# Основы ООП в Ruby. Private, protected, public. Instance variables. Перегрузка  операторов

Ruby — объектно-ориентированный язык программирования, поэтому знание парадигмы объектно-ориентированного программирования (ООП) является обязательным.

Тема объектно-ориентированного программирования достаточно сложна, поскольку в ней переплетаются множество подтем, которые, разделить нельзя, поэтому данная глава будет иметь итерационной подход к изложению материала, то есть мы будем двигаться по спирали захватывая все больший и большый объем информации.

### ****Что есть Объектно-ориентированное программирование?****

Объектно — ориентированное программирование — это программирование основанное на использовании объектов — экземпляров абстрактных типов (классах). Объект в ООП — это программная модель какого-то реально существующего объекта, например вас или меня, при этом будучи объектами реального мира мы будем являться экземплярами некоторого типа (вида) — Человека Разумного. Объекты являющиеся экземплярами одного класса называются родственными или братьями, а класс от которого они произошли, если провести аналогию, является шаблоном ДНК человека в котором описываются все общие моменты производных объектов, однако каждый объект может иметь свои отличия от другого объекта того же типа. Все люди одинаковые в том, что  имеют две ноги, два глаза и штуку на животе именуемую пупком, однако у всех людей различается цвет глаз, отпечатки пальцев, имя, характер, лицо. Не смотря на эти различия между людьми, мы все-таки отличаем других людей, скажем от макак или павианов. Все общее в людях и общий набор свойств определяется в классе, а все различное определяется в самих объектах.

ООП является значительным шагом в перед, который позволяет писать программисту не просто код, а описывать реальную модель взаимодействия объектов. Объектно — ориентированное программирование повышает читабельность кода за счет его систематизации и сходства с тем, как человек мыслит и представляет мир.

### ****Три кита объектно-ориентированного программирования: Инкапсуляция, Наследование, Полиморфизм****

Для начала давайте создадим код, на котором мы будем разбирать примеры,он будет состоять из двух классов: Monkey и Human, где Human, согласно революционной эволюционной теории Чарльза Дарвина наследуется от класса Monkey:

class Monkey

def initialize(height, weight)

@height = height

@weight = weight

end

def height

@height

end

def weight

@weight

end

def height=(height)

@height = height

end

def weight=(weight)

@weight = weight

end

end

class Human < Monkey

def initialize(height, weight, name)

super(height, weight)

@name = name

end

def name

@name

end

def name=(name)

@name = name

end

end

m = Monkey.new(120,55)

h = Human.new(180, 90, 'Ivan')

### ****Что есть Инкупсуляция?****

Инкапсуляция — это самый большой и жирный кит объектно-ориентированного программирования. Инкапсуляция заключается в том, что присущие функциональному программирования данные и функции над ними превратились в свойства и методы объекта, которые заключены в объект (запаяны в капсулу). Доступ к этим свойствам и методам доступен только через специально определяемые интерфейсы, таким образом скрывается внутреннее устройство объекта. если вы заходите покушать, вам не нужно знать все методы вашего организма по перевариванию пищи,вам не нужно обращаться к языку, чтобы тот чувствовал вкус,к слюнным железам, чтобы те выделяли слюну смачивающую пищу, к нижней челюсти, чтобы та пережевывала пищу до необходимой кондиции, снова к языку и пищеводу, чтобы пища попала в желудок, затем к желудку, чтобы тот обработал пищу желудочным соком и так далее. Все, что вам нужно — это воспользоваться специальным интерфейсом — ртом, положив туда пищу, все, внутренние методы вашего организва запустятся сами этим одним единственным интерфейсом (методом — аксессором, от access — доступ). Ваши свойства, как экземпляра Хомо Сапиенса, это размеры элементов вашего тела, цвет кожи, цвет глаз, родинки и веснушки, ваше имя и фамилия, ваше физическое,  духовное и психологическое состояние. Вы хороший, здоровый, темноглазый, бледнолицый, девяносто килограммовый и сто-восьмидесята сантиметровый Иван — и все, что я перечислил — это ваши свойства. Ваши методы — это ходьба, бег, употребление пищи, сон, чтение, писание, программирование и так далее. Свойства к которым можно получить доступ называются открытыми, открыты они не из-за какой-то своей особенности, а из-за того, что для них определены методы — аксессоры. Посмотрте код Monkey и вы увидите там четыре метода аксессора: #weight, #height, #weight= и #height=. Первых два называются ридерами (от read — читать), они занимаются тем, что предоставляют вам текущее значение одноименных свойств объекта @weight и @height. Другие два метода, #height= и #weight= являются аксессорами — врайтерами (от write — писать) и из задача состоит в определении (записи) нового значения свойствам объекта. Без методов аксессоров вы не можете получить доступ к свойствам объекта, вы не можете знать имя девушки, и есть ли у нее парень, пока не воспользуетесь соответствующими методами аксессорами: #name и #has\_boyfriend?. Аксессоры read и write также часто называют get и set — аксессорами (от get — получать, set — устанавливать).

### ****Расследование что есть Наследование****

Наследование — чуть менее упитанный кит ООП, тем не менее он очень важен. Наследование заключается в эволюции классов. Как человек эволюционировал из обезъяны, так и один класс может наследоваться от другого. Наследование заключается в передаче всей структуры класса Monkey в класс Human, при этом наследование реализуется очень просто, при определении класса Human достаточно дописать < Monkey, что и указывает на наследование класса Human от класса Monkey. Говорят, что Monkey наследует Human, а Human наследуется от Monkey. Говорят, что Monkey — супер класс или родительский класс, а Human — подкласс, субкласс или дочерний класс. На практике это означает, что для Human полностью копируется реализация класса Monkey. Такое копирование и есть Наследованием.

Наследование — очень полезная вещь, так как: предоставляет возможность повторного использования кода, спасает от повторений в коде. Скажем имея класс Human, мы можем наследоваться от него классами WhiteHuman и BlackHuman, которые будут иметь абсолютно одиинаковые свойства, за исключением цвета кожи и формы носа. Вместо написания двух новых классов, вы просто наследуетесь ими от базового класса — Human дополняя его новыми свойствами и методами, либо переписывая старые.

class BlackHuman < Human

def initialize(height, weight, name)

super(height, weight, name)

@skin\_color = :black

end

def skin\_color

@skin\_color

end

def skin\_color=(skin\_color)

@skin\_color = skin\_color

end

end

На случай, если темнокожего человека зовут Майкл Джексон, мы определяем метод-аксессор #skin\_color=, который позволяет изменить цвет кожи.

### ****Полиморфизм — все течет, все меняется****

Полиморфизм — самый щуплый кит объектно-ориентированного программирования, который тесно связан с китом наследования и делает наследование еще более мощным. Полиморфизм (поли — много, морфе — форма) заключается в том, что наследуясь, в классе — потомке (дочернем классе или подклассе или субклассе или …) вы можете переопределять унаследованные свойства и методы. Во всех трех примерах выше имеется метод #initialize. Метод #initialize, согласно договоренности, является тем методом, что автоматически выполняется при создании нового экземпляра класса, такие методы еще называют методами — конструкторами. В классах Monkey и Human данный метод занимается тем, что принимает данные переданные в метод .new при создании нового объекта и устанавливает эти данные в свойства создаваемого объекта. Поскольку у экзмепляров классов Monkey и Human набор свойств разный, то и разное количечество аргументов должно быть принято, от чего следует в классе Human переопределить данный метод таким образом, чтобы он мог задавать дополнительные свойства. В классе BlackHuman мы также переопределяем метод #initialize, однако он сам устанавливает свойству @skin\_color по умолчанию значение :black. В обоих классах Human и BlackHuman метод #initialize снабжен выражением super. Super — это очень полезное выражение, которое позволяет в контексте метода вызывать одноименный метод из родительского класса, другими словами, super позволяет реализовать что-то вроде «наследования методов» при котором мы в контексте переопределенного метода, вызываем старый метод и дополняем его. Все эти операции по переопределению и носят имя полиморфизм.

### ****Области видимости и типы переменных****

Невозможно объяснять более-менее сложные вещи не объяснив о том, что такое область видимости и какие переменный бывают. Область видимости — это способность переменной быть использованой в контексте определенного фрагмента кода. Класс, объект, процедура, метод, код, вложеный в блоки условных операторов и циклов — все это может быть областью видимости. В Ruby существует 5 типов переменных:

**variable** — локальная переменная, она доступна только в той области видимости, где была определена, а также во всех вложенных областях видимости.

**@variable** - переменная объекта (экземпляра класса), имена таких переменных должны начинаться с одного знака @ — это еще одно соглашение в Ruby, которое вы обязаны знать. переменные экземпляра класса доступны только в том объекте, где они определены и вроженных в него областях видимости.

**@@variable** — переменная класса, имена таких переменных должны начинаться с двух символов @. Их область видимости — класс в котором они определены и все экземпляры данного класса.

**$variable** — глобальные переменные, их имена должны начинаться с символа $, а область видимости — вся программа.

Constant - константы это не переменные, но предназначение похожее — хранить ссылку на объект. Имена констант должны начинаться с большой буквы. Область видимости константы такая же как и у глобальной переменной, единственное отличие заключается в том, что значение константы следует использовать для «статики». При попытке изменить значение константы, значение будет изменено, однако будет возвращено предупреждение. В большинстве других языков программирования констаты не могут переопределяться.

### ****Создание объектов****

В Ruby существут предопределенные типы данных, которые были рассмотрены в прошлой главе и абстрактные — то, что мы рассматриваем в данной главе. Отличие в создании базовых (предопределенных) типов и абстрактных заключается в том, что для базовых типов существует специальный упрощенный синтаксис:

5.object\_id #=> 11

"string".object\_id #=> 82742600

[1,2,3].object\_id #=> 82800010

Просто используя тот или иной литерал, мы автоматически создаем соответствующий литералу объект. Работая с абстрактными типами данных, нам необходимо явно определять объект при помощи метода .new класса, примеры:

|  |
| --- |
| m = Monkey.new(120,55)  h = Human.new(180, 90, 'Ivan') |

m и h являются объектами, соответственно, классов Monkey и Human, что означает, что они имеют тип Monkey и Human. На самом деле m и h являются всеголишь переменными ссылающимися на реальные объекты в памяти компьютера, которые не имеют имен, но имеют уникальные идентификаторы объектов, по которым к ним происходит обращение. Несмотря на это m и h, для простоты можно также называть объектами.

Создавая объект (экземпляр класса) нам необходимо воспользоваться методом .new. Метод .new принимает аргументы для создания класса, которые затем могут быть использованы в методе #initialize и создает новый объект.

### ****Методы классов и методы объектов****

Вы наверняка заметили, что в данном учебнике перед именами одних методов я ставлю знак #, а перед именами других я ставлю точку. Это такая договоренность, согласно которой имена методов вне текста программы начинаются с # если метод принадлежит объекту и с . , если метод принадлежит классу. Метод класса отличается от метода объекта тем, что приемником метода класса является класс, а приемником метода экземпляра класса является экземпляр класса (объект). Хочу заметить, что в Ruby даже сами классы являютя объектами. Отличие классов от объектов заключается в том, что каждый класс является экземпляром базового класса Class. Любой метод объявленный следущим образом:

def method\_name()

end

является методом экземпляра класса. Для того, чтобы объявить метод класса, вам необходимо использовать следущую конструкцию:

def self.method\_name()

end

, либо следущую:

class BlackHuman < Human

class << self

def obj\_count

count = 0

ObjectSpace.each\_object(self){|obj| count += 1}

end

end

def initialize(height, weight, name)

super(height, weight, name)

@skin\_color = :black

end

def skin\_color

@skin\_color

end

def skin\_color=(skin\_color)

@skin\_color = skin\_color

end

end

bh = BlackHuman.new(180,90,"John")

bh2 = BlackHuman.new(185,90,"Robert")

bh3 = BlackHuman.new(170,70,"Thomas")

puts BlackHuman.obj\_count # 3

Как видно из примера, вызов методов класса не отличается от вызова методов объекта, единственное отличие — это приемник метода.

### ****Указатель self****

В предыдущем примере вы могли видеть указатель self. self — это указатель на текущий объект, объект, в контексте которого проиходит действие. Когда вы пишите:

class ClassName

def self.method\_name()

end

end

Это в конечном счете интерпретируется как:

class ClassName

def ClassName.method\_name()

end

end

Когда вы используете self в контексте метода экземпляра класса, то self будет ссылаться на текущий экземпляр класса, пример:

class ClassName

def obj\_info

puts "ObjectID:   #{self.object\_id}"

puts "ObjectType: #{self.class}"

end

end

o = ClassName.new

puts o.object\_id #=> 83402010

o.obj\_info

#ObjectID:   83402010

### #ObjectType: ClassName

### ****Метакласс он же Singleton — класс (класс одиночка), он же Eigenclass, он же Virtual Class, он же …****

Объект, по сути является копией класса. При этом объект не может иметь методов, он использует методы объявленные в классе. Такой способ работы, очевидно, основан на желании экономии оперативной памяти, ведь вместо полного копирование объектом методов класса, объект просто обращается к классу за методами. Ruby обладает грандиозными возможностями метапрограммирования, например, вы можете объявлять методы для уже существующего объекта. Поскольку объект не может иметь методов, а методы объявляются для определенного одного объекта, а не для всех, то была введена такая концепция как метаклассы. Метакласс — это такой невидимый класс, который наследуется от того класса, от которого произошел объект. Другими словами, каждый объект имеет за собой метакласс и по настоящему является его экземпляром, что скрывается. Сам метакласс наследуется от класса, которому, якобы реально принадлежит объект. Уникальные методы объекта называются в честьэтого самого singleton — класса singleton-методами. Ниже приведен пример создания singleton метода:

def bh3.upcase\_name

@name.upcase

end

puts bh3.upcase\_name #=> THOMAS

puts bh2.upcase\_name #=> NoMethodError

Как видите из примера, объект bh3 являющийся экземпляром класса BlackHuman имеет уникальный метод #upcase\_name, которого не имеют другие объекты того же типа. Это и есть singleton-метод, который определен в singleton классе. Обычно Singleton класс не заметен, вызвам метод #class для bh3, мы получим имя класса BlackHuman, но не singleton класс. Для того, чтобы получить метакласс объекта, следует воспользоваться методом #singleton\_class:

singleton = bh3.singleton\_class #=> #<Class:#<BlackHuman:0x827c59c>>

singleton.superclass #BlackHuman

Первая строка кода возвращает полное имя класса, которое состоит из имени самого класса и всех его супперклассов. Метаклассы не имеют имени, а имеют лишь идентификаторы, что и показано в выводе: 0x827c59c.

Еще один полезный метод для работы с метаклассами — это метод #singleton\_methods, который возвращает все уникальные методы объекта, то есть те, которые не определены в его классе, то есть те, которые определены в его метаклассе, пример:

|  |
| --- |
| bh3.singleton\_methods #=> [:upcase\_name] |

Когда вы объявляете методы класса в один из описанных выше способов, то эти методы класса определяются в его метаклассе, поскольку, как уже говорилось, каждый класс также является объектом.

### ****Техника MonkeyPatching****

Данная техника не рекомендуется к использованию, хотя иногда она бывает самым удачным решением. Данная техника основана на том, что вы можте изменять реализацию класса непосредственно во время исполнения кода и в любом месте программы, где к классу имеется доступ. Ниже приведен простой пример MonkeyPathching’а:

class String

def upcase\_reverse

self.upcase.reverse

end

end

"hello".upcase\_reverse #OLLEH

"ruby".upcase\_reverse #YBUR

Опасным моментом использования MonkeyPatching’а является переопределение существующих методов, особенно когда переопределение происходит произвольно, то есть объявляя метод вы переопределяете уже существующий, о существовании которого вам не было известно.

### ****Немного Метапрограммирования****

Метапрограммирование — это техника написания программ, которые могут изменять сами себя или другие программы во время выполнения. Ruby предоставляет очень мощные возможности метапрограмирования, благодаря чему можно творить действительно волшебные вещи.

В примерах выше мы определяли различные методы аксессоры для доступа к свойстам объектов. Для каждого свойства нам необходимо было определить 2 метода аксессора для получения и установки значения свойству объекта. Представьте, что у объекта имеется несколько десятков свойств, тогда вам придется объявлять множество методов, которые могут составлять большую часть кода класса. Для получения методов аксессоров без необходимости определения каждого из них имеются специальные методы — генераторы аксессоров: .attr\_reader и .attr\_writer, которые создают соответственно методы чтения и установки свойств объекта, примеры:

class Monkey

def initialize(height, weight)

@height = height

@weight = weight

end

attr\_reader :height, :weight

attr\_writer :height, :weight

end

class Human < Monkey

def initialize(height, weight, name)

super(height, weight)

@name = name

end

attr\_reader :name

attr\_writer :name

end

Методы .attr\_reader и .attr\_writer принимают в качестве аргументов символы соответствующие именам свойств объекта. В Ruby также существует метод .attr\_accessor, который объединяет в себе .attr\_reader и .attr\_writer, пример:

class Monkey

def initialize(height, weight)

@height = height

@weight = weight

end

attr\_accessor :height, :weight

end

Как работают генераторы аксессоров, я покажу на примере attr\_accessor, который объединяет функционал обоих read и write генараторов:

class Monkey

def initialize(height, weight)

@height = height

@weight = weight

end

def self.attr\_accessor(\*args)

args.each do |accessor|

define\_method accessor do

instance\_variable\_get "@#{accessor}"

end

define\_method "#{accessor}=" do |value|

instance\_variable\_set "@#{accessor}", value

end

end

end

attr\_accessor :height, :weight

end

m = Monkey.new(120,100)

m.height #=> 120

m.height = 160

m.height #=> 160

Методы .define\_method, .instance\_variable\_set и .instance\_variable\_get являются одними из наиболее популярных методов в метапрограммировании на Ruby. Метод .define\_method используется для объявления методов, он получает в качестве аргумента имя метода, который необходимо создать и блок кода в который помещен код метода. Использование .define\_method должно быть оправдано, поскольку, во-первых, стандартный синтаксис с def более лаконичный, а во-вторых методы определенные с помощью .define\_method создаются медленнее, чем методы объявленные с помощью def, что снижает производительность приложения.

**.eval в метапрограммировании**  
.eval — очень интересный и очень мощный метод, который принимает строку кода и выполняет ее как код, простой пример:

eval ("puts \"ruby\"") # ruby

Не смотря на свою мощь, .eval применяется достаточно редко из-за своей небезопасности. Вместо .eval рекомендуется использовать .class\_eval, #instance\_eval и .module\_eval. Разница между этими методами заключается в ограниченном контексте выполнения, .class\_eval выполняет код для текущего класса, .instance\_eval для текущего экземпляра класса, .module\_eval для модуля. Поскольку класс и модуль являются объектами, то и для них доступен метод .instance\_eval.

module Mod

end

class Monkey

include Mod

end

dm = <<CODE

def hello

puts 'hello'

end

CODE

dm2 = <<CODE

def bye

puts 'bye'

end

CODE

dm3 = <<CODE

def my\_f

puts 'ruby'

end

CODE

m = Monkey.new(120, 80)

Monkey.instance\_eval(dm)

Monkey.hello #hello

Monkey.class\_eval(dm2)

#Monkey.bye #=> NoMethodError

m.instance\_eval(dm)

m.hello # hello

Mod.instance\_eval(dm)

Mod.hello # hello

Mod.module\_eval(dm3)

#Mod.my\_f #=> NoMethodError

m.my\_f # ruby

Monkey.singleton\_methods #=> [:hello]

Mod.singleton\_methods #=> [:hello]

m.singleton\_methods #=> [:hello]

#instance\_eval всегда работает в метаклассе объекта, из-за чего методы определенные с помощью #instance\_eval всегда определяются как методы класса или модуля, что на самом деле означают, что они определены в метакласе класса и метаклассе модуля. .class\_eval всегда выполняется в контексте самого класса, именнопо этой причине объявляемые с его помощью методы являются методами экземпляра класса. .module\_eval абсолютно аналогичен методу .class\_eval, только вместо класса простором для его действия является модуль. Вам следует запомнить эти различия и использовать #instance\_eval, .class\_eval и .module\_eval, вместо .eval.

### Примеры

Я не хочу скучно приводить примеры использования. В пояснении разницы между декларациями видимости методов мне помогут самые обыкновенные хоббиты. Они дружественны и открыты для других, но и у них есть также личная жизнь. Многое из их личной жизни доступно для других хоббитов так как в своем хоббитовом царстве они могут бродить по жилищам друг друга. Конечно, каждый хоббит имеет что-то очень личное, такое, что знает один лишь он и никому и никогда не раскроет своей тайны. Такие особенности устройства поведения хоббитов очень похожи на методы в Ruby. Давайте представим каждого хоббита как объект. Кстати, вот и класс для создания хоббитов:

class Hobbit

  def initialize(name, rooms, has\_ring)

    @name, @rooms, @has\_ring = name, rooms, has\_ring

  end

  def name

    @name

  end

  def name\_of(hobbit)

    hobbit.name

  end

  def rooms\_of(hobbit)

    hobbit.rooms

  end

  def hobbit\_has\_ring?(hobbit)

    hobbit.has\_ring?

  end

  protected

  def rooms

    @rooms

  end

  private

  def has\_ring?

    @has\_ring

  end

end

Каждый хоббит имеет 3 атрибута. Имя хоббита (name) декларировано как public, число комнат в доме хоббита (rooms\_of) декларировано как protected и наличие кольца (hobbit\_has\_ring?) — private. name, rooms\_of и hobbit\_has\_ring? — это методы аксессоры для получения значений соответствующих атрибутов объекта хоббита. Теперь давайте создадим пару хоббитов:

irb(main):001:0> require ‘hobbit.rb’  
=> true  
irb(main):002:0> frodo = Hobbit.new(‘Frodo’, 3, true)  
=> #  
irb(main):003:0> samwise = Hobbit.new(‘Samwise’, 2, false)  
=> #

### Публичные (public) методы

В Ruby все методы, для которых явно не декларирована видимость являются public, это означает, что метод доступен ото всюду, из любой части кода, которая знает какому объекту принадлежит метод. Мы можем легко узнать имена каждого хоббита:

irb(main):004:0> frodo.name  
=> «Frodo»  
irb(main):005:0> samwise.name  
=> «Samwise»

### Защищенные (protected) методы

Что насчет комнат в их домах? Эта информация доступна только через protected-методы:

irb(main):006:0> frodo.rooms  
NoMethodError: protected method `rooms’ called for #  
from (irb):6  
from :0  
irb(main):007:0> frodo.rooms\_of(frodo)  
=> 3  
irb(main):008:0> frodo.rooms\_of(samwise)  
=> 2

Как вы видите, мы не можем вызвать вызвать protected — метод объекта непосредственно, но один объект может вызвать защищенный метод для себя или любого другого родственного объекта. Таким образом хоббиты скрывают количество комнат в своих домах от других живых существ, но между собой у них в этом секрета нет. Если перейти на чисто программерскую терминологию, то public-методы доступны глобально, а protected могут вызываться только через методы родственных объектов (произошедших от того же класса).

### Приватные (private) методы

Что насчет наличия кольца у хоббита? Метод доступа к этой информации помечен как private. Как же мы можем получить к ней доступ?

irb(main):013:0> frodo.has\_ring?  
NoMethodError: private method `has\_ring?’ called for #  
from (irb):13  
from :0  
irb(main):014:0> frodo.hobbit\_has\_ring?(frodo)  
NoMethodError: private method `has\_ring?’ called for #  
from ./hobbit.rb:19:in `hobbit\_has\_ring?’  
from (irb):14  
from :0  
irb(main):015:0> frodo.hobbit\_has\_ring?(samwise)  
NoMethodError: private method `has\_ring?’ called for #  
from ./hobbit.rb:19:in `hobbit\_has\_ring?’  
from (irb):15  
from :0

Мы не можем получить доступ к методу has\_ring? непосредственно, и не можем получить доступ через другие методы объекта, даже если хоббит спрашивает сам себя. Давайте добавим в класс Hobbit новый метод i\_have\_the\_ring?

class Hobbit

  def i\_have\_the\_ring?

    has\_ring?

  end

end

Теперь у frodo есть способ проверить есть ли у него кольцо:

irb(main):021:0> frodo.i\_have\_the\_ring?  
=> true

В Ruby protected методы остаются открытыми для других объектов того же типа. По этому доступ к protected методу гарантирован пока есть объект — «приемник» (hobbit в hobbit.rooms) произошедший от того же класса что и объект метод которого мы хотим вызвать.

Private — методы позволяют вызывать себя только в контексте собственного объекта. Мы не можем вызвать их как: hobbit.has\_ring? или даже так: self.has\_ring?, но мы можем воспользоваться private методом просто вызвав его в контексте объекта: has\_ring?.

### Заключение

public — полностью открытые методы. Различие между protected и private может вызывать проблемы если вы не осведомлены о нем. В Ruby декларация private является даже более строгой, чем private в языке Си. Декларация protected является золотой серединой между public и private позволяя одному хоббиту узнать число комнат в доме другого ;-)

Я использую protected-методы когда они не предназначены для непосредственного вызова и я не пишу для них тестов. Я пишу тесты для public-методов которые вызывают методы обозначенные как protected. Кроме того, я избегаю private-методов потому что я не могу представить потребность блокировать доступ родственных объектов к методам друг-друга (хотя я вероятно должен если я не проверяю их), кроме того из-за использования private методов могут возникать проблемы с дебагом.