

Strings and Variables

```
class Player
```

```
  attr_accessor :name
  attr_reader :health
```

```
  def initialize(name, health=100)
    @name = name.capitalize
    @health = health
    @found_treasures = Hash.new(0)
  end
```

Inside the *initialize* method we store an object's state in **INSTANCE VARIABLES**. The values assigned to those instance variables are unique to each object. Instance variables always start with the @ sign, and they spring into existence the first time you assign to them. Unlike a method's local variables which evaporate when the method is finished, instance variables live for the life of the object. We can refer to instance variables in any instance method. For example, we can refer to *@health* in the *to_s* method.

```
  def to_s
    "I'm #{@name} with health = #{@health}, points = #{points},
    and score = #{score} as of #{time}."
  end
```

```
  def time
    current_time = Time.new
    current_time.strftime("%I:%M:%S")
  end
```

A **LOCAL VARIABLE** is local to the method it is declared in. It is not accessible outside the method. Therefore, we cannot call *current_time* in the *to_s* method; instead, we call *time* (the name of the method).

```
  def strong?
    @health > 100
  end
```

```
  def blam
    @health -= 10
    puts "#{@name} got blammed!"
  end
```

Double-quoted STRINGS allow you to embed variables, substitute in the value of any Ruby expression, and use various escape sequences. Ruby evaluates (interpolates) Ruby code within a double-quoted string using *#{}*.

```
  def w00t
    @health += 15
    puts "#{@name} got w00ted!"
  end
end
```

In a **single-quoted STRING**, you have to use a backslash to escape any apostrophes so that Ruby knows that the apostrophe is actually *in* the string literal instead of marking the end of the string literal. You also have to use + to concatenate (link together) single-quoted strings. For example: *puts '\m ' + name + ' with health = ' + health.to_s + '.'*

```
player1 = Player.new("moe")
player2 = Player.new("larry", 60)
player1.blam
player2.w00t
puts player1
puts player2
```

Classes

`class Player` ←

`attr_accessor :name`
`attr_reader :health`

`def initialize(name, health=100)` ←
 `@name = name.capitalize`
 `@health = health`
 `@found_treasures = Hash.new(0)`
`end`

Built-in Ruby class

`def to_s`
 `"I'm #{@name} with health = #{@health}, points = #{points},`
 `and score = #{score} as of #{time}."`
`end`

`def time`
 `current_time = Time.new`
 `current_time.strftime("%I:%M:%S")`
`end`

`def strong?`
 `@health > 100`
`end`

`def blam`
 `@health -= 10`
 `puts "#{@name} got blammed!"`
`end`

`def w00t`
 `@health += 15`
 `puts "#{@name} got w00ted!"`
`end`

`end`

`player1 = Player.new("moe")`
`player2 = Player.new("larry", 60)`
`player1.blam`
`player2.w00t`
`puts player1`
`puts player2`

CLASSES are used to easily and consistently create new objects and encapsulate an object's behavior in methods. A class is like a blueprint for instantiating objects (instances). Class names start with an uppercase letter, and multi-word class names have each word capitalized (e.g., ClumsyPlayer).

The **initialize method** is a special "constructor" method. You never call it directly. Instead, Ruby will automatically call *initialize* when you call the *new* method on the class. You define an initialize method to initialize your object's state.

To create a new object, call the **new method** on the associated class.

What does this Player class do?

It allows us to consistently **create player objects** with state and behavior, like this:



Methods

```
class Player
  def initialize(name, health=100)
    @name = name.capitalize
    @health = health
    @found_treasures = Hash.new(0)
  end

  def time
    current_time = Time.new
    current_time.strftime("%I:%M:%S")
  end

  def blam
    @health -= 10
    puts "#{@name} got blammed!"
  end

  def w00t
    @health += 15
    puts "#{@name} got w00ted!"
  end

  def strong?
    @health > 100
  end

  def points
    @found_treasures.values.reduce(0, :+)
  end

  def to_s
    "I'm #{@name} with health = #{@health}, points = #{points},
    and score = #{score} as of #{time}."
  end

  def score
    @health + points
  end
end

player1 = Player.new("moe")
player2 = Player.new("larry", 60)
puts player1
puts player2
```

A **METHOD** is a set of expressions that returns a value. It's a reusable chunk of code that you call by name. A method should do one thing well.

To define your own method, (1) start with the keyword **def**, (2) followed by the method name and the **method's parameters** between parentheses, and (3) finish with the keyword **end**. Method names start with a lowercase letter, followed by a mix of letters, numbers, and underscores.

You **call a method** by using the syntax:
`method_name(parameter1, parameter2, ...)`

A **default parameter** value can be specified in a method definition to replace the value of a parameter if it is not passed into the method.

Methods that end with a **question mark** return true or false. Methods that end with an **exclamation point** (such as *reverse!*) permanently change the value of the object on which it is called.

Method calls can be **chained** together.

Methods invoke (call) other methods by simply using the method name.

The **new method** automatically calls the *initialize* method. Any parameters passed in to the *new* method are passed along to *initialize*.

All classes inherit a default **to_s method** (which returns a cryptic string representation of the object). We chose to define a custom *to_s* method. To call the *to_s* method, we pass the player object to *puts*.

Attributes & Instance Methods

```
class Player
```

```
attr_accessor :name ← Readable and writable attribute  
attr_reader :health ← Read-only attribute
```

Outside of a class, we refer to the state of an object as its **ATTRIBUTES**.

```
def initialize(name, health=100)  
  @name = name.capitalize  
  @health = health  
  @found_treasures = Hash.new(0)  
end
```

An object's state is stored in its **instance variables**.

```
def to_s  
  "I'm #{@name} with health = #{@health},  
    points = #{points}, and  
    score = #{score} as of #{time}."  
end
```

```
def strong?  
  @health > 100  
end
```

```
def blam  
  @health -= 10  
  puts "#{@name} got blamed!"  
end
```

```
def woot  
  @health += 15  
  puts "#{@name} got wooted!"  
end
```

```
def points  
  @found_treasures.values.reduce(0, :+)  
end
```

```
def score ←  
  @health + points  
end
```

We refer to an object's methods as its **INSTANCE METHODS** because they must be called with the object (instance) as the receiver, such as *player1.blam*. Instance methods have access to the object's instance variables, which means the instance methods act upon the object's state.

A **VIRTUAL ATTRIBUTE** (in this case an accessor) typically derives its value from other instance variables. To the outside world, it appears to be an attribute like any other. But internally, it doesn't have a corresponding instance variable.

```
end
```

Arrays

An **ARRAY** is a collection of objects. Like all objects in Ruby, it has its own state and behavior.

```
class Game
  attr_reader :title

  def initialize(title)
    @title = title
    @players = []
  end
```

Defines an instance variable that starts off as an **empty array**. This is the array's initial state, like so:

`@players = []`

using the append operator:

```
def add_player(a_player)
  @players << a_player
end
```

or using the push method:

```
def add_player(a_player)
  @players.push(a_player)
end
```

An instance method then adds player objects to the empty array instance variable.

`@players = [player1, player2, player3, player4]`

You can **add objects to an array** by using either the append operator or push (a built-in Ruby method).

```
strong_players, wimpy_players = @players.partition { |player| player.strong? }
end
```

This single line block **creates 2 arrays**. The *strong_players* array contains all the player objects for which the method *strong?* returns true:

`strong_players = [player1, player4]`

The *wimpy_players* array contains all the other player objects:

`wimpy_players = [player2, player3]`

Conditionals

```
require_relative 'player' ←  
require_relative 'die'
```

When you use more than one file for your program, you need to tell Ruby where to find the other files. In this case we're telling our program to find the *player.rb* **FILE RELATIVE** to the current file (they're in the same directory) and load the file. (Note that you don't need to include the *.rb* extension when requiring a file.) Loading the *player.rb* file causes the *Player* class to be defined.

```
class Game
```

```
  attr_accessor :title
```

```
  def initialize(title)
```

```
    @title = title
```

```
    @players = []
```

```
  end
```

```
  def add_player(a_player)
```

```
    @players.push(a_player)
```

```
  end
```

```
  def play
```

```
    puts "There are #{@players.size}  
        players in #{@title}: "
```

```
    @players.each do |player|
```

```
      puts player
```

```
    end
```

```
    @players.each do |player|
```

```
      die = Die.new ←
```

Creates a new *Die* object and assigns it to the variable named "die."

```
      case die.roll ←
```

Calls the *roll* method on the new *Die* object.

```
      when 1..2
```

```
        player.blam
```

```
      when 3..4
```

```
        puts "#{player.name} was skipped."
```

```
      else
```

```
        player.w00t
```

```
      end
```

```
      puts player
```

```
    end
```

```
  end
```

```
end
```

This **CONDITIONAL** sets up three possible paths through the game. Based on the outcome of the "roll," the game either *blams*, *skips*, or *w00ts* the player object.

Modules

game_turn.rb

```
require_relative 'player'
require_relative 'die'

module GameTurn
  def self.take_turn(player)
    die = Die.new
    case die.roll
    when 1..2
      player.blam
    when 3..4
      puts "#{player.name} was skipped."
    else
      player.w00t
    end
  end
end
```

game.rb

```
require_relative 'player'
require_relative 'game_turn'

class Game
  def play
    puts "There are #{@players.size}
          players in #{@title}: "
    @players.each do |player|
      puts player
    end

    @players.each do |player|
      GameTurn.take_turn(player)
      puts player
    end
  end
end
```

MODULES are an organizational aid that allow you to group together methods. A module is like a class, but a module cannot be instantiated. In other words, you don't create objects from modules. Technically speaking, a module is a namespace where methods (and other things) can live without worrying about clashing with other similarly-named things in the program. Module names always start with an uppercase letter, and multi-word module names have each word capitalized.

Module methods are defined on the *self* variable, unlike regular instance methods.

player1

player2

player3

What does this iterator do?

Each player object in the players array is passed in as a block parameter to the *GameTurn* module's *take_turn* method as its method parameter.

When calling a **module method**, be sure to capitalize it.

```
class Game
```

```
  def play(rounds)
    puts "There are #{@players.size}
      players in #{@title}: "

    @players.each do |player|
      puts player
    end
  end
```

A **BLOCK** is a chunk of code (expressions) between braces (single-line blocks) or between *do* and *end* (multi-line blocks) that is associated with a method call, such as *upto* or *each*.

Methods such as **each** that act like loops—methods that execute a block repeatedly—are called **iterators**.

Block parameters, located between the vertical bars, are local variables, so they can only be used within this method.

What does this block do?

Each *player* object in the *@players* array is passed in to the block as a block parameter. Then, for that object we call the *to_s* method printing out the player's info like so:

```
@players = [ player1, player2, player3 ]
```

1 2 3

```
@players.each do |player|
  puts player
end
```

1 2 3

I'm Moe with a health of 100 and a score of 103.
I'm Larry with a health of 60 and a score of 65.
I'm Curly with a health of 125 and a score of 130.

```
  1.upto(rounds) do |round|
    puts "\nRound #{round}:"
    @players.each do |player|
      GameTurn.take_turn(player)
      puts player
    end
  end
end
```

What does this block do?

An example of a **block in a block**. We want to iterate through a number of "rounds" (the first block) where each round then iterates through all the players (the second block) to give them each a *GameTurn*.

repeat entire process for each round

```
[ player1, player2, player3 ]
```

Die.new
die.roll
blam, skip or woot


```
class Game
```

A **BLOCK** is a chunk of code that is called on a method.

```
def print_name_and_health(player)
  puts "#{player.name} (#{player.health})"
end
```

```
def print_stats
  puts "\n#{@title} Statistics:"
```

What does this block do?

```
strong_players, wimpy_players = @players.partition { |player| player.strong? }
```

An example of a **single line block**. You can think of the curly braces as the do and end. This block creates 2 arrays. The *strong_players* array contains all the player objects for which the method *strong?* returns true. The *wimpy_players* array contains all the other player objects.

```
puts "\n#{strong_players.size} strong players:"
strong_players.each do |player|
  print_name_and_health(player)
end
```

What do these two blocks do?

Iterate through each player object in the *strong_players* array, and print the player's name and health.

```
puts "\n#{wimpy_players.size} wimpy players:"
wimpy_players.each do |player|
  print_name_and_health(player)
end
```

Iterate through each player object in the *wimpy_players* array, and print the player's name and health.

```
puts "\n#{@title} High Scores:"
@players.sort.each do |player|
  formatted_name = player.name.ljust(20, '.')
  puts "#{formatted_name} #{player.score}"
end
end
end
```

What does this block do?

Sorts the player objects in the *@players* array, and then iterate through each one printing each player's name (formatted as specified) and score.

Symbols & Structs I

treasure_trove.rb

class object

struct's attributes

```
Treasure = Struct.new(:name, :points)
```

A constant that references an array.

```
module TreasureTrove
  TREASURES = [
    Treasure.new(:pie, 5),
    Treasure.new(:bottle, 25),
    Treasure.new(:hammer, 50),
    Treasure.new(:skillet, 100),
    Treasure.new(:broomstick, 200),
    Treasure.new(:crowbar, 400)
  ]
end
```

A **STRUCT** is a convenient way to bundle related attributes. Think of it as an object without behavior. Calling **Struct.new** with a list of attributes returns a Class object, in this case *Treasure*, that has accessor methods defined for the attributes *:name* and *:points*. The class object can then be used to create specific instances (objects) of the Struct, such as a pie worth 5 points.

SYMBOLS are a way to name things (or identify something in your code). Symbols last for the life of the program. A symbol cannot be changed (it's immutable). Symbols start with a colon.

What does this code do?

It creates an array called *TREASURES* with 6 treasure objects:

```
TREASURES = [
  [pie, 5], [bottle, 25], [hammer, 50], [skillet, 100], [broomstick, 200], [crowbar, 400]
]
```

game.rb

local variable

name of module

scope resolution operator

name of constant in module, which is an array of treasure objects

```
class Game
  treasures = TreasureTrove::TREASURES
  puts "\nThere are #{treasures.size} treasures to be found:"
  treasures.each do |treasure|
    puts "A #{treasure.name} is worth #{treasure.points} points"
  end
end
```

Each treasure object is passed in as a block parameter.

What does this iterator do?

It iterates through the treasures array and prints out each object's name and point value like so:

```
There are 6 treasures to be found:
A pie is worth 5 points
A bottle is worth 25 points
A hammer is worth 50 points
A skillet is worth 100 points
A broomstick is worth 200 points
A crowbar is worth 400 points
```

Symbols & Structs II

treasure_trove.rb

```
Treasure = Struct.new(:name, :points)
```

```
module TreasureTrove
  TREASURES = [
    Treasure.new(:pie, 5),
    Treasure.new(:bottle, 25),
    Treasure.new(:hammer, 50),
    Treasure.new(:skillet, 100),
    Treasure.new(:broomstick, 200),
    Treasure.new(:crowbar, 400)
  ]

  def self.random
    TREASURES.sample
  end
end
```

Module methods are defined on the *self* variable, unlike regular instance methods.

Since *TREASURES* is a constant that references an array, we can call any array methods, in this case the *sample* method.

What does the random method do?

It returns a random treasure object, such as the bottle object.

TREASURES = [

pie 5	bottle 25	hammer 50	skillet 100	broom stick 200	crowbar 400
----------	--------------	--------------	----------------	-----------------------	----------------

]

game_turn.rb

```
module GameTurn
  def self.take_turn(player)
    die = Die.new
    case die.roll
    when 1..2
      player.blam
    when 3..4
      puts "#{player.name} was skipped."
    else
      player.woot
    end
    local variable
    name of module
    name of module method
    treasure = TreasureTrove.random
    puts "#{player.name} found a #{treasure.name} worth #{treasure.points} points."
  end
end
```

bottle
25

What does this code do?

Prints out the random treasure object's name and points, such as:

Curly found a bottle worth 25 points.

Hashes

player.rb

```
class Player
```

```
  def initialize(name, health=100)
    @name = name.capitalize
    @health = health
    @found_treasures = Hash.new(0)
  end
```

Even though a Treasure object is a Struct with two attributes, we want to pass in the entire Treasure object. So the method only needs one parameter.

```
  def found_treasure(treasure)
    @found_treasures[treasure.name] += treasure.points
```

key
syntax for summing the
existing amount (points) and
the new amount (points)

value

What does the found_treasure method do?

It adds the key-value pair of our randomly found treasure object to the @found_treasures hash. The key becomes the name of the treasure object, for example bottle, and the value is the sum of the points of that treasure.

@name	
@health	
@found_treasures	
key	value

@name	
@health	
@found_treasures	
key	value
bottle	25
hammer	50

Then the method prints out the found treasures name and points, and the contents of the entire hash, like so:

```
  puts "#{@name} found a #{treasure.name} worth #{treasure.points} points."
  puts "#{@name}'s treasures: #{@found_treasures}"
end
```

Curly found a hammer worth 50 points.
Curly's treasures: {:bottle=>25, :hammer=>50}

game_turn.rb

```
module GameTurn
  def self.take_turn(player)
    ...
    treasure = TreasureTrove.random
    player.found_treasure(treasure)
  end
end
```

treasure
object

What does this code now do?

The take_turn method now calls the found_treasure method on the player object taking a turn, passing in the randomly-found treasure to the method, which in turn adds it to the @found_treasures hash.

Hashes & Iterators

Online Ruby Programming Course
<http://pragmaticstudio.com/ruby>
Copyright © The Pragmatic Studio

player.rb

```
class Player
```

```
  def initialize(name, health=100)
    @name = name.capitalize
    @health = health
    @found_treasures = Hash.new(0)
  end
```

```
  def found_treasure(treasure)
    @found_treasures[treasure.name] += treasure.points
    puts "#{@name} found a #{treasure.name} worth #{treasure.points} points."
    puts "#{@name}'s treasures: #{@found_treasures}"
  end
```

Reduces the hash values
down to a single number.

```
  def points
    @found_treasures.values.reduce(0, :+)
  end
```

What does the points method do?

Iterates through the values in a player's
@found_treasures hash (in this case, the
treasures' points) and sums them:

75

Examples of virtual accessors that return
a value derived from an instance variable.

```
  def total_points
    @players.reduce(0) { |sum, player| sum + player.points }
  end
```

two block parameters

What does this method do?

Iterates through each player and sums up their
points, for example Moe's points + Curly's
points + Larry's points.

game.rb

```
class Game
```

```
  def print_stats
```

```
    puts "\n#{total_points} total points from treasures found"
    @players.each do |player|
      puts "\n#{player.name}'s point totals:"
      puts "#{player.points} grand total points"
    end
  end
end
```

What does this method do?

Using the methods defined above in the *Player*
class, it prints out the grand total of all the
points for all the players. Then, it iterates
through each player object and print out the
player's name and their total points:

500 total points from treasures found
Curly's point totals:
75 grand total points

@name	
@health	
@found_treasures	
key	value
bottle	25
hammer	50

found_treasure

Custom Iterators

player.rb

```
require_relative 'treasure_trove'
```

```
class Player
```

```
  def points
```

```
    @found_treasures.values.reduce(0, :+)
```

```
  end
```

```
  def found_treasure(treasure)
```

```
    @found_treasures[treasure.name] += treasure.points
```

```
    puts "#{@name} found a #{treasure.name} worth #{treasure.points} points."
```

```
  end
```

```
  def each_found_treasure
```

```
    @found_treasures.each do |name, points|
```

```
      yield Treasure.new(name, points)
```

```
    end
```

```
  end
```

```
end
```

two block parameters

What does this method do?

It iterates over each key-value pair in the `@found_treasures` hash and for each pair it creates a new *Treasure* object (defined as a Struct). The name of the new *Treasure* object is the same as the key in the `@found_treasures` hash and the points are the same as the value in the hash. In this example, we'd have two new objects.

hammer
50

bottle
25

game.rb

```
class Game
```

```
  def print_stats
```

```
    puts "\n#{total_points} total points from treasures found"
```

```
    @players.sort.each do |player|
```

```
      puts "\n#{player.name}'s point totals:"
```

```
      player.each_found_treasure do |treasure|
```

```
        puts "#{treasure.points} total #{treasure.name} points"
```

```
      end
```

```
      puts "#{player.points} grand total points"
```

```
    end
```

```
  end
```

```
end
```

The block parameter is the *treasure* object (Struct) created in the `each_found_treasure` method.

What does this code do?

For each player, it calls the `each_found_treasure` iterator method with a block that takes a *treasure* as a block parameter. The iterator then **yields** each unique treasure to the block such that it prints out the points and names on a per-treasure basis like so:

Larry's point totals:
200 total skillet points
50 total hammer points
25 total bottle points
275 grand total points

studio_game.rb

```
knuckleheads = Game.new("Knuckleheads")
knuckleheads.load_players(ARGV.shift || 'players.csv')
loop do
  puts "\nHow many game rounds? ('quit' to exit)"
  answer = gets.chomp.downcase
  case answer
  when /\d+$/
    knuckleheads.play(answer.to_i)
  when 'quit', 'exit'
    knuckleheads.print_stats
    break
  else
    puts "Please enter a number or 'quit'"
  end
end
```

← Loads the players **from the CSV file** given as a command-line argument or defaults to loading the players in the "players.csv" file.

← Prompts user for input.

← Formats the user's input.

← Uses a conditional case statement to direct the program's flow through 3 possible paths.

game.rb

```
def load_players(from_file)
  File.readlines(from_file).each do |line|
    add_player(Player.from_csv(line))
  end
end
```

The *load_players* method returns an array of lines from the file and iterates through each of them. The *add_player* method adds each player, per instructions from the *from_csv* method, to the *@players* array.

player.rb

```
def self.from_csv(string)
  name, health = string.split(',')
  new(name, Integer(health))
end
```

The *from_csv* is a class-level method that takes a string formatted as CSV. Then splits each line and assigns its fields to *name* and *health* variables. The *Integer()* method will raise an exception if health is not a valid number.

studio_game.rb

knuckleheads.save_high_scores

Saves the file.

game.rb

default
parameter

```
def save_high_scores(to_file="high_scores.txt")
  File.open(to_file, "w") do |file|
    file.puts "#{@title} High Scores:"
    @players.sort.each do |player|
      file.puts high_score_entry(player)
    end
  end
end
```

Opens the file.

Prints a heading.

Iterates through each player in the *@players* array (in sorted order) printing off their information per instructions in the *high_score_entry* method and **outputs it to a CSV file.**

Inheritance I

clumsy_player.rb

child class (subclass) inherits from parent class (superclass)

```
class ClumsyPlayer < Player
  attr_reader :boost_factor
```

```
  def initialize(name, health=100, boost_factor=1)
    super(name, health)
    @boost_factor = boost_factor
  end
```

```
  def w00t
    @boost_factor.times { super }
  end
```

```
  def found_treasure(treasure)
    damaged_treasure = Treasure.new(treasure.name, treasure.points / 2.0)
    super(damaged_treasure)
  end
end
```

Inheritance offers a convenient way to share code. It allows you to model is-a relationships between parent classes and their children. For example, in the game a *ClumsyPlayer* is a specialized type of *Player*. Child classes inherit methods from their parents, and can override those methods or add new methods.

What do these 3 methods do?

Initializes a clumsy player the same way as normal players with a name and initial health value. Calling **super** from inside a method calls the method of the same name in the parent class, passing in any parameters. Then, it initializes clumsy players only with a number representing a boost factor.

Every time a clumsy player is w00ted, he is given the *boost_factor* number of w00ts. For example, if his *boost_factor* is 5, he is given 5 boosts in health every time he is w00ted.

Creates a new *Treasure* object (named *damaged_treasure*) with the same name as the original treasure but with only half of the original treasure's points. That object is then passed to the default *found_treasure* method defined in the *Player* class.

player.rb

```
def found_treasure(treasure)
  @found_treasures[treasure.name] += treasure.points
  puts "#{@name} found a #{treasure.name} worth #{treasure.points} points."
end
```

The method in the parent class (in this case, *found_treasure* in the *Player* class) is called the "default" method.

Inheritance II

berserk_player.rb

```
class BerserkPlayer < Player
  def initialize(name, health=100)
    super(name, health)
    @w00t_count = 0
  end

  def berserk?
    @w00t_count > 5
  end

  def w00t
    super
    @w00t_count += 1
    puts "#{@name} is berserk!" if berserk?
  end

  def blam
    if berserk?
      w00t
    else
      super
    end
  end
end
```

player.rb

```
class Player
  def initialize(name, health=100)
    @name = name.capitalize
    @health = health
    @found_treasures = Hash.new(0)
  end

  def blam
    @health -= 10
    puts "#{@name} got blammed!"
  end

  def w00t
    @health += 15
    puts "#{@name} got w00ted!"
  end
end
```

What do these 4 methods do?

It initializes a berserk player the same way as normal players with a name and initial health value. Then, unique to berserk players, it creates an instance variable to track the w00t count setting the initial value to 0.

Returns true if the *w00t_count* is greater than 5. Otherwise, the method returns false.

First, *super* invokes the *w00t* method in the *Player* class. Next, 1 is added to the *@w00t_count* instance variable. Finally, if calling the *berserk?* method returns true, then a warning message is printed out.

If calling the *berserk?* method returns true, then this method will call the *w00t* method in the *BerserkPlayer* class. Otherwise, it will call the default *blam* method in the *Player* class.

Methods in the parent class are called "**default**" methods. They can be overridden in child classes.

Mixins

playable.rb

```
module Playable
  def w00t
    @health += 15
    puts "#{@name} got w00ted!"
  end

  def blam
    @health -= 10
    puts "#{@name} got blammed!"
  end

  def strong?
    @health > 100
  end
end
```

Mixins offer a way to share code (methods) across classes that aren't necessarily in the same inheritance hierarchy. A mixin is a module that gets included (mixed in) inside a class definition. How do you know what should be a mixin? Look for "-able" behavior in a class, in this case, play-able behavior.

Unlike typical module methods that are defined on self, we want these methods to become instance methods of the class they're mixed into. Therefore, there is no *self*.

These methods only work if the class they're mixed into (called the host class) has **instance variables** named *@name* and *@health*.

```
module Playable
  def w00t
    self.health += 15
    puts "#{name} got w00ted!"
  end

  def blam
    self.health -= 10
    puts "#{name} got blammed!"
  end

  def strong?
    health > 100
  end
end
```

OR

It's generally considered better design if mixins rely on **attributes** (getter and setter methods) rather than instance variables. That way, if an instance variable changes it doesn't break the mixin.

To read the value of an attribute, using **self** is optional. You could read the value of the *name* attribute by calling either *name* or *self.name*. Using *self* in this case isn't required because *self* is always the implicit receiver of a method call.

When assigning a value to an attribute, you must use **self**. If you try to assign to *health* (without the *self*), Ruby will treat *health* as a local variable and the *health* of the actual *Player* object won't be updated.

The *strong?* method needs to read the value of the *health* attribute, so **self** isn't required in this case.

player.rb

```
class Player
  include Playable

  attr_accessor :name, :health

  def initialize(name, health=100)
    @name = name.capitalize
    @health = health
    @found_treasures = Hash.new(0)
  end
end
```

Include makes all the methods in the module (*Playable*) available as instance methods in the class (*Player*).

If we want our module to use attributes instead of instance variables, the attributes need to be both readable and writable (*attr_accessor*).

Testing I

```
require_relative 'player'

describe Player do
  before do
    @initial_health = 150
    @player = Player.new("larry", @initial_health)
  end

  it "has an initial health" do
    @player.health.should == 150
  end

  it "has a string representation" do
    @player.to_s.should == "I'm Larry with a health of 150 and a score of 155."
  end

  it "computes a score as the sum of its health and length of name" do
    @player.score.should == (150 + 5)
  end

  it "increases health by 15 when w00ted" do
    @player.w00t

    @player.health.should == @initial_health + 15
  end

  it "decreases health by 10 when blammed" do
    @player.blam

    @player.health.should == @initial_health - 10
  end

  context "created with a default health" do
    before do
      @player = Player.new("larry")
    end

    it "has a health of 100" do
      @player.health.should == 100
    end
  end
end
```

Requires the code in the *player.rb* file.

The **Describe** method defines an example group. It takes a block (code between *do* and *end*) that makes up the group.

The **before** method allows us to share common setup code across code examples. It runs once before each of the examples. Instance variables set in **before** are accessible in the code examples.

A **code example** starts with *it* and an arbitrary string. It takes a block and inside the block we put an example of how the code is intended to be used and its expected behavior. It first invokes your code and gets back some results. It then checks that the results are what you expect.

RSpec arranges things so that every object has a **should** method. Here we are calling *should* on the result of the score method.

An **expectation** expresses how you expect the code to behave.

The **equality operator (==)** compares the result we get with the result we expect.

A **context** is a way to organize similar code examples. It typically has its own before block that sets up the "context" for the enclosed code examples.

Testing II

player_spec.rb

describe Player do

context "with a health greater than 100" do

before do

@player = Player.new("larry", 150) ← Sets up a player with an initial health of 150.
end

it "is strong" do

@player.should be_strong ← Calling *be_strong* calls the *strong?* method on the *@player* object. And because we use *should*, it expects the result of calling the method to return true.
end

context "with a health of 100 or less" do

before do

@player = Player.new("larry", 100)
end

it "is wimpy" do

@player.should_not be_strong ← Expects the result of calling the *strong?* method on the *@player* object to return false.
end

end

game_spec.rb

describe Game do

before do

@game = Game.new("Knuckleheads")
@initial_health = 100
@player = Player.new("moe", @initial_health)

@game.add_player(@player)

end

it "w00ts the player if a high number is rolled" do

Die.any_instance.stub(:roll).and_return(5)

@game.play ← Forces a specific number to be rolled. In this case, 5.

@player.health.should == @initial_health + 15

end

end

The **before** block is called before every example runs, so the instance variables set in before are accessible in the examples.

Sets up a game (named Knuckleheads) with one player object (with the name moe and an initial health of 100) and adds the player to the *@players* array.

Expects that if we call *play* and *roll* a 5, then the player's health should increase by 15.