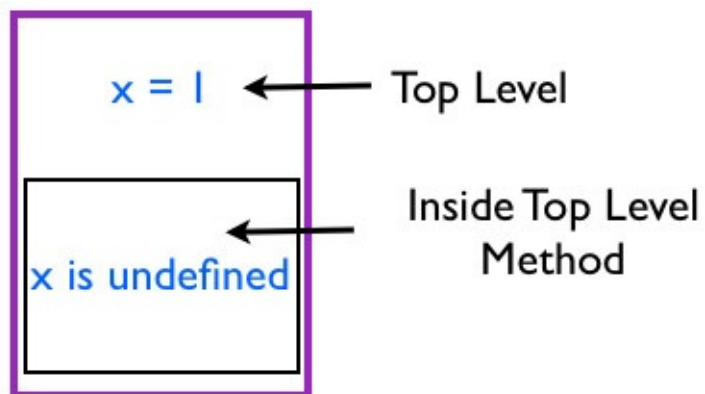
Let's look at an example where we have a local variable at the top level and see if we can access it inside a top level method.

```
x = 1
p x
def test
  p x
end
test
```

This prints 1 and then the error.

```
NameError: undefined local variable or method 'x' for main:Object
```

Thus, we can see that the top level local variable `x` is not accessible inside the top level method.



The value of `x` at the Top Level and Method

We can verify this fact by asking Ruby.

```
x = 1
p "Local variables at the top level : #{local_variables}"
def test
  p "Local variables inside the test method : #{local_variables}"
end
test
```

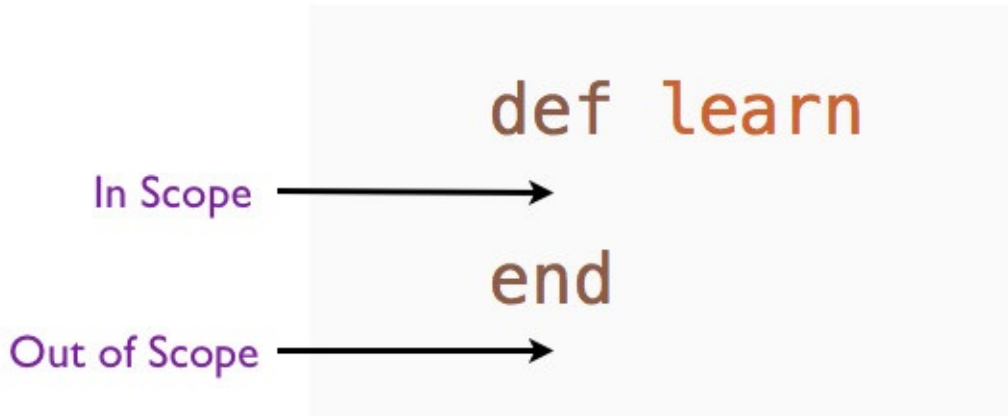
This prints:

```
Local variables at the top level : [:x]
```

Local variables inside the test method : []

This shows that at the top level, there is a local variable x, whereas, inside the top level method, there is none. This is an example where self does not change but scope changes.

Any variables defined within a method goes out of scope when we exit the method. The local variable is garbage collected. For instance, any variable defined in a learn method is not available outside of that method.



Scope of Variables Inside a Top Level Method

The diagram illustrates the life time of a variable defined within a method.

Inside a Top Level Method

Local Variables in Different Scopes

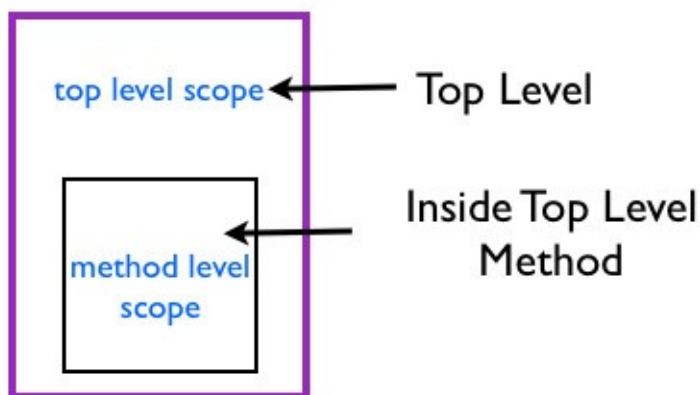
What happens if we define a local variable inside the top level method? Let's write a simple program that defines local variable at the top level and inside a top level method.

```
x = 1  
  
p "At top level x is : #{x}"  
  
def test  
  x = 2  
  p "Inside the top level method x is : #{x}"  
end  
  
test  
  
p "Back at the top level x is : #{x}"
```

This prints:

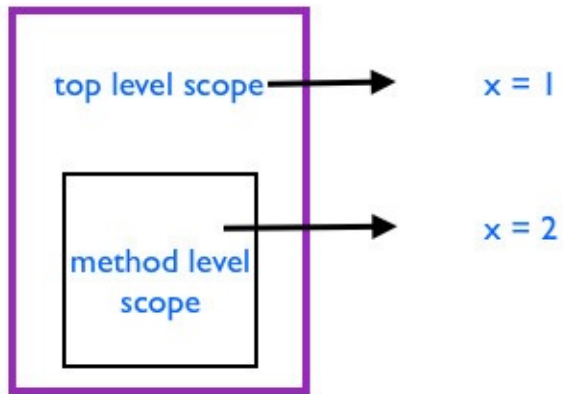
```
At top level x is : 1  
Inside the top level method x is : 2  
Back at the top level x is : 1
```

Top level has its own scope. The method definition has its own scope.



The Scope at the Top Level and Inside Method

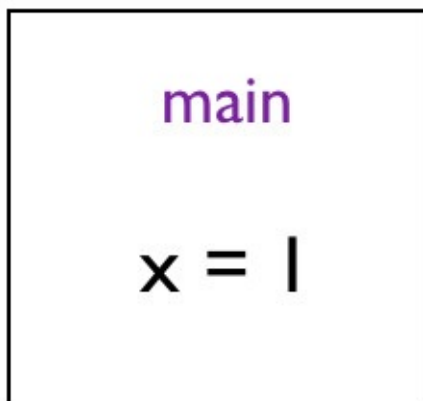
That's why we see x is 1 at top level and it is 2 inside the method definition. The variable names are the same. However, the x at the top level and the x inside the method are variables in different scopes. Thus, they are different.



Different Local Variables with Same Name

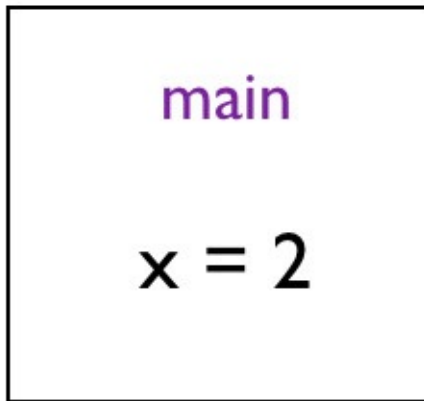
This is an example where self does not change, but scope changes. We know that the self is main at the top level as well as inside the method.

The Top Level



The self and Local Variable

The Top Level Method



The self and Local Variable

Let's trace the local variables in different scopes as the program executes.

```
x = 1

p "At the top level local variables:"
p local_variables

def test
  x = 2
  p "Inside the top level method, local variables:"
  p local_variables
end

test
```

This prints:

```
At the top level local variables:
[:x]
Inside the top level method, local variables:
[:x]
```

The name of the local variables is the same. From the previous experiment we know they are different.

Inside a Class

Visibility of Local Variable

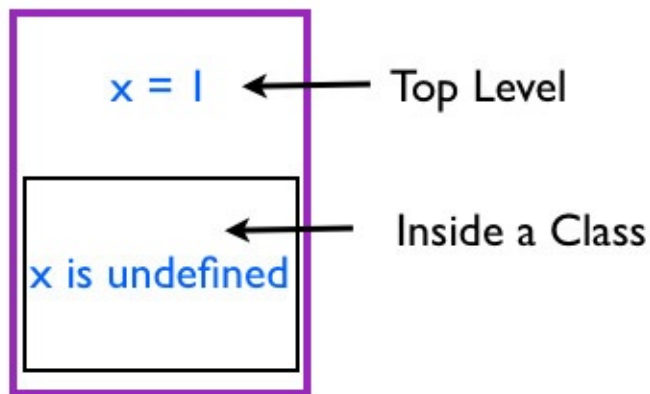
What happens when we try to access a local variable defined at the top level from a class?

```
x = 1  
  
class A  
  p x  
end
```

This gives us the error:

```
NameError: undefined local variable or method 'x' for A:Class
```

The local variable `x` defined at the top level is not visible within the class.



The value of `x` at the Top Level and Class

This is an example where both self and scope changes.

Local Variables in Different Scopes

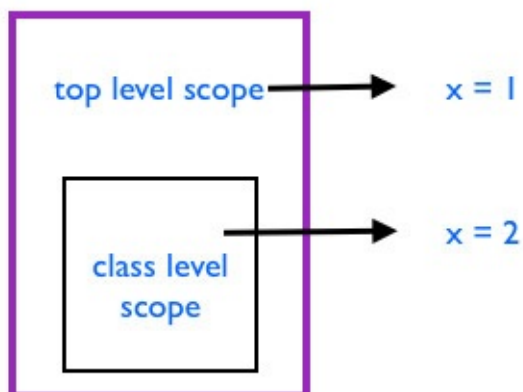
Let's write a simple program that defines a local variable at the top level and inside a class.

```
x = 1  
  
p "At the top level, x is #{x}"  
  
class A  
  x = 2  
  p "Inside the class, x is : #{x}"  
end  
  
p "Back at the top level, x is #{x}"
```

This prints:

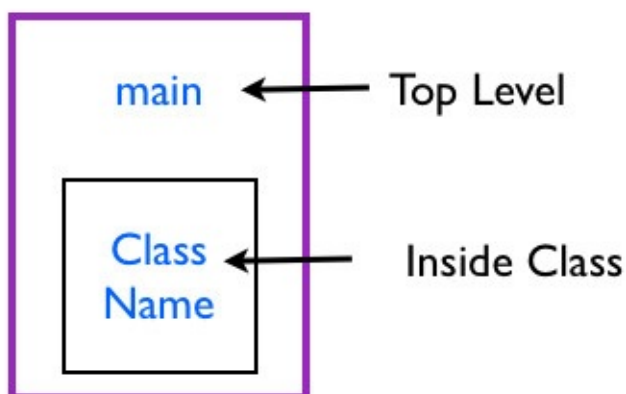

```
At the top level, x is 1
Inside the class, x is : 2
Back at the top level, x is 1
```

The value of x inside the class is different from the x at the top level. Variables inside the class have its own scope. The x at the top level is different from the x inside the class.



Different Local Variables with Same Name

This is an example where both self and scope changes. The self at the top level is main and the self inside the class is A.



The self at the Top Level and Inside Class

Let's trace the local variables.

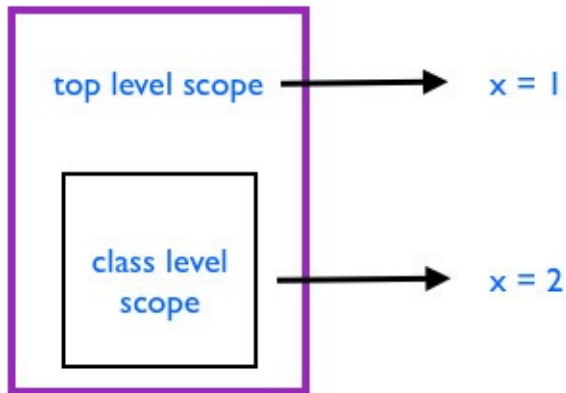
```
x = 1
p "Local variables at the top level : #{local_variables}"

class A
  x = 2
  p "Local variables inside the class : #{local_variables}"
end
```

This prints:

```
Local variables at the top level : [:x]
Local variables inside the class : [:x]
```

The name of the local variable is the same. But they are different because they belong to different scopes. The first one belongs to the top level and the second one belongs to the method level scope.



Different Local Variables with Same Name

Let's take a look an example.

```
x = 1
p "Local variables at the top level : #{x}"

class A
  x = 2
  p "Local variables inside the class : #{x}"
end

p "Local variables at the top level : #{x}"
```

This prints:

```
Local variables at the top level : 1
Local variables inside the class : 2
Local variables at the top level : 1
```

Inside a Module

Visibility of Local Variable

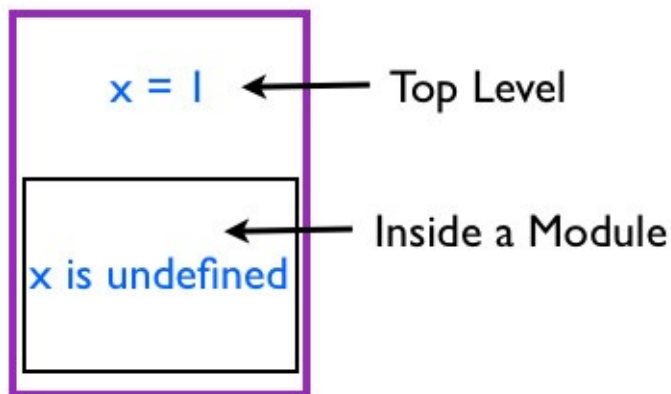
What happens when we try to access a local variable defined at the top level from a module?

```
x = 1
module B
  p x
end
```

This gives us the error:

```
NameError: undefined local variable or method 'x' for B:Module
```

The local variable `x` defined at the top level is not visible within the module.



The value of `x` at the Top Level and Module

This is an example where both self and scope changes.

Local Variables in Different Scopes

Let's write a simple program that defines local variable at the top level and inside a module.

```
x = 1
p "At the top level, x is : #{x}"

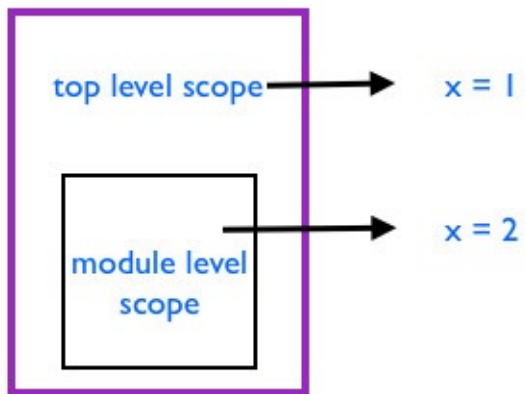
module B
  x = 2
  p "Inside the module, x is : #{x}"
end

p "Back at the top level, x is : #{x}"
```

This prints:

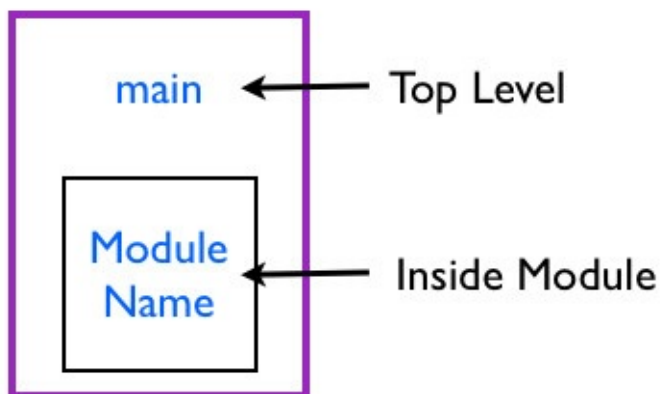
```
At the top level, x is : 1
Inside the module, x is : 2
Back at the top level, x is : 1
```

The variables inside Module also have its own scope.



Different Local Variables with Same Name

The self and scope both change in this example. The self is main at the top level and the self is B inside the module.



The self at the Top Level and Inside Module

Discovery Exercise

Write a program to trace the local variables for the above example.

The Grand Example

Let's combine top level, class definition and method definition into one example.

```
x = 1

p "At the top level, x is : #{x}"

def test
  x = 2
  p "Inside top level method, x is : #{x}"
end

test

class A
  x = 3
  p "Inside the class, x is : #{x}"
end

module B
  x = 4
  p "Inside the module, x is : #{x}"
end

p "Back at the top level, x is : #{x}"
```

This prints:

```
At the top level, x is : 1
Inside top level method, x is : 2
Inside the class, x is : 3
Inside the module, x is : 4
Back at the top level, x is : 1
```

We see that top level, top level method, class and module have their own scopes and the value of x is specific to their own scope.



In the last example, does the self and scope change together? If so, what are their values as the program executes?

Key Takeaways

Here is a summary of how self and scope changes.

Self	Scope	Where
No Change	Changes	Top Level and Top Level Method

Changes	Changes	Top Level and Inside Class
Changes	Changes	Top Level and Inside Module

Summary

In this chapter, we experimented with the scope of variables in four different scenarios.

1. At top level scope
2. Method definition scope
3. Class definition scope
4. Module definition scope

We found that each of these, have their own local variables. The class, module or def keyword creates a new local scope.

Scope of Variables Redux

In this chapter, you will learn about the visibility of local variables in the context of dynamic language constructs such as `define_method`, `Class.new` and `Module.new`.

Scope

How does dynamically defining methods, classes and modules affect the local variable visibility?

At the Top Level

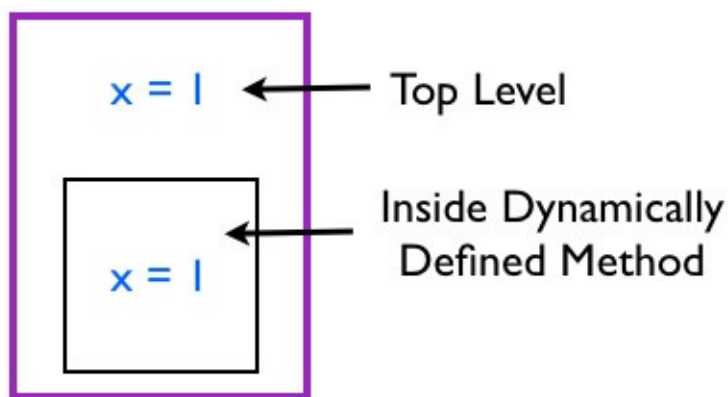
Visibility of Local Variable

```
x = 1  
  
p "At top level x : #{x}"  
  
define_method(:test) do  
  p "Inside top level method x : #{x}"  
end  
  
test
```

This prints:

```
At top level x : 1  
Inside top level method x : 1
```

The local variable is visible inside the dynamically defined method.



The value of x at the Top Level and Inside Method

We can verify it by checking the local variables.

```
x = 1  
  
p "At top level local_variables is : #{local_variables}"  
  
define_method(:test) do  
  p "Local variable inside the method : #{local_variables}"  
end  
  
test  
p "Back at the top level local_variables is : #{local_variables}"
```

This prints:

```
At top level local_variables is : [:x]  
Local variable inside the method : [:x]  
Back at the top level local_variables is : [:x]
```

The local variable defined at the top level is visible inside the dynamically defined method.

The Value of Self

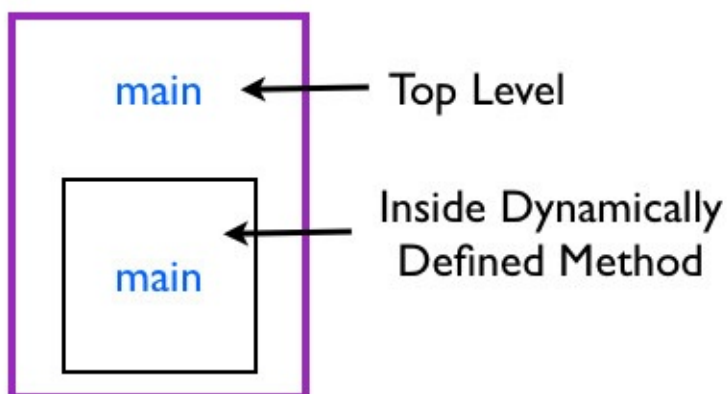
Does the value of self change? Let's check:

```
x = 1
p "At top level self : #{self}"
define_method(:test) do
  p "Inside top level method self : #{self}"
end
test
```

This prints:

```
At top level self : main
Inside top level method self : main
```

The value of self remains the same.



The self at the Top Level and Inside Method

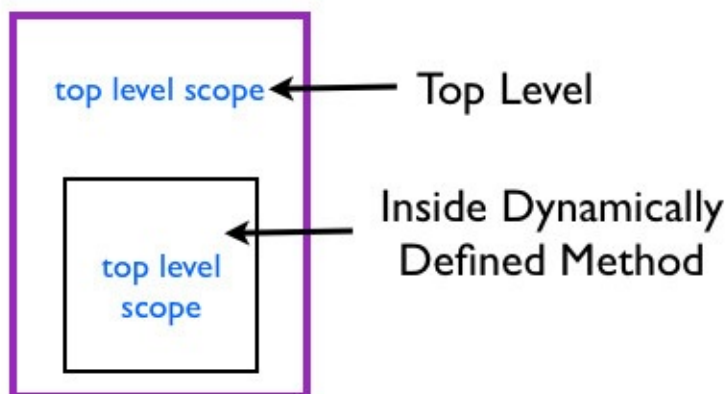
Inside the Top Level Method

```
x = 1  
p "At top level x is : #{x}"  
  
define_method(:test) do  
  x = 2  
  p "Inside the top level method x is : #{x}"  
end  
  
test  
p "Back at the top level x is : #{x}"
```

This prints:

```
At top level x is : 1  
Inside the top level method x is : 2  
Back at the top level x is : 2
```

The scope did not change. Thus, the x at the top level and inside the method is the same.



The Scope at the Top Level and Inside Method

At the Top Level

Inside a Class

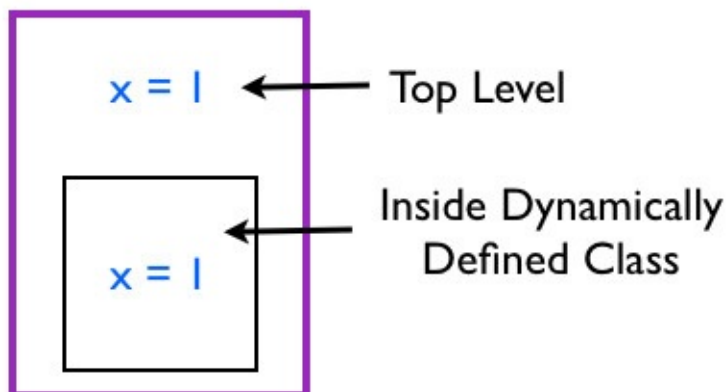
Let's check if we can see the local variable declared at the top level from inside a dynamically defined class.

```
x = 1
p "At top level x : #{x}"
Car = Class.new do
  p "Inside the Car class x : #{x}"
end
```

This prints:

```
At top level x : 1
Inside the Car class x : 1
```

The local variables defined at the top level is visible inside the Car class.



The value of x at the Top Level and Inside Class

We can print the local variables to verify that the variable x is the same.

```
x = 1
p "At top level : #{local_variables}"
Car = Class.new do
  p "Inside the Car class : #{local_variables}"
end
```

This prints:

```
At top level : [:x]
Inside the Car class : [:x]
```

Value of Self

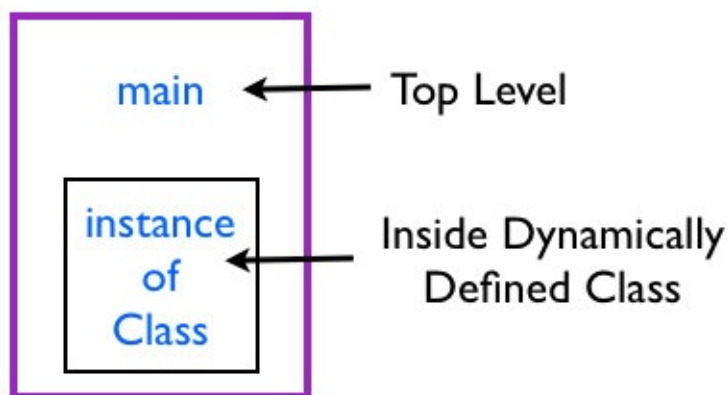
Does the value of self change? Let's check:

```
x = 1
p "At top level, self : #{self}"
Car = Class.new do
  p "Inside the Car class, self : #{self}"
end
```

This prints:

```
At top level, self : main
Inside the Car class, self : #<Class:0x007fff331c45f0>
```

The self changes from main to an instance of Class.



The self at the Top Level and Inside Class

Variable with Same Name Inside a Class

What happens when you have a variable with the same name inside the class?

```
x = 1
p "At top level x : #{x}"
Car = Class.new do
  x = 2
  p "Inside the Car class x : #{x}"
end
p "Back at the top level : #{x}"
```

This prints:

```
At top level x : 1
Inside the Car class x : 2
Back at the top level : 2
```

This changes the value of the same local variable defined at the top level.

At the Top Level

Inside a Module

Let's check if we can see the local variable declared at the top level from inside a dynamically defined module.

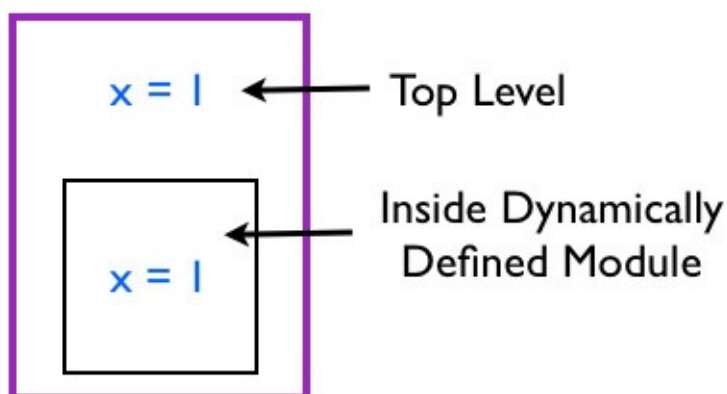
```
x = 1
p "At top level x : #{x}"

Driveable = Module.new do
  p "Inside the Driveable module x : #{x}"
end
```

This prints:

```
At top level x : 1
Inside the Driveable module x : 1
```

The local variables defined at the top level is visible inside the do-end block of creating a module.



The value of x at the Top Level and Inside Module

We can print the local variables to verify it.

```
x = 1
p "At top level : #{local_variables}"

Driveable = Module.new do
  p "Inside the Driveable module : #{local_variables}"
end
```

This prints:

```
At top level : [:x]
Inside the Driveable module : [:x]
```


Value of Self

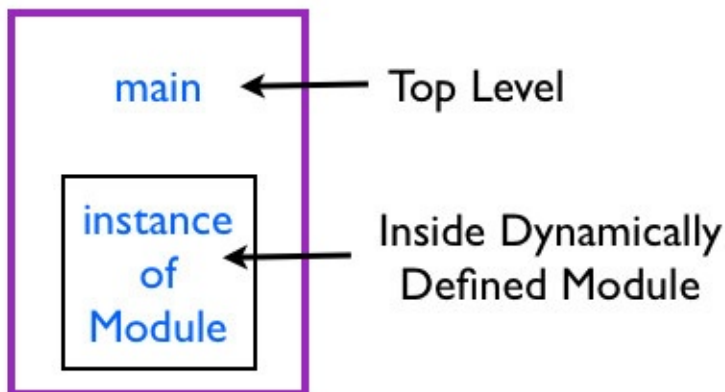
Does the value of self change inside the do-end block? Let's check:

```
x = 1
p "At top level, self : #{self}"
Driveable = Module.new do
  p "Inside the Driveable module, self : #{self}"
end
```

This prints:

```
At top level, self : main
Inside the Driveable module, self : #<Module:0x007f840>
```

The self changes from main to an instance of Module.



The self at the Top Level and Inside Module

Variable with Same Name Inside a Module

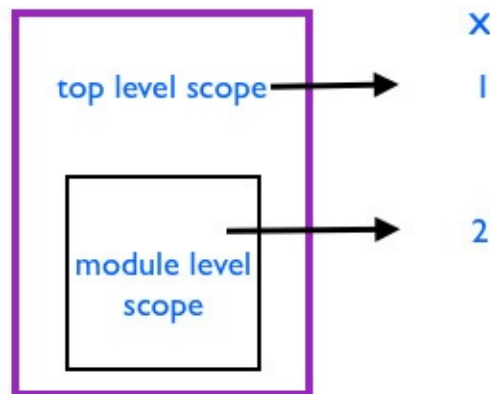
What happens when you have a variable with the same name inside the do-end block of creating a Module?

```
x = 1
p "At the top level, x : #{x}"
Driveable = Module.new do
  x = 2
  p "Inside the Driveable module, x : #{x}"
end
p "Back at the top level, x : #{x}"
```

This prints:

```
At the top level, x : 1
Inside the Driveable module, x : 2
Back at the top level, x : 2
```

This changes value of the same local variable defined at the top level.



Same Local Variable in Top Level and Module Level Scope

Key Takeaways

- The local variable defined at the top level is visible inside the dynamically defined method.
- The scope does not change. Thus, the x at the top level and inside the method is the same.
- The local variables defined at the top level is visible inside the do-end block of creating the Car class.
- The scope does not change. Thus, the x at the top level and inside the do-end block of creating a new instance of Class is the same.
- The self changes from main to an instance of the Class.
- The local variables defined at the top level is visible inside the do-end block of creating the Driveable module.
- The scope does not change. Thus, the x at the top level and inside the module is the same.
- The self changes from main to an instance of Module.

Here is a summary of what happens to self and scope when we use dynamic language constructs of Ruby.

Self	Scope	Where
No Change	No Change	Top Level and Top Level Method
Changes	No Change	Top Level and Inside Class
Changes	No Change	Top Level and Inside Module

Summary

In this chapter, you learned that when we dynamically define a method, class or module, the scope does not change. The dynamic creation of a method does not change the value of self, it remains main. For dynamic creation of class and module, the self changes as summarized in the table.

Dynamic Construct	Self
Class.new	Instance of Class
Module.new	Instance of Module

Every Object is an Instance of a Class

In this chapter, you will learn that every class is an instance of a Ruby built-in class called `Class`.

User Defined Class

Let's take a look at user defined classes.

```
class Car
  def drive
    puts 'driving...'
  end
end
```



We can send the **drive()** message to the car instance **car**.

```
car.drive
```

This prints:

```
driving...
```

We created an instance of our car class and called the **drive()** method.

Car Class is an Object

The class `Car` we defined is an object. If that is the case, then the `Car` class must be an instance of some class. What is that class?

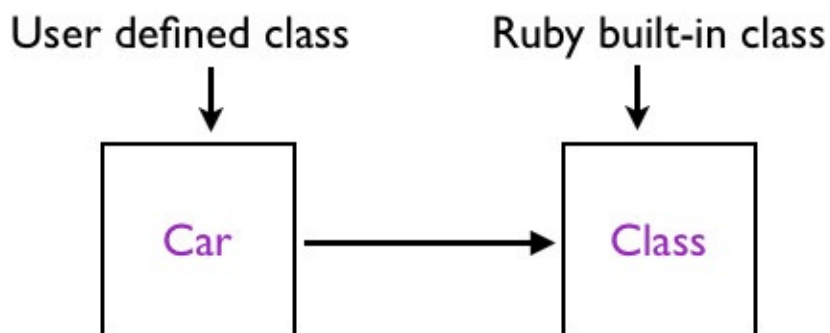
```
class Car
  def drive
    puts 'driving...'
  end
end

p Car.class
```

This prints:

```
Class
```

The `Car` class is an instance of a class called **Class**.



Car class is an instance of Class

The class Keyword

Why is Car class an instance of Ruby's built-in Class? When you use the Ruby language keyword **class**, Ruby uses Class to create Car object. Ruby does something like this:

```
Car = Class.new
```

If you print the class of Car:

```
p Car.class
```

It prints:

```
Class
```

keyword class class name

↓ ↙

class Car

Ruby uses Class to Create Object

Creating Car Class using Class

We can rewrite the example like this:

```
Car = Class.new do
  def drive
    p 'driving'
  end
end

car = Car.new
car.drive
```

This also prints:

```
driving
```


Rhonda Asks

Why would you want to create a Car class using Class.new ?

The reason is that the scope of variables defined before the class definition is visible inside the do-end block.

```
x = 1

Car = Class.new do
  p x
end
```

This prints 1, but what happens if we define a Car class using the **class** keyword?

```
x = 1

class Car
  p x
end
```

This results in:

```
NameError: undefined local variable or method 'x' for Car:Class
```

The x is not visible inside the class definition. The reason is that the **class** keyword creates a new scope. Let's verify this by writing a program. We can print the local variables at the top level and inside the Car class.

```
x = 1

p 'Local variables at top level'
p local_variables

class Car
  p 'Local variables inside the class'
  p local_variables
end
```

This prints:

```
Local variables at top level
[:x]
Local variables inside the class
[]
```

This shows that at the top level, we have x as the local variable, whereas, inside the Car class, there is none.

Scope of Local Variable Inside a Block

Let's write a simple program to check the local variables at the top level and inside the do-end block of a Class.new call.

```
x = 1
```

```
p 'Local variables at top level:'  
p local_variables  
  
Car = Class.new do  
  p 'Local variables inside the do-end block'  
  p local_variables  
end
```

This prints:

```
Local variables at top level:  
[:x]  
Local variables inside the do-end block  
[:x]
```

This shows that we have the same local variable x at the top level as well as the do-end block.

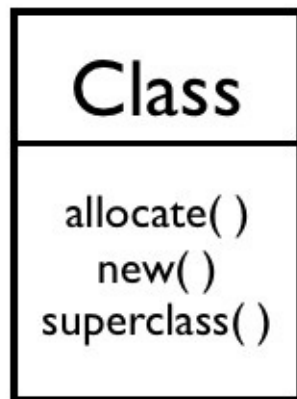
Methods Defined in Class

The **Class** is Ruby's built-in class that provides the **new()** method that we can use to instantiate the car object.

```
p Class.public_instance_methods(false).sort
```

This prints:

```
[:allocate, :new, :superclass]
```



Methods Defined in Class

As a developer you will not call **allocate()** method. You will use the **new()** and **superclass()** methods.

The new Instance Method

Since `Car` is an object you can call the instance method **new** like this:

```
car = Car.new
```

Because **new** is an instance method provided by Ruby's built-in class called **Class**. The above example is like:

```
car_class = Class.new do
  def drive
    p 'driving'
  end
end

car_object = car_class.new
car_object.drive
```

Class names in Ruby must begin with Capital letter. That's the reason we don't name a class with lowercase like `car_class`, we use `Car`. In this example, the name `car_class` is used to make it clear that the class `Car` is an object that can respond to **new** message.

Summary

In this chapter we saw that the car object is an instance of a user defined Car class. The class can either be user defined or Ruby built-in classes. In the next section, we will see that Ruby built-in classes are also objects.

Instance Methods and Instance Variables

In this chapter, you will learn where the instance methods and instance variables live in Ruby.

Greeting Example

Let's look at a simple example that we can use to experiment and learn.

```
class Greeter
  def initialize(text)
    @text = text
  end

  def greet
    @text
  end
end

greeter = Greeter.new('Hi')
p greeter.class
```

This prints:

```
Greeter
```

Instance Methods of Greeter Class

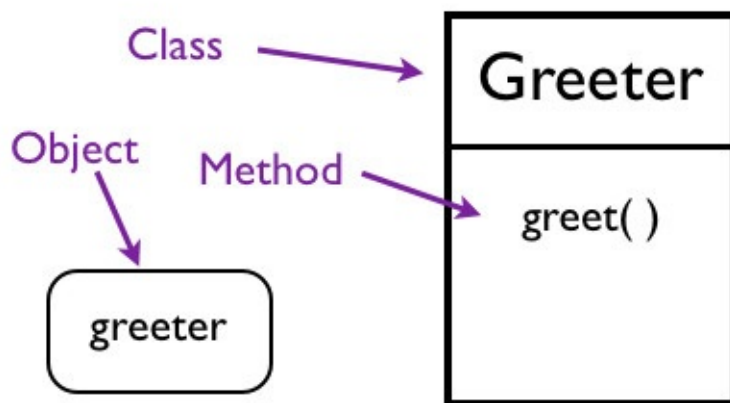
We know the instance of Greeting; the **greeter** object gets created using the Greeter class. We can also get the instance methods as follows:

```
p greeter.class.instance_methods(false)
```

This prints:

```
[:greet]
```

The methods defined in a class becomes instance methods available to the objects of that class.



Methods Live in Greeter Class

Instance Variables of Greeter Object

Let's look at the instance variables of the object o.

```
p greeter.instance_variables
```

This prints:

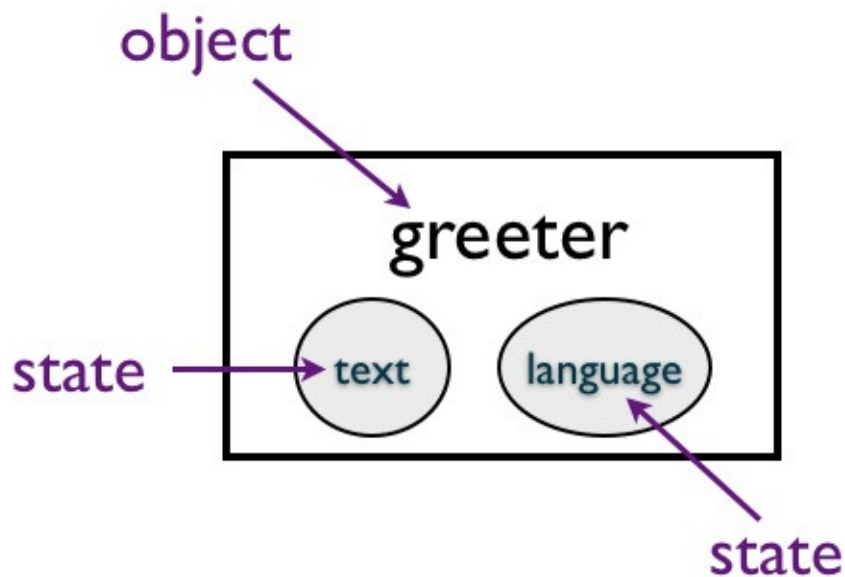
```
[:@text]
```

The instance variables live in the specific objects we create.

Fabio Asks

Can we have many instance variables in a class?

Yes. For instance, we could have text and language as the instance variables.



Objects have Unique State

In code, it would look like this:

```
class Greeter
  def initialize(text, language)
    @text = text
    @language = language
  end

  def greet
    "In #{@language}, it's #{@text}"
  end
end

greeter = Greeter.new('Hi', 'English')
p greeter.welcome
```

This prints:

```
In English, it's Hi
```

Instance Methods of String

Ruby's built-in classes also have instance methods. Let's experiment with the Ruby built-in String class.

```
> s = 'hi'
=> "hi"
> s.instance_methods
NoMethodError: undefined method `instance_methods' for "hi":String
from (irb):3
from /Users/bparanj/.rvm/rubies/ruby-2.3.0/bin/irb:11:in `
```

We get an error when we call **instance_methods** on the string object. Let's call **instance_methods** on the String class.

```
> String.instance_methods
=> [:<=>, :==, :===, :eql?, :hash, :casecmp, :+, :*, :%, :[], :
[]=, :insert, :length, :size, :bytesize, :empty?, =~, :match, :succ, :succ!, :next, :next!, :up
```

We see lots of methods defined in the String class. Let's call the **length** method on the string object.

```
> s.length
=> 2
```

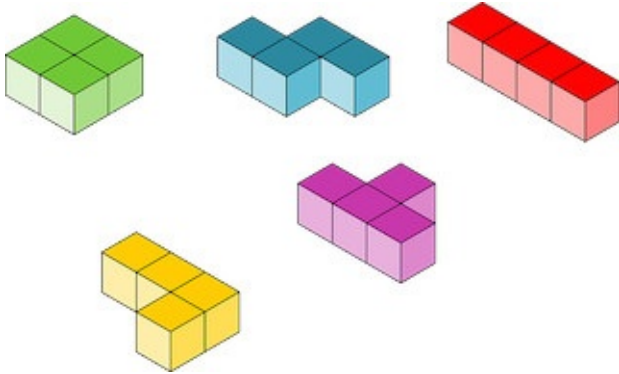
It prints 2.

Summary

In this chapter, we were able to query for instance variables and instance methods. We learned that the instance methods live in the class and the instance variables live in the object. Objects share instance methods. Instance variables are not shared between objects.

Block Object

In this chapter, you will learn about block objects and delayed execution of the code in a block object.



What is a Block?

A block is a chunk of code that is enclosed between the curly braces or do-end. Let's create a simple block that prints hi.

```
{ puts 'hi' }
```

If you run this program, you will get the error:

```
syntax error, unexpected tSTRING_BEG, expecting keyword_do or '{' or '('
```

We enclosed a chunk of code using the curly braces, but, it is not valid syntax in Ruby.

Converting a Block into an Object

We can use Proc, lambda or the literal constructor -> to convert a block into an Object.

```
-> { puts 'hi' }
```

If you run this program, you will not get any syntax error. But, it will not print anything to the standard output.

Delayed Execution

Why did not print hi? If you had written:

```
puts 'hi'
```

Running this will print hi. What is the difference? We converted the block into an object. What is this object? Let's find out.

```
p -> { puts 'hi' }
```

This prints:

```
#<Proc:0x007fb1f0@untitled 2:16 (lambda)>
```

The output shows the memory location of the Proc object, but it also has lambda. Let's make the output easy to read. We can check the class of the returned Proc object.

```
greet = -> { puts 'hi' }  
puts greet.class
```

We assigned the value returned after conversion to a variable. Then, we print the class of that object. This prints:

```
Proc
```

Now we know that using the literal constructor created a Proc object. We can execute the code by sending a **call** message to the Proc object.

```
greet = -> { puts 'hi' }  
greet.call
```

This prints:

```
hi
```

This is how block objects exhibit delayed execution by nature. The above example is equal to this:

```
def greet  
  puts 'hi'  
end  
  
greet
```

This version of the example has a name for the method that we can call, the **greet** method. The Proc version of the example has no name for the method. The block we converted into an object is an anonymous function. It has no name. The greet variable is a pointer to an

anonymous function. We call the anonymous function by sending **call** message to it.



Fabio Asks

Why do we need to assign the Proc object to a variable?

Why not just do this:

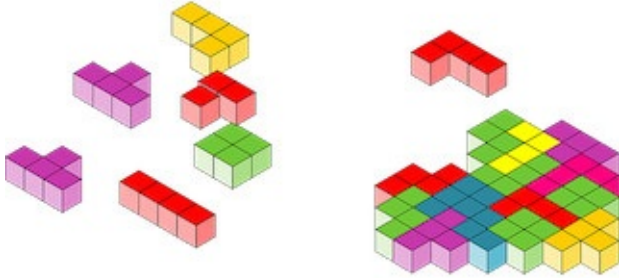
```
-> { puts 'hi' }.call
```

Yes, this will work. We are calling the anonymous function by sending the **call** message to it. In this case, we are not taking advantage of delayed execution that block objects provide us.

Rhonda Asks

What is the advantage of assigning it to a variable?

Methods can take this proc object as an argument and execute them by sending the **call** message. Thus, the code can be re-used in different scenarios.



Summary

In this chapter, you learned how to convert a block into an object and delayed execution of the code in a block object.

Closures

In this chapter, you will learn about the basics of closures and how you can use them to execute code in different execution contexts.

What is closure?

A closure is an anonymous function that carries its creation context where ever it goes.

Block Objects are Closures

Changing the Value Outside a Block

Let's write a simple program to illustrate what happens to the block object when we change the values of a local variable.

```
x = 0  
seconds = -> { x }  
p seconds.call
```

This prints:

```
0
```

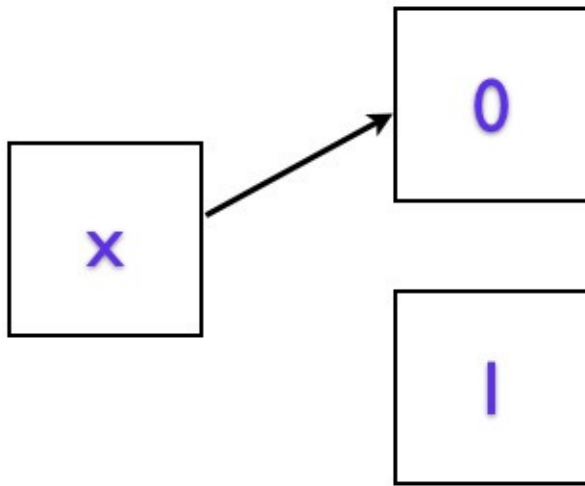
Let's change the value of x and print it.

```
x = 0  
seconds = -> { x }  
p seconds.call  
  
x = 1  
p seconds.call
```

This prints:

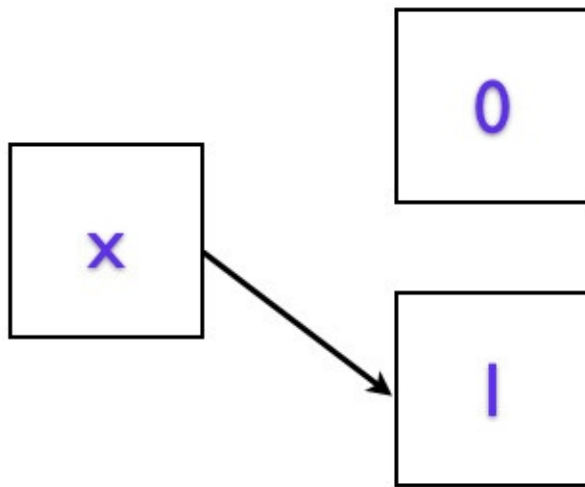
```
0  
1
```

The value of x is changed to 1 after the block object is created. But, the change is reflected when we execute the code in the block object. This illustrates that the identifier x is actually a reference to Fixnum object.



Reference to Fixnum Object

If you change that reference to point to a different Fixnum object, it will point to it.



Reference to Fixnum Object

Changing the Value Inside a Block

Let's write a simple program to illustrate what happens when the value changes inside the block.

```
x = 0
seconds = -> { x += 1 }
p seconds.call
p seconds.call
```



```
p seconds.call  
p seconds.call
```

This prints:

```
1  
2  
3  
4
```

The counter increases by one on each call.

Carrying the State Around

The block encapsulates the state. Earlier, we saw that we can pass this Proc object as an argument to a method. What happens when we have another variable with the same name in that method?

```
x = 0  
seconds = -> { x += 1 }  
  
def tester(s)  
  x = 100  
  p s.call  
  p s.call  
  p s.call  
  p s.call  
end  
  
tester(seconds)
```

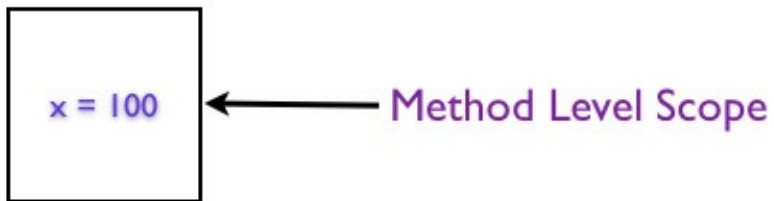
This prints:

```
1  
2  
3  
4
```

The variable `x` with the value 100 inside the `tester` method did not have any effect on the Proc object. This illustrates an important concept. The block object carries the state found at the point of its creation. In our example, it is this line:

```
seconds = -> { x += 1 }
```

At this line, the value of `x` was 0. It carries this value into the new scope of the `tester` method. We already know that methods create a new scope and `x = 100` is in a new scope. The identifier names are the same but they belong to different execution contexts. One at the top level scope and the other at the method definition scope.

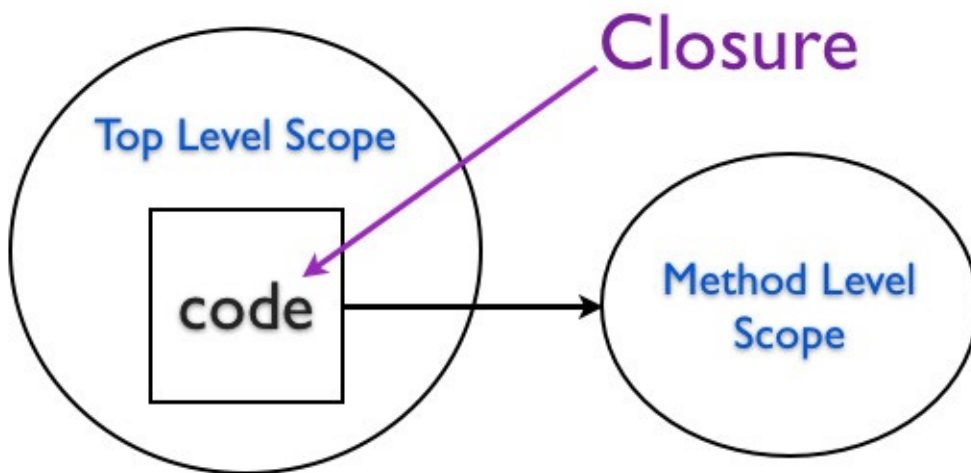


The Value of `x` in Different Scopes

The block object encapsulates the state. In this case, the value of `x`. The `x` gets incremented every time we call the block object by sending the **call** message. When a piece of code carries its creation context around with it like this, we call it closure.

Insight

You can execute code in a different execution context without using **eval** by using closures.



Executing Code in Different Scope

We captured the binding at the top level scope in a block object and executed the code in the block object in the method level scope.

Twin Analogy

Haylee and Kaylee are twins who live together in San Francisco.



Haylee is going on a business trip to New York.



She packs a tooth paste that is 100% full in her suitcase. The packing of the suitcase is the creation of the proc object using the literal constructor, ->. The top level context is San Francisco. The InNewYork class represents New York location.

```
toothpaste_level = 100  
p "In SF : #{toothpaste_level}"
```

```
brush = -> { toothpaste_level -= 5 }
brush.call

p "After brushing in SF : #{toothpaste_level}"

class InNewYork
  def get_ready(block)
    p "Brushing in NY"
    current_level = block.call
    p "In NY : #{current_level}"
  end
end

InNewYork.new.get_ready(brush)
p "In SF : #{toothpaste_level}"
```

This prints:

```
In SF : 100
After brushing in SF : 95
Brushing in NY
In NY : 90
In SF : 90
```



When Haylee brushes her teeth in New York, the mirror image of that toothpaste in San Francisco is affected. Kaylee in San Francisco observes the toothpaste usage of her twin in New York. Of course, in reality physical objects cannot be in two locations at the same time. But, that's how closures behave in a programming environment.

Evaluating Code using Binding Object

Execute Code in Top Level Context

The block objects provide a **binding** method that we can use to execute code. Here is an example to illustrate that concept.

```
x = 1
o = -> { x }

def tester(b)
  x = 10
  eval('x', b)
end

p tester(o.binding)
```

This prints 1. Inside the tester method the value of x is 10. But, we execute code in the top level execution context by passing in the binding of the block object. Thus, the value of x is 1 inside the tester method.

Execute Code in Method Level Context

How can we switch the execution context to the method level scope? Here is an example.

```
x = 1
o = -> { x }

def tester
  x = 10
  eval('x', binding)
end

p tester
```

This prints 10. The **binding** call inside the **tester** method provides the execution context within that method. Thus, the x = 10 initialized value is available in the method level scope.

Execute Code in an Object Context

In the previous chapter, we could not call the private **binding** method of an object. Here is that example.

```
class Car
  def initialize(color)
    @color = color
  end

  def drive
    'driving'
  end
end

car = Car.new('red')
p eval("@color", car.binding)
```

This gave us the error:

```
NoMethodError: private method 'binding' called for #<Car:0x0570 @color="red">
```

We can make this example work by accessing the binding within the car object.

```
class Car
  def initialize(color)
    @color = color
  end

  def drive
    'driving'
  end

  def context
    binding
  end
end

car = Car.new('red')
p eval("@color", car.context)
```

This prints red. We can also use the binding of a block object to access the instance variable.

```
class Car
  def initialize(color)
    @color = color
  end

  def drive
    'driving'
  end

  def null_proc
    -> { }
  end
end

car = Car.new('red')
p eval("@color", car.null_proc.binding)
```

This also prints red. This illustrates the concept that the null_proc is a closure. Let's check the class of the null_proc.

```
p car.null_proc.class
```

This prints Proc. The null_proc is bound to the execution context that can access the instance variable of the car object. The **binding** instance method on the Proc class exposes the execution context.

You can also replace the existing null_proc implementation with:

```
Proc.new{ }
```

This creates a proc object from an empty block. This will still work.

Summary

In this chapter, you learned the basics of closures. You also learned how to execute code in different contexts using closures.

Focus on Messages

In this chapter, you will learn how to write message centric Ruby programs.

Avoid Over Emphasis on Objects

Alan Kay coined the term Object Oriented Programming. He has expressed regret that he overemphasized the benefits of objects.

I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea. The big idea is "messaging". The Japanese have a small word - ma - for "that which is in between" - perhaps the nearest English equivalent is "interstitial".

The key in making great and grow-able systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.

– Alan Kay



Sending a Message

In Smalltalk, you can send factorial message to a number object. In Ruby, when you do the same:

```
3.factorial
```

You will get an error:

```
NoMethodError: undefined method 'factorial' for 3:Fixnum
```

Open Fixnum Class

Ruby is flexible, it has open classes. We can open the Fixnum class and define factorial method.

```
class Fixnum
  def factorial
    (1..self).reduce(1, :*)
  end
end
```

We can now send factorial method to the Fixnum object.

```
p 3.factorial
```

This prints:

```
6
```

The above solution looks much more elegant than doing:

```
Factorial.compute(3)
```

Using Refinements

It is debatable whether opening a Fixnum class to define factorial is a good idea. It is better to use refinements instead to avoid global impact in your programs.

```
module MyModule
  refine Fixnum do
    def factorial
      (1..self).reduce(1, :*)
    end
  end
end

using MyModule

p 3.factorial
```

This prints the same value 6. There is a drawback to this approach. We need to know the name of the module in the using declaration before we can call the **factorial** method. This creates a dependency. It's a trade-off you need to make between reducing the impact vs knowing the name of a module.

Summary

In this chapter, you learned how you can make your Ruby programs more message centric. We can use open classes and refinements for this purpose. The real power is in the messaging.

Self and Scope Toolbox

A good grasp of how the self and scope changes is helpful to become good at Ruby. The table below summarizes how they change.

Self	Scope	Tool
Changes	No Change	instance_eval, class_eval
Changes	Changes	class, def, module
No Change	No Change	block, Class.new, Module.new, define_method
No Change	Changes	Recursion, Top level method

We have discussed most of the tools in this book.



As you continue your learning, you now have a way to put the tools into the appropriate slot. This helps you to choose the right tool for a given situation.

Retry Library

In this chapter, we will apply the concepts we have seen so far in the book to develop a simple Retry library.

Failure Handler Proc

Sometimes you want to do something after your retry attempts has failed in your code. You may want to log the exception to a remote server or simply write to a log file. You could then investigate the problem later. Let's write a simple failure call back proc object that will handle the failure when we connect to a remote web service.

```
uri = 'www.google.com'  
query = 'rich'  
MAX_RETRY_ATTEMPT = 3  
  
failure_call_back = -> { raise "Timeout Error: Cannot reach service #{uri} #  
{query.inspect.to_s} after #{MAX_RETRY_ATTEMPT} retry attempts." }  
failure_call_back.call
```

You can call the `failure_call_back` proc object immediately.

Using a Class

But being able to call it later is more useful. It allows our library to be generic and agnostic to the application specific variables. Let's change our code to make it context independent.

```
# Application specific code below
uri = 'www.google.com'
query = 'rich'
retries = 3

failure_call_back = -> { raise "Timeout Error: Cannot reach service #{uri} #
{query.inspect.to_s} after #{retries} retry attempts." }

# Library code begins
class RetryMe
  def self.retry(failure_call_back)
    # Retry implementation goes here
    # On failure the following call back is executed
    failure_call_back.call
  end
end

# Application code below
RetryMe.retry(failure_call_back)
```

This prints:

```
RuntimeError: Timeout Error: Cannot reach service www.google.com "rich" after 3 retry attempts.
```

This works, but this approach is not message centric. How can we improve the design? Can we use a module?

Using a Module

```
uri = 'www.google.com'
query = 'rich'
retries = 3

failure_call_back = -> { raise "Timeout Error: Cannot reach service #{uri} #
{query.inspect.to_s} after #{retries} retry attempts." }

module RetryMe
  def retry(failure_call_back)
    failure_call_back.call
  end
end

include RetryMe
retry(failure_call_back)
```

This gives an error:

```
syntax error, unexpected '(', expecting end-of-input
retry(failure_call_back)
  ^
catch_exception.rb:20: in `block in <top (required)>': undefined method `each' for nil:NilClass
```

We cannot use `retry` as the method name because `retry` is a Ruby keyword. Let's rename the **retry** method to **attempt**.

```
uri = 'www.google.com'
query = 'rich'
retries = 3

failure_call_back = -> { raise "Timeout Error: Cannot reach service #{uri} #
{query.inspect.to_s} after #{retries} retry attempts." }

module RetryMe
  def attempt(failure_call_back)
    failure_call_back.call
  end
end

include RetryMe
attempt(failure_call_back)
```

This results in the expected output.

```
# RuntimeError: Timeout Error: Cannot reach service www.google.com "rich" after 3 retry attempts
```

This approach requires `include RetryMe` statement before we can use the **attempt** method.

Opening the Kernel Module

We can add our method to the Kernel Module.

```
uri = 'www.google.com'
query = 'rich'
retries = 3

failure_call_back = -> { raise "Timeout Error: Cannot reach service #{uri} #
{query.inspect.to_s} after #{retries} retry attempts." }

module Kernel
  def attempt(failure_call_back)
    failure_call_back.call
  end
end

attempt(failure_call_back)
```

This works. However, there is a drawback. This makes it available everywhere.

Using Refinements

We can restrict access to our methods by using refinements.

```
uri = 'www.google.com'
query = 'rich'
retries = 3

failure_call_back = -> { raise "Timeout Error: Cannot reach service #{uri} #
{query.inspect.to_s} after #{retries} retry attempts." }

module RetryMe
  refine Object do
    def attempt(failure_call_back)
      failure_call_back.call
    end
  end
end

using RetryMe
attempt(failure_call_back)
```

This also works. If you compare this solution to the module based solution, there is only one difference. The keyword `using` is used instead of `include`.

Retryable Gem

If you browse the source code for `retryable gem`, you will find that it used the Kernel module approach in the first version. The 2.0 version uses the module approach where `Retryable` module has a **`retryable`** class method.

Summary

In this chapter, we developed the core of a simple Retry library that takes advantage of delayed execution of proc objects. The failure call back proc object is called only when all the attempts has failed. We briefly saw how the concepts we learned in this book is used in the wild.

Basics for Ruby Object Model

This section will cover the basics of Ruby needed to learn the Ruby Object Model. It does not go into too much detail of the Ruby Object Model. However, it does provide the concepts that are important to learn the Ruby Object Model.

Introduction

Ruby's Object Model was influenced by Smalltalk. The Smalltalk object model follows a set of simple rules that are uniformly applied. The rules are:

Rule 1

Everything is an object.

Rule 2

Every object is an instance of a class.

Rule 3

Every class has a superclass.

Rule 4

Everything happens by sending messages.

Rule 5

Method look-up follows the inheritance chain.

Ruby Object Model

In Ruby, Rule 1 is not applicable. We have seen that everything is not an object. We can say:

- Everything in the inheritance hierarchy is an Object.
- Receiver and Sender in a message sending interaction are objects.
- Every class is an object. In other words, every class is an instance of a Ruby built-in class called Class.

We have already discussed the Rule 3 and 4 in the previous section. In this section, you will learn about Rule 2 and 5.

Class Methods

In this chapter, we will answer the question: Where does the class methods live?

Instance Method

Let's define an instance method **drive()** in Car class.

```
class Car
  def drive
    p 'driving'
  end
end

p Car.instance_methods(false).sort
```

This prints:

```
[:drive]
```

There is no surprise here, if we define an instance method, it shows up in the output.

Class Method

What if we had defined a class method **drive()** instead?

```
class Car
  def self.drive
    p 'driving'
  end
end

p Car.instance_methods(false).sort
```

This prints:

```
[]
```

This is because there are no instance methods in Car. There is class method **drive()**. The question is where does the class methods like **drive()** live?

The Singleton Class

The class methods live in singleton class. We can use a special syntax that gives us a reference to the singleton class as follows:

```
class Car
  def self.drive
    p 'driving'
  end
end

singleton_class = class << Car
  self
end

p singleton_class.instance_methods(false).sort
```

This prints:

```
[:drive]
```

We can see that the singleton class holds the class method we have defined in the Car class. Ruby 1.9 introduced **singleton_methods** that is an alternative to **class <<** syntax. We can use it like this:

```
p Car.singleton_methods
```

This prints:

```
[:drive]
```

Different Ways to Define Class Method

To illustrate the point made above, we can define the class method for Car in it's singleton like this:

```
class Car
end

class << Car
  def drive
    'driving'
  end
end

p Car.drive
```

This is same as this:

```
class Car
  def self.drive
    'driving'
  end
end

p Car.drive
```

And so is this:

```
class Car
  class << self
    def drive
      'driving'
    end
  end
end

p Car.drive
```

The last form of defining a class method will lead to difficulty in maintenance. Because it is difficult to determine whether it is an instance method or a class method. This happens when the class methods are way down below its **class << self** declaration.

Summary

In this chapter, we designed experiments to answer the question: Where does the class method live? We found that they live in singleton class.

Singleton Methods

In this chapter, you will learn about singleton methods and class methods and how they relate to each other.

Class Method

Shifting Perspective

Let's define a class method drive in Car class.

```
class Car
  def self.drive
    p 'driving'
  end
end
```

How can we ask Ruby for the class methods defined in Car class? We cannot do:

```
Car.class_methods
```

We will get NoMethodError. We have to use **singleton_methods**.

```
class Car
  def self.drive
    p 'driving'
  end
end

p Car.singleton_methods
```

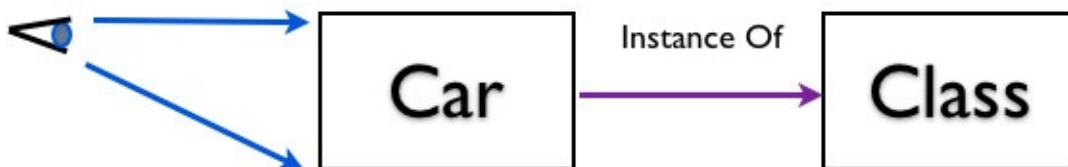
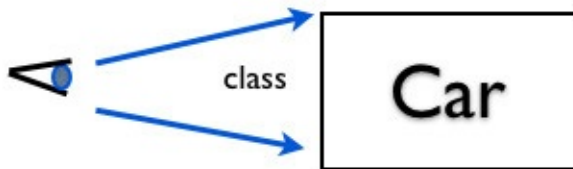
This prints:

```
[:drive]
```

We can call the class method like this:

```
Car.drive
```

To view the **drive()** class method as a singleton method, we need to shift our perspective. We shift our perspective from Car class to Car as an instance of Class.



Shift in Perspective

We can define the drive class method like this:

```
Car = Class.new
class << Car
  def drive
    p 'driving'
  end
end
Car.drive
```

This also prints:

```
driving
```

We now see that Car is an instance of class **Class** so it is a singleton method from that perspective. To make this concept clear, if we create a Bus class that is an instance of Class:

```
Bus = Class.new
Bus.drive
```

This prints:

```
NoMethodError: undefined method 'drive' for Bus:Class
```

The **drive** class method is not available for Bus class or any other instances of Class.

Alternative Way to Define Class Method

Instead of using **class** << syntax, we can also define a class method like this:


```
Car = Class.new
```

```
def Car.drive  
  p 'driving'  
end
```

```
Car.drive
```

Singleton Class and Class Method

We can also define a class method by defining a method inside the singleton class like this:

```
class Car
  class << self
    def drive
      p 'driving'
    end
  end
end

p Car.singleton_methods
```

This prints:

```
[:drive]
```

The effect is the same as **Class Method** section, we can still call the **drive()** method like this:

```
Car.drive
```

Singleton Method for an Object

Let's define a singleton method called **drive** for a specific instance of Car class like this:

```
class Car
end

c = Car.new

def c.drive
  'driving'
end

p c.singleton_methods
```

This prints:

```
[:drive]
```

We can call the singleton method drive like this:

```
p c.drive
```

This prints:

```
driving
```

Since this is a singleton method, the **drive()** method is not available for other instances of Car. This implies that we cannot do this:

```
b = Car.new
b.drive
```

We get the error:

```
NoMethodError: undefined method 'drive' for Car.
```

Define a Method in Singleton Class

We can do what we did in previous section like this:

```
class Car
end

c = Car.new

class << c
  def drive
    'driving'
  end
end

p c.drive
```

This prints:

```
driving
```

Let's look at the singleton methods for Car class.

```
p c.singleton_methods
```

This prints:

```
[:drive]
```

This is the same as the previous section. They both illustrate different ways to define a method in the singleton class.

Mixin a Module

An alternative to defining a singleton method using:

```
class << obj
```

construct is to mix-in the method from a module. Here is an example:

```
module Driveable
  def drive
    'driving'
  end
end

class Car
end

c = Car.new

class << c
  include Driveable
end

p c.drive
```

This prints driving.

```
p c.singleton_methods
```

This prints [:drive]. This does the same thing we did in the previous section.

Different Approaches

We have seen three different approaches:

- Defining a class method in a Class.
- Defining a singleton method for a specific car object.
- Using mix-in to define a singleton method.

Let's now combine them all into one grand example:

```
module Stoppable
  def stop
    'brake failure, cannot stop'
  end
end

class Car
  def self.start
    'starting'
  end
end

c = Car.new

def c.fly
  'flying'
end

class << c
  include Stoppable

  def drive
    'driving'
  end
end

p Car.singleton_methods
```

This prints `[:start]`. Let's print singleton methods for `c`, the specific instance of car object.

```
p c.singleton_methods
```

This prints:

```
[:fly, :drive, :stop]
```

We can filter out the methods included in the module by passing `false` to the `singleton_methods()`.

```
p c.singleton_methods(false)
```

This prints:

```
[:fly, :drive]
```

These statements:

```
p Car.start  
p c.drive  
p c.stop  
p c.fly
```

will print:

```
starting  
driving  
brake failure, cannot stop  
flying
```

The first call is a class method call and the other three are singleton method calls.

Introspect Singleton Class

We can also ask Ruby for the **singleton_class** of a class like this:

```
class Car
end

p Car.singleton_class
```

This prints:

```
#Class:Car
```


Display Singleton Class

Let's look at a simple example for displaying the name of the singleton class:

```
class Car
  class << self
    def class_name
      to_s
    end
  end
end

p Car.class_name
```

This prints:

```
Car
```

Dynamic Singleton Method

We can also use `define_singleton_method()` to dynamically define singleton method. Here is an example that defines `to_s` singleton method in Car class:

```
class Car
end

Car.define_singleton_method(:class_name) do
  to_s
end

p Car.class_name
```

This still prints:

```
Car
```

Combo Example

Let's combine the two above examples into one example:

```
class Car
  class << self
    def class_name
      to_s
    end
  end
end

Car.define_singleton_method(:whoami) do
  "I am : #{class_name}"
end

p Car.whoami
```

This prints:

```
I am : Car
```

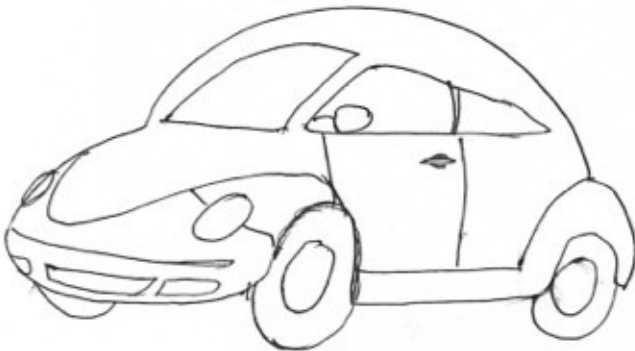
Singleton Method for String

As a last example, let's define a singleton method on Ruby's built-in string class.

```
car = 'Beetle'  
car.define_singleton_method(:drive) { "You are driving : #{self}" }  
p car.drive
```

This prints:

```
You are driving : Beetle
```



Let's check the singleton methods for this specific string object.

```
p car.singleton_methods
```

This prints:

```
[:drive]
```

Key Takeaway

- Class methods and singleton methods are the same.

Summary

In this chapter, we saw different ways to define class methods and singleton methods. Class methods are just methods on the singleton class. We also learned that the singleton methods live in singleton class.

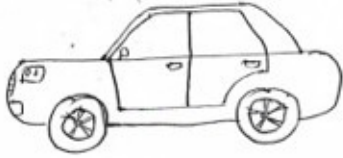
Objects and Inheritance Hierarchy

In this chapter, you will learn that everything in an inheritance hierarchy is an object.

User Defined Class

Let's define a user defined class.

```
class Car  
end
```



Object is the Parent of Car

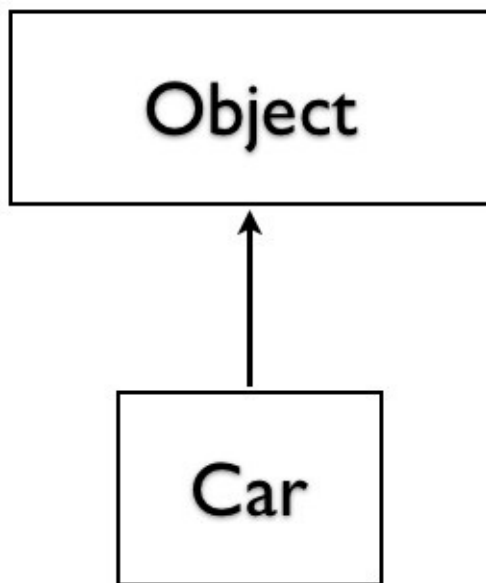
This user defined Car class is part of an inheritance hierarchy. Thus, we can ask Ruby for it's super-class.

```
class Car
end

p Car.superclass
```

This prints:

Object



Object is Super Class of Car

Implicit Parent

The Car class implicitly extends from Ruby's built-in Object class. It's as if you had written code like this:

```
class Car < Object  
end
```

Car is an Object

We found out that the Car class is part of an inheritance hierarchy. If everything in an inheritance hierarchy is an object, then the Car class must be an object. If Car class is an object, it must be an instance of some class. What is that class? We can ask Ruby:

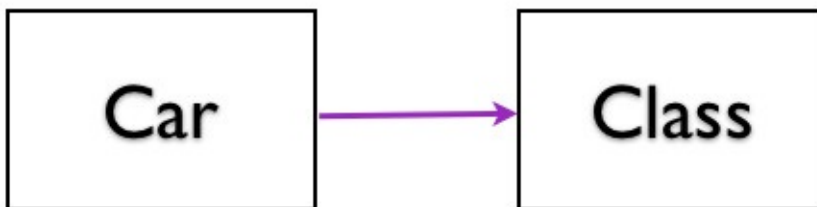
```
class Car
end

p Car.class
```

This prints:

```
Class
```

The Car class is an instance of Ruby's built-in class called Class.



Car is an Instance of Class

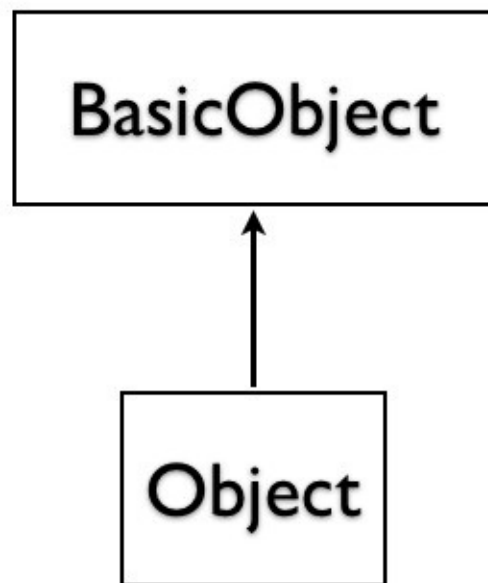
BasicObject is the Parent of Object

We know that super-class of Car is Object. The Object also has a super-class.

```
p Object.superclass
```

This prints:

```
BasicObject
```



BasicObject is Super Class of Object

Object is an Instance of Class

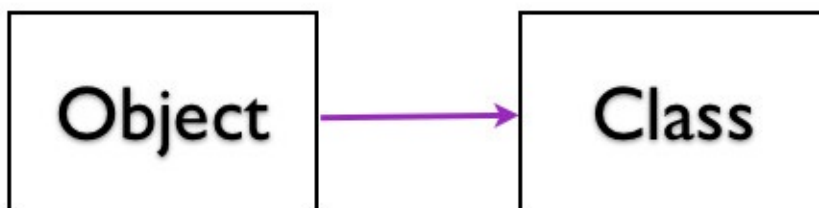
The Ruby's built-in Object is also part of an inheritance hierarchy. It must also be an object.

```
p Object.class
```

This prints:

```
Class
```

The Object is an instance of Ruby's built-in class called Class.



Object is an Instance of Class

BasicObject Has No Parent

The Ruby's built-in BasicObject is the root of the inheritance hierarchy.

```
p BasicObject.superclass
```

This prints:

```
nil
```

The nil indicates that BasicObject has no parent.



BasicObject is an Instance of Class

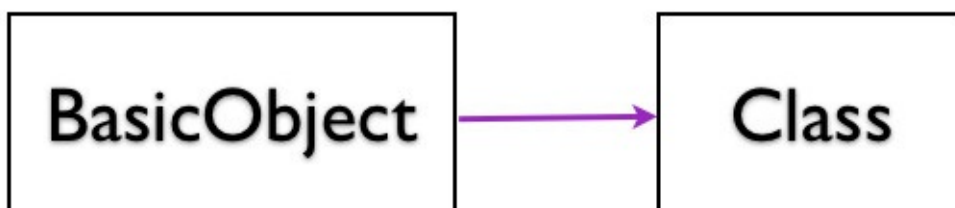
BasicObject is also an object, since it has sub-classes and is part of the inheritance hierarchy. What is the class used to create an instance of BasicObject? We can ask Ruby:

```
p BasicObject.class
```

This prints:

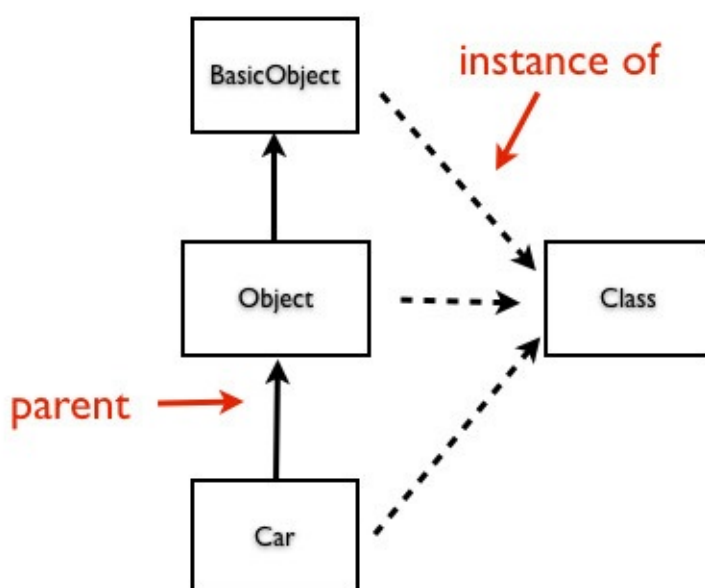
```
Class
```

The BasicObject is an instance of Ruby's built-in class called Class.



BasicObject is an Instance of Class

Visual Summary



Inheritance Hierarchy and the Class

Fabio Asks

Why does user defined classes use Class as the template to create an instance?

You define classes in Ruby using the **class** keyword. This is the reason that the class you define becomes an instance of the Ruby's built-in class called `Class`.

Rhonda Asks

Why does Ruby's built-in classes use Class as the template to create an instance?

The reason is the same as the reason for user defined classes. The **class** keyword defines the Ruby's built-in objects like Object and BasicObject.

Key Takeaways

- User defined classes and Ruby's built-in classes are objects.
- User defined classes and Ruby's built-in classes are instances of class called Class.

Summary

In this chapter, you learned that everything in the inheritance hierarchy is an object.

Class, Object and Module Hierarchy

In this chapter, you will learn about the hierarchy of Ruby built-in classes, Class, Object and Module.

Object is an Instance of Class

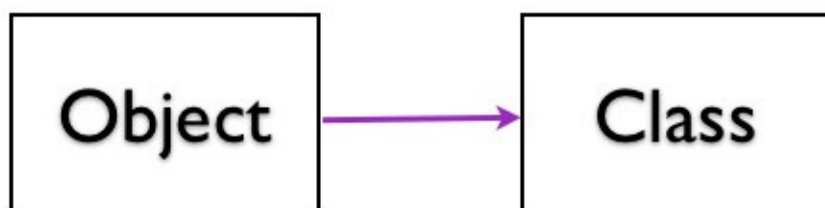
In the previous chapter we experimented with user defined classes. We learned that user defined classes implicitly extend from Object.

What is the Ruby's built-in Object's class? In other words, Object is an object, so it must be an instance of some class, what is that class? We can ask Ruby:

```
p Object.class
```

This prints:

```
Class
```



Object is an Instance of Class

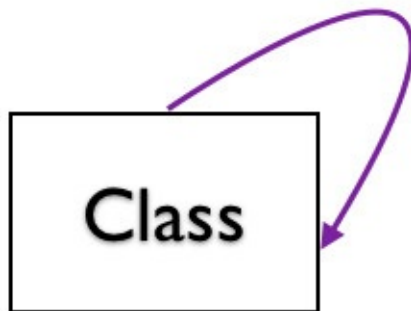
Class is an Instance of Class

The Ruby's built-in `Class` itself is an object. We can find out the class used to make instances of `Class`.

```
p Class.class
```

This prints:

```
Class
```



Class is an Instance of Class

This seems to be like the chicken and egg problem. How is **Class** created from `Class`? But Ruby is consistent. Whenever you use the language construct **class** to create a `Class`, Ruby uses `Class` to create instances.

The class can be either user defined or the existing Ruby's built-in classes.

```
name = 'Bugs Bunny'  
p name.class
```

This prints:

```
String
```

`String` is the Ruby's built-in class. The above code is the same as doing this:

```
name = String.new('Bugs Bunny')  
p name.class
```

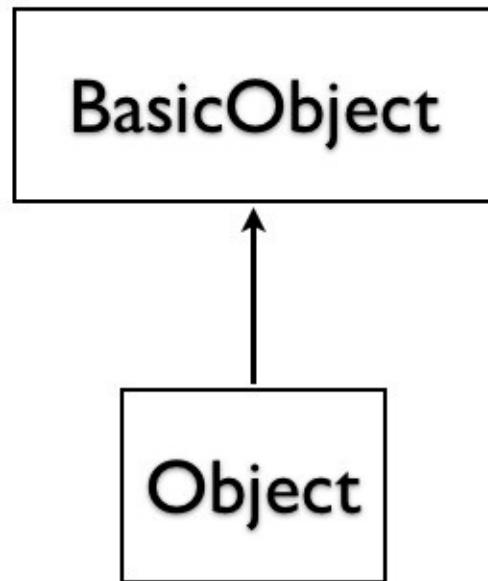
BasicObject is Parent of Object

In the previous chapter, we saw that Object was the super-class of any user defined class. What is the super-class of Object?

```
p Object.superclass
```

This prints:

```
BasicObject
```



BasicObject is Super Class of Object

BasicObject is the Root

What is the super-class of BasicObject?

```
p BasicObject.superclass
```

This prints:

```
nil
```

This means **BasicObject** is the root of the hierarchy. It does not have a parent.



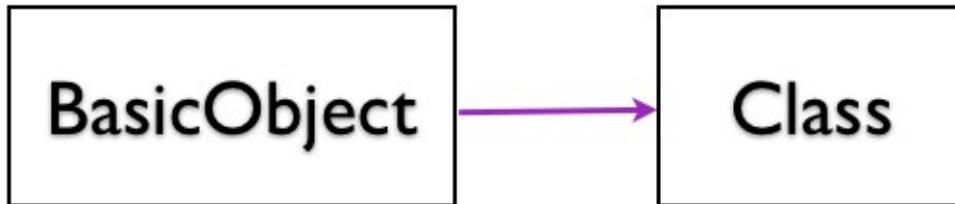
BasicObject is an Instance of Class

The BasicObject is an instance of Class. You can verify it like this:

```
p BasicObject.class
```

This prints:

```
Class
```



BasicObject is an Instance of Class

Module is the Parent of Class

What is the super-class of Class?

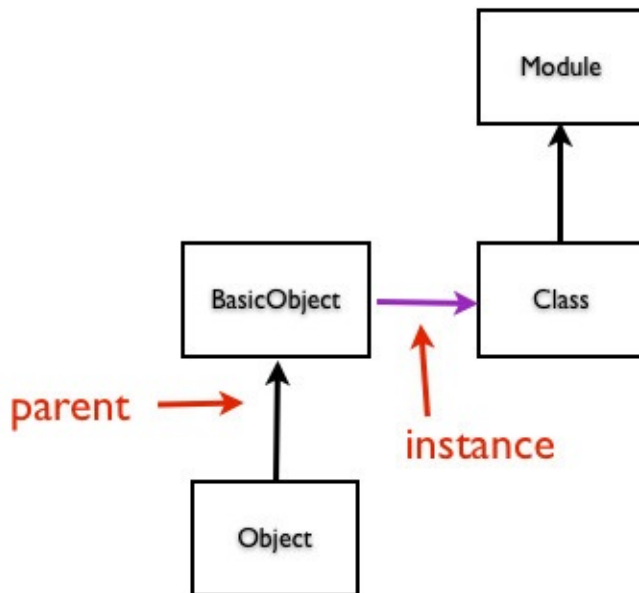
```
p > Class.superclass
```

This prints:

```
Module
```

Visual Summary

Here is the visual summary of what we have learned so far.



Hierarchy of Class, Object and Module

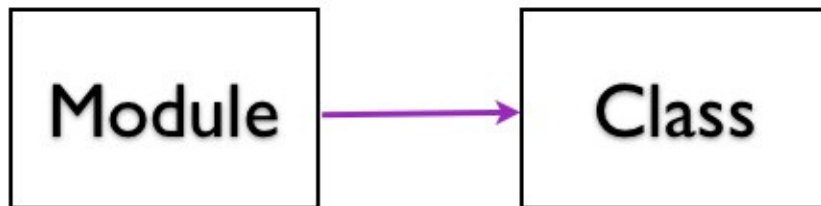
Module is an Instance of Class

The class Module is an object. What class is it an instance of?

```
p Module.class
```

This prints:

```
Class
```



Module is an Instance of Class

Why is Module a Class?

How can module be a class? Let's say we have a Vehicle module:

```
module Vehicle
  def wheels
    1000
  end
end

p Vehicle.class
```

This prints:

```
Module
```

The Module is a class because Ruby defines Module like this:

```
class Module
end
```

Can we create an instance of Module class? Yes, we will see that next.

How to Create a Module Instance?

Instead of doing this:

```
module Vehicle
  def wheels
    100
  end
end

class Car
  include Vehicle
end

c = Car.new
p c.wheels
```

We can do the same thing we did above like this:

```
Vehicle = Module.new do
  def wheels
    100
  end
end

class Car
  include Vehicle
end

c = Car.new
p c.wheels
```

Fabio Asks

Why would you want to create a module the second way?

The short answer is blocks are closures. We have access to the variables before the do-end block.

Methods in Module, Class and Object

Let's now compare the methods in Module, Class and Object. Here are the methods in Module.

```
p Module.public_methods(false).sort
```

This prints:

```
[:allocate, :constants, :nesting, :new, :superclass]
```

Here are the methods in Class.

```
p Class.public_methods(false).sort
```

This prints:

```
[:allocate, :constants, :nesting, :new, :superclass]
```

Here are the methods in Object.

```
p Object.public_methods(false).sort
```

This prints:

```
[:allocate, :new, :superclass]
```

You can see we can create instances of Module, Class and Object. Because they have the method **new()**. You can also notice that class and module have the same set of public methods.

Dynamic Creation of Car Class

We already know that user defined classes are instances of Class. Instead of doing this:

```
class Car
  def drive
    p 'driving...'
  end
end

c = Car.new
c.drive
```

We can do this:

```
Car = Class.new do
  def drive
    p 'driving...'
  end
end

c = Car.new
c.drive
```

Both versions of the Car examples print:

```
driving...
```

Rhonda Asks

Why would you want to create a class the second way?

The short answer is blocks are closures. We have access to the variables before the do-end block.

Summary

In this chapter you learned the Ruby's built-in inheritance hierarchy. This consists of Class, Object, Module and BasicObject. You also saw how we can create modules and classes by creating them on the fly and adding methods to it.

Hierarchy of Class Methods

In this chapter, you will learn that class methods have their own inheritance hierarchy. We will also see that the sub classes inherit the class methods from its parent.

Class of a Singleton Class

Let's consider the example we saw in the previous chapter.

```
class Car
  def self.drive
    'driving'
  end
end

singleton_class = Car.singleton_class
```

What class is this singleton class an instance of? Let's ask Ruby:

```
p singleton_class.class
```

This prints:

```
Class
```

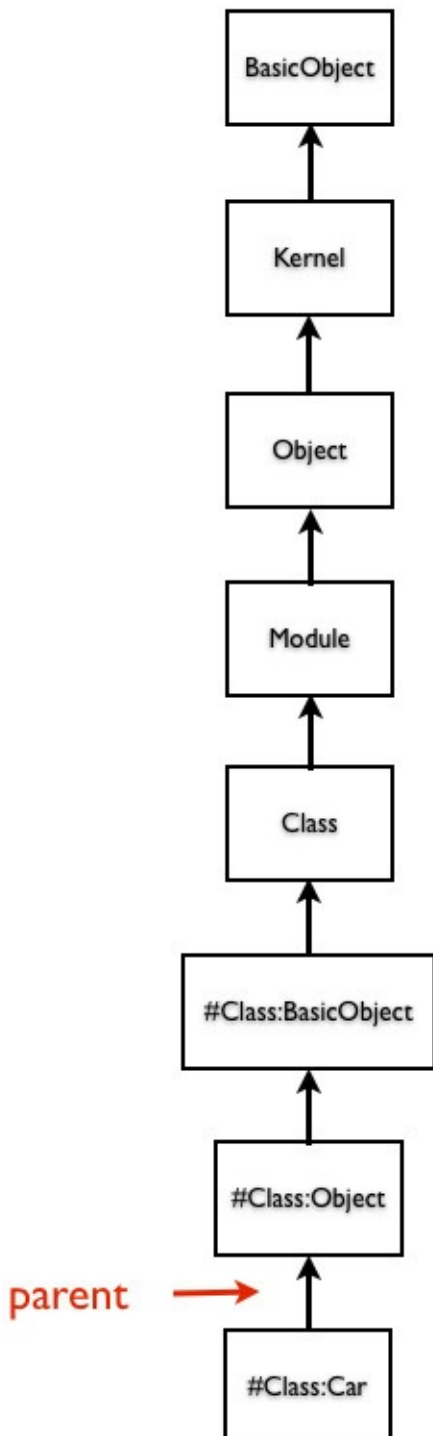
Object Hierarchy of Singleton Class

What is object hierarchy of the singleton class? Let's ask Ruby:

```
p singleton_class.ancestors
```

This prints:

```
[#<Class:Car>, #<Class:Object>, #<Class:BasicObject>, Class, Module, Object, Kernel, BasicObject]
```



There is a whole new hierarchy consisting of singleton classes for Car, Object and

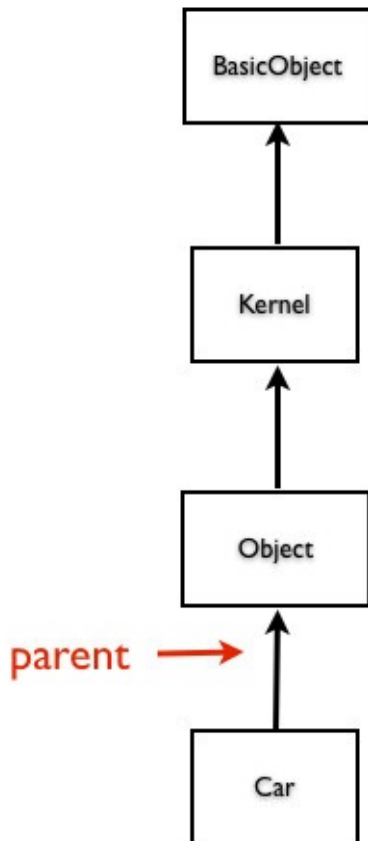
BasicObject. Compare this to the ancestors of a normal class:

```
p Car.ancestors
```

This prints:

```
[Car, Object, Kernel, BasicObject]
```

Now we don't see any singleton classes in this case.



Ancestors of Car

One Level Up

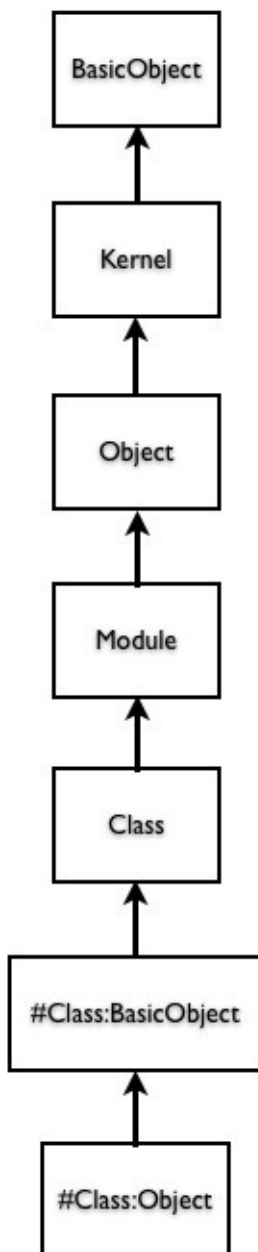
We can go one level up and do a similar experiment.

```
class Object
  def self.drive
    p 'driving'
  end
end

singleton_class = Object.singleton_class
p singleton_class.ancestors
```

This prints:

```
[#<Class:Object>, #<Class:BasicObject>, Class, Module, Object, Kernel, BasicObject]
```



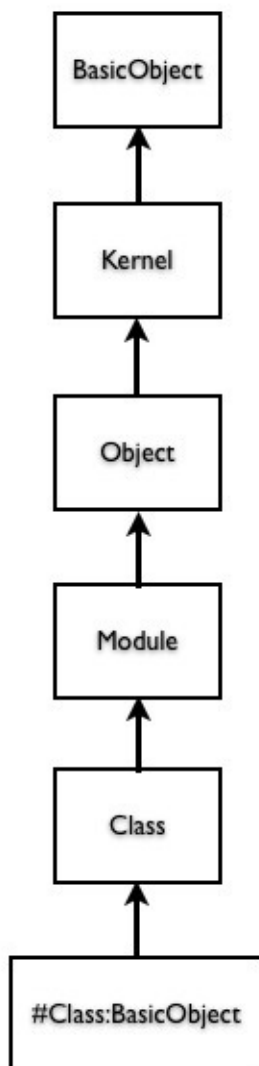
One More Level Up

Let's go one more level up and repeat a similar experiment.

```
singleton_class = BasicObject.singleton_class  
p singleton_class.ancestors
```

This prints:

```
[#<Class:BasicObject>, Class, Module, Object, Kernel, BasicObject]
```



Practical Use

Let's define a class method in Ruby built-in BasicObject.

```
class BasicObject
  def self.drive
    p 'driving...'
  end
end

class Car
end

Car.drive
```

This prints:

```
driving
```

Sub classes can call its parent's class methods. If you are familiar with ActiveRecord library of Rails, we often do something like this:

```
module ActiveRecord
  class Base
    def self.find(id)
      'finding...'
    end
  end
end

class Car < ActiveRecord::Base
end

Car.find(1)
```

Now you know how it works.

Same Sender and Receiver

How are we able to call ActiveRecord class methods without a receiver? Can we say Sender and Receiver are the same in this case? For example:

```
class Car < ActiveRecord::Base
  has_many :wheels
end
```

The concept slightly varies because sub classes can call class methods without a receiver. You can say sub class is a super class. This is the reason why you can consider the sender and receiver to be the same.

Summary

In this chapter, we learned that class methods have their own inheritance hierarchy. We also saw that the sub classes inherit the class methods from its parent.

The Method Lookup

In this chapter, you will learn the introspection abilities of Ruby in the context of Ruby Object Model. We will query for ancestors. Class hierarchy determines the method look-up in Ruby.

Define Method in Object

We already know that class Object is the super class of any user defined class. Let's define a **greet** method by opening the Ruby's built-in Object class.

```
class Object
  def greet
    puts 'hi'
  end
end
```

User Defined Class

Let's create a class and call the method **greet()** on it.

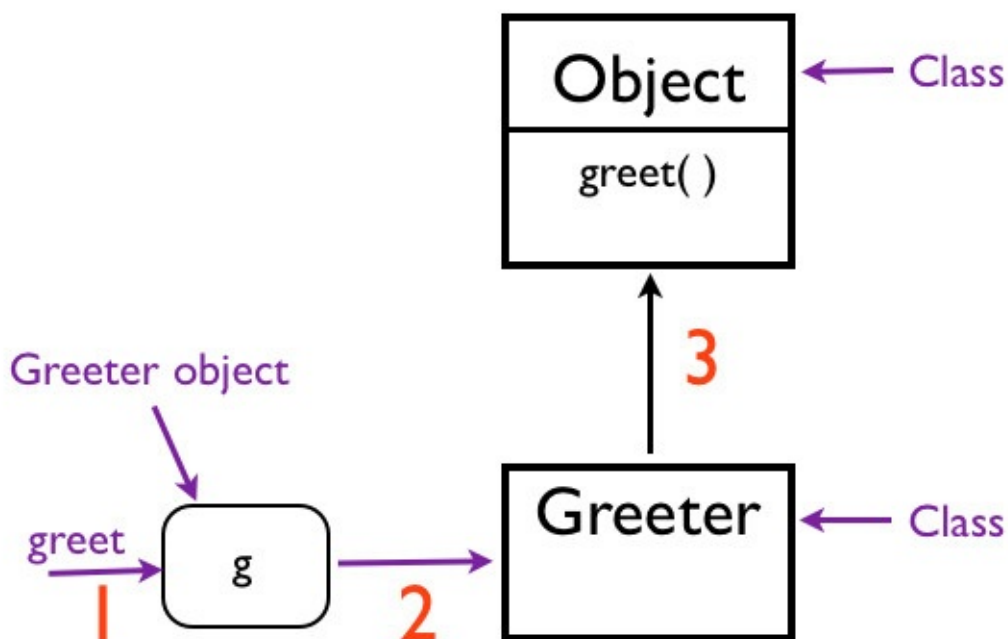
```
class Greeter
end

g = Greeter.new
g.greet
```

This prints:

```
hi
```

The variable **g** has an instance of Greeter class. When we call the **greet()** method, the value of self changes to the Greeter object **g**. Then Ruby looks for the **greet()** method in the Greeter class. It does not find it there. It goes to the super-class of Greeter which is the Ruby's built-in Object class. It finds the **greet()** method in Object class, it calls that method.



Method Lookup

Method Lookup Path

We can ask ruby for its method look up path like this:

```
class Object
  def greet
    p 'hi'
  end
end

class Greeter
end

p Greeter.ancestors
```

This prints:

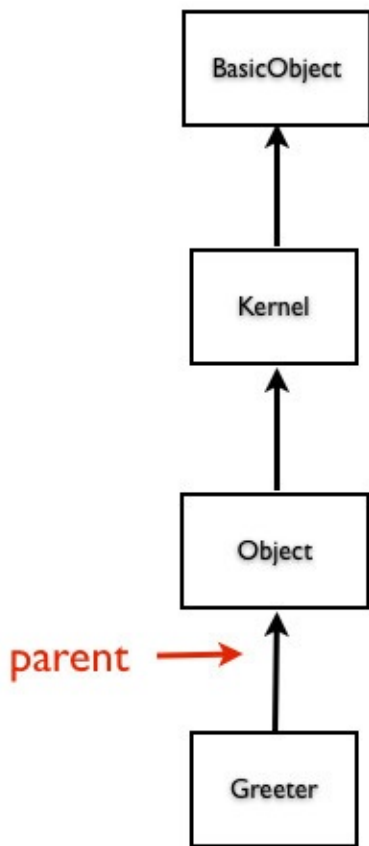
```
[Greeter, Object, Kernel, BasicObject].
```

The order in which the classes and modules appear in the array represents the order in which Ruby searches for the methods. The method look up sequence is as shown below:

1. Greeter Class
2. Object Class
3. Kernel Module
4. BasicObject Class



Visual Summary



Method Lookup Path

Summary

In this chapter, you learned the introspection abilities of Ruby in the context of Ruby Object Model. We were able to query for ancestors. Why do we need to worry about class hierarchy in Ruby? Because it determines the method look-up in Ruby.

Object Oriented Programming Revisited

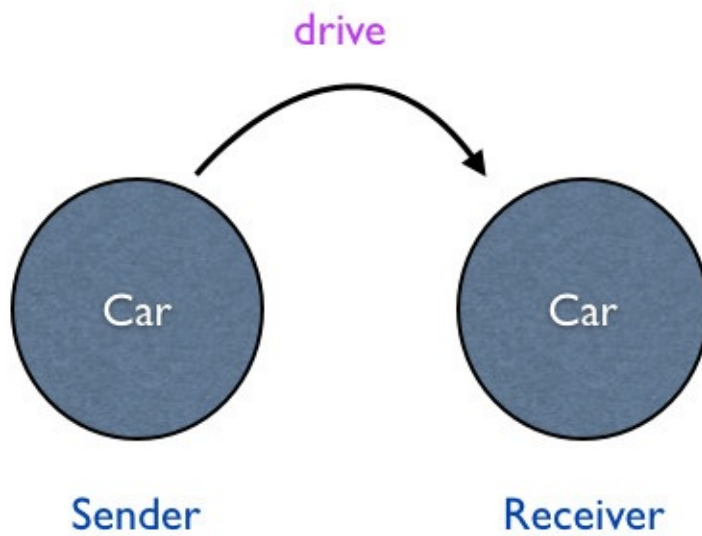
This section reviews some concepts using examples used in section 2 of this book.

Modeling the Real World

In this chapter, you will learn how we model the real world using objects that send messages to communicate.

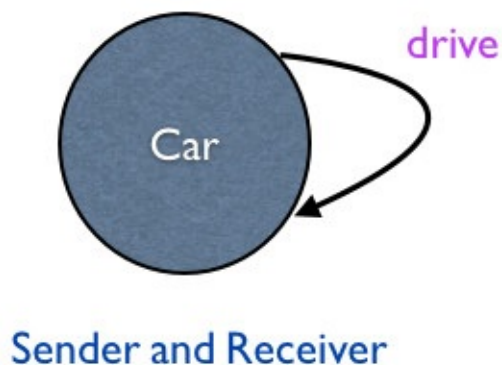
Self Driving Car

The example we saw in previous section looks more like a self driving car.



Same Sender and Receiver

The Car object sends **drive()** to itself.



In code, it will look like this:

```
class Car
  def self.drive
    p 'driving'
  end

  p "Sender is : #{self}"
  p "Receiver is : #{binding.receiver}"
  Car.drive
end
```

This prints:

```
Sender is : Car  
Receiver is : Car  
driving
```

You can see that the sender and receiver is the Car class. The **binding** method provides us the execution context. We can retrieve the receiver object from the execution context.

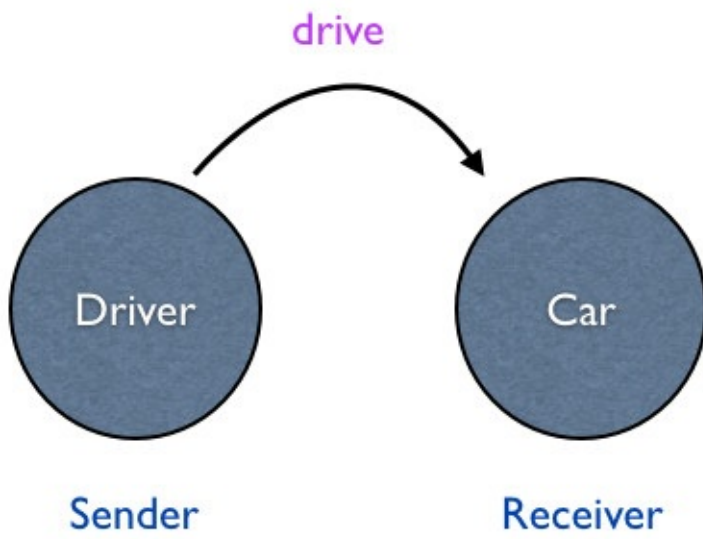
Driver and Car

In reality, it's the driver who drives the car.



Driver drives the Car

Sender and Receiver



Driver drives the Car

Intent vs Implementation

What is intent?

I want to drive the car. We don't reach into the transmission and pull levers to drive the car. We step on the gas to drive the car. Your grandma can drive the car without knowing the details of the engine. She expresses her intent by using the public interface of the car.

What is implementation?

The things under the hood of a car is the implementation. Only your car mechanic knows about the details of the car engine. You may be aware of the **3.0 litre V-6** engine, but you have no idea how it works.

Background Job Processing

Let's see an example for code that reaches into the implementation details of a method.

```
MyQueueClass.instance.enqueue(job)
```

This code knows that the implementation of the background processing class uses Singleton pattern. How can we fix this problem? To express the intent without any dependency on the implementation, we can re-write the code.

```
MyQueueClass.enqueue(job)
```

In this case, the client of the MyQueueClass is independent of the implementation details. We now have an intent revealing interface. This interface is stable. Tomorrow you may decide not to use the Singleton pattern, if so, the interface will not change.



Write an example program to illustrate the fact that a driver drives a car. **Hint:** It will be like the Teacher-Student example we saw in *Message Passing* chapter.

Summary

In this chapter, we discussed how we model the real world using objects that send messages to communicate. We also saw how to separate the intent from the implementation to define intent revealing interface.

Resources

Abstraction

[Basics of Abstraction](#)

[Single Purpose Principle](#)

[The Art of Uniform Interface](#)

Coupling Basics

[Dependency Direction](#)

[Concrete Class vs Abstract Messages](#)

Object Oriented Design Basics

[Flexible Design](#)

[Localized Change vs Additive Change](#)

[Open Closed Principle](#)

[The Three Basic Rules for a Good Design](#)

Key Takeaways

- Class acts as a template used to create objects.
- Class describes the behavior and state that an object can hold.
- Every object is an instance of a class.
- Instance methods live in the class.
- Instance variables live in the object.
- Instance variables are unique to each object.
- By default, instance variables are hidden from the outside.
- We can expose instance variable via an accessor.
- You can over-ride the `to_s` method to customize the inspect message.
- Everything is an object is true for Smalltalk but not for Ruby.
- Every sender and receiver in a message passing interaction is an object.
- Sender can be explicit or implicit.
- Sender is the owner of the scope where the message originated.
- The dot notation makes sending messages explicit.
- If the receiver and the sender is the same, you can omit the receiver and the dot.
- There is always a receiver.
- There is always a sender.
- There is always a message that passes between the sender and the receiver.
- You cannot provide an explicit receiver to call a private method.
- You have to call the private method in functional form.
- Everything in the inheritance hierarchy is an Object.
- Receiver and Sender in a message sending interaction are objects.
- Every class is an object. In other words, every class is an instance of a Ruby built-in class called Class.
- Every object is an instance of a class.
- Every class has a superclass.
- Everything happens by sending messages.
- Method lookup follows the inheritance chain.
- Class methods and singleton methods are the same.
- User defined classes and Ruby's built-in classes are objects.
- User defined classes and Ruby's built-in classes are instances of class called Class.

Essential Book Series

Essential Object Oriented Analysis

This book covers the following topics.

- Domain Object
- Parts of Speech Technique
- Case Study : Buffet R Us
- How to Identify Services
- CRC Technique
- Interviewing Domain Experts
- Conceptual Category List
- Static Modeling
- Class Design
- Finding Operations from the Static Model
- Abstraction
- Choose Good Names
- Encapsulation
- Polymorphism
- Interfaces
- Domain Model
- Effective Use of Inheritance

Essential Object Oriented Design in Ruby

This book covers the basic Object Oriented Design concepts using Ruby programming language. Topics covered are:

- Basics of Abstraction
- Single Purpose Principle
- Stepwise Refinement
- Dependency Inversion Principle
- Basic Three Rules of Design
- The Art of Uniform Interface
- Localized Change vs Additive Change
- Coupling Basics : Dependency Direction
- Concrete Class vs Abstract Messages
- Flexible Design
- Open Closed Principle

Test Driven Development in Ruby: A Gentle Introduction for Beginners

This book covers the following topics:

- Kata
- What is a Coding Kata?
- Why Coding Kata?
- What is a Domain?
- What is a Problem Domain?
- What is a Solution Domain?
- What vs How
- Why distinguish the What and How?
- How to Separate the What from How?
- Focus of What
- Focus of How
- A Brief Introduction to TDD
- What is TDD?
- Why TDD?
- What are the steps in TDD?
- Why write a failing test first?
- How to Write a Failing Test?
- How to Make the Test Pass?
- How to Get All Benefits of TDD?
- Why is TDD Difficult to Learn?
- How TDD Separates the What from How?
- Problem Solving Skills
- How Does Problem Solving Skills Fit into TDD?
- Why do we Need to Separate these Four Phases?
- How Do You Analyze the Problem?
- How Much Analysis is Enough?
- Basics of Test Driven Development
- Designing Test Cases
- Why Design Test Cases?
- How Many Test Cases Do We Need?
- What Should Be the Sequence of Tests?
- What is a Starter Test?
- What is a Next Test?
- What is a Story Test?
- Assertion
- How Many Assertions Can You Use in a Test?
- Canonical Test Structure
- Minimal Implementation
- Why do we Aim for Simplicity?
- Ways to do Minimal Implementation
- Getting it Right
- Common Beginner Mistakes
- Techniques in TDD
- Obvious Implementation
- Fake it Till You Make It
- Triangulation

Essential SQL: A Gentle Introduction for Beginners

This book covers the following topics:

- Create, Insert and Select Statements
- Sorting
- Search Conditions
- Advanced Search Conditions
- Wildcard Search
- Calculated Fields
- Aggregate Functions
- Grouping Data
- Subqueries
- Joins
- Inner Joins
- Advanced Joins
- Update and Delete
- Constraints
- Indexes

Essential Git: Introduction to Git Basics for Beginners

This book covers the following topics.

- Create a Project
- Checking Status
- Making Changes
- Staging Changes
- Un-staging Changes
- Committing Changes
- Tracking Changes to Files
- Committing
- History
- Find the Hash for Previous Version
- Tagging Versions
- Revert Local Changes
- Undoing Staged Changes
- Reverting Committed Changes
- Removing Commits
- Remove Unnecessary Tag
- Modify Commits
- Moving Files
- Creating Branches
- Switching Branches
- Dealing with Different Changes
- Viewing Diverging Branches

- Merging
- Conflict
- Rebasing
- Create a Remote Repository
- Push Branch to Remote
- Get all Remote Branches
- Keep Branches Up to Date
- List all Branches

Essential Ruby

This book covers the following topics.

- What is a Class?
- What is an Object?
- Creating an Object?
- State and Behavior
- Hidden Instance Variables
- Sending a Message to a Receiver
- Message Passing
- Inheritance
- Module
- Symbol
- The yield Keyword
- Everything is not an Object
- Top Level Context
- Code Execution
- Binding
- Pseudo Variables
- The Default Receiver
- Message Sending Expression
- The Self at the Top Level
- The Dynamic Nature of Self
- When does Self Change
- The main Object
- Message Sender at the Top Level
- Top Level Methods
- Same Sender and Receiver
- Private Methods
- Scope of Variables
- Every Object has a Class
- Instance Methods and Instance Variables
- Block Object
- Focus on Messages
- Self and Scope
- Retry Library

- Class Methods
- Singleton Methods
- Objects and Inheritance Hierarchy
- Class, Object and Module Hierarchy
- Hierarchy of Class Methods
- The Method Lookup
- Modeling the Real World

Table of Contents

[Introduction](#)

[Object Oriented Programming](#)

- [What is a Class?](#)
- [What is an Object?](#)
- [Creating an Object](#)
- [State and Behavior](#)
- [Hidden Instance Variables](#)
- [Sending a Message to a Receiver](#)
- [Message Passing](#)
- [Inheritance](#)
- [Module](#)

[Essential Ruby](#)

- [Symbol](#)
- [The yield Keyword](#)
- [Everything is Not an Object](#)
- [Top Level Context](#)
- [Code Execution](#)
- [Binding](#)
- [Pseudo Variables](#)
- [The Default Receiver](#)
- [Message Sending Expression](#)
- [The Self at the Top Level](#)
- [The Dynamic Nature of Self](#)
- [When does self Change?](#)
- [The main Object](#)
- [Message Sender at the Top Level](#)
- [Top Level Methods](#)
- [Same Sender and Receiver](#)
- [Private Methods](#)
- [Scope of Variables](#)
- [Scope of Variables Redux](#)
- [Every Object Has a Class](#)
- [Instance Methods and Instance Variables](#)
- [Block Object](#)
- [Closures](#)
- [Focus on Messages](#)
- [Self and Scope](#)
- [Retry Library](#)

[Basics for Ruby Object Model](#)

- [introduction](#)
- [Class Methods](#)

[Singleton Methods](#)

[Objects and Inheritance Hierarchy](#)

[Class, Object and Module Hierarchy](#)

[Hierarchy of Class Methods](#)

[The Method Lookup](#)

[Object Oriented Programming Revisited](#)

[Modeling the Real World](#)

[Resources](#)

[Key Takeaways](#)

[Essential Book Series](#)