Scripting in RGSS Ruby for Intermediates and Experts

0th Edition

Boris Mikić alias Blizzard

Zagreb, 25.9.2007

Contents:

1. Introduction

Are you reading this to **learn** how to script? Then this is **NOT** for you. If you want to understand this e-book fully, you need basic scripting knowledge. You can read this, of course, but you'll end up wasting your time by not understanding even half of it.
Instead, this e-book will teach you how to become a better scripter. There are far too many scripters who know just a few basics and in the best case they can make a CMS. I will teach you how to handle data, how to make windows look nice and handle them efficiently, how to avoid lag, how to hunt down bugs and how to not be an idiot when scripting. At the end of each chapter there is a little summary, also teaching you how to choose between RAM and CPU specific for that chapter.
Note that I will explain you several things, but these are only the basics. You can't become a better scripter just by reading this e-book, it will only give you a head start in becoming a better scripter. You simply can't become a better scripter without practical experience.

1.1. Why RAM or CPU?

This is the most common question when programming. You will always have to choose between saving RAM or saving CPU. Should your program work faster or should it be smaller? Usually programmers avoid re-processing data by storing data. That way CPU time is saved and RAM is used instead. You will learn when it does make sense to store data and when it doesn't.
RGSS Ruby is a scripting language. A *Scripting Language* is a programming language that is not being compiled, but interpreted. Interpreted means each line of code gets separated, compiled and executed on the fly. For those of you who have programmed in compiling languages like C, C++, BASIC (which was earlier a scripting language!), Fortran, Pascal, etc. you will most probably remember the short time the compiler needs to translate your program code into machine code (depending on the size of your code, it doesn't have to be short…). Using RAM sometimes can be more reasonable in RGSS, because storing data into RAM and loading it into CPU cache/registers is being executed the same way in every programming language. But careful! Storing and Loading data is about 10 to 15 times slower than executing simple operations like summation.

2. Compatibility

This chapter will show you how to make your scripts work with scripts of other scripters more conveniently. If you don't make your scripts compatible, most won't be used. If a user has three cool scripts from three other scripters that work together just fine and yours just won't fit in, he will kick yours most likely and keep using the other three. Another reason for compatibility is to save your time and effort. If your script doesn't work with another one, you will need time to merge it with the other ones for that user.
If your script can be configured and/or has options, make a working precondition and turn off all critical options that need to be first set up correctly by users. It's always a good idea to make your script work *Plug 'n' Play* as most people don't bother reading instructions.

2.1. Aliasing

Aliasing is a built-in method to give methods an alias name. *alias* is a reserved word in RGSS Ruby.

```
alias not_for_underaged alcohol
```

This will give the method *alcohol* the alias *not_for_underaged*. Now your class instance can call the method *alcohol* through its alias *not_for_underaged* as well.

```
child.not_for_underaged
child.alcohol
```

Both will result in the same. Of course you are not limited to class methods. That what is so interesting for compatibility is an attribute of the aliasing method. Even if you redefine the original method, the alias will still execute the original method.

```
def alcohol
  loop do
    drink
    have_fun
    if enough or evening_over?
      break
    end
  end
end
# break point 1
alias not_for_underaged alcohol
# break point 2
def alcohol
  not_for_underaged
  if promille > 2.5
    hangover
  end
end
```

This code will do following:
- define *alcohol*
- give *alcohol* the alias *not_for_underaged*
- redefine *alcohol* (***not_for_underaged* will still execute the old method!**)
- note how the new method *alcohol* calls its old definition through *not_for_underaged*

Now let's analyze the code more deeply. Let's try following code:

```
child.alcohol
child.not_for_underaged
```

It will result in following actions:

- before breakpoint 1

```
# start alcohol
  loop do
    drink
    have_fun
    if enough or evening_over?
      break
    end
  end
# ERROR! Undefined method not_for_underaged!
```

- before breakpoint 2

```
# start alcohol
  loop do
    drink
    have_fun
    if enough or evening_over?
      break
    end
  end
# start not_for_underaged
  loop do
    drink
    have_fun
    if enough or evening_over?
      break
    end
  end
```

- all

```
# start new alcohol
  not_for_underaged
  # start not_for_underaged
  loop do
    drink
    have_fun
    if enough or evening_over?
      break
    end
  end
```

```
  # continue in new alcohol
  if promille > 2.5
    hangover
  end
# start not_for_underaged
  loop do
    drink
    have_fun
    if enough or evening_over?
      break
    end
  end
```

Can you see why it helps the compatibility? You can put into the end of methods or at their beginning some of your code. Of course, you can get more advanced if you need to process data differently.

```
def alcohol
  loop do
    drink
    have_fun
    if enough or evening_over?
      break
    end
  end
  if promille > 2.5
    hangover
  end
end
alias not_for_underaged alcohol
def alcohol(age = 17)
  age < 18 ? go_home : not_for_underaged
end
```

By default calling *alcohol* will always execute *go_home* (no argument was specified, default is 17 which is less than 18, in other words; you don't get a drink if you don't tell your age). Calling *alcohol(18)* will result in executing the old code of *alcohol*. Keep in mind, you cannot only add code at the end or beginning of the method, you can also override, branch or just manipulate them in any way.
From my experience so far, you can't alias module methods if they were defined using *def self.something*.

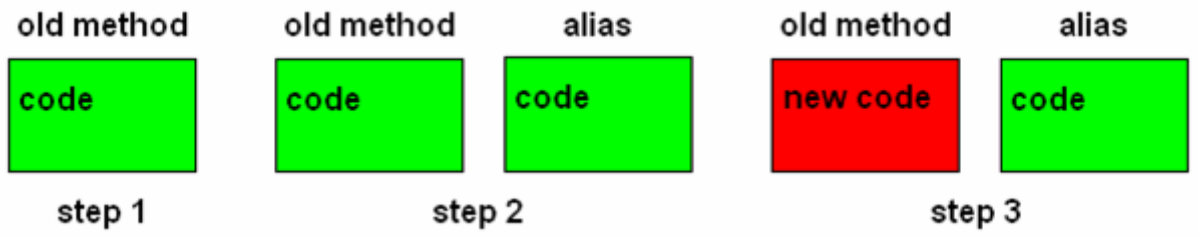


figure 2.1.1. – aliasing and the change of code

Keep in mind that if you alias a method that was already defined in a superclass and change the superclass' method afterwards or alias it, you will get a conflict between them and parts will not be executed at all.

2.2. Thinking in Advance

It's not really necessary, but it can be of big help sometimes. If you are using several of your own methods in more than one script, it might be a good idea to always alias them with the same name. Users will automatically get the *Stack level too deep* error (see ***2.4.*** for more information). What you can do is to include a global variable (i.e. *$my_script*) and set it as condition in all your other scripts to allow this part of code.

```
# my_script
def my_script
  do_something
end
if $my_extra_do_something != true
  alias do_something_more do_something
  def do_something
    do_this_and_that
    do_something_more
  end
  $my_extra_do_something = true
end
# my_2nd_script
def my_2nd _script
  do_something_new
  do_something
end
if $my_extra_do_something != true
  alias do_something_more do_something
  def do_something
    do_this_and_that
    do_something_more
  end
  $my_extra_do_something = true
end
# my_3rd_script
def my_3rd _script
  do_something
  do_it_again
end
if $my_extra_do_something != true
  alias do_something_more do_something
  def do_something
    do_this_and_that
    do_something_more
  end
  $my_extra_do_something = true
end
```

The aliasing of *do_something* will happen once whatever order your scripts are put into the editor. This can also help a lot if other scripters are using your aliased method. If any script already aliased this method, the new one won't. That results in more code than necessary, but it can improve compatibility incredibly if used cleverly.

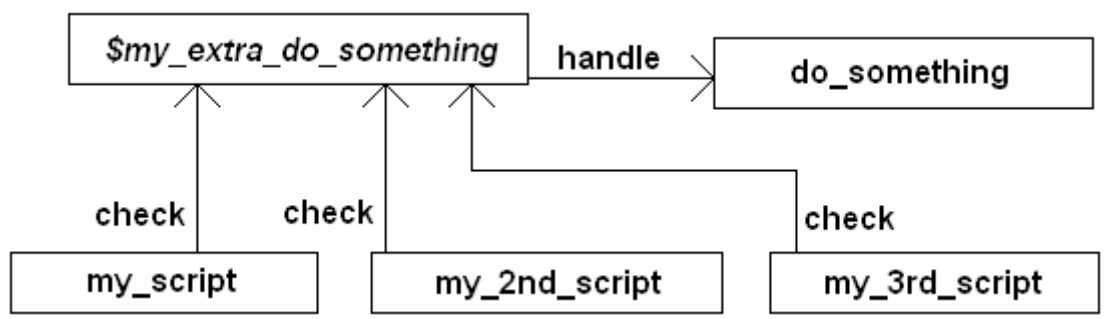


figure 2.2.1. – interacting scripts as block scheme

2.3. Names and Problems

The title says everything. When aliasing methods, you should pay attention to how you name the new aliases for your methods. Always a good idea would be using the *originalmethodname_yourname_yourscriptname* pattern. Or you can develop a unique style. I name my methods using the *originalmethodname_myscriptname_later* pattern. I also cut initialize to init, update to upd and dispose to disp, because those are the methods I mostly have to alias, so I don't have to type it out (see any of my scripts for more examples).
The usefulness of naming your methods correctly is that the names don't get too long (as in the SDK pattern for example where you have to include e.g. the date and class name in the alias name), they still are easy to find and there is as good as no chance that an accidental *Stack level to deep* error occurs (see ***2.4.*** for more information).

2.4. "Stack level too deep"

First a note for everybody who knows other programming languages as well: **RGSS Ruby does NOT support recursion!** Here is a fine example for recursion in C/C++. This would be the normal way to make a function that calculates the factorial of a number:

```
int factorial(int x)
{
    int i, result = 1;
    for (i = 2, i <= x, i++)
    {
        result *= i;
    }
    return result;
}
```

This is the same function working with recursion:

```
int factorial(int x)
```

```
{
    if (x <= 0) return 1;
    return x * factorial(x - 1);
}
```

Recursion allows calling the same method over and over as long as there is place on the *stack* (see ***chapter 8*** for more information). Here is the function in RGSS:

```
def factorial(x = 1)
  result = 1
  for i in 2...x
    result *= i
  end
  return result
end
```

This would be the same code in RGSS if it would support recursion:

```
def factorial(x = 1)
  return 1 if x <= 0
  return x * factorial(x - 1)
end
```

You should have seen my face when I found out that RGSS doesn't support it. And since it doesn't, the *Stack level too deep* error can arise. Following examples of code will raise that error:

```
def method_1
  method_1
end
```

```
def method_1
  method_2
end
def method_2
  method_3
end
def method_3
  method_1
end
```

```
alias new_method_1 method_1
def method_1
  method_2
end
def method_2
  new_method_1
end
```

Note that recursive codes MUST HAVE a condition of aborting. If a function just calls itself over and over without ever returning, it's malfunctioning. I haven't taken this into account on the three examples above.

2.5. RAM or CPU?

Now you have learned the basics how to make your script way more compatible with other's scripts as well as your own. You should know understand aliasing, what it does and how to use it correctly. Making your script more compatible won't either make you need to use more RAM nor more CPU. The worst thing that can happen is that you have multiple times the same piece of code in several scripts which is no big deal if they can work together or are not being executed at all due to smart aliasing.

3. Data Processing and Storage

In this chapter you will learn how to handle data and data processing. In other words, this chapter IS *RAM or CPU*. You will learn how you can decrease the number of coded lines.

3.1. Why "looping"?

The answer is very simple. Look at these two examples of code.

```
call_my_method
x += do_this
y += do_that
call_my_method
x += do_this
y += do_that
call_my_method
x += do_this
y += do_that
```

```
x = (x + 1) * 3 + 1
x = (x + 1) * 3 + 2
x = (x + 1) * 3 + 3
x = (x + 1) * 3 + 4
```

There's a lot of unnecessary code and with time your code gets hard to overview even if it's quite simple. In RGSS Ruby we can solve this problem with looping.

```
for i in 0...3
  call_my_method
  x += do_this
  y += do_that
end
```

```
for i in 1..4
  x = (x + 1) * 3 + i
end
```

Since there are two types of loops (determined, undetermined) we cannot only use looping to simplify coding, but also to run an undetermined number of code repetition.

```
loop do
  x = rand(10000)
  if x == 6284
    break
  end
end
```

This code will run as long as the random number is not exactly 6284. This is just a brief explanation what loops are and can be used for (see ***chapter 8*** for more information).

Side note: Don't get confused by the range definition. 0…3 will iterate from 0 to 2, that means excluding 3 while 1..4 will iterate from 1 to 4, that means including 4. Keep that in mind.

3.2. Why Methods/Functions?

Again the answer is very simple. If you have some piece of code that is used often, it's a good idea to group this piece of code into a *function*. For classes functions are called *methods*, but that's basically the same. Note that it doesn't make much sense to put some random code into a function, focus on really grouping code. When you are naming a function you should give it a name that kind of reflects what it's doing. This will help you make your code more readable and easier to overview. Another useful idea is to break down code into logic groups and finally into functions. Here is an example of code that was grouped and broken down into various methods. The possibilities are countless.

```
def remove_enemy_from_map(enemy)
  id = enemy.get_sprite_id
  if id >= 0
    enemy.set_fading_flag
    enemy.character.active = false
    enemy.freeze_current_action
    @spriteset.sprite_takeover_fade(prepare_sprite_fade(id))
    remove_enemy_character(enemy.id)
    @hud.kills += 1
  end
  return id
end
```

3.3. How to handle Data efficiently

There are many possible ways in which you can store and use data. Keep in mind that RGSS Ruby is a scripting language and its code is being interpreted and not compiled. This causes a much slower processing, up to 10 times slower or even more. That's why it is very important that your code is fast.
One common problem in this area is data structure. The most commonly used structures are *list*, *hash* and *stack*. Since RGSS Ruby's *Array* class supports the stack commands *push* and *pop*, you can use the array class for any type of arrays, lists and stacks.
Lists work on the *FIFO* principle (*First In First Out*). The first data that is added into the list is the first one that gets removed again. Imagine a pipe where you put balls into it. The ball you put first in will come out as first on the other side.
Stacks work on the *LIFO* principle (*Last In First Out*). The last data that was added will be the first one that is removed again. Imagine a few plates. If you respect the rule that you can take only one plate at once, you can only take the plate that is on top.
Hash is a completely different story. Hash works on the principle of *spread addressing.* While in an array data is stored sequentially, hash stores data anywhere. It uses a *key* to access data. Naturally such an access can very effective if the data you handle is very large, changed rarely and the so-called hash table is filled about 75%. Arrays and Hash both have the same average case of accessing data, though hashs can have a complexity of O(n) in worst case. In case of RGSS Ruby in RPG Maker mostly you have only small amounts of data. 1 MB (which would make 250000 4-byte integers) is a relatively small

amount of data if you consider that the average RAM today everybody has, is 512 MB. Even though RAM means *Random Access Memory* and data can be accessed very quickly, no matter where it is stored, there is a problem with CPU *L1* and *L2 cache* which will work much slower if you have to load data from all over the RAM, since the *L2 cache* load data blocks from the RAM. *L1* loads blocks from *L2*. This ensures optimal CPU performance as *L1 cache* memory is several times more expensive than *L2 cache* memory which again is several times more expensive than normal RAM. That's why nobody is making RAM out of the same technology that's used for *L1* or *L2 cache*: Financially it simply doesn't pay off. This is also the reason why CPUs with a bigger cache memory are valued very high and why it's said that it's optimal; it's the optimal solution between time and money. If some required data isn't in the cache, this is called a *cache miss* and a new block needs to be loaded which requires time. So use hash only if it is really necessary, really makes sense, really would result in a code that is much easier to overview and improve the performance otherwise. A nice example would be a very fast collision detection if you have a hash which's keys are map coordinates and all characters on the map at this specific position are stored in an array in this hash value.

Never use hash for configurations! Don't let users set up data and/or configurations with hash! Those would be very bad mistakes and only show people who have understandings of this method that you don't. The problem is that hashs just are not efficient enough in comparison to arrays if you don't use large amounts of data which you rarely will in case RGSS Ruby.

There is a clever alternative to hash, which works much faster and is even much easier to overview. This alternative is excellent for configurations by the user and mapping data similar to how hash actually works. The actual trick is that the *data mapping* IS spread, but the actual data in the RAM is stored as one block. In the praxis it has proven to be way easier to use by users. This alternative is a simple *conversion method*. As argument the method gets a key which it will transform to the corresponding value. The only disadvantage of such a method compared to hash is that it's constant data. You cannot change the *mapping* during the game at all. But every non-constant configuration is stored in special classes anyways (see ***7.2.*** for more information) so this method is as good as 100%-ly in advantage compared to hash configurations.

Example: Let's say you want equipment parts to trigger some special skills. An example would be a method like using the template *when EQUIP_ID then return SKILL_ID*:

```
def trigger_skill(id)
  case id
  when 1 then return 12
  when 4 then return 31
  when 7 then return 9
  when 12 then return 4
  when 81 then return 27
  end
  return 0
end
```

An equivalent with hash:

```
TRIGGER_SKILL = {}
TRIGGER_SKILL[1] = 12
```

```
TRIGGER_SKILL[4] = 31
TRIGGER_SKILL[7] = 9
TRIGGER_SKILL[12] = 4
TRIGGER_SKILL[81] = 27
```

Or the other way hash can be defined:

```
TRIGGER_SKILL = {1 => 12, 4 => 31, 7 => 9, 12 => 4, 81 => 27}
```

The last example is awful. It's quite hard to overview, especially for somebody who is not a programmer. The second example could pass, but I would say using *ANSI syntax* like in the first example should be easiest to understand. You can almost read it like "when id is 1 then skill is 12, when id is 4 then skill is 31…" The second example isn't as easy to read. Using a conversion method will make codes easier to overview and especially configurations the user has to set up. It also uses only the RAM required for code execution, which DOES take more space than the hash would, but it's stored as a data block and that way it will be way quicker in the CPU cache. Also keep in mind that memory storing and loading operations are 10 to 15 times slower than arithmetical-logical operations. Accessing one hash piece requires some time, another one far away takes the same amount of time again, etc. while accessing one and the same method a couple of times is much faster. That in both cases the parameter carry-overs needed for method execution (yes, getting a hash element by using a key is a method) took also time to be stored on the system stack as well as the return address to the current command was not taken into account as it happens in both cases and doesn't give any of those methods an advantage (see **6.2.** for more information).
Another data management possibility is using arrays. You can easily manipulate data by adding it at the end of the array, at the beginning, removing it from the end or removing it from the beginning. Here is a simple explanation of those 4 commands:

```
push    – add new item at the end
unshift – shift array and add at the beginning
pop     – remove last item from the end
shift   – remove first item and shift array
```

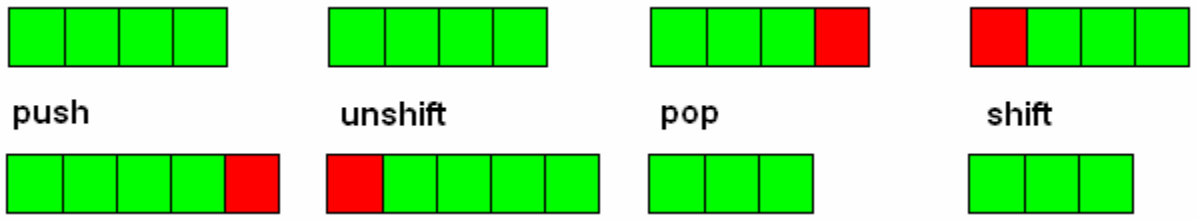


figure 3.3.1. – how push, unshift, pop and shift work

Read the RMXP Help File for more information. Note that you can add anything into an array. Your arrays can consist out of many various instances of any classes. There's nothing that prevents you to have 3 integers, one actor and 5 strings in an array. There's also the possibility to use an array if a method is supposed to return more than just one value.

```
def throw_2_dices
  return [rand(6) + 1, rand(6) + 1]
end
```

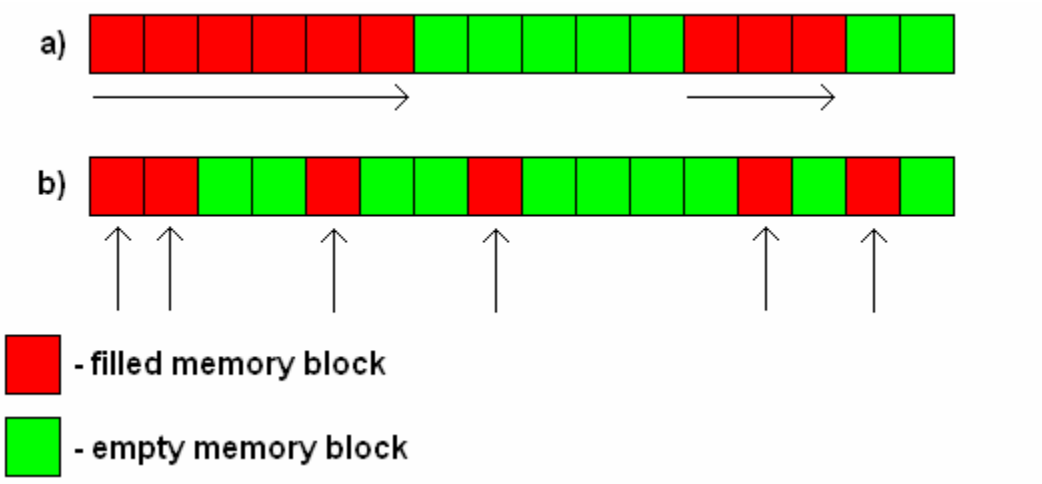


figure 3.3.2. – data accessing and reading a) array b) hash

3.4. Module or Class?

A question many scripters would kill for to get the answer. And in the end it's so simple. Use classes to process instances of data. For example, each actor is one instance, like a number or a bunch of grouped numbers. Imagine an array of 100 numbers free to your disposal. Classes are just bigger and even support complex processing via *methods*. A method is exactly the same as a *function*. The only difference is that a method directly processes the instance, doesn't need the class instance passed as argument and only gets processed by the instances of this specific class. One class can't use the method of another class. For example the class *Bitmap* will throw an *undefined method* error if you try the *size* method on it while on an instance of *Array* it will work just fine.
Modules are another story. Modules don't contain data (but they can contain some critical data), they are only a bunch of methods that can be used anywhere to process anything. There are many scripters who use modules for constants for configuration. This is a good solution if you have 10 or more options to avoid some configuration conflicts with other scripts, but for less it doesn't make much sense.
Examples of classes are the various windows in RPG Maker. Here is an example of loading an instance and working with it.

```
win = Window_Status.new
win.refresh
win.x += 16
```

In case of using a module it's different. You don't need an instance of the module to access it.

```
Input.trigger?(Input::B)
```

Keep in mind that when you define module methods, you can either use the normal way like *def method_name* and *module_function :method_name*, directly *def self.method_name* or *def NAME_OF_MODULE.method_name*.
Most probably you have noticed the *Input::B*. This is the way constants are access from within modules. The **::** is called *scope operator*. So modules are mostly used for a collection of methods and functions. You can also create methods like *def self.something* in classes and use them directly as if it was a module.
See ***chapter 8*** for more information about *Object Oriented Programming*.

```
class Game_Party
  def self.load_from_savefile(file)
    tmp = Marshal.load(file)
    $game_party = tmp if tmp.is_a?(Game_Party)
  end
end
Game_Party.load_from_savefile(file)
```

This example will work just fine, you can try it out. Just be sure to use it on a real *IO* instance. You can use a save game file and call it as long as *EOF* (*End Of File*) is reached. You will find out that the party status from the save game file was loaded.

3.5. RAM or CPU?

In this chapter you were made familiar with some basic terms of data handling. Data can become a cruel enemy is there's too much of it. Better let the CPU convert data instead of the RAM looking for it, especially if it's constant data. Data that is converted within the CPU already is in its cache while the data from the RAM is far away from that. But that makes only sense if the data is spread. Make hash your last solution in programming, not the first one.

4. Windows

No, it's not the operating system. This chapter will teach you how to handle windows, since they are actually consisting out of several sprites and sprites are the ones that can cause the most lag, especially if they are large.

4.1. The Basics

Windows are not as simple as it seems at first. Here are a few things you should always keep in mind as they will help you all the time.

1. Windows are several sprites, many windows will cause lag.
2. Windows use a bitmap called *contents*, it's size is the same as the window's, but smaller by 32 pixels in width and 32 pixels in height with an x and y offset of 16. If a window is smaller than 33×33 pixels and you try to create a bitmap by the default way, you will get the error c*ould not create Bitmap*.
3. Drawing windows' contents (called refreshing) is time consuming and can cause lag. Best if you refresh display only when it changes.
4. NEVER refresh a window all the time! Update yes, refresh no.
5. Moving windows is time consuming and can cause lag.
6. Changing the x and y attributes will cause the window to move, while changing the ox and oy attributes will cause the bitmap within to move, not the window itself.
7. The window's z position is a third dimension value and determines if a window should be displayed over other windows, sprites, etc. A higher value will bring it *closer to the player*.
8. If the bitmap within the window exceeds the window size, little arrows from the windowskin will be displayed. You just have to keep the bitmap size under control to prevent this.
9. Turning a window *active* will cause the cursor (if there is one) to blink while turning it inactive will cause the cursor to freeze its animation. To remove the cursor by default you need to set the window's *index* to -1. For this always use *self.index* and never *@index* directly. Note that only *Window_Selectable* has cursor handling methods, *Window_Base* has none. Though, you can create you own Window classes that can work with cursors from *Window_Base* if you want.
10. Turning a window into *pause* mode will cause the little indicator at the bottom to appear which is known from the message window when it waits for user input.
11. Dispose windows if you don't use them often or don't need them. Don't forget to set the variable containing the instance to nil. Clever window disposal and creation is the key to a lag free window system, even if it is animated (e.g. moving windows).
12. Don't make windows bigger than you actually have to. Bigger windows cause more lag.

In RGSS there is a superclass called *Window* which has all the basics inherited. Any instances is created internally with several sprites which is very fast due to the fact that *Window* is compiled into the RGSS10XX.dll. *Window_Base* is a class with enhanced methods to support easier working with it. *Window_Selectable* is a subclass of *Window_Base* and is used for windows with selections, because it already has several helpful methods predefined.

4.2. The wrong Way to create a Window

Window creation is not loading instances. It is connected to coding a window display. The most often used template to create windows is.

```
class My_Window < Window_Base
  def initialize
    super(x, y, w, h)
    self.contents = Bitmap.new(width - 32, height - 32)
    self.contents.font.name = 'Arial'
    self.contents.font.size = 22
    refresh
  end
  def refresh
    self.contents.clear
    self.contents.draw_text(x, y, w, 32, 'string')
  end
end
```

If you create a window *the wrong way*, it will just look ugly (*figure 4.2.1.*). Imagine a refresh method like this one:

```
  def refresh
    self.contents.clear
    self.contents.draw_text(13, 2, 128, 32, 'This')
    self.contents.draw_text(53, 7, 128, 32, 'window')
    self.contents.draw_text(111, 5, 128, 32, 'looks')
    self.contents.draw_text(140, 12, 128, 32, 'awful!')
  end
```

figure 4.2.1. – this window looks awful

Check often how your window looks like. It's always a good idea to use consistent offsets between various drawings. This code will generate a window like on *figure 4.2.2.*.

```
  def refresh
    self.contents.clear
    self.contents.draw_text(0, 0, 128, 32, 'This')
    self.contents.draw_text(72, 0, 128, 32, 'window')
    self.contents.draw_text(0, 28, 128, 32, 'looks')
    self.contents.draw_text(72, 28, 128, 32, 'better!')
  end
```

figure 4.2.2. – this window looks better

The 128 and 32 serve for a special purpose. The 128 is the width of the drawing. If your text is too big to fit in, it will be squeezed. The 32 serves for the height. You can leave this value usually 32, since it doesn't have too much of an effect if this value is higher than necessary. Usually this height is the font size, though some fonts might be actually bigger when drawn than the font size. That way it's possible that a font's size is 24, but the font's drawing takes up a height of 26.

figure 4.2.3. – squeezed text

figure 4.2.4. – height too small

I'm just happy that windows in RGSS are limited to rectangles… Find out more wrong ways to create windows for yourself. And for God's sake, don't really use them!

4.3. Do they look nice?

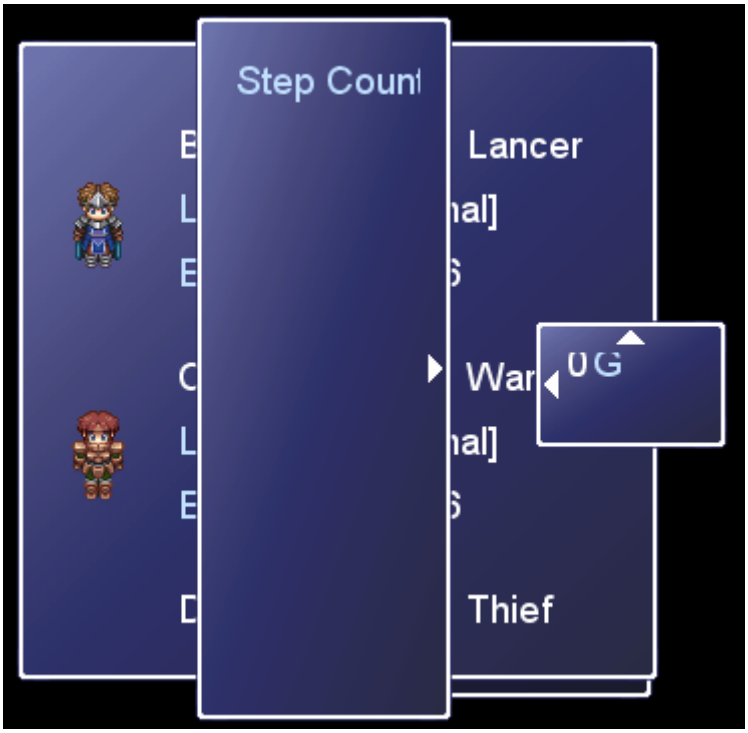


figure 4.3.1. – awful window order

They surely do not. At least not on *figure 4.3.1.*. What's wrong with the picture? I mean why does it look awful? There are three things that need to be considered:

a) window size
b) window position
c) window bitmap offset

Your best bet is to make all windows have size dividable with 32 (64, 96, 128…) or in some special case 16 can be enough (48, 64, 80, 96, 112…). It is important that you never make them less than 33×33, other wise you will get an error during the bitmap creation as said before.

Window position is a slightly different story. It is also a good idea to make windows have positions (x and y coordinates) dividable by 8. If you animate window movement, you can disregard this rule of esthetic as long as the final position of the window still respects this rule.

The last thing would be the problem with bitmaps exceeding the window size (and position, ox and oy). Usually you should avoid this to happen. The only situation where it makes sense to have bitmaps larger than the window are scrollable windows. These are not really *rules*, they are there to help you create an interface somebody can look at without getting a heart attack.

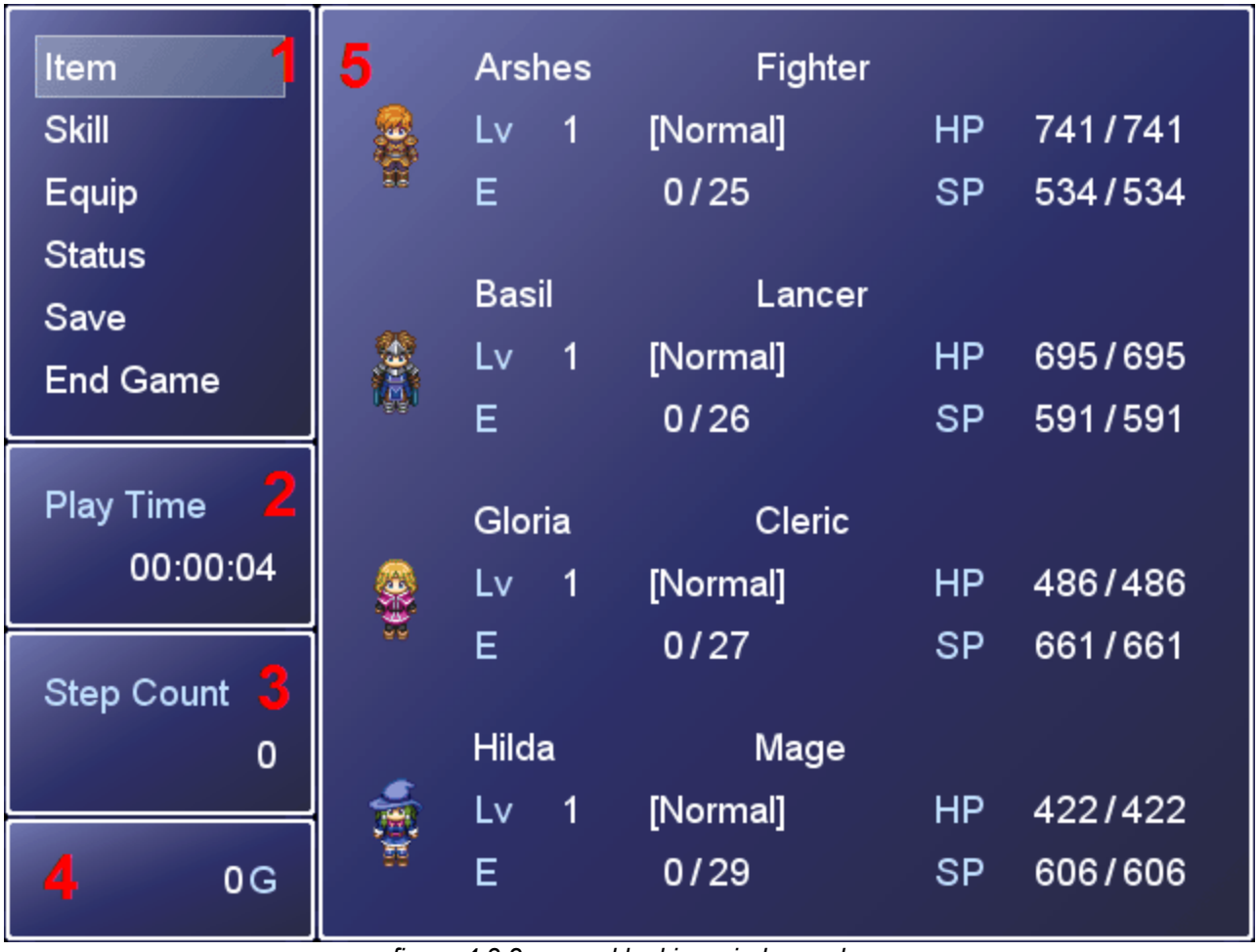


figure 4.3.2. – good looking window order

Doesn't *figure 4.3.2.* look much better? The window properties are:

1. x = 0, y = 0, width = 160, height = 224
2. x = 0, y = 224, width = 160, height = 96
3. x = 0, y = 320, width = 160, height = 96
4. x = 0, y = 416, width = 160, height = 64
5. x = 160, y = 0, width = 480, height = 480

Most probably you have noticed that all numbers have a common divisor: 32. The next problem would be something almost every beginner scripter has encountered. Do you recognize the situation from *figure 4.3.3.*?

figure 4.3.3. – problems with arrows

As already said a few times, this happens, because the bitmap is bigger than the window size allows. In this specific case the window is 96×64 and the bitmap is larger than 64×32. One easy way to avoid this unwanted effect is to ensure the bitmap's size in def initialize.

```
def initialize
  …
  self.contents = Bitmap(width - 32, height - 32)
  …
end
```

As you can see the entire trick consists out of the red bolded part. *width* and *height* are methods defined within the Window class and return the width and height of the current window if the words *width* and *height* are not being used as local variables. This will create sometimes bigger bitmaps than necessary, but at least it will ensure that no glitches occur and no arrows are being displayed.
The last problem is again the window position. Take a look at figure 4.3.4..

figure 4.3.4. – a picture which sight can kill babies

Overlapping windows are no problems in general. The problem occurs with windows that don't look like they are supposed to be overlapped.

figure 4.3.5. – looks as if the designer just messed up

Figure 4.3.5. looks as if the designer just messed up and not as if they are supposed to be overlapped. *Figure 4.3.6.* shows a much better overlapping look.

figure 4.3.6. – nothing special, but looks much better

4.4. Window or Sprite

Use a window if you want the window skin to be used. If you want a transparent background or display a special image as background, don't use a window with opacity = 0, but use a sprite. Be aware that you will run the risk that you will have to code special methods again (like *draw_actor_hp*) or copy them from the *Window_Base* class in the case you are using them. Although it might prove even better to code special methods which also use a smaller font, etc. But be careful as most probably you will end up with more coded lines and therefore a code that's harder to understand. In any case this will decrease the lag of your interface EVEN THOUGH you have more code (see ***5.4.*** for more information).

4.5. HUD Basics

There are 4 things you need to keep in mind when working with HUDs.

1. If your HUD can be turned off, make sure you DISPOSE the window/sprite and not make it just invisible. It will result in a little bit more code and way less lag.
2. Display those informations which the player needs access all the time. Displays like *Strength* are pointless as it isn't vital that the player knows the exact value all the time and it takes up space on the screen.
3. Refresh HUDs ONLY if data has changed. The map is already burdened with all the event sprites, the last thing it needs is to refresh your HUD every second or worse, every frame. To do so, store all the data that is displayed into instance variables and compare them in the *update* method with the current values. If any value has changed, refresh the HUD and store the new values. You can go beyond that and refresh only the section that has changed. This will result in more code, but can decimate even the refresh time. i.e. Imagine 20 values from which each needs 0.01 seconds to be drawn. That would result in 0.2 seconds to redraw the whole HUD and this would cause visible lag each time to do so. If you redraw just the one that has changed, it will need only 0.01 seconds and go unnoticeable. You can remove parts of the display on a *Bitmap* instance by using *fill_rect(x, y, width, height, Color.new(ANY, ANY, ANY, 0))*. This will remove the given section of the bitmap and you can draw the new information. If you want to go even beyond that, you can skip refreshing static words like *HP*, *SP* or *EXP*.

Depending on how your gradient bars/look like, you might not even need to remove them with *fill_rect* before redrawing them. (I use a trick for my HUD where I draw the HP value on the gradient bar. That way by just redrawing the gradient bar I already removed the *HP/MAXHP* display as well. I also don't need to clear that section of the gradient bar first due to the way I let them be drawn.)

4. Dilemma whether to use a sprite or a window (see ***4.4.*** for more information).

4.6. The Problem with non-vital Information being displayed

First off you need to decide which data is vital and which isn't. A HUD that contains a gold display with an icon will cause lag each time it is redrawn, it will take up screen space and it will only be refreshed after each battle anyway. In case of an ABS as battle system it can work out if it is kept small, since it changes often. But in the end, why does the player in a dungeon need to know the exact wealth of his heroes ALL THE TIME? He can easily call the menu and take a look at it if it really interests him. As said, only in an ABS it makes sense and only if it is kept as small as possible. The most vital informations are *HP*, *SP*, *EXP* and other battle related values like *Limit-Break* gauge, *Soul Rage* percentage, *Stamina*, etc. It really doesn't make too much sense to use a HUD system if no ABS is being used. HUDs for mini-games are ok, HUDs for special options like stamina on the map are ok. Everything else should be considered very well if it is worth to increase the lag on the map which is already overfilled with lag-causing events and event sprites anyway. Not to talk about the huge *Tilemap* of a 200×200 sized map.
The real problem with non-vital information being displayed is screen space. Less space to see what's actually happening in the game will decrease the gaming experience. So don't overfill the screen, keep HUDs as small as anyhow possible.

4.7. RAM or CPU?

In this chapter you should get an idea how to make windows look professional. Different than in nature-mapping, randomness is a big no-no when creating a window interface. Don't display useless or not so important informations and don't refresh them all the time. Use sprites if your background is supposed to be transparent and/or you are using a custom image as background. You can just draw the image on the sprite. And don't forget the 32 pixels rule. It can save you quite some designing time if you have *only* 20×15 (640/32×480/32) possible positions instead of 640×480.

5. Lag-weg

This chapter will teach you about the worst archenemy of every programmer and scripter: **Lag**. *Lag-weg* literally means *Lag-away* as *weg* means *away* in German.

5.1. Algorithm complexity

This field is complex enough to write a 20 pages essay about it, so I will give only a brief explanation what it means and what it's good for without using the appropriate terminology. It's quite simple: Algorithm complexity is used to determine formally an algorithm's speed depending on the given parameters. Check out this code:

```
p 1
if a
  s = Sprite.new
end
c = d + 11
```

What do you think its complexity is? It's O(1). Now this one:

```
for i in 0…n
  f += determine_special_string(n)
end
```

The algorithm complexity is O(n). What's with this example?

```
for i in 0…n
  for j in 0…n
    for k in 0…n
      make_some_lag
    end
  end
end
```

This algorithm's complexity is $O(n^3)$. Simply said, algorithm complexity tells you how the number of execution repeating depends on the number of processed data. In the last example an *n* of 2 would cause 8 iterations while an *n* of 3 would already cause 27 iterations and an *n* of 4 would cause 64 iterations. Can you see the pattern? Do you see how it increases rapidly? An *n* of 10 would cause already 1000 executions of the *make_some_lag* method. The most common complexity is (n^2) There were many algorithms developed that have a lower complexity. One of the favorites are algorithms with complexity of $O(n \times \log_2(n))$. The *log* (*logarithm*) function is the opposite operation of an exponent. For example, those equations here are correct and actually mean the same thing:

$$\log_2 32 = 5$$
$$32 = 2^5$$

The equally colored numbers are the same in both equations. You could read it like this: “If the base is 2, then the exponent needed to get 32 would be 5.” (see ***chapter 8*** for links about logarithmic algebra). All in all, this complexity is popular as it works like a damper. Examples of this would be: $\log_2(2) = 1$, $\log_2(32) = 5$, $\log_2(1024) = 10$, $\log_2(1048576) = 20$. If *n* becomes 1000, only 10 iterations are needed. If *n* becomes even 1000000. the number of iterations is just 20. The *n* multiplication in this complexity will have a heavy impact on the number of iterations, this is true, but the complexity $O(n \times \log_2(n))$ is much closer to $O(n)$ than it is to $O(n^2)$.
The field of algorithm complexity also does *best-worst-average case scenario analysis*. The explanation what that means is also quite simple. Image, that you are searching an array for one specific number. The best case scenario would be if the number was at the very first position. The worst case scenario would be that it’s at the last position or not there at all, since then you would need to iterate through all of the array’s elements only to find out it’s not there. Average case scenario is (best + worst) / 2 or in other words: In average the searched number is exactly in the middle. This is how algorithm complexity is usually being determined. Each factor or lower complexity components are being removed as their influence decreases with the number of iterations.
Let’s say the average case scenario’s complexity is $5/4*n^4+2*n^2+43$. The algorithm complexity is simply $=(n^4)$ as any factors and lower complexity components are being removed. See ***chapter 8*** for links about algorithm complexity.

5.2. What lags and why it lags (RGSS specific)

In common, graphics are known to cause the most lag, except for algorithms with a high complexity. The problem is that graphic handling is much more complex than it seems. A simple method like *set_pixel* in the *Bitmap* class causes a chain reaction. The *Graphics* module handles the display in RGSS. Changing one pixel using the method mentioned before will cause a change of data in the class instance itself, in the window/sprite that handles the *Bitmap* instance and finally an update in the *Graphics* module which needs to update the screen display by redrawing it with all the new information the next time *Graphics.update* is called. This is also the reason why invisible windows/sprites lag and each refresh lags.
Specifically in RGSS sprites and windows cause the most lag. The highest lag can be created in an interface with many created windows or the map itself with many events. Each event has a sprite assigned. Each sprite causes lag, regardless if it has a graphic or not since the testing of this sprite and it’s updating already takes quite some CPU time. Updating the event itself is quite fast, since the Interpreter class only needs to acquire commands and the execute commands. A map with 100 events that are being updated, but without sprites at all will cause unnoticeable lag. A map with 100 events and sprites without graphic that are not being updated will also cause little lag, but it might be noticed. This shows that just the mere presence of sprites causes lag. A map with 100 events and 100 visible sprites would look awful. This is the reason why the most Anti-Lag Systems are *Event-Anti-Lag Systems* that prevent updating events and their sprites outside of the screen, since they player doesn’t see them anyway. The makers don’t really care what happens to the event, they only use it to determine whether the sprite should really be updated or not. *Blizz-ABSEAL* has a more powerful technique to prevent lag. Instead of just disabling the update of those sprites, the sprites are being disposed and removed from

memory completely when the event is out of the updating range. The event update is also being restricted during that time.
So keep in mind that graphics are what causes most lag in RGSS. Especially sprites and windows which **consist out of several sprites** (see ***chapter 4*** for more information)! That's why you should concentrate on disposing invisible windows and sprites. This will increase the number of lines of your script, but also make it so fast that many will envy you for its performance power.

5.3. Decrease Process Time

Remember this: Executing each line is a nuisance for RGSS. If you want to speed your script you can do that in various way, The first thing you can do is to change your strings. One quoted strings are processed faster by the engine.

```
bitmap.draw_text(0, 0, 608, 32,
    'This string is being processed faster ' + "than this one.\n")
```

Use double quoted strings only if you use character substitutions (like *"\n"*) or interpolations/variable embedding (like *"#{some_numeric_calculation_or_value}"*). Don't use variable embedding if you have just one value. You can just use value.to_s which will convert value into a string.

```
bitmap.draw_text(0, 0, 128, 32, "#{actor.hp*100/actor.maxhp}%")
```

```
bitmap.draw_text(0, 0, 128, 32, $game_party.gold.to_s)
```

The first code above would draw out the percentage of the remaining HP of actor with a *%* character, the second would draw out just the number of gold the party currently has. Using something like this is unnecessary:

```
bitmap.draw_text(0, 0, 128, 32, "#{$game_party.gold}")
```

Or worse:

```
bitmap.draw_text(0, 0, 128, 32, "#{$game_party.gold}".to_s)
```

Another example on how to decrease process time obviously less code. With less code I don't mean a shorter script, but removing useless lines and unnecessary temporary variables (see ***5.4.*** for more information). The only thing that's more powerful, is conditional branching. Even if you have a formidable number of coded lines, your code will still be quite fast if most of it isn't even executed if not necessary. You can also decrease process time by using memory blocks instead of spread memory addressing (see ***3.3.*** for more information).
A trick to decrease process time is to store method results, so those methods don't need to be executed all over again. Instead of using a code like this:

```
for actor in $game_party.actors
  actor.hp -= calculate_special_skill_damage(skill)
end
```

You should rather use a code like this:

```
result = calculate_special_skill_damage(skill)
for actor in $game_party.actors
  actor.hp -= result
end
```

Though, always keep in mind that this only makes sense with more complex calculations. Storing and loading memory between CPU and RAM is about 10 to 15 times slower than simple arithmetical-logical operations. In the first code the method gets executed each time for each actor. In the second code the method gets executed only once and the result will be used later. This might not be a big decrease of processing time, but imagine you execute one method for each skill you have in your database which is, let's say, 200 and within this method you execute another thing like that for each of the, let's say, 300 animations you have in you database. Something like this will result in 200×300 = 60000 executions of the actual part for the animations. Do you want your script to take 60000 times longer for a part of code? I don't think so. Note that this does only make sense if the result really is the same for each actor like in the given example.
Here is an example where this won't help though:

```
for actor in $game_party.actors
  actor.hp -= calculate_special_skill_damage(actor.id)
end
```

```
result = calculate_special_skill_damage(actor.id)
for actor in $game_party.actors
  actor.hp -= result
end
```

The second code will either give you an error (*undefined method 'id' for nil:NilClass*) or worse: It will go through. In this case the result depends on which actor it is, so you can't calculate it in advance and store the result. You have no other choice than to calculate it each time for each actor, since it depends on the actor.
Another place where you can improve the performance would be stopping to process data more than once. This mostly happens if you don't fully understand how RMXP works. Here is an example of processing the same thing two times.

```
number = 5
"#{number}".to_s
```

This is unnecessary. Embedding and calling .to_s do the same, only the embedding is an operation that needs more time to be executed. Use to_s, it's about twice as fast. But be careful! Check out the next two codes.

```
nums = [0, 1, 2, 3]
"#{nums[0]} #{nums[1]} #{nums[2]} #{nums[3]}"
```

```
nums = [0, 1, 2, 3]
nums[0].to_s + nums[1].to_s + nums[2].to_s + nums[3].to_s
```

What do you think? Which one will work faster? You might believe it to be the second one and you are right. But it will work faster by less than 2%. Why is that so? It's pretty simple. to_s might be an operation that needs less time to be executed than variable embedding, but the concatenation of the strings (that means one is put at the end of the other using the + operator) will cause a CPU time loss. So, if you need to convert a variable to a string, use to_s, but if you need to convert it to a string and concatenate with another string, it pretty much doesn't matter which method you use. Just be sure not to use both.

5.4. Don't fear more Code

A problem many of you have encountered is most probably the problem when your code gets bigger and bigger. There are two different types of *bigger getting* codes.
The first is the one that has redundant lines. You could have used a different idea or method to process your data. You could have used less memory to process your data. It isn't bad if you have such lines, but try to keep them to a minimum. Most of those lines get created by your missing knowledge of how computers work and how RGSS interprets commands. When you have more experience and go through your older codes, you will often find lines where you will think or say "Hah! Why did I do THAT?! I could have easily done it that way, saved myself that temporary variable and had only 4 commands instead of 7…" or "Hey… I actually don't need this variable…" or…
The second type of code is what I call separation code. Of course there are things you can code *elegantly* with only several lines of code like a slanted gradient bar. But how much sense does it really make? This type of code uses conditional branching to separate code processes. Your elegant code might be able to process *this* and *that* and *that thing*, too, but if there is a better solution for *that* and *that thing*, why should an universal code process them if they could be coded more efficiently? Use conditional branching to separate them, let only *this* be processed by your basic code, create a new code for *that* and *that thing* instead. This will result in more code, but a better script as well. Example: A Gradient Bar Styler with 2 styles.

```
class Bitmap
  def gradient_bar_2(x, y, w, h, fill_rate, c1, c2)
    # fill background with black color
    fill_rect(x, y, w, h, Color.new(0, 0, 0))
    # iterate through x coordinates, but not for all, only for the
    # fill_rate
    for i in 0...(w * fill_rate).to_i
      # iterate through y coordinates
      for j in 0...h
        # how the calculation works:
        # 1. base color is the color of c1
        # 2. get the difference between c2 and c1
        # 3. depending on which iteration it is, a color between
```

```
        # c1 and c2 will be chosen
        # 4. what "* i / w" is for: it will multiply the difference
        # with the "percentage" of the current iteration and modify
        # the difference
        r = c1.red + (c2.red - c1.red) * i / w
        g = c1.green + (c2.green - c1.green) * i / w
        b = c1.blue + (c2.blue - c1.blue) * i / w
        # if the second style is used
        if $style == 1
          # This will modify the colors depending on which "row" is
          # currently being drawn. "style 1" will make the upper
          # lines darker than lower lines. The lowest lines are in
          # the normal color. The result is a multi gradient.
          r = r * j / h
          g = g * j / h
          b = b * j / h
        end
        set_pixel(x+i, y+j, Color.new(r, g, b))
      end
    end
  end
end
```

Drawing pixel per pixel (this is what *set_pixel* does) is very time consuming and can cause incredible lag. *set_pixel* is almost 40% faster than *fill_rect* and that means, use *set_pixel* if you need to draw one pixel, but use *fill_rect* if you need two or more pixels. Here is a solution which will cause *style 1* to still consume as much time as before, but *style 0* will be drawn MUCH faster.

```
class Bitmap
  def gradient_bar_2(x, y, w, h, fill_rate, c1, c2)
    fill_rect(x, y, w, h, Color.new(0, 0, 0))
    # check for style 1 already at the beginning
    if $style == 1
      for i in 0...(w * fill_rate).to_i
        for j in 0...h
          r = (c1.red + (c2.red - c1.red) * i / w) * j / h
          g = (c1.green + (c2.green - c1.green) * i / w) * j / h
          b = (c1.blue + (c2.blue - c1.blue) * i / w) * j / h
          set_pixel(x+i, y+j, Color.new(r, g, b))
        end
      end
    # this piece will get executed if "style 0" is being used.
    else
      for i in 0...(w * fill_rate).to_i
        r = c1.red + (c2.red - c1.red) * i / w
        g = c1.green + (c2.green - c1.green) * i / w
        b = c1.blue + (c2.blue - c1.blue) * i / w
        fill_rect(x+i, y, 1, height, Color.new(r, g, b))
      end
    end
  end
end
```

By comparing those codes directly (without the commented lines) you will notice that the second code is longer by 4 lines, but it works much faster in case *style 0* is being used. But hey, you don't have to believe me that code 2 is MUCH faster with drawing *style 0*. TO see it for yourself, just copy the first code and try each style. Then copy the second code and try them again. You will notice incredible lag with style 1 in both codes, but style 0 will work in the second code unbelievably faster. If you are going to try it out, make a new RMXP project and additionally use this drawing control code as help. Just add it above main as well as one of the codes for the gradient above.

```
$style, $c1, $c2 = 1, Color.new(255, 0, 0),  Color.new(255, 255, 0)
$sprite = Sprite.new
$sprite.bitmap = Bitmap.new(128, 16)
$sprite.z = 1000
class Scene_Map
  alias _5_4_test update
  def update
    $sprite.bitmap.gradient_bar_2(0, 0, 128, 16, 0.8, $c1, $c2)
    _5_4_test
  end
end
```

5.4.1. – the evil, lag causing gradient bar (left style 0, right style 1)

You may think this is not much of an improvement, but imagine a gradient bar styler with 6 styles where 5 styles could have been coded better.
If you are a caring scripter and keep your older scripts up to date, often you will see your older codes and have the itches to improve them. You will just feel the need to make them better, to make them reflect your current state of knowledge. (i.e. You won't find my Credits script anywhere on the web anymore. The improved Picture Movie Scene now does what my Credits script did and much more.)

5.5. RAM or CPU?

Lag is a problem everywhere in programming, not only game development. Lag is only a laic term to describe that the CPU isn't powerful enough to process data fast enough that it will go through *unnoticed*. Learn how programming works, learn how algorithms work, learn how your programming language works and learn how a CPU works and you are one step closer to decrease lag permanently. Keep in mind that this chapter isn't the only one that explains the problem with lagging, but it is a chapter to give you a better insight into the *why* and what you can do about it. Technically spoken, this entire e-book is just one big Lag-weg e-book.

6. Wannabe-Cool Scripting

You are a scripter/programmer, but before all, you are a human being like everybody else. There just are people in the world who seem to believe they would be better than other for either no real or some really idiotic reason. You won't come over as *cool* if you're being an idiot and rude to people. In this chapter we will work on your relations to the world around you, since it doesn't evolve around you: You evolve around it.

6.1. Scripts with {} Brackets

To clear out all misunderstandings first, a *{…}* command will do the same as *do … end* will. Those commands are used to define block commands. Block commands are being put together and interpreted altogether. This CAN improve performance if used cleverly, but doesn't have to. Any local variables defined within a block are removed when exiting the block.

```
loop {
  a = 0 if a == nil
  a += 1
  break if a == 100
}
a.ceil
```

This will give you an *undefined method or variable* error for *a*. For more specific details, see the RMXP Help File. Keep in mind that {} brackets are also used to determine a hash data structure as *Hash* is an arbitrary class in RGSS Ruby very similar to *Array* which can be defined by using [] brackets. {} brackets are also used when embedding variables into strings (see ***5.3.*** for more information). As you can see, {} brackets have various functions. So using {} brackets in scripts is in no way wannabe-cool scripting even if it might seem like that.

6.2. One Line Functions/Methods

One line methods and functions seem at first sight only a good waste of space. To make you understand why they are very time-consuming operations, you first need to understand what happens when a method is called. I will give you only a brief explanation on a basic CPU architecture, since you can write several books on CPU architecture.
A CPU has usually a specific number of registers as working memory where it saves the currently processed data (note that various CPUs have different numbers of registers, i.e. *Pentiums 4* have 128, older *ARM* CPUs have 37, etc.) Those registers are closer to the CPU than the so called *L1 cache*. A CPU directly operates over those registers. Registers usually have a size of 32 bits (or 4 Bytes), but a new generation of computers with 64 bit registers are already available on the market nowadays.

1) Your script code is not being executed as is, it needs to be translated first. Since RGSS Ruby is an object oriented language (see ***chapter 8*** for more information), the translation is even more complex. In a couple of translations your code gets usually first translated into a more simple code (with usually more lines of code) in which i.e. your

loops are completely simplified. In the end the code is translated into assembler commands and then into machine code. Some interpreters without built-in optimizers can even translate code directly into assembler code which takes less, but makes the CPU work slower as redundant commands are being executed.

2) When you use a function call, this is what happens in your PC:
- All commands in the *pipeline* need to be truncated, which causes losing 20 CPU cycles on a *Pentium 4* and that means the method call already takes the same amount of time like 20 summation operations.
- The CPU stores the address of the current command position into the RAM onto the stack which takes 10 to 15 times longer than letting the CPU add two numbers.
- Now all arguments that are passed onto the function need to be added onto the stack as well which will take some CPU time again.
- Now the so-called *program counter* (which determines the address of the next command to be executed) is being set to the address of the function's translated code.
- The *pipeline* gets filled and that means the CPU needs to wait for another 20 cycles until the next command can be executed.

3) Now the function gets executed. When the end of the function is reached a very similar process is being executed to restore the old location so the old code can be continued. Also any returning values are being stored onto the stack, so the CPU can access the results of the function call after it has returned to the position where the method originally was called from. Again the *pipeline* needs to be filled with the pending commands where it stopped before jumping to the function's address.

All of this might give the impression that functions are a bad concept, but this is not true. If there were no functions, there would be code repeating which needs MUCH more RAM. It's more economical to spend some CPU time for this instead of very much RAM. Also, this makes programming easier. The next two codes are completely idiotic.

```
def sum_two_numbers(a, b)
  return a + b
end
```

```
class Scene_Load < Scene_File
  def commandnewgame_partysetup
    $game_party.setup_starting_members
  end
end
```

Not even a very bad programmer would use something like this. There is absolutely no excuse using something as bad as this, especially since all what the second code does is an unnecessary reroute of a method call. The only situation where it makes sense to make a one-line method is re-rerouting variables from within other classes so the code is easier to read. So avoid use if anyhow possible.

```
class Game_Enemy
  def gold
    return $data_enemies[@enemy_id].gold
  end
end
```

For more information about CPU architecture, see ***chapter 8***.

6.3. Too many pointless and useless Commands

I just love stuff like this. People who consider themselves great scripters and somehow managed to fool other people into believing the same, yet they need 20 lines of code to make something work that could have been done with 10. It wouldn't be so bad if it was not obvious that it could have been done. I will not reveal the name of the author of the following code, but I think you know it already. The code will allow you to switch the party leader by cycling through all party members if you press L or R (Q or W). Take a look at the next code and most probably you will find a way to shorten it yourself.

```
class Game_Party
  def shift_forward
    @actors << @actors.shift
    $game_player.refresh
  end
  def shift_backwards
    @actors.insert(0, @actors.pop)
    $game_player.refresh
  end
end
class Scene_Map
  alias_method :s____partycycling_scnmap_update, :update
  def update
    if Input.trigger?(Input::L)
      $game_party.shift_forward
    elsif Input.trigger?(Input::R)
      $game_party.shift_backwards
    end
    s____partycycling_scnmap_update
  end
end
```

Isn't it just beautiful? Oh yes, it is. The following code will do the same thing.

```
class Scene_Map
  alias upd_ptc_later update
  def update
    if Input.trigger?(Input::L)
      $game_party.add_actor($game_party.actors.shift.id)
    elsif Input.trigger?(Input::R)
      $game_party.actors.unshift($game_party.actors.pop)
      $game_player.refresh
    end
    upd_ptc_later
  end
end
```

This takes us to the point of this chapter: Use things that are already there. I used the *add_actor* method from *Game_Party* which has already the *$game_player.refresh* call

included. Since there is no method to add an actor at the first position of the party and shift each actor by one position, I had to remove the last actor (*pop*) and put him at the first position (*unshift*) (see ***3.3.*** for more information about working with arrays). Use what you already have, don't write new stuff if you don't need it. A method call is a very expensive operation, especially a one-line method (see ***6.2.*** for more information). Note how I was able to manipulate the *@actors* array by calling the methods *pop* and *unshift* even though the *@actors* array is a private instance variable in *Game_Party*, it's only available for reading, not for writing. That means you can do *$game_party.actors*, but you can't do *$game_party.actors = something* (see ***7.8.*** for more information).

6.4. Too much SephirothSpawn

Please see ***6.2.***, ***6.3.***, ***6.5.*** and ***6.7.*** for more information.

6.5. Avoid being an Idiot

What I understand under being an idiot when scripting is first off all ignorance, secondly arrogance and lastly being obnoxious. If you are neither, you can skip this part.
If somebody has discovered a bug, **LOOK INTO IT**. Don't just say "It's your fault, my script works fine." It's fine to say "Try getting the newest version first.", maybe you have fixed it already. If he's using the newest version already, look into it. There are often situations where you haven't considered a condition or where a line is just faulty. It is also a very common occurrence that changing code (even if it was actually a bug fix!) causes new bugs. The bigger and more complex a script the more likely a bug is to appear. The more scripts you have, the more likely that they interfere with each other. Making your own scripts compatible with each other will turn out quite handy if you have released 5 or more 1000+ lines scripts or scripts that have a very complex structure (i.e. like an encoder or encryptor).
If somebody asks you to help him with a script that is not yours, you don't need to turn the person down in a mean way like "Huh?! This is not my script. Ask the maker. Hmpf, kids these days!" It's not hard to just type down the reason even if you just make it up. Of course it's better to ask the maker, but you can always say "I can look into it, but I can't promise anything. Your best chance is to ask the maker himself, he should know his script best." This has nothing to do with scripting, it has to do with having manners. The worst thing that can happen is to see a well-known and respected scripter do something like this. A disappointment that crushes your world into pieces that makes you wanting to never become like this.
Nobody will use your scripts if you're being rude to them. You're not the only RGSS scripter in the world. And if you are reading this e-book, you're most probably not one of the best either.

6.6. Re-invent the Wheel

Many might want to avoid that, but why?! There is a good point for each scripter to *re-invent the wheel*. The point is **NOT** that you have it easier and to have all you need to make a good script, the point is to learn something. If you just use the code of other's, you won't get far. Your scripts will be unoriginal, debugging will be a nuisance due to your lack of

understanding what actually happens in RGSS and your scripts will lag, because the other scripter might have messed up with his tool for you or they might just lag, because you don't know how to fix it. Don't get me wrong, tools are good, but try to create your own tools, not to use too many tools from other scripters. A much better idea for getting a tool would be to get it and check out the code instead of using it. You might learn something.
You like SephirothSpawn's slant bars? Or maybe Trickster's? Or maybe mine? Why don't you try it yourself? Get each of our codes, compare them, check out what they do and how they do and (maybe most important) why they do the things they do. Then try to make your own. Yes, you just have re-invented the wheel, but it turned out good you did. You have learned how to work with a bitmap and create bitmaps via script.
Another problem is the problem "But there IS already a script for this. Why should I make another one? It will be the same…" Wrong. Your codes will never be the same. It might turn out similar, but never the same (except you copy-paste, you thief!). Your ideas of how i.e. a party switcher should look like or how it should work will most probably be completely different from somebody else's ideas. Your main goal in this case should be "I want to make it better than that other guy." Learn from the mistakes others have made. Improving doesn't only mean to add functions or make it look different. Improving can also mean to get rid of useless functions and add really useful stuff instead.
On the other hand, making it easier for other scripters by making a complex tool is fine. But just making a few methods for them won't help them much in improving. Of course this applies to something like making a collection of methods and letting everybody just use what he likes, not to the case when somebody is asking "How can I do this and that? Can you help me?" Ask, it costs nothing and you will learn something.
I am always proud of a scripter who has asked me for help, maybe even for half of the script, but in the end he makes something on his own and mostly in a completely different way. I am proud that he has asked me, but in the end, he was his own teacher. Why did it turn out like this? He asks me for one thing, then another, then another, etc. and during that he gathers enough knowledge that he realizes his approach wasn't such a good idea as he could have made it much easier and better. This scripter could have just used my script, but he didn't. He learned how it works and wanted to try it out himself.

6.7. Enforcing Standards

DO NOT ENFORCE STANDARDS! Having a standard is fine, using a standard is fine, but enforcing a standard is the *Microsoft way* to do it: People have to use it else they are just screwed. People are beings who have ideals and individual characteristics. Enforcing your standard **WILL** make you many enemies who do not agree with your philosophy and make many people not using your scripts out of principle or just because they cannot being bothered to implement your standard into their games, because it is none and doesn't work with all the other scripts. And you're turning out to be an ignorant asshole by always saying "You need this, you need that to make my script work, because it's the standard and I don't go out of the standard.", especially if it's not a standard. For more information how to avoid begin an idiot, see ***6.5.***.
Don't get me wrong, standards are fine, but self-made standards surely are not. Standards do not make any sense if not EVERYBODY is using and following them. If you don't have the authority to enforce a standard, don't do it!
Interesting side notes: SDK (Standard Development Kit) is an ironic name, because it actually doesn't fulfill the definition of Standard*. There is no such thing as* standard dependent*. The SDK has only caused more*

*incompatibility issues so far by separating SDK and non-SDK scripts, no matter if they were aliased (for more about aliasing see **2.1.**) or not. SDK 2.x is much more incompatible, even with SDK 1.x scripts. Even today there is no standard defined for programming as it is a quite creative industrial branch. Today only several methods and approaches exist for programming. Standards don't make sense if not everybody accepts and uses them. This cannot be achieved by enforcing standards, especially if you claim your standard to be flawless even though it's obvious that it isn't. Setting standards should not be attempted by people who are not professionals in this area and who do not have the authority to do so. If the best of the best could not have created a standard within over 50 years of programming, how are a few amateurs supposed to be capable of doing so?!*

6.8. Scripts the World doesn't need

This is a matter hard to explain. Examples might be the best solution to understand what I mean:
Instead of dying from HP reaching zero, when SP reach zero, the hero dies.
Sometimes it's not easy to judge, but sometimes it's so obvious. The main reason in not making a script is that nobody's ever going to use it. Would you use this script? I wouldn't. Even from the view of the player I wouldn't want to see this function in a game. It would just annoy instead of adding spice to the game. Think about it.
The human mind is capable of inventing anything, even within abstract relations and that means they don't have to make sense. There are just scripts the world doesn't need. Another similar problem is the problem with scripts having pointless options. Would you ever want a skill to target 2 random heroes, the hero next one of them, the enemy you choose and one random enemy next to the enemy you've chosen?! Sorry, but this is idiotic.
Think about if a script really has a point to be made and think about how much sense it does to include options and which ones for that script. It's fine to have a few vital options for the user to customize it and/or make it maybe compatible with other scripts. But having too many options will only confuse the user, prevent him from using your script and most probably giving you a headache implementing them.
Imagine a 100 lines script with 30 options (that means actually only 70 lines of code). Does the user really need to set up if he wants to have a 13 or 14 frames transition, since it looks annoying if any other value is used anyway?! If there really is a user who needs this option, he will ask you and you will tell him to change just a line or two and it will work instead of you losing half an hour implementing this option and three other ones nobody's ever going to use. It's always better to have too few options than too many. If people need some special options that many would use, they will tell you. Good software development can be achieved through continuous work with the clients more easily than just you making it initially work with anything. You will simply waste your time for nothing.
Most people don't even read instructions and don't even know of what your script is capable of, especially if you include 30 pointless options. It's pretty discouraging if you get the new script that makes you shadows and reflections and see 5 pages of instructions with 50 options that don't make sense at all. Normal users are not scripters, they are just happy with the *black box* you give them and it works fine by setting up just three or four options. Stick to quality, not quantity.

6.9. RAM or CPU?

This chapter shows you that you are not superior to other people. It shows you that it's not cool to make your script big if it's unnecessary. Don't make big scripts just for the sake of

being big and don’t unnecessary process data. If you’re a good scripter, it doesn’t mean that you’re scripts need to be big, but to be small. It also doesn’t mean you’re better than other humans. Respect them as they respect you. If nobody is using the script you have put effort into, because you’re being an idiot and having no social manners, there’s no point for you to script for others anymore. Script for yourself and enjoy your loneliness and ego yourself, they don’t need it.

Ah right, I forgot… In this case use your both, your CPU and your RAM as much as possible. Process things before you act and store important things like social manners. You can store this into your RAM right now: You are not better than others, whatever you may think. Keep a low profile and be quiet. You will be praised more and earlier if you stay humble. If something goes on your nerves, just ignore it. Someday when you’re better, you will realize it doesn’t matter anyway. Or you can go and start a revolution if it does matter. That’s one thing that can never be too late.

7. Hints and Tricks

This chapter will show you some interesting ideas how to make your scripts work faster, be more compatible, be shorter and on top of that have a cool and professional looking code. Most probably you will enjoy this chapter most as it teaches you actually something to script and doesn't teach you the theory behind all of it too much. This chapter also includes a lot pf practical experience.

7.1. Pen on Paper or train your Brain

Have you ever tried making a script straight out of your head? Well, I have. I did it actually a lot of times, almost every time I am working on a script. There's a reason why I have chosen this "hint and trick" as first. The reason is that everything starts here, in your head. First there was the idea and in the end there was the script. What's in-between is for you to decide.

Let's say you have an idea and you want to make your script work this way. You start scripting. Suddenly you get into a dead end, since your idea somehow doesn't work. This happened to me with my first bigger script: The *Soul Rage System*. I was working on it two days until I had to enough. I realized that it would take less time to use a different approach and start the script almost completely from scratch again than removing the last couple of problems and bugs that were causing me a headache.

The point here is that I didn't know the battle scripts and I tried an own approach while I could have simply re-used the already existing scripts for more convenience and better compatibility. Or if I had first put everything down on paper, most probably I would have come to that idea earlier. Since my head was busy figuring out how to make it work, it had no time to think about how to make it really work. I was so concentrated on the details that I couldn't see the system as a whole. You see, the advantage of putting everything on paper is that you don't have to keep it in your head all the time. The human mind is capable of memorizing between 5 and 9 unrelated memories in one moment. It works similar to RAM in your PC. If something new needs to be put into the RAM, something old either needs to be deleted or stored on the Hard Drive for later. It's similar with our brain. Those unrelated memories are first *stored* into short-time memory in which they can stay for about 30 seconds before they get *deleted*. Through repetition of one memory it gets stored into long-time memory. This is how human beings memorize and learn things.

Now imagine what happens if you have those 5 to 9 unrelated memories and then comes another and then another and so on. It's hard to keep track of all of this unless you switch those memories between current, short-time and even long-time memory. This process is quite exhausting and most probably you will have to *start over* a couple of times (i.e. "Ok, let me see again… This is like that and that works…"). You have only two options. Either you just work it out this way and your brain will stay in form or you can just be more systematical and put everything on paper to create a program flow which you can easily track, modify and overview.

When we get older our brain starts to be more lazy and lazy. It's because of the exhaustion of memorizing things and thinking during the years. The human brain does more work in the first three years than in all the rest of its life. The brain of a newborn child is hyperactive. During the day it literally absorbs all kind of information and during the night while the child is asleep, those informations get post-processed and memorized. The only

reason why we believe to become smarter while we age is because we earn experience and through this we learn how to systemize our thoughts and doing, so it's not as exhausting as before, mostly even more effective. With fewer resources, we can do more. You can systemize your thoughts in your imagination, imagining colors, shapes, program flows, code and everything else or you can put it on paper. As already mentioned, keeping everything in your head is more exhausting, but also keeps your brain in shape. I recommend using this method as long as you feel capable of using it. When it becomes too much (which will be when you turn 20 or 21), your best bet is to switch the tactics. Start using pen and paper for a change. You will be surprised how much easier it will suddenly be to come up with ideas, yet you will still have ideas. It can be especially useful if you have to write down (or draw down) *all possible scenarios* if you have like 20 of them.
On the other side a common problem of people who work too much on paper is that they start being less creative. Their brain gets lazy. So when you feel that you are not getting any good ideas anymore, try switching tactics again, it might be just what you needed. If you always use the systematical approach, your brain gets used to one and the same pattern which it already knows. That's why it gets lazy, there's nothing new to learn or to do (beside the fact that it gets boring as well). The best method, of course, would be a combination, but it's up to you to decide if you will constantly work with half power or use your brain's potential as much as possible and then giving it a break. Take into account that the constant method could give only moderate results. You could just become tired while scripting constantly even though not with full potential.
All in all, you have to decide which you will trust more: Your imagination and creativity or your eyes and experience. Or will you be smarter than everybody else and use them both?

7.2. "Game_System" is your save data's best friend

How many times has it happened that some configurations need to be saved and you use something like this here?

```
$dns = My_new_DNS.new
class Scene_Save < Scene_File
  alias write_save_data_my_new_dns write_save_data
  def write_save_data(file)
    write_save_data_my_new_dns(file)
    $dns = Marshal.load(file)
  end
end
class Scene_Load < Scene_File
  alias read_save_data_my_new_dns read_save_data
  def read_save_data(file)
    read_save_data_my_new_dns(file)
    Marshal.dump($dns, file)
  end
end
```

Actually be happy if you made it this way, since you used aliasing… But this is not a good idea. You will destroy all your old save files in a way they will cause a crash as soon as you try opening them in *Scene_File*. This is what would be much smarter to do:

```
class Game_System
  attr_reader :dns
  alias init_my_new_dns initialize
  def initialize
    init_my_new_dns
    @dns = My_new_DNS.new
  end
end
```

The DNS data will be saved along with the instance of the *Game_System* class which is already saved by default. The worst that can happen is an *undefined method "whatever" for nil:NilClass* error, since it was tried to execute a method on *$game_system.dns*, but *@dns* is still uninitialized in the save file. It will load normally, it will work normally until the script tries to execute a method. Here is an alternate solution to prevent EVEN save file corruption:

```
class Game_System
  attr_reader :dns
  alias init_my_new_dns initialize
  def initialize
    init_my_new_dns
    init_dns
  end
  def init_dns
    @dns = My_new_DNS.new
  end
end
```

This code might first seem inefficient due to a one-line-method. To clear things up: **It IS inefficient!** But look at this little expansion here:

```
class Scene_Map
  alias main_my_new_dns main
  def main
    $game_system.init_dns if $game_system.dns == nil
    main_my_new_dns
  end
end
```

A save file that didn't have DNS before would initialize it as soon as it is loaded. I personally do not use something like that. Old save file corruption is a sacrifice I am ready to take if I can avoid one line methods and messing with scene initialization. I value higher compatibility with other scripts more than keeping old save files. You could use this code as well:

```
class Game_System
  attr_accessor :dns
end
class Scene_Map
  alias main_my_other_dns main
    $game_system.dns = My_other_DNS.new if $game_system.dns == nil
```

```
    main_my_other_dns
  end
end
```

Interesting enough that you wouldn't need to alias the *initialize* method in *Game_System* anymore. But the price might be higher than it seems. The DNS will be loaded the first time the *Scene_Map* is being executed, not earlier and this leave time for a bug and incompatibility issues with other scripts. The shortest solution is not always the best.
You are not limited to the *Game_System* class. Use it mostly for configurations and similar things. If you wish to add a new attribute to your party like an array of party skills, it would be a better idea to add this array into the *Game_Party* class instead so the engine and your scripts are more congruent.

7.3. Boolean Algebra

First be aware that in RGSS Ruby there are several notations for one and the same thing.

```
not a                          = !a          (complement)
a and b                        = a && b      (disjunction)
a and not b or not a and b     = a ^ b       (exclusive conjunction)
a or b                         = a || b      (inclusive conjunction)

complement              = NEGATION operation
disjunction             = AND operation
exclusive conjunction   = XOR operation
inclusive conjunction   = OR operation
```

The left ones are called *ANSI Syntax*, the second are called *Classic Syntax*. I preferred *ANSI Syntax*, since it makes the code more readable. I have switched to *Classic Syntax*, since there is a bug with the interpretation (see ***7.4.*** for more information) and I suggest you to do the same. NEGATION has the highest priority, next is AND, then comes XOR and OR has the lowest priority. You can manipulate priorities if you use () brackets. These are the properties of Boolean Algebra:

```
a and true          = a                       - neutral element true
a or false          = a                       - neutral element false
a and b             = b and a                 - commutative
a or b              = b or a                  - commutative
a and (b or c)      = (a and b) or (a and c)  - distributable
a or (b and c)      = (a or b) and (a or c)   - distributable
a or not a          = true                    - complementary
a and not a         = false                   - complementary
```

These are the Boolean Operations:

```
a or true           = true          - domination
a and false         = false         - domination
```

a and a	= a	- idempotent
a or a	= a	- idempotent
not (not a)	= a	- involution
a and (not a or b)	= a and b	
a or (not a and b)	= a or b	
a and (a or b)	= a	- absorption
a or (a and b)	= a	- absorption
a and (b and c)	= (a and b) and c	- associability
a or (b or c)	= (a or b) or c	- associability
not (a and b)	= not a or not b	- De Morgan's Law
not (a or b)	= not a and not b	- De Morgan's Law
(a and b) or (a and not b)	= a	- simplification
(a or b) and (a or not b)	= a	- simplification

You don't need to know all of those, you can conclude them by remembering only the properties. Use them well, they will help you in coding.

7.4. The evil Bug in "if" Branching

During my time I have found one serious bug in RGSS's interpretation of conditional branching. Just think about this piece of code here? What do you think? How will RGSS REALLY interpret it?

```
print false and true ? 1 : 2
```

If you think it will interpret it as if there is written *p (false and true) ? 1 : 2* which would cause the evaluation of *(false and true)* first, then you're wrong. The result of this line will be a window with *false* written on it. Take a look at this example:

```
a = nil
b = true
c = (a != nil and b ? 1 : 2)
print c
```

Now you would assume the result to be *2*, since *a* is nil and therefore the condition is not fulfilled. The print command will print out *false* again. Why? I don't know. There seems to be a problem with comparisons, since there ARE () brackets and that means the parenthesis is fine. The next code would work fine and print out *2*.

```
a = nil
b = true
c = (a != nil and b) ? 1 : 2
print c
```

Now check out this code:

```
a = nil
b = true
```

```
c = 3
if a == nil or b and c == nil
  print 1
else
  print 2
end
```

What do you think will be printed? You think it will be *2*? You are right. Again a bug as it seems. Oddly enough the next code will work fine and print *1*, even though all that was changed is the position of two conditions:

```
a = nil
b = true
c = 3
if a == nil or c == nil and b
  print 1
else
  print 2
end
```

There seems to be a problem with evaluation of conditions that include *false* and *and*. The actual bug is a problem with priorities when *false* is one of the values of the comparisons. If you use *&&* and || instead of *and* and *or*, you will see a big surprise: **IT WORKS FINE!** The reason is that *&&* has a higher priority than *and* and || has a higher priority than *or*. Here are two last examples to prove this theory:

```
a = nil
b = true
c = 3
if b and c == nil or a == nil
  print 1
else
  print 2
end
```

```
a = nil
b = true
c = 3
if c == nil and b or a == nil
  print 1
else
  print 2
end
```

Both codes work fine. I suggest you use () brackets each time to go sure, take a good look at your conditioning and use the () brackets when something like this could happen, work normally and use the brackets only if your conditions start behaving funny or you can just switch to *Classic Syntax* like I did (see ***7.3.*** for more information).
Keep in mind that I encountered this bug numerous times, I only wasn't aware that he was the cause until recently. Usually I put all my *and* conditions at the first place and after them the *or* conditions, so I would say I didn't encounter it too often (see ***7.5.*** for more

information why exactly I used this order of the conditions). Note that this bug is in all RGSS dynamic link libraries (dll) released up to this day.

7.5. First this or that? – When "if" goes crazy

A nice optimization of code can be achieved at conditional branching. You can imagine that an *if* statement is read from left to right by RGSS. If the first condition is true and there's an *and* before the next one, the next one will be read as well until a statement results in true or false. An interesting property is that a statement will automatically result in false if all conditions are connected by *and*s and only one condition results in false. Now when you consider that RGSS checks conditions from left to right then it would be wise to put the condition that is false most often of the time at the left-most position. That way RGSS needs to check a minimum number of conditions to skip that branch. It's a similar story if conditions are connected with *or*s. In this case put the condition that is most often true at the left-most position. That way if the first condition is true, the branch can be executed without the need to check all the other conditions first.
Another method how to decrease condition processing time is to use the advice given above and additionally put methods at those places where it will be check at least.

```
if moving? and not @solid
…
def moving?
  …
end
```

The method *moving?* will be checked first, then the class variable @*solid*. This is wrong. This code is better:

```
if not @solid and moving?
```

In this case the method *moving?* will be checked only when @*solid* is false. Moving might be more often false than @*solid* true, but consider that *moving?* could be a method with 20 lines of code. Even in the case that @*solid* was false only every 40 frames, it would still be an improvement, because a comparison is executed faster than a method with 20 *who-knows-what-they-do* lines.

7.6. The Trick with "unless" – De Morgan's Law

Note that De Morgan's Law is mere one part of *Boolean Operations* (see ***7.3.*** for more information).

```
if not a and b
```

```
if not (a and b)
```

```
unless a and b
```

```
unless a or not b
```

To keep it simple, I will come to the point. Code 2 and 3 are exactly the same, code 1 and 4 are exactly the same. *unless something* is the same as *if not (something)*. De Morgan's Law as one of the Boolean Operations tells you following:

```
not (a and b) = not a or not b
```

```
not (a or b) = not a and not b
```

This Law is universal for any number of arguments:

```
not (a and b and c and ... and z) = not a or not b or ... or not z
```

```
not (a or b or c or ... or z) = not a and not b and ... and not z
```

To apply this one RGSS together with using *unless*:

```
if a and b or not c = if not ((not a or not b) and c) =
 = unless (not a or not b) and c
```

Note that *and* has a higher priority than *or*, so the *or* part now needs to be put in () brackets. Though, you can still make it look different than the example above:

```
unless (not a or not b) and c = unless not (a and b) and c
```

Keep in mind that *a* is the same as *not (not a)*. A good advice would be to learn the Boolean Operations, so you don't need getting a headache each time you need to use them.

7.7. Comparisons

First off, it's not the same which class instances you compare. Have you ever noticed that something like *1 == 1* would work out, but something like *Sprite.new == Sprite.new* wouldn't? The reason is as follows: There are two types of objects. The first types are simple to understand. Any instance of a class that can be considered as an original. If you create two sprites and compare them with ==, you will get false as result, since those two objects are not the same one. This usually works for all classes, except the second type. Those classes usually cannot be loaded into variables as instances and each of those classes with the same value are considered equal. As all numbers are instances of the class *Numeric*, *1 == 1* will result in true. Those ones are not originals. Each number will give the result true if compared with another number of the same value. Also an interesting attribute is that *1 == 1.0* would also result in true. Non-Original classes are all *Numeric*, the *String*, *NilClass*, *TrueClass* and *FalseClass* classes. You should also consider that using something like *bitmap == bitmap.clone* will usually result in true as well.
The reason for this is how the classes actually work. Original classes are being created in the RAM and a pointing address is being returned to where in the RAM the instance was

stored. Passing on such variables as arguments into methods will cause that this very class instance is being operated. This is called *call by reference* while the other type is called *call by value* (see ***chapter 8*** for more information). *Call by value* creates copies from the objects when they are passed on as arguments into a method. Modifying such a copy within a method will have no effect on the real class instance. This is why bitmap == bitmap.clone could work out as the clone method only does a so-called *shallow copy*. Shallow copy, which would be the opposite of *deep copy*, only copies the pointers, but not the variables which they point to.
Note that using comparison with an array or hash causes each value to be compared with the other. In other words if the comparison or all elements at the same positions or the same keys results in true, then the arrays or hashs are equal as well.

7.8. Instance Variable Access aka Encapsulation

Have you ever noticed special definitions like *attr_reader*, *attr_writer* and *attr_accessor*? Check out the next two codes.

```
class Hack_Noob
  def initialize
    @tools = Hacker_Tools.new('n00b')
    @log = Data_Log.new('')
    @hacked = []
  end
  def tools
    return @tools
  end
  def log=(data)
    @log = data
  end
  def hacked
    return @hacked
  end
  def hacked=(ary)
    @hacked = ary
  end
end
```

```
class Hack_Expert
  def initialize
    @tools = Hacker_Tools_XP.new('XP')
    @log = Data_Log.new('deny 1337 5p34k')
    @hacked = []
  end
  attr_reader   :tools
  attr_writer   :log
  attr_accessor :hacked
end
```

You might not be able to see the difference here. Well, there actually is none: The codes work exactly the same. The *attr_something* are special methods in RGSS Ruby which allow you to quickly define public instance variables. *attr_reader* will allow reading the variable,

attr_writer will allow writing into the variable and *attr_accessor* will allow accessing the variable which means reading and writing.
But be careful, sometimes encapsulation is necessary. Let's take the *sp*= method from *Game_Battler* as an example.

```
class Game_Battler
  def sp=(sp)
    @sp = [[sp, maxsp].min, 0].max
  end
end
```

So, what does this method do? It defines how an *sp*= call should handled. The value will not bluntly be stored as *@sp*, but it will be corrected first. It will not go under 0 and not above *maxsp* (which is a method as well). Keep in mind that using an *attr_accessor :sp* after the definition of this method will cause it to be rendered obsolete like there has been a redefinition of the method (which technically has been). It is possible to alias a method (see ***2.1.*** for more information) that has been defined over *attr_something*. This code will work out just fine.

```
class God < Heaven
  attr_writer :new_prayer
  alias new_prayer_auto_fulfill_for_good_people new_prayer=
  def new_prayer=(prayer)
    if rating(prayer.person) == 'good person'
      fulfill(prayer)
    elsif rating(prayer.person) == 'bad person'
      @new_prayer = prayer
    end
  end
end
```

7.9. Powerful implemented Iterator Method "*each*"

What would you say if I told you I could speed up the loops in your script by let's say up to 200%? Well, I can. As RGSS Ruby is an interpreted language (see ***1.1.*** for more information), the *for i in something* needs to be interpreted as well which is slower than the compiled *each* methods. The syntax of the *each* iterators can be confusing in the beginning, but as soon as you understand what is what, it's as easy as the *for* iterator. The following *for* iterator definitions are equal to the given *each* iterator definitions below.

```
for i in start..end_inclusive
  ...
end
(start..end_inclusive).each {|i|
  ...
}
```

```
for i in start...end_exclusive
  ...
end
```

```
(start...end_exclusive).each {|i|
  ...
end
```

```
for element in ary
  ...
end
ary.each {|element|
  ...
}
```

```
for i in 0...ary.size
  ...
end
ary.each_index {|i|
  ...
end
```

```
for value in hash.values
  ...
end
hash.each_value {|value|
  ...
}
```

```
for key in hash.keys
  ...
end
hash.each_key {|key|
  ...
end
```

In the examples above you should see that the iterator definition change from *for* to *each* looks like this.

```
for i in something
  code
end
something.each {|i|
  code
}
```

Of course you can use a *do … end* instead of *{ … }* as they are identical.

```
for i in something
  code
end
something.each do |i|
  code
end
```

The explanation of each *each* iterator is quite simple. *each* iterates through all array elements or all numbers of a defined range just as the *for* iterator does.
each_index automatically iterates through all indices of the given array. The most interesting part here is that due to the conversion to a range (*0…ary.size*) in the *for* iterator, *each_index* can be faster up to 200%.
each_value iterates through all values of a hash and is about than 50% faster than *values.each* as the conversion of the hash to an array using *values* does not happen.
Same goes for *each_key*. It is about 50% faster than *keys.each* as the conversion of the mapping data to an array does not happen. So, if you want to speed up your script, exchange *for* iterators with *each* iterators
There are more interesting methods, see the *Enumerable* module in the RMXP Help File.

7.10. Bug Hunter

Debugging a script or program is an art for itself. Did it ever happen to you that you tried to track down a bug through half of all RTP scripts for 2 hours? Well, it has to me.
The first step to repair a bug is obviously to cause one. You first need to find out which are the circumstances of the bug. Does is happen during battle when you try to use a skill? Or does it happen on the map when a text message should be displayed? Or does that option in your CMS just not work? To find a solution for a problem you first need a clear definition of the problem. I will use an example of a weird bug I have been hunting once. Suddenly my character *Lucius* couldn't be inflicted with a status effect anymore. It didn't work with skills and it didn't work directly with events.
The second step is to find the cause of the bug. You have to find out which piece of code doesn't work right and how it conflicts with everything else if it does. This can be a very painful process which could take hours if you use the wrong approach. The worst scenario is that you need half an hour to locate the bug's cause by using a systematic approach. Let's think about it. What makes *Lucius* different from all the other characters? Then it came to my mind: *Lucius* is able to use either two one-handed weapons or one two-handed weapon. Could it have been that I have messed up something when I implemented this system? There was a lot of room for this, since I had aliased quite a number of *Game_Actor* methods to make this system work. But this conclusion is still is of no help. To save my time by avoiding looking for the cause of the problem where isn't, I first made a copy of my *Scripts.rxdata* and removed the two-onehanded/one-twohanded system. Suddenly it worked again. That means this system really is causing the bug. So I've put a couple of controls printings in the *skill_effect* method of *Game_Battler*. I let the script print out *Lucius*' states array a couple of times during the method execution. I noticed that it just didn't change after the call of the *add_state* method. I have also checked a couple of other things like printing out the skill's *plus_state_set*. So I have taken a look at the *add_state* method. But I wasn't able to find anything wrong about it. After about half an hour of more control prints and going trough the whole *Game_Actor* class I finally found the cause. The problematic method was *equip*. Yes, the cause of that nasty problem which I had for quite some time was a method that had almost nothing to do with status effects. The only call related to this area was *update_auto_state* and this was the problem. I can't remember exactly what the problem there was, but if I remember right, it was the built-in RGSS conditioning bug (see ***7.4.*** for more information) that caused me the problem as some simple parenthesis and some special exception handlings fixed it.

The third step is to model a solution. Sometimes it's just a typing mistake, sometimes it's just a little problem that can be fixed by changing a few lines slightly. But sometimes it's a bug that requires remodeling a whole subsystem or method, since the approach you were using was just not working without a bug or if that method is written quite messy, so a bug is almost not to be avoided.
The fourth and last step is to implement the solution and test the script again. Keep in mind that adding new things and editing old ones is the most common time for a bug to appear. Let's say you've found a bug in your minimap script. You fix it by editing a conditional branch. Suddenly you find out there is another bug. What happened? Your edit of the conditional branch wasn't fully thought through and you didn't make it right so another (mostly similar) bug appeared.
Here is the syntax which you need to use to quickly print out some data.

```
p data
p data.inspect
p "#{data1} - #{data2}"
p data1.to_s + ' - ' + data2.to_s
p "#{data1.inspect} - #{data2.inspect}"
p data1.inspect + ' - ' + data2.inspect
```

Note that the first two will give the same print out, the third and fourth will and the last two will. The last two work on the same principle as the first two do. If you use variable embedding into strings or the *to_s* method, it will print out the data often differently than if you use the *inspect* method. I recommend the *inspect* method. Note that you can leave out the *inspect* method if you use only one variable as *p* automatically does the conversion via *inspect*. See the RMXP Help File for more information.
But sometimes using simple print outs doesn't help much. If your system requires is a graphically dependent system (i.e. a passability grid), it might be a good idea to use a sprite with a bitmap of size 640×480 to draw some things on the screen so you can quickly see what's going on in your script. You can take a look at my *Quick Passability Test*, included in *Tons of Add-ons* to get the idea of how it is done. This is usually useful for bug hunting as well as during the development itself.

7.11. Global, Local, Instance Variable or Constant?

First off, you need to know how define each type of variable. Global variables have their variable names beginning with a *$*. Local variable have a first non-capital letter. Instance variables have their names beginning with an @. And finally constants have a capital first letter. What's the difference between all of these? Global variables are visible in every script. Instance variables are a little bit more limited. They are visible within a class, but not outside of it. Though you have the possibility to make them public instance variables by using the three *attr_something* methods (see ***7.8.*** for more information). You can access them from anywhere. Local variables are only visible within the method or block they were created in. They don't exist outside of this method or block (except if they are returned or passed on as argument into another method or block). Constants are visible within the environment they were created in. If you create a constant outside of every class definition, you can access it from anywhere just like a global constant. If you create it within a class definition, you can access it directly if you are within the class or using the *scope operator ::*

(see ***chapter 8*** for more information about *Object Oriented Programming*) by using *NAME_OF_CLASS::NAME_OF_CONSTANT*. Same goes for modules. The values of constants cannot be changed during the execution of the script.
Automatically raises the question when to use each type. Constants are great for constant configurations like i.e. an array of IDs for skills which can attack all enemies and party members at the same time. Global variables are best for some special things like turning on/off a script locally where these changes don't need to be saved (see ***7.2.*** for more information) or for script presence recognition to enhance compatibility (see ***2.2.*** for more information), but they are to be avoided if possible. Local variables are great for temporary data. Instance variables are used for class attributes and special flags where it would be a waste to use global variables.
Another question that raises here is "Why not just use global variables everywhere to save oneself the trouble?" The answer is very simple. It is highly inefficient. A CPU works that way that it only can store a specific number of variables for current work. If every variable was global, the CPU would spend very much time on storing and loading those variables from the RAM which would cause an incredible speed loss (see ***chapter 8*** for more information).

7.12. Inside-Outside or Outside-Inside

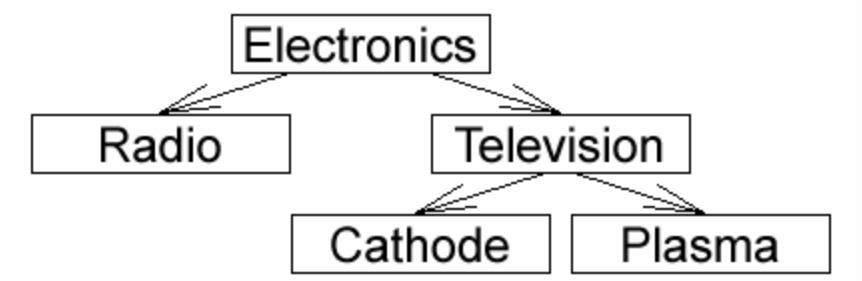
The approach you are using to make your script work can make a big difference. There are two main approaches. By using the first (also called *From the Inside to the Outside*) you are attempting to make all the subsystems work and then move on into combining those components into bigger systems and so on. This approach can be useful as specific subsystems can be tested quite conveniently before they are integrated into the bigger system. The bigger system, then, can be tested quite conveniently as well, since it's safe to say that the subsystems are working. But this approach is only useful for smaller system. In bigger systems it could easily happen that something wasn't considered and one component needs to be rewritten.
In bigger systems it is usual that the *From the Outside to the Inside* approach is used. Using this approach you first define the outer systems like graphical interface. With a working graphic shell everything is easier. Every subsystem can directly be integrated into the big system and makes incompatibility bugs between those two types of system less frequent. The disadvantage of this approach is that the system can be less flexible. A change on the outer system mostly has the consequence that each subsystem needs to be changed as well. All of this increases the chance of a bug. That's why this is suited for systems which have an exact and precise definition: In big systems. If a system is defined and thought through very good, many changes won't be necessary. The disadvantage of this is that usually more time needs to be invested into planning than scripting. It's hard to make it work if you just head-start into scripting. Decide for yourself which approach might be the best for your system.
My personal advice is to use the approach *Inside-Outside* as the only script I had to use the *Outside-Inside* approach was *Blizz-ABS* and the unfinished *Semi-TBS*. For the *Semi-TBS* I had quite a good specification of the outer system. Details came later when I needed them.

7.13. “Uh, what does this knob do?”

Most probably you have often encountered something in the RTP scripts that you didn’t understand quite well. The key in learning things yourself is to experiment. If you are unsure about how a method works and which results it gives, you can either take a look at the RMXP Help File or try experimenting with it yourself. Often you won’t find in the Help File what you are looking for and you will be forced to experiment with it if you want to find out more. Keep in mind that experimentation is a very time consuming learning process, though the quality of the experience you earn through it is of much better quality than if you just copy-paste things. See ***6.6.*** for more high quality script learning experience.

7.14. About Superclasses and Mix-ins

First of all you need to understand the principle of *Superclasses* and *Subclasses*. Let’s take this simple example. Imagine that we create a class, an object that is called *Electronics*. Now you create two subclasses; *Radio* and *Television*. And lastly we create two subclasses of *Television*; *Cathode* and *Plasma*. Now let’s go all the way back. *Plasma* and *Cathode* are both *Televisions*. *Radio* and *Television* are both *Electronics*. Do you see the structure? Both *Radio* and *Television* have the attributes and properties of *Electronics* while both *Cathode* and *Plasma* have the attributes of *Television* (*figure 7.14.1.*).
Electronics is the highest superclass. *Radio* and *Television* are subclasses of *Electronics* and *Television* is the superclass of *Cathode* and *Plasma*. *Cathode* and *Plasma* are subclasses of both *Television* and *Electronics*, only that *Television* is the direct superclass while *Electronics* is a higher superclass.



7.14.1. – superclass inheritance by subclasses

Now, why did we complicate it this way? It’s simple. Let’s say those are the definitions of those classes.

```
class Electronics
  def turn_on
  end
  def turn_off
  end
end
class Radio < Electronics
  def change_volume(new_volume)
  end
  def switch_radiostation(new_station)
  end
```

```
end
class Television < Electronics
  def change_volume(new_volume)
  end
  def switch_channel(new_channel)
  end
end
class Cathode < Television
  def display_on_cathode_pipe
  end
end
class Plasma < Television
  def display_on_plasma_lcd
  end
end
```

Note how both *Radio* and *Television* have *change_volume* defined. This is because of two reasons. The first would be what if we made a *Hover* class? Well, *Hover* would be a subclass of *Electronics*, but you can't really change the volume of a *Hover*, can you? Keep in mind, that if we would have only used *Radio* and *Television*, this would have worked out though. So, to keep your code shorter you can simply define the *change_volume* in the *Electronics* class, it's no big deal. It's better if you make your code shorter by avoiding the definition of a method more than once even though the original method isn't used by all subclasses or the superclass itself. Now the interesting thing is that all instances of *Cathode* and *Plasma* can call the methods already defined in *Television* and *Electronics*. The same way *Television* and *Radio* can call any method defined in *Electronics*. They also have all the instance variables from their superclasses available.
The next thing to mention is something that's called method overloading (or overriding). Let's say, we redefine a few methods in *Plasma*.

```
class Plasma
  def turn_on
    super
    change_volume(100)
  end
  def turn_off(flag)
    super()
    if flag
      change_volume(0)
    end
  end
  def switch_channel(new_channel)
    change_volume(0)
    super
    change_volume(100)
  end
end
```

Note how it's not necessary to write *class Plasma < Television* as it was already defined, though nothing special will happen if you do it again.
Now, what's happening here? *turn_on* will first turn the *Television* on and then change the volume to 100 if it's used on a *Plasma* TV. The *super* calls the method of the superclass. In

other words you can simply enhance already existing methods similar like aliasing (see ***2.1.*** for more information). Of course you don't have to call the superclass' method. You can just redefine the method and that's it.
There's only one thing you need to keep in mind. Take a look at the redefinitions of *turn_off* and *switch_channel*. The *turn_off* method from *Plasma* needs an argument passed on. Since the superclass' method doesn't have an argument specified you necessarily have to specify that no argument is passed on by using *super()*. If you leave out the () brackets like in the redefinition of *switch_channel* then all arguments are passed on the superclass' method as is.
The last thing to mention in this are would be *mix-in modules*. Those are normal modules defined like modules are defined, though most of the methods are defined as private (without the *self* in *def self.something*), If this module is included in a class it can use it methods. Let's create a couple of new classes.

```
class Person
  def initialize(gender)
    @gender = gender
  end
end
class John < Person
  def initialize
    super('male')
  end
end
class Mike < Person
  def initialize
    super('male')
  end
end
class Cindy < Person
  def initialize
    super('female')
  end
end
```

Now why should we use modules for this? We can define everything we need over classes and superclasses. That's not quite correct. There is a little exception where we just can't do that. Let's say that *Mike* is a *Painter*, *John* a *Musician* and *Cindy* is their teacher, so she's both *Painter* and *Musician*. You see the problem incoming here? It would be hard to define methods for *Painter*, *Musician* and both if they were just classes. So let's use the mix-in module approach.

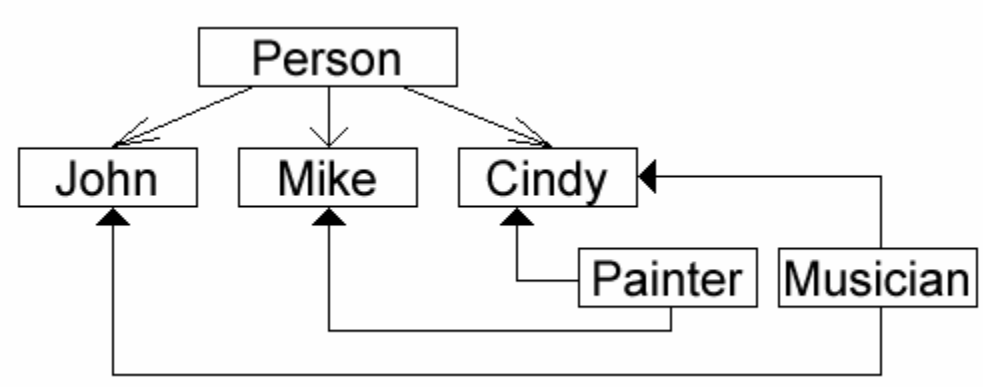
```
module Painter
  def mix_colors
  end
  def draw_picture
  end
end
module Musician
  def choose instrument
  end
```

```
  def read_notes
  end
  def play_music
  end
end
```

And how can we now make *John* a Musician, *Mike* a *Painter* and *Cindy* both? We just *include* the necessary module (*figure 7.14.2.*).

```
class John
  include Musician
end
class Mike < Person
  include Painter
end
class Cindy < Person
  include Painter
  include Musician
end
```

The last thing that needs to be taken care of is method overloading. It's pretty simple. The last definition has the highest priority. Modules have priority over superclasses and classes have priority over modules.
This approach is not recommended as saving class instances will cause the included module to be saved as well and will give an error when you try to load a save file if the module definition was removed from the scripts. I only mentioned this method to make you familiar with it and show you that something like this is possible, not that you go out there and use it. It can only do heavy damage to compatibility and is rarely unavoidable. It's better to redefine the class itself.



7.14.2. – how mix-ins work

7.15. NFS – Need for Sorting

Have you ever tried to make a sorting method for *Array* in RGSS? There is no need for it as there are two compiled methods. One if *sort* and the other is *sort!*. They do both about the same thing, only that *sort!* sorts the array destructively which means that the array itself will be modified while *sort* only returns a sorted array without changing the original. If the array

consists out of numeric values, they will be sorted ASCDESC#####. You can simple reverse the order if you use the method *reverse* or *reverse!* on the array. Keep in mind that methods that end with an *!*, execute everything destructively. You cannot only sort numeric array, you can sort any kind of data by specifying a block where you can define a comparison conditions and results. Let's say we want to sort the party members descending by agility. If the agility is equal, we want it to sort ascending by strength. If the strength is equal, we want it to sort ascending by name and if even the names are equal then descending by ID number.

```
$game_party.actors.sort! {|a, b|
  if a.agi > b.agi
    -1
  elsif a.agi < b.agi
    +1
  elsif a.str > b.str
    +1
  elsif a.str < b.str
    -1
  elsif a.name > b.name
    +1
  elsif a.name < b.name
    -1
  else
    a.id <=> b.id
  end
}
```

Each actor get compared to another where *a* is one actor and *b* the other. If the agility of actor *a* is greater than the agility of actor *b* then the result is -1 which means *a* will get shifted more to the left. In the opposite case it's +1 (you can leave out the + sign). Then the same comparison is down for strength, only this time the strength comparison results' signs are switched. This will cause the one with less strength to be more left in the array. Same goes for name where the *smaller* name is defined by comparison of strings. i.e. the string "coffee" is smaller than "dog" as the first letters are c and d and c is smaller than d. If the first letter is the same, then the next letter is taken into account. If both words are the same up to one point, then the shorter string is considered as smaller. Don't get confused with the <=> operator. The next three codes are equivalent in the purpose of comparison.

```
a <=> b
```

```
a > b ? +1 : (a < b ? -1 : 0)
```

```
(a - b) / (a - b).abs
```

```
a - b
```

This knowledge can sometimes be very useful, so it's a good idea to memorize it.

7.16. CPU or RAM?

That’s it. You’ve tasted a bit of experience that awaits you if you continue progressing as scripter. Always try to keep the CPU requirement low and RAM usage to a minimum. If you can’t do one of those, decide which is more important in this case.

8. Useful Links

Since this e-book explains just a few basics, here are several links for those who are eager to learn more.

Ruby
http://en.wikipedia.org/wiki/Ruby_(programming_language)
Stack
http://en.wikipedia.org/wiki/Stack_(data_structure)
Recursion
http://en.wikipedia.org/wiki/Recursion_(computer_science)
Interpreter
http://en.wikipedia.org/wiki/Interpreter_(computing)
Compiler
http://en.wikipedia.org/wiki/Compiler
Machine Language
http://en.wikipedia.org/wiki/Machine_language
CPU
http://en.wikipedia.org/wiki/Central_processing_unit
http://en.wikipedia.org/wiki/CPU_design
Object-oriented programming
http://en.wikipedia.org/wiki/Object-oriented_programming
RAM
http://en.wikipedia.org/wiki/Random_access_memory
Array
http://en.wikipedia.org/wiki/Array
Hash
http://en.wikipedia.org/wiki/Hash_table
HUD
http://en.wikipedia.org/wiki/HUD_(computer_gaming)
Algorithm complexity
http://en.wikipedia.org/wiki/Complexity
http://en.wikipedia.org/wiki/Computational_complexity_theory
Logarithm
http://en.wikipedia.org/wiki/Logarithm
Boolean Algebra
http://en.wikipedia.org/wiki/Boolean_algebra_(structure)
Call by Value/Call by Reference
http://en.wikipedia.org/wiki/Evaluation_strategy

9. Summary

Don't compare yourself with other scripters. You might lose your motivation by thinking "I will never get as good as him…" On the other hand you could also DO compare yourself with other scripters. You might be the type of person who says "I want to do better. I can do better. I will do better." instead. It's up to you. None of the good and known scripters today was just born with his skills. He has earned them through work and experience. This e-book is a part of my knowledge as a gift to all of you. It's not only what I have learned from scripting in RGSS, it's also what I have learned from programming in general, from what I have been taught in college and from life itself. Every scripter is unique and his experience and skills are incomparable with the experience and skills of any other scripter. One day you will just realize "Hey, I became better than that other guy..."

The RPG Maker XP Help File is always a big help when you get into a dilemma asking yourself "Can I do that?" or "Does that class support that?" or... That's why I have included an enhanced English Help File with an extra chapter by an unknown author and the Scripts.rxdata with English commented code, improved iteration coding, recoded to avoid the built-in RGSS bug with conditioning and overworked code to shorten the and improve the code. Many of you might not have a legal copy of RPG Maker XP and therefore not the English files. **But this does not mean that I encourage illegal copies! I only give support regardless of the fact that your copy is legal or not! If you like the RMXP editor and engine, buy it!** Reading through the Help File once completely will help you a lot. You will find many useful methods that can give you results, instead of using your own code for that. I also included a copy of the RGSS102E.dll

By reading this e-book I hope right now you got the itches to make something more than just a CMS. And even if it's a CMS, I hope its moving windows that move from behind a window to the side and then back over the other window will look cool, since they don't lag at all.