CHAPTER 9

DEFINING CLASSES & CREATING OBJECTS

Introduction to Computer Science Using Ruby

Creating Classes

- Ruby has built-in classes, but you can create your own
- Imagine organizing a database for bank accounts
  - Create a class describing the properties and behaviors of "all" bank accounts

Defining Classes

- The next example shows you how to **define** your own class
- This is the way you **create** a new class

Example 9.1: Class Definition Syntax

```
1  class Classname
2     def initialize(var1, var2, ..., varn)
3        @variable_1 = var1
4        @variable_2 = var2
5        ...
6        @variable_n = varn
7     end
8
9     def method_1
10       # code
11    end
12
13    def method_2
14       # code
15    end
16 end
```

Class Definition: Example 9.2

- We will explain **class generation** and **object instantiation(s)** using an example of a bank account management system
- First, create a Class called **Account**
  - Note the Capital letter!!

Example 9.2: Account Version #1

```
1 class Account
2   def initialize(balance)
3     @balance = balance
4   end
5 end
```


Class Definition

- The variables inside the parenthesis after initialize are the **parameters** that are assigned when instantiating an object

```
1 class Account
2   def initialize(balance)
3     @balance = balance
4   end
5 end
```

constructor

Properties of an Instantiation

- An object will have a variable called "**balance**" with an **initial value** which you have to assign using a parameter

```
1 class Account
2   def initialize(balance)
3     @balance = balance
4   end
5 end
```


Class Instantiation

- The special character, @, is used to indicate that it is a parameter available to all methods of the class that are used by the object
- Variables starting with @ are called **Instance Variables**
  - They are available to ALL methods within the class

```
1 class Account
2   def initialize(balance)
3     @balance = balance
4   end
5 end
```

Class Instantiation

- You can instantiate an object of the Account class the same way you create new strings and arrays:

```
bob = Account.new(10.00)
```

(Note: You did NOT have to define a Method called "new". That is done for you by Ruby.)

Class Instantiation

- The parameter passed in the parenthesis will become the initial balance of Bob's account

```
bob = Account.new(10.00)
```


Data and Methods

- Now that we know how to define the Account class, we should consider its functionality:
  - What variables do we need?
  - What methods would be useful?
- No class needs particular variables and methods
  - The **constructor** is the exception to this rule
  - Classes are used to group **functionality** and **data** associated with it in one compartmentalized structure
  - Methods and variables are dictated by this goal

Data and Methods

- The Account class could use more variables to store information such as:
  - Name
  - Phone number
  - Social security number
  - Minimum required balance

Example 9.3: Account Version #2

```
1 class Account
2   def initialize(balance,name,phone_number)
3     @balance = balance
4     @name = name
5     @phone_number = phone_number
6   end
7 end
```

constructor

We insert two new variables to the class

Example 9.3: Account Version #2

```
1 class Account
2   def initialize(balance, name, phone_number)
3     @balance = balance
4     @name = name
5     @phone_number = phone_number
6   end
7 end
```

name and phone_number will help make each instantiation unique

Data and Methods

- New instantiation of an object from the Account class:
  `bob = Account.new(10.00, "Bob", 7166349483)`
- Regretfully, there is absolutely *nothing* we can do with this class, except for instantiating new objects
- It would be useful to have some real functionality (i.e., being able to withdraw and deposit)

Example 9.4: Account Version #3

```
1 class Account
2   def initialize(balance, name, phone_number)
3     @balance = balance
4     @name = name
5     @phone_number = phone_number
6   end
7
8   def deposit(amount)
9     # code
10  end
11
12  def withdraw(amount)
13    # code
14  end
15 end
```

Data and Methods: Implementing Methods

- Once the details of the Account class are finalized, a programmer can use it without knowing the code
  - They only need to know:
    - Data needed to initialize the class
    - Data needed for each method in the class

Example 9.5: Account Version #4

```
 1 class Account
 2   def initialize(balance, name, phone_number)
 3     @balance = balance
 4     @name = name
 5     @phone_number = phone_number
 6   end
 7
 8   def deposit(amount)
 9     @balance += amount
10   end
11
12   def withdraw(amount)
13     @balance -= amount
14   end
15 end
```


Now, initialize the classes to use these methods:

```
irb(main):003:0> require 'account_4.rb'
=> true
irb(main):004:0> mary_account =
  Account.new(500, "Mary", 8181000000)
=> #<Account:0x3dfa68 @balance=500,
  @name="Mary", @phone_number=8181000000>
irb(main):005:0> mary_account.deposit(200)
=> 700
irb(main):006:0> mary_account
=> #<Account:0x3dfa68 @balance=700,
  @name="Mary", @phone_number=8181000000>
```


Data and Methods: Implementing Methods

Now, let's create a method to make the output simple:

Example 9.6: Display method

```
1 def display()
2   puts "Name: " + @name
3   puts "Phone Number: " + @phone_number.to_s
4   puts "Balance: " + @balance.to_s
5 end
```

Data and Methods: Implementing Methods

- Let's use the new display method to output the account data in the objects:

```
bob_account = Account.new(500, "Bob",
  8181000000)
mary_account = Account.new(500, "Mary",
  8881234567)
bob_account.withdraw(200)
mary_account.deposit(200)
bob_account.display()
mary_account.display()
```


Data and Methods: Implementing Methods

- We will move money from Bob to Mary's account:
  - Two methods are called: withdraw & deposit

```
bob_account = Account.new(500, "Bob",
  8181000000)
mary_account = Account.new(500, "Mary",
  8881234567)
bob_account.withdraw(200)
mary_account.deposit(200)
bob_account.display()
mary_account.display()
```


Data and Methods: Implementing Methods

- We could make a method that does both at the same time, but this would mean the method calls **two different instances (objects)** of the same class
- A method can call multiple different instances of the same class by **passing objects as parameters** into the method
  - In our case, we need two instances of the same class, so we will transfer one as a parameter

Example 9.7: Transfer Method

How to pass in the object:

```
1 def transfer(amount, target_account)
2   @balance -= amount
3   target_account.deposit(amount)
4 end
```

None of our defined methods returned a value to the invoking statement. To obtain this value, a method must be defined that returns a value.

Example 9.8: Status Method

The implementation for our method:

```
1 def status
2   return @balance
3 end
```

The return construct returns the value of `@balance` to the invoking statement. Because there is no local overriding parameter called `@balance`, the global value for `@balance` is accessed.

Summary

- Classes can be created by a **definition process** via the **constructor**
- Classes are meant to **group data and method**s together
- The process of instantiating objects creates **compartmentalized** objects with their data
- Once an object has been created, it abstracts the details **away from** the program that uses it
  - You can use an object without seeing the details of that object directly