Introdução

Olá! Bem vindo a Bíblia do RGSS! Aqui você vai aprender muita coisa sobre a linguagem. Gostaria de agradecer a você que mostrou interesse para aprender e baixou este tutorial, ele foi feito com muita vontade para que muitos makers possam aprender esta grande linguagem de progamação!

Muitos sabem que há um tempo, o RGSS era uma linguagem que poucos dominavam, mas hoje as coisas mudaram, já existem muitos tutoriais sobre o assunto pela internet. Espero que este tutorial seja definitivo para sua aprendizagem, pois deu muito trabalho organizar os documentos para reuni-los aqui. Fique atento para a próxima atualização!

É importante dizer que este tutorial é comprometido para aqueles que têm uma noção mais avançada do RGSS, ou de nível básico-intermediário, mas não impede que seja estudado por iniciantes. Um tutorial voltado aos que não sabem nada será iniciado ainda.

As pessoas que participaram deste documento são: Jão e DarkChocobo. Cada um cedeu seus materiais para que esta grande coletânea fosse feita. Desejo que isto sirva para você aprender bem mais!

Vale lembrar que este tutorial não está completo, pegamos tudo que havíamos postado e juntamos, há certas aulas de classes que não precisam ser mostrados por serem pequenas e não tão importantes, para ver todo o conteúdo restante do RGSS, é só acessar o arquivo de ajuda já anexado.

Aula 1 – Considerações Iniciais

Índice de Tópicos:

Tópico 1 – O que é Ruby?

Ruby é uma linguagem de programação interpretada, com tipagem dinâmica e forte, orientada a objetos com vastas semelhanças com Perl, SmallTalk e Python.
Projetada tanto para a programação em grande escala quanto para codificação rápida, tem um suporte a orientação a objetos simples e prático. A linguagem foi criada pelo japonês Yukihiro Matsumoto, que aproveitou as melhores idéias das outras linguagens da época. Esta linguagem possui vastos repositórios de bibliotecas disponíveis em sites como Ruby Forge e Ruby Application Archive (RAA). Existe, ainda, uma ferramenta bastante útil para instalação de bibliotecas, chamada Ruby Gems, o software mais famoso desenvolvido em Ruby é o Ruby on Rails

Tópico 2 – O que é RGSS?

RGSS (Ruby Game Scripting System) é uma biblioteca que usa a Linguagem de Script Ruby Orientada a objetos para desenvolver jogos 2D para a Plataforma Windows. RGSS leva você a construir inteiramente sistemas de jogos com originalidade mais facilmente que outras linguagens. Ruby é uma linguagem simples e de fácil aprendizado para os iniciantes, sendo esta uma poderosa e sofisticada ferramenta.

Tópico 3 – Sintaxe

Considerações Iniciais
Ruby é uma linguagem que diferencia letras maiúsculas das minúsculas. Diferente de outras linguagens, em Ruby você pode usar comentários e espaços em qualquer local sempre que necessário. Quebras de linhas (quando aperta enter) podem ser usadas como espaços desde que esteja claro que o comando ainda não acabou, senão, elas serão consideradas uma finalização do comando.

Identificadores
Identificadores no Ruby podem ser comparados a 'Comandos dos Eventos', eles representam cada ação do script e fazem com que ele funcione. Identificadores são representados por qualquer palavra desde que comecem com uma letra ou um underline(_).

Exemplo:
```
Ruby_é_simples
```

Comentários
Todo e qualquer comentário será totalmente ignorado, fazendo com que sirvam apenas para orientação. Comentários podem ser chamados de duas formas:

1 – Desde que o primeiro dígito da linha seja um ‘jogo da velha’(#) toda a linha será um comentário.

Exemplo:

```
# Este é um comentário de linha.
```

2 - É utilizado o comando ‘=begin’ para representar o início de um comentário, e é utilizado o comando ‘=end’ para representar o fim do mesmo.

Exemplo:

```
=begin
Este tipo de comentário pode durar quantas linhas quiser.
Ele é muito utilizado para instrução de uso dos scripts.
=end
```

OBS: O editor de scripts do RPG Maker XP e VX utiliza uma tonalidade verde de cor para representar comentários.

Palavras Reservadas

As palavras reservadas não podem ser utilizadas para nomear classes, variáveis, entre outros. Entretanto, variáveis com prefixo $ ou @(serão vistos mais adiante) não são consideradas reservadas.

As palavras reservadas estão listadas abaixo:

```
BEGIN    class    ensure   nil      self     when
END      def      false    not      super    while
alias    defined? for      or       then     yield
and      do       if       redo     true
begin    else     in       rescue   undef
break    elsif    module   retry    unless
case     end      next     return   until
```

OBS: Não é necessário decorar todas as palavras reservadas, pois elas irão aparecer no editor em uma tonalidade de cor diferenciada das demais.

Aula 2 – As Variáveis e Suas Funções

Índice de Tópicos:

Tópico 1 – O que são e para que servem as Variáveis?

As variáveis carregam dados necessários para o funcionamento correto do jogo. As variáveis podem ser consideradas containers de dados, elas ficaram guardando seus dados até que você queira utilizá-los. Em Ruby, as variáveis têm o mesmo propósito das variáveis usadas pelos eventos do RPGMaker, guardar dados para que eles possam ser usados futuramente para um funcionamento correto do jogo, apesar de não serem usadas de mesma forma.

Tópico 2 – Os Tipos de Variáveis

Em Ruby, existem vários tipos diferentes de variáveis, e qualquer uma dela pode ter vários tipos de valores diferentes. Existem 4 tipos diferentes de variáveis, são elas: Variável Global, Variável de Instância, Variável Local, e Constantes.

Variáveis Globais

As variáveis globais são aquelas que poderão ser utilizadas por qualquer parte do programa (incluindo até mesmo pelos eventos). As variáveis globais são identificadas por conter um prefixo "$" em seu nome.

Exemplo:

```
$global_variable          # Esta é uma Variável Global.
```

Variáveis de Instância

As variáveis de instância pertencem a um específico objeto , elas podem ser acessadas apenas pela classe a qual elas pertencem (classes serão vistas na próxima aula). As variáveis de Instância são identificadas por conter um prefixo "@" em seu nome.

Exemplo:

```
@instance_variables          # Esta é uma Variável de Instância.
```

Variáveis Locais

As variáveis locais são aquelas que podem ser usadas apenas no método a qual ela pertence (métodos serão vistos na próxima classe). As variáveis locais não contém qualquer prefixo em seu nome, mas devem ser iniciadas por uma letra minúscula ou um underline(_).

Exemplo:

```
local_variable          # Esta é uma variável local.
```

Constantes

Constantes são variáveis que pertencem a classes, classes que incluem modules e modules (serão vistos em aulas futuras). Constantes são identificadas por começarem seu nome com uma letra maiúscula.

```
Constante          # Esta é uma Constante.
CONSTANTE          # Constantes também podem ter seu nome formado por
letras maiúsculas.
```

Para chamar uma constante a partir de um objeto externo, utilize "::"

```
Module::Constante
```

OBS: Você pode não ter entendido este exemplo, pois modules serão visto somente em aulas futuras.

Tópico 3 – Valores das Variáveis

As variáveis podem ter vários tipos diferentes de valores, cada tipo de variável pode conter qualquer tipo de valor desejado. Os possíveis tipos de valores que as variáveis podem conter são: Valores Numéricos, Strings, Expressões Regulares, Arrays, Hashes, Ranges, Símbolos e Pseudo-variáveis.

Valores Numéricos

Valores Numéricos são aquelas variáveis que representam números, exatamente igual às variáveis do RPG Maker, elas podem representar números inteiros, decimais e até mesmo negativos. É importante lembrar que as variáveis do RPG Maker são alojadas numa variável global pelo RGSS, por isso que é possível a mudança do valor das variáveis através dos scripts, logo, tudo que o comando "Alterar Variáveis" dos eventos faz é mudar o valor de uma variável global (algo que também poderia ser feito pelo comando "Chamar Script", já que tudo que ele faz é alterar o valor de uma variável).

Exemplo:

```
0                # Este é um valor Numérico dado às variáveis.
16               # Este é um valor Numérico dado às variáveis.
-42               # Este é um valor Numérico dado às variáveis.
```

OBS: O Ruby ignora qualquer underline (_) que esteja no meio de um valor numérico, isso pode ser útil para separar valores grandes, como no exemplo a seguir:

```
1_000_000_000          # Ambas as formas de escrever
1000000000             # resultarão em um valor de um bilhão.
```

Strings

Strings são nada mais, nada menos que textos (expressões literárias). As sctrings podem ser usadas colocando-se aspas ou aspas-duplas (' e ") entre o texto.

Exemplo:

```
'Esta é uma String'
"Esta também é uma string"
```

A diferença de usar aspas ou aspas-duplas é que as aspas-duplas pode ser utilizados códigos dentro das strings, já textos em aspas comuns irão mostrar exatamente o que está dentro das aspas.

Exemplo:

```
sabor = "doce."
'Suco #{valor}'          # O programa irá interpretar desta maneira:
Suco #{sabor}
"Suco #{valor}"          # O programa irá interpretar desta maneira:
Suco doce.
```

OBS: Em aulas futuras irei explicar o exemplo acima, só quis explicar que ao usar aspas-duplas, o Ruby interpreta um texto com comandos dentro.

Quando dois Strings estiverem separados por um espaço em branco, o Ruby interpretar como uma só string.

Exemplo:

```
 'Suco' 'doce.'          # O programa irá interpretar desta maneira:
'Suco doce.'
```

Expressões Regulares

Qualquer texto representado dentro de barras (/) são consideradas Expreções Regulares. Em RGSS as expreções regulares não são muito utilizadas, por tanto irei explicar este tipo de valor somente em aulas futuras.

Exemplo:

```
/my name is dark chocobo/
```

Array

Array é um tipo de valor muito intereçante, pois ela em si não tem qualquer valor, na verdade as Array servem para guardar vários valores em uma única variável, incluindo outras Arrays. As Arrays podem guardar uma quantidade ilimitada de valores em uma única variável, colocando estes valores numa ordem específica destro de chaves, cada valor é separado por uma vírgula.

OBS: É importante lembrar, que cada valor da array pode conter qualquer tipo diferente de valor, incluindo outras Arrays.

Exemplo:

```
 [[0, 16, -42]
['Esta é uma String', "Esta também é uma string", 'Suco' 'doce.']
[16, "Suco doce.", -42, 1_000_000_000]
# Todas estas acima são Arrays.
# Note como cada valor é separado por vírgula independente do tipo de
valor.
```

Como os valores da Array ficam em grupo, cada um deles tem um ID (é o número de sua ordem na lista, começando do 0)

Exemplo:

```
[0, 16, -42]
# Nesta Array, 0 (zero) tem ID 0, pois é o primeiro da lista.
```

```
# 16 tem ID 1, pois é o segundo.
# -42 tem ID 2, pois é o terceiro.
# Agora vamos para uma outra Array:
[16, "Suco doce.", -42, 1_000_000_000]
# Nesta Array, 16 tem ID 0, pois é o primeiro da lista.
# "Suco doce." tem ID 1
# -42 tem ID 2
# 1_000_000_000 tem ID 3
```

Exemplo:

```
 [4, [0, 16, -42], 'Suco doce.', 1_000_000_000]
# Preste bem atenção no valor de ID 1 desta Array.
# O valor de ID 1 é: [0, 16, -42]
# É uma Array dentro da própria Array.
# Neste caso ela funcionará do mesmo jeito, sem qualquer problema.
```

OBS: Note que, como as Arrays não têm limite de tamanho, uma Array pode conter quantas Arrays você quiser dentro dela, e estas Arrays podem, por sua vez, ter outras Arrays dentro de si e assim por diante (apesar de isso não ser nem um pouquinho comum de encontrar)

Nota Final: Ainda há mais o que falar sobre Arrays, mas para entende-la melhor devemos primeiro concluir nossos estudos sobre variáveis, por tanto, vamos adiante.

Hash

Hashes são muito parecidas com as Arrays pelo fato delas também guardarem vários valores dentro de si, porém, as Hashes, diferente das Arrays, tão têm uma ordem fixa por IDs, Porém as Hashes são mais complexas de serem compreendidas, então vamos passar para o próximo valor e explicarei hashes em aulas futuras.

Range

Range são operadores utilizados entre dois valores para determinar um novo valor que seria todos os valores entre os dois valores determinados. Simplificando, uma range é todo o valor entre um início e um fim especificado. Uma range é identificado por reticências (...) ou dois pontos finais (..), no caso da reticências, a range terá o valor inicial até o final, mas nos dois pontos duplos a range terá o valor somente dos valores entre os dois especificados. (Eu sei que está meio difícil para entender, mas tudo ficará mais simples após o exemplo)

Exemplo:

```
1 .. 20   # Neste caso, a range terá todo o valor de 1 à 20
1 ... 20   # Neste caso, a range terá o valor ENTRE 1 e 20 (de 2 à 19)
```

OBS: As ranges não são usadas como as outras variáveis comuns, são criadas especificamente para o comando for do Ruby. (Este, será explicado no próximo tópico desta aula.)

Símbolos

Símbolos são o único tipo de valor com que vocês não devem ter qualquer tipo de preocupação, eles são instâncias de uma classe interna do RGSS e são os únicos objetos (do RGSS) que irão voltar exatamente da forma que são sempre que forem chamados.

Exemplo:

```
:class
:lvar
:method
:$gvar
:@ivar
:+
```

Pseudo-Variáveis

Diferentes dos valores vistos acima, estes são valores especiais conhecidas como Pseudo-Variáveis. Pseudo-Variáveis não podem ser alteradas. As Pseudo-Variáveis são: self, nil, true e false.

Self – Self é o próprio objeto, exemplo: as variáveis também podem representar classes (mas isso será visto na próxima aula), então quando você escreve 'self' dentro desta classe, está se referindo à variável que está representando esta classe. Não irei me aprofundar neste tipo de valor, pois para isso, primeiro você deve saber sobre classes, que só serão dadas na próxima aula.

Nil – Sempre que uma variável não declarada for chamada, ela será considerada Nil, logo, nil pode ser considerado como um NADA, como se a variável não tivesse nenhum valor (nem zero).

True/False – São valores especiais que significam Verdadeiro (true) ou Falso (false), são usados em ocasiões especiais, como em condições para definir coisas como, por exemplo, se uma janela de comando estiver ativa ou não. Estes valores são utilizados em uma variável global para representar as Switches do RPG Maker (true = ON, false = OFF), logo, tudo que o comando "Alterar Switches" dos eventos faz eh alterar o valor de uma variável global de true para false(e vice-versa) do mesmo jeito que o comando "Alterar Variáveis" já citado acima.

Tópico 4 – Trabalhando com as Variáveis

Como você já sabe, as variáveis servem para alojar valores de diversas formas, esta parte da aula, lhe ensinará como criar as variáveis e alterar seus valores.

As variáveis representam valores que podemos armazenar para usá-los em todas as coisas. Para atribuir propriedades, condições, etc.

Antes de mais nada devemos declarar uma variável. Declarar, em Ruby, é uma expreção dada quando eu crio uma Variável. Para criar uma Variável simplesmente equaliza-a a um valor desejado, você pode dar qualquer nome e qualquer valor a Variável que estiver criando, dês de que siga as regras citadas para cada tipo de variável no começo da aula.

Exemplo:

```
variável = 5
```

Com isso, eu acabo de criar uma variável local com nome de ‘variável’, e seu valor é 5. Que tal começar a colocar as aulas em prática? Crie um novo script em seu jogo acima de todos os outros scripts, e cole o código já dito acima (variável = 5). Na linha abaixo coloque o comando:

```
p variável
```

O comando ‘p’, criará uma janela padrão do Windows no jogo (que aparecerá logo ao iniciar o jogo, pois está no primeiro script) exibindo o valor de variável. Então dê OK no editor e teste o jogo, veja como apareceu o valor citado na variável 5.

Agora apague o ‘p variável’, vamos criar uma nova variável. Coloque o seguinte código:

```
teste = variável + 2
p teste
```

Agora dê um teste no jogo. Apareceu o número 6, pois se ‘variável’ tem o valor de ‘5’, e ‘teste = variável + 2’, então ‘teste = 7’. É simplesmente uma equação matemática de primeiro grau.

Agora, apague o último código citado, vamos alterar o valor de ‘variável’ sem precisar criar uma segunda variável. Logo abaixo de ‘variável = 5’, coloque:

```
variável += 2
```

Com isso, o valor de variável, será adicionado em 2, este comando também pode ser substituído por ‘variável = variável + 2’, pois em Ruby existe sempre várias formas de fazer cada coisa que você quiser, apenas optamos pelo caminho mais simples e curto.

Abaixo segue uma lista dos comandos matemáticos que o Ruby consegue ler:

```
+, -, *, /, %, **, &, |, ^, <<, >>, &&, ||
```

OBS: Em outras aulas serão explicados o significado de cada operador, por hora, somente os que realmente importam são: +, -, * e /; que representam respectivamente: soma, subtração, multiplicação e divisão.

Condições

As condições do Ruby são geralmente usas juntamente com variáveis de todos os tipos. As condições do Ruby são utilizadas da mesma forma que as condições dos eventos do RPG Maker, porém com mais especificações e possibilidades.

Em Ruby existem quatro formas diferentes de criar condições, são elas: Condição Completa, Condição de uma Linha, Condição Rápida e Condição de um Comando. OBS: Na verdade, as quatro formas de condições não têm nomes específicos, mas fica mais fácil ensinar, dando um nome a cada uma.

Condição Completa: É a mais simples de fazer, e mais comum, apesar de ser a maior. Ela consiste no uso da palavra ‘if’ no começo da estrutura, e logo depois da condição.

Exemplo:

```
if variável == 5
  p 'sim'
end
```

Agora vamos explicar o significado desta expreção:
Na primeira linha, a palavra ‘if’ representa o início de uma condição, e ‘variável == 5’ representa a condição dada para que a próxima linha aconteça, note que é utilizado dois ‘=’ já que é uma condição, pelo contrário, o programa iria pensar que você estava querendo mudar o valor de ‘variável’ para 5.

A segunda linha é o que irá acontecer no caso de variável ser 5, no caso, irá aparecer uma janela padrão do Windows com a mensagem ‘sim’.
A palavra ‘end’, na terceira linha, representa o final da estrutura, caso a condição de ‘variável’ ser 5, não seja cumprida (se ‘variável’ não for 5) o programa irá pular a linha ‘p ‘sim’’ e irá direto para o ‘end’, que é o final da estrutura.
Cole isto logo abaixo daquele antigo comando ‘variável = 5’ e dê um teste para ver o que acontece. Agora mude o ‘variável = 5’ para ‘variável = 3’ e dê um teste. Como variável não é 5, o programa não mostrou a mensagem sim, pois a condição de variável ser 5 não foi cumprida.
Agora apague a condição criada (o último código citado) e cole esta nova condição abaixo:

```
if variável == 5
  p 'sim'
else
  p 'não'
end
```

Agora altere o valor de variável para 5 e teste, depois para outro valor que não seja 5, e teste. ‘else’ é uma palavra usada para representar uma exceção à condição da estrutura. Agora retire o ‘else’ e coloque no lugar isto: ‘elsif variável == 3’.
Agora você está disendo ao programa que se variável for 5, a mensagem será ‘sim’, “mas se” for 3, a mensagem será ‘não’. Porém, como você não especificou uma exceção para suas duas condições, o programa pulará direto pro ‘end’ no caso de variável não for 3, nem 5. Você pode usar quantos ‘elsif’ você quiser por exceção, mas else, somente um, ou nenhum; e será sempre obrigatório um ‘end’ em casa condição.
Você também pode dar mais do que uma condição por estrutura com as palavras ‘and’ e ‘or’.

Exemplo:

```
if variável == 5 or variável == 3
  p 'sim'
end
```

Neste caso, se variável for 5 ou 3, a mensagem ‘sim’ será exibida.

Exemplo:

```
if variável == 5 and teste == 3
  p 'sim'
end
```

Neste caso eu usei duas variáveis para a condição, a mensagem ‘sim’ será exibida se ‘variável’ for 5, e ‘teste’ for 3. Lembre-se de que você deve sempre declarar a variável antes de usá-la (teste = 3), caso contrário acontecerá um erro, pois o programa não conseguirá encontrar a variável.
Você pode também utilizar outros comandos ao invés de ‘==’, os mais comuns e usados estão listados abaixo:

```
==   # igual à
>    # maior que
<    # menor que
>=   # maior ou igual que
<=   # menor ou igual que
!=   # diferente de
```

Aula 3 – Classes, Módulos, Métodos e Herança

Índice de Tópicos:

Tópico 1 – Classes

Como foi visto na aula 01, Ruby é uma linguágem interpretada por objetos, as classes podem ser consideradas estes objetos. Para os iniciantes eu costumo diser que as classes são o que representam os próprios scripts, logo, cada classe representa um script, mas que só funcionam corretamente quando usados juntos.
Da mesma forma que as variáveis, é necessário declarar uma classe antes de poder utiliza-la. Você declara uma classe quando utiliza a palavra 'class' seguido do nome da classe(separado por um espaço). O nome da classe deve começar com uma letra maiúscula, e é necessário o uso da palavra 'end' após a declaração da classe para determinar o término da classe.

Exemplo:
```
class Nome_da_classe
end
```

Ao declarar uma classe já declarada você poderá alterar a classe sem apaga-lá.

Tópico 1.1 – Representação de Classes por Variáveis

Algo não especificado na aula 02 foi que as variáveis também podem representar as classes como um valor, da mesma forma que strings, valores numéricos, arrays, etc..
Para declarar variáveis com valor de uma classe você deve equalizá-la ao nome da classe seguida de um método da superclasse, o '.new'.

Exemplo:
```
$game_temp = Game_Temp.new          # Trecho retirado da classe
Scene_Tile, linha 115.
```

OBS: Método e Superclasses serão vistos ainda nesta aula, portanto não se preocupem em entender isto agora, apenas quero explicar como representar uma classe por variáveis, pois isto será importante para o entendimento dos tópicos 2 e 3 desta aula.

Tópico 2 – Módulos

Módulos são objetos muito semelhantes às classes, a única diferença, é basicamente o modo de ser usada. Os módulos também devem ser declarados antes de seu uso, e seguem as mesmas regras que as classes, para serem declaradas, exceto pelo uso da palavra 'module', ao invés de 'class'.

Exemplo:

```
module Nome_do_módulo
end
```

Ao declarar um módulo já declarado você poderá alterar o módulo sem apagá-lo.

OBS: Os módulos não podem ser representados por variáveis pela forma com que são utilizados (será visto mais tarde)

Tópico 2.1 - Incluindo Módulos Classes

Módulos podem ser incluídos nas classes escrevendo, dentro das classes, a palavra include e o nome do módulo desejado, separados por um espaço. Ao incluir um módulo á uma classe, a classe herdará todos os métodos e variáveis de instância contidas no módulo. Variáveis e métodos já declarados serão apagados e substituídos pelos métodos e variáveis do módulo.

Exemplo:

```
include Math          # Math é o nome de um módulo.
```

OBS: Se o método do módulo a ser incluído já existir por meio de uma herança de uma superclasse, o método do módulo terá prioridade, logo, o método da superclasse será excluído.

Tópico 3 – Métodos

Os Métodos é o que fazem as classes e módulos funcionarem, sem eles, nada aconteceria. Os métodos podem ser comparados com os comandos dos eventos do RPGMaker, pôs é a través deles que tudo funciona, cada método tem uma função específica, como por exemplo: verificar se o jogador apertou alguma tecla, atualizar os valores das janelas a cada frame(frame é o tipo de contagem de tempo utilizado pelo RPGMaker), mudar valores de variáveis, etc..

OBS: Cada comando dos eventos do RPGMaker chama um método para realizar uma ação desejada.

Os métodos são, geralmente, declarados dentro das classes e módulos, não existe nenhum método declarado fora de classes e métodos nos scripts padrões do RPGMakerXP & VX, apesar de ser possível declará-los fora das classes e métodos.

Para declarar os métodos é utilizado a palavra ‘def’ e logo depois o nome do método(separado por um espaço), e um ‘end’ para determinar o final do método.

Exemplo:
```
def nome_do_método
end
```

Cada método só poderá ser chamado a partir da classe ou módulo a qual ele pertence, se ele foi declarado de fora de uma classe e módulo, ele poderá ser chamado de qualquer lugar, incluindo pelos eventos. (porém, existe outra forma mais utilizada para chamar métodos por eventos)

Chamando Métodos

Os métodos poderão ser chamados simplesmente por escrever o nome do método desejado.

Exemplo:
```
# Esté é um método
def nome_do_método
end

# Para chamar este método se utiliza o comando:
nome_do_método
```

Métodos carentes de valores

Alguns métodos necessitam de determinados valores para seu correto funcionamento, estes valores devem ser determinados ao chamar o método. Para determinar os valores basta colocá-los dentro de parênteses logo após o nome do método, separados por vírgulas no caso de mais de um valor; os valores necessitados também devem estar especificados quando declarado o método.

Exemplo:
```
# Esté é um método
def nome_do_método(valor)
end

# Para chamar este método se utiliza o comando:
nome_do_método # Neste caso ocorrerá um erro, pois não foi
especificado o valor
nome_do_método(5) # Neste caso, o método será chamado com o valor 5
```

Note no exemplo acima, que foi escrito ‘valor’ dentro dos parênteses, assim, será declarada uma variável chamada ‘valor’, dentro deste método. No caso acima ‘valor’ será 5, já que o método foi chamado juntamente com o número 5.

Outro Exemplo:
```
# Esté é um método
def nome_do_método(valor1, valor2, valor3)
end

# Para chamar este método se utiliza o comando:
nome_do_método(16, “String”, [4, 22]) # Note que qualquer tipo de
valor pode ser utilizado.
```

Para encerrar o assunto, irei mostrar um método retirado de uma classe do RGSS.

Exemplo:

```
# Método 'remove_actor' da classe Game_Party.
# Este método é utilizado para remover um herói do grupo.
# Ele é chamado sempre que nós utilizamos o comando
# mudar membro nos eventos e escolhemos expulsar um membro do grupo.
# note a variável 'actor_id' na declaração do método
  def remove_actor(actor_id)

    @actors.delete(actor_id) # Este comando exclui o membro do
                                                  # grupo de ID
definido ao chamar o método.

    $game_player.refresh    # Este comando será explicado nesta mesma
aula.

end  # Este 'end' determina que acaba aqui o método, ele já fez tudo
que tinha que fazer.
```

OBS: Ao declarar um método já declarado, o novo método irá substituir o já declarado, a não ser que seja utilizado o comando 'alias'. (será visto ainda nesta aula)

Tópico 3.1 – Utilização de Métodos a partir de um Objeto Externo

Agora que já sabemos o que são classes, módulos e métodos; irei explicar como eles funcionam, pois as classes e módulos, como já foi explicado, não são nada sem os métodos.

Chamando métodos de módulos.

Como já foi explicado um método é chamado somente por escrever-mos seu nome, mas para isto o método deve ser chamado da mesma classe em que foi declarado, exemplo, eu não posso chamar o método 'gain_gold'(da classe Game_Player) utilizando seu nome na classe Game_Actor, ocorrerá um erro de 'método não declarado' (undefined method). Porém, esta regra limita-se às classes, os módulos podem ter seus métodos chamados por escrever o nome do módulo, seguido do método desejado. (separados por um ponto)

Exemplo:

```
Graphics.transition(10) # Trecho retirado da classe Scene_Base, linha
36.
```

'Graphics', é o nome de um módulo, 'transition' é o nome de um método do módulo Graphics, e 10 é o valor requerido pelo método 'transition'.

Chamando métodos de classes.

Existe uma forma muito utilizada para chamar métodos das classes, a partir de uma outra classe ou de um módulo, é a utilização de uma variável para representar uma outra classe. Como já foi explicado no tópico 1.1, variáveis podem representar classes. Após equalizar a variável a uma classe, devem-se utilizar regras parecidas com o modo de chamar métodos de módulos, porém, como nós temos uma variável, para representar uma classe, deve ser citado o nome dessa variável ao invés do nome da classe.

Exemplo:

```
$game_player = Game_Player.new          # Trecho tirade da classe
Scene_Tile, linha 125.

# em outra parte do mesmo script encontramos a seguinte linha:

$game_player.moveto($data_system.start_x, $data_system.start_y) #
Linha 216.

# Como a variável $game_player representa a classe Game_Player,
# foi só utilizar um ponto e o nome do método para chamá-lo (.moveto)
# note que foi necessário utilizar 2 valores neste método:
# ($data_system.start_x, $data_system.start_y)
# Eles representam as coordenadas x e y em que irá começar o jogo,
# pois tirei esta linha de dentro do método da Scene_Title que faz o
'new_game'.

# Note também que a variável $game_player, que representa a classe
Game_Player,
# É uma variável global, e pode ser acessada de qualquer lugar,
incluindo dos eventos
```

Vamos fazer um teste, crie um novo evento, e utilize o comando '$game_player.moveto($data_system.start_x, $data_system.start_y)', e teste o jogo. O método utilizado é o que move o personagem até o seu local de origem, nas especificações $data_system.start_x, $data_system.start_y. Tente agora substituir estes valores por outros valores quaisquer, exemplo: '$game_player.moveto(6,4)'.
O método 'moveto' é o método utilizado para fazer o 'teleport' dos eventos, porém ele só move o personagem de posição no mapa, outros métodos são utilizados juntos para que isto aconteça, explore os métodos das classes Game_Player, Game_Party e Game_Actor e você encontrará alguns métodos bem legais.

Tópico 3.2 Métodos Privados

Através dos métodos privados são criadas as chamadas "Variáveis de Instância Públicas", que são variáveis de instância da classe que podem ser acessadas mais facilmente de objetos externos.
Para criar uma variável de instância pública utilize o comando de atributo dentro da classe a qual a variável de instância pertence (fora de qualquer método). Segue abaixo os 3 tipos existentes de comandos de atributos:

attr_writer: Define uma variável de instância pública que poderá ser alterada a partir de qualquer classe ou módulo.

attr_reader: Define uma variável de instância pública que poderá apenas ser lida por qualquer classe ou módulo.

attr_accessor: Define uma variável de instância pública que poderá ser lida e alterada por qualquer classe ou módulo.

Para definir a variável de instância a ser pública coloque o nome dela, sem o "@", escrita em uma string, ou com um ":" no início, separada por um espaço do comando de atributo.

Exemplo:

```
# Trecho abaixo retirado a classe Game_Party, linhas 16 até 20.
  attr_reader   :gold
  attr_reader   :steps
  attr_accessor :last_item_id
  attr_accessor :last_actor_index
  attr_accessor :last_target_index
```

Tópico 4 – Superclasses / Herança

Superclasses são aquelas classes que servem de base para uma criação facilitada para as seuas "classes filhas", que irão herdar todos os métodos já declarados. As classes filhas devem ser declaradas juntamente com o símbolo < separando seu nome, do nome de sua classe pai.

Exemplo:

```
class Window_Status < Window_Base # Trecho retirado da classe
Window_Status, linha 7.
# No exemplo acima, Window_Status é a classe filha e Window_Base é a
sua superclasse.
```

Para exemplo de superclasses temos Window_Base e Scene_Base, todas as Windows e Scenes são derivados delas. As classes filhas podem utilizar todas as classes já declaradas pela sua "classe mãe", ou "Superclasse".

Exemplo:

```
# Método retirado da classe Window_Status, Linhas 20 à 29.

  def refresh
    self.contents.clear
    draw_actor_name(@actor, 4, 0)
    draw_actor_class(@actor, 128, 0)
    draw_actor_face(@actor, 8, 32)
    draw_basic_info(128, 32)
    draw_parameters(32, 160)
    draw_exp_info(288, 32)
    draw_equipments(288, 160)
  end

=begin
Se você abrir seu RPGMaker VX e ir na classe Window_Status, não irá
encontrar a declaração dos métodos draw_actor_name, draw_actor_class e
draw_actor_face; pois estes métodos pertencem à classe Window_Base,
porém, a classe Window_Status, é filha de Window_Base, logo, ela tem
permição para utilizar seus métodos.
=end
```

As classes filhas também irão herdar as variáveis de instância de sua superclasse, pois as variáveis de instância(para quem não se lembra) são aquelas que pertencem unicamente à classe a qual foi declarada.
Ao declarar um método já declarado por sua superclasse, o novo método irá substituir o já declarado, a não ser que seja utilizado o comando 'alias'. (será visto ainda nesta aula) Porém o método já declarado poderar ser chamado pela palavra super.

Exemplo:

```
# Método 'initialize' da classe Window_Status
  def initialize(actor)
    super(0, 0, 544, 416)
    @actor = actor
    refresh
  end

# O método 'initialize' já foi declarado pela sua superclasse.
# Utilizando 'super(0, 0, 544, 416)' será o mesmo que chamar
'initialize(0, 0, 544, 416)'
# Pois que o comando 'super' chama o mesmo método, porém, de sua
superclasse.
```

Se você observar os scripts padrões do RPG Maker notará que existem muitas classes que são classes filhas e que suas superclasses são, por sua vez classes filhas de outras superclasses, como exemplo de Window_EquipItem, que é filha de Window_Item, que é filha de Window_Selectable, que é filha de Window_Base, que é filha de Window, que é filha de Object, que é filha de Kernel. (as classes Window, Object e Kernel são classes internas do RGSS, e são impossíveis de serem observadas pelo editor do maker.)

Tópico 5 – Alias

Alias é um comando que serve para alterar o nome dos métodos já criados tanto por superclasses quanto pelas próprias classes. Este comando é muito utilizado por criadores de scripts para não alterar os métodos já existentes nas classes que forem alteradas, apenas adicionando mais comandos aos mesmos. Quando um alias for utilizado ele deve estar dentro da mesma classe que o método em qual o alias deverá alterar o nome, já que não é permitido o uso de expreções como 'objeto.método'.
Para utilizar o comando alias basta escrever a palavra 'alias' seguida do nome a ser alterado do método e o nome original do método.

Exemplo:

```
# Trecho retirado do meu script 'Tempo de Jogo', linhas 70 à 74.
  alias dc_tempo_de_jogo_start start
  def start
    dc_tempo_de_jogo_start
    @playtime_window = Window_PlayTime.new(@gold_window.x,
@gold_window.y - 88)
  End
```

O alias mudou o nome do método original da classe Scene_Menu de 'start' para 'dc_tempo_de_jogo_start', logo depois foi declarado o novo método 'start', e nele, é chamado o método 'dc_tempo_de_jogo_start', ou seja, o método 'start' original, já que seu nome mudou; e depois vem o comando '@playtime_window = Window_PlayTime.new(@gold_window.x, @gold_window.y - 88)'.
Em outras palavras, é adicionado o comando '@playtime_window = Window_PlayTime.new(@gold_window.x, @gold_window.y - 88)' ao método 'start' da classe referente.

Aula 4 – Estruturas Condicionais e de Repetição

Índice de Tópicos:

Tópico 1 – Como Funcionam

Na aula 2 foi visto basicamente para que servem e como funcionam as estruturas condicionais, nesta aula iremos nos aprofundar neste vasto assunto.

Basicamente, para que as condições ocorram, sempre serão necessários os pseudovalores true e false (vistos na aula 2). Eles decidirão se a condição citada será verdadeira ou não. Os valores true e false sempre irão substituir 2 valores que estejam separados por um operador.

Exemplo:

```
variável == 5          # Os valores 'variável' e '5' estão separados
por um operacional '=='
```

Logo, só acontecerão condições se a expressão citada na condição tiver valor de 'true' ou 'false'.

Tópico 1.1 – Operadores

A função dos operadores condicionais é de substituir os 2 valores que ele separa por um valor 'true' ou 'false', de acordo com o operador e valores utilizados.

Operadores:

```
==      # Igual à
!=      # Diferente de
>       # Maior que
<       # Menor que
>=      # Maior ou igual que
<=      # Menor ou igual que
```

OBS: Tenha cuidado para não confundir os operadores condicionais com os operadores matemáticos. O operador '==', por exemplo, é um operador condicional, já o operador matemático '=', é utilizado para definir o valor de uma variável, como ja foi visto em aulas anteriores.

Para uma maior compreensão do assunto, aqui vai alguns exemplos:

```
3 == 3                                  # true
4 == 2                                  # false
5 == [5]                                # false ('5' é um número, e
'[5]' é uma array)
1_000_000 == 1000000                    # true
variável == variável                    # true
"string" == "string"                    # true
'string' == ""                          # false
```

Tópico 2 – Tipos de Estruturas Condicionais

Como visto na aula 2, existem vários tipos de estruturas condicionais, qualquer uma poderá ser utilizada para qualquer situação, porém algumas serão basicamente mais simples de serem feitas.

Tópico 2.1 – If / Unless

If

If, é a estrutura condicional mais utilizada pelo RGSS, para utilizá-lo basta escrever a palavra 'if' e logo depois os valores, separados por um espaço.

Se os valores resultarem em 'true', o que está dentro da estrutura acontecerá, vejamos um exemplo já citado na aula 2:

```
if variável == 5
  p 'sim'
end
```

Como os valores 'variável' e '5' estão separados por um operador condicional, o programa irá verificar se o valor de variável é igual a 5, se for, a espreção será substituída por um 'true', logo, a espreção 'p 'sim'' será utilizada, mas se 'variável' não for igual a 5, a espreção será substituída por um 'false', logo, a espreção 'p 'sim'' será ignorada.

Vamos à aula prática, crie um script acima de todos os outros em seu projeto (acima de todos os outros para que ele ocorra antes dos outros), e digite a seguinte espreção:

```
p 5 == 5
```

Agora teste o jogo, verá que o valor que apareceu na tela foi 'true', pôs como "5 é igual a 5", a espreção foi substituída pelo valor 'true'. Agora mude a espreção colocando outros valores no lugar dos cincos e teste para ver o que acontece, coloque também números diferentes como '4 == 6', e outros tipos de valores que você aprendeu na aula

2, como strings e arrays, por exemplo. Teste também outros operadores, como por exemplo ‘2 > 1’.

OBS: Note que você não pode utilizar os operadores ‘>’, ‘<’, ‘=>’ e ‘<=’ com valores de arrays e strings, porém, você pode utilizar ‘==’ e ‘!=’, mesmo que não esteja utilizando junto com valores iguais.

Exemplo:

```
 “string”   ==   [1, 2, 3]                          # false
“RPGMaker XP” == “RPGMaker VX”     # false
[5, 4, 6]   ==   [7, 2, 0]                          # false

“string”   !=   [1, 2, 3]                          # true
“RPGMaker XP” != “RPGMaker VX”     # true
[5, 4, 6]   !=   [7, 2, 0]                          # true
```

Unless

Basicamente ‘Unless’ é exatamente o contrário de ‘if’.

Exemplo:

```
# As 2 espreções abaixo resultariam num mesmo resultado:

if variável == outra_variável     # se variável for igual a outra
veriável

unless variável != outra_variável     # se a variável não for
diferente da outra variável
```

Utilização Alternativa das expressões

Como já dito anteriormente existem vários tipos de condicionais além do tradicional explicados acima. Estas formas alternativas são utilizadas com o mesmo conceito das espreções já explicadas, porém com poucas alterações

Condições de uma linha

Estas são utilizadas quase da mesma forma que a tradicional explicada acima, porém, só têm suporte à um comando. Para utilizá-la faça da mesma forma que foi esplicado, porém, coloque tudo na mesma linha com a palavra ‘then’ logo após os valores condicionais.

Exemplo:

```
# condicional comum:
if variável == 5
  p ‘sim’
end

# condicional de uma linha:
If variável == 5 then p ‘sim’ end     # se variável for igual a 5, p
‘sim’
```

Ocorrerá exatamente a mesma coisa que a condicional comum, porém, utilizando um espaço de linhas menor, como mostra o exemplo acima.
Você também pode utilizar um else, para definir uma exceção:

```
# condicional comum:
if variável == 5
  p ‘sim’
else
  p 'nao'
end

# condicional de uma linha:
if variável == 5 then p “sim" else p 'não' end     # se variável for
igual a 5, p ‘sim’, se não, p 'não'
```

OBS: Exceções serão vistas ainda nesta aula

Condições de um comando

Outra forma de resumir uma condicional comum à uma linha é simplesmente colocar o resultado antes da palavra ‘if’.

Exemplo:

```
# condicional comum:
if variável == 5
  p ‘sim’
end

# condicional de uma linha:
p ‘sim’ if variável == 5     # p ‘sim’, se variável for igual a 5
```

Tópico 2.2 – And / Or

As palavras ‘and’ e ‘or’ são extensões para criar condições mais elaboradas e específicas, quando necessário. Elas devem ser colocadas após a condição, e exigem uma segunda condição.

Exemplo:

```
if variável == 5 and outra_variável == 3
end
```

ou

```
if variável == 5 or outra_variável == 3
end
```

Elas representam uma maior necessidade de requisitos para a estrutura. O ‘And’ (em inglês, ‘e’) indica que ambas as condições deverão ser cumpridas para a estrutura ser ativada. O ‘Or’ (em inglês, ‘ou’) indica que a estrutura será ativada se qualquer uma condição for ativada.

Exemplo:

```
# A estrutura abaixo será ativada se ‘variável’ e ‘outra_variável’
forem,
# respectivamente iguais a 5 e 3.
if variável == 5 and outra_variável == 3
end
```

```
# A estrutura abaixo será ativada se 'variável' ou 'outra_variável'
for,
# respectivamente 5 ou 3.
if variável == 5 or outra_variável == 3
end
```

And e Or podem ser usadas quantas vezes forem necessárias, para não haver estruturas dentro de estruturas.

Exemplo:

```
# Lembrando que você aprendeu na aula 2 que variáveis de tipos
diferentes serão
# sempre variáveis distintas, mesmo que tenham o mesmo nome.

if variável == 5 and @variável == 5 and $variável == 5 and VARIÁVEL ==
5
end
```

ou

```
if variável == 5 or @variável == 5 or $variável == 5 or VARIÁVEL == 5
end
```

Você pode criar, também, estruturas com 'and' e 'or' juntas.

Exemplo:

```
if variável == 5 or @variável == 5 and $variável == 5 or VARIÁVEL == 5
end

=begin
Para entender este caso, você deve prestar muita atenção,
A estrutura só será ativada se uma das variáveis 'variável' ou
'@variável' forem 5,
E se, além disto, uma das variáveis '$variável' ou 'VARIÁVEL' forem 5.
=end
```

Tópico 2.3 – Condições Rápidas

Estas condições são utilizadas em casos raros, em que o scripter quer criar uma condição com uma exceção de um único comando. Para utilizá-la não é necessário a palavra 'if', apenas a condição(valor, operador, valor), uma interrogação('?'), o comando em caso de a condição ser verdadeira, dois pontos(':'), e o comando de exceção.

Exemplo:

```
variável == 5 ? método : outro_método
# É claro que você não é obrigado chamar um método, você pode também
# utilizar outro comando, como alterar o valor de uma variável:

variável == true ? variável = false : variável = true
# O exemplo acima é um comando normalmente usado para variáveis que
# representam switches, para mudarems eu valor independente de seu
valor atual:
# Se ela for true, se transforma em false, mas se não for true,
# se transforma em true, pôs logicamente é false.
```

OBS: O comando 'Toogle Switch'(do maker 2003) foi extinto exatamente pelo fato de ele poder ser facilmente feito por scripts, como mostra o exemplo acima.

É sempre bom lembrar que, como o ruby ignora os espaços em branco sempre que há necessidade de um novo valor, as condições rápidas podem ser expressas da seguinte forma, facilitando sua compreensão:

```
variável == true ?     # Se variável for true...
variável = false :      # Ela se transforma em false,
variável = true         # Se não, ela se transforma em true
```

Logo, as condições rápidas também podem ser utilizadas várias veses de uma só vez: Código:

```
# O trecho abaixo foi retirado da classe Window_Base do RMXP, linhas
301 a 305.
# Desenhar HP
self.contents.font.color = actor.hp == actor.maxhp ?
Color.new(64,255,128) :
                                          actor.hp <= actor.maxhp /
5 ? knockout_color :
                                          actor.hp <= actor.maxhp /
2 ? crisis_color :
                                          normal_color
# Note como as condições são repetidas várias vezes.
# As condições rápidas são apenas para diminuir o número de linhas
utilizadas
# por condições mais simples, pois, apesar de este exemplo estar em
várias linhas,
# ele poderia ter sido feito em apenas uma linha, eliminando os
espaços.

# Você não é obrigado a entender este exemplo agora, pôs já é algo
mais avançado,
# e por tanto não será cobrado até que você se especialize em scripts.
```

Tópico 2.4 – Case

A estrutura condicional Case, tem o mesmo objetivo da estrutura if, porém mais fácil de programar em certas situações. Com ela você irá definir um comando (ou mais) para o caso de cada possível valor de uma determinada variável. Em outras palavras, executará um comando diferente para cada valor que a variável tiver. Para usá-lo, utilize a palavra case, seguida de uma variável, e nas linhas abaixo, as palavras when, seguidas do possível valor da variável, e no final um 'end', para determinar o fim da estrutura.

```
# Trecho retirado da classe Scene_Tile, do RMVX, linhas 66 à 73.
case @command_window.index     # Caso o cursor esteja no botão:
when 0                                          # 0, executa o
método 'command_new_game'
  command_new_game
when 1                                          # 1, executa o
método 'command_continue'
  command_continue
when 2                                          # 2, executa o
método 'command_shutdown'
  command_shutdown
end
```

Você pode utilizar vários valores em cada ‘when’, para não precisar repeti-lo, utilizando todos os valores possíveis separados por vírgulas. Vamos fazer um teste, abra o editor de seu projeto e vá na linha 67, e onde tem o 0(zero), coloque 0, 1, 2; e teste o jogo. Você perceberá que não importa qual botão você clicar, todas irão redirecioná-lo ao new game, pôs você definiu que se o cursor estiver em qualquer um dos botões 0, 1 e 2; eles irão levá-lo para o método ‘new_game’ (que está logo abaixo do when 0, 1, 2).

Você poderá também utilizar um ‘else’, entre o último ‘when’ e o ‘end’, para determinar uma exceção. (exceções serão vistas ainda nesta aula)

Exemplo 1: (string)

```
sabor_do_biscoito = "morango"

case biscoito
when "chocolate"
  # o biscoito é de chocolate
when "morango"
  # o biscoito é de morango
when "limão"
  # esse biscoito é ruim >_<
End
```

Exemplo 2: (range)

```
preço_do_biscoito = 5

case biscoito
when 0 ... 2
  # que biscoito barato
when 3 ... 6
  # o biscoito está num preço normal
when 7 ... 1_000_000
  # biscoito de ouro
end
```

Tópico 2.5 – Exceções

Exceções são o que acontecerão no caso de a condição da estrutura não for cumprida corretamente. As estruturas condicionais poderão, ou não ter exceções (exceto a ‘condição rápida’, que é obrigatório o uso de exceções). Para definir exceções utilize a palavra ‘else’.

```
if variável == 5
  # variável é  5
else
  # variável não é 5
end
```

Exceções podem ser utilizadas também nas estruturas case.

```
case power_ranger
when "Azul", "Verde", "Vermelho", "Preto"
  # Homem
when "Rosa", "Amarelo"
  # Mulher
else
end
```

Exceções Condicionais

As Exceções condicionais são um tipo de exceção utilizado somente nas condicionais if e unless. Elas representam uma segunda condição dentro da exceção da condição principal. Para determinar uma exceção condicional utilize a palavra 'elsif', ao invés de 'else', seguido da condição.

Exemplo:

```
if variável == 0
  # condição
elsif variável == 1
 # exceção condicional
else
  # exceção
end
```

Note que o exemplo acima poderia ser representado da seguinte forma:

```
if variável == 0
  # condição principal
else
  if variável == 1
    # segunda condição dentro da exceção da condição principal
  else
    # exceção da segunda condição
  end
end
```

Tópico 3 – Estruturas de Repetição

As estruturas de repetição são estruturas que irão executar os comandos determinados repetidamente um determinado número de vezes, ou até você determinar o seu término. Existem vários tipos de estrutura de repetição, cada uma é trabalhada de uma forma diferente, existindo assim uma estrutura diferente para cada caso.

Tópico 3.1 – Loop

Os loops são estruturas em que deverão ser definidos alguns comandos e os mesmos ficarão sendo executados, até que o comando de finalização do loop seja chamado. Em Ruby, os loops são trabalhados de forma exatamente igual aos loops dos eventos do RPGMaker. Para iniciar um loop, use o comando 'loop do', e use a palavra 'break' para "quebrar" o loop, não esqueça de que o loop, por ser uma estrutura, precisa de um end, para determinar seu fim.

Exemplo:

```
# Trecho retirado da classe Scene_Base, do RMVX, linhas 16 até 21.
loop do
  Graphics.update
  Input.update
  update
  break if $scene != self
end
```

Note que o comando ‘break’, está em uma condição: ‘break if $scene != self’, ou seja, o loop só será quebrado no caso de a variável ‘$scene’ for diferente (!=) de ‘self’. ‘self’ será visto mais profundamente no futuro, ‘self’ representa a própria classe em que foi escrito.

Caso a condição de ‘$scene’ ser diferente de ‘self’ não for cumprida, chegará no final sem um ‘break’, logo, ela irá recomeçar a partir do primeiro comando. (neste caso, ‘Graphics.update’)

Tópico 3.2 – While

While é uma estrutura muito semelhante ao loop, porém, é trabalhada de forma diferente. Em while, você deve descrever uma condição para uma quebra de estrutura logo após a declaração do início da estrutura.

Exemplo:

```
# Trecho retirado do script Main, do RMXP, linhas 36 à 38.
while $scene != nil
  $scene.main
end
```

Na mesma linha da declaração do início da estrutura, foi declarada uma condição para a quebra da estrutura (se $scene != nil). A estrutura será repetira sempre que a condição não for cumprida, e será quebrada no caso de esta condição não for cumprida, ou no caso do uso da quebra de estruturas ‘break’.

Tópico 3.3 – For

A estrutura diferente das citadas acima, por trabalhar com um valor que irá mudar de acordo com o número de vezes que a estrutura já foi repetida. Para chamar esta estrutura, utilize a palavra ‘for’, seguida de uma variável que não precisa estar declarada, neste caso, você já estará declarando a variável; depois a palavra in, e por último uma range ou uma array, pré determinada.

A variável escolhida terá, inicialmente, o primeiro valor da range ou array. A cada vez que a estrutura for repetida, a variável declarada, passará a ter o próximo valor da range ou da array, até chegar em seu último valor, neste caso, a estrutura será quebrada.

Exemplo:

```
# Exemplo retirado da classe Window_Command, do RMXP, linhas 31 à 33.
for i in 0...@item_max
  draw_item(i, normal_color)
end
```

Percebe-se que o método ‘draw_item’, é um método que necessita de dois valores, por isso os parênteses, com as duas variáveis separadas por vírgula, ‘i’ e ‘normal_color’.

Cada vez que a estrutura for repetida, a variável ‘i’ terá um valor diferente, sendo este valor um valor que virá dês de 0, até o valor da variável @item_max, por exemplo: se @item_max for 3, a cada vez que a estrutura for repetida, i terá um valor de: 0, depois 1, depois 2, e por último 3. Logo, se @item_max for 3, a estrutura acima seria o mesmo que o exemplo abaixo:

```
draw_item(0, normal_color)
draw_item(1, normal_color)
draw_item(2, normal_color)
draw_item(3, normal_color)
```

Este tipo de estrutura é muito útil, quando você quer utilizar várias veses um mesmo comando, porém com valores diferente, exatamente como os exemplos citados acima.

Aula 5 – Revisão e Classes do RGSS

Índice de Tópicos:

Tópico 1 – Classes e Módulos

Ruby é uma linguagem de programação que se baseia num sistema de objetos, as classes. Cada possível valor será considerado uma classe, ou seja, um objeto. Cada classe tem vários métodos, que são os comandos possuídos pelas classes, sem os métodos, as classes não seriam nada, e vice-versa. Módulos são parecidos com as classes, porém, estas não precisarão ser declaradas, além de poderem ser utilizadas a partir de qualquer parte do jogo. Você também pode chamar constantes dos módulos, utilizando o comando 'Módulo::Constante'.

Tópico 2.1 – Herança

Classes que têm uma herança em relação a outras, terão todos os métodos de sua superclasse, ou "classe mãe". As classes também podem herdar os métodos de um Módulo ao utilizar o comando 'Include.Método' dentro da classe desejada.

Tópico 2 – Revisão e Complemento

Existem 4 tipos di variáveis: local, de instância, global e constante. Variáveis locais devem ser declaradas e usadas num mesmo método de uma classe ou de um módulo(variável). Variáveis de instância devem ser utilizados somente na classe ou método em que foram declaradas(@variável). Variáveis de instância pública são determinadas variáveis de instância que poderão ser acessadas por outras classes ou módulos(@classe.variável). Variáveis globais poderão ser acessadas por qualquer classe, ou até mesmo pelos eventos($variável). Constantes são variáveis que terão sempre um determinado valor (CONSTANTE ou Constante ou Módulo::Constante).

Tópico 2.1 – Valores Numéricos

Valores numéricos são aqueles representados por números. Existem 2 tipos de valores numéricos: Integer e Float.

Integer (Números Inteiros) são números que podem ser negativos, ou positivos, exemplo:

```
... , -3 , -2 , -1 , 0 , 1, 2 , 3 , ...
```

Float (Números Decimais) são números que estão entre um número inteiro e outro, exemplo:

```
... , 0.7 , 0.8 , 0.9 , 1.0, 1.1 , 1.2 , 1.3 , ...
```

OBS: Note que um valor Float não se transformará em integer se ele terminar com '.0'. Note também que uma Integer, quando dividida por um valor inadequado (10/3, por exemplo) será arredondado (o resultado seria 3), porém, em float(10.0/3) resultaria no valor correto(3.333...). Logo, o valor Float é utilizado somente em raros casos, o HP dos personagens, por exemplo, é uma Integer. (Imagine se seu HP ficasse "3.333..." entendeu agora porque ele fica em Integer ao invés de Float?)

Os Integer são divididos em dois grupos: Fixnum, e Bignum. Fixnum serão Integers pequenas (com valores pequenos), enquanto Bignums (valores raramente usados) são números que excedem o limite da memória para estes valores.

Exemplo:

```
1, 2, 3, 4, 5, 6, ...          # Fixnum
1_000_000_000_000              # Bignum
```

OBS: Lembre-se que Bignum e Fixnum são classes filhas de Integer, logo, um Float nunca será um Bignum ou Fixnum, exemplo:

```
1_000_000_000_000            # Bignum
1_000_000_000_000.0          # Float
```

Representação das classes que representam valores numéricos:

```
Numeric   >   Integer   >   Fixnum
                          >   Bignum
          >   Float
```

Tópico 2.2 – Strings

Strings são valores em textos (palavras, etc.). Devem ser colocados dentro de parênteses. As strings podem exibir valores de variáveis utilizando o comando '#{variável}' dentro da string.

Exemplo:

```
memória_ram = 516
"A sua memória RAM é de: #{memória_ram} Mb."
  # O programa interpretará o texto acima desta forma:
  # "A sua memória RAM é de: 516 Mb."
```

```
  # É claro que alterando o valor de 'memória_ram' obteríamos outros
valores
memória_ram = 256
  # Agora, o programa interpretará o texto da seguinte forma:
  # "A sua memória RAM é de: 256 Mb."
```

Métodos

A classe String possui alguns métodos extremamente úteis para casa tipo de situação, estes métodos podem ser complexos para serem compreendidos, e não serão utilizados com grande freqüência no ruby, portanto não é obrigatório o compreendimento de cada um deles. Os principais são: downcase/upcase, sub/gsub e scan.

Downcase e Upcase

Downcase e Upcase alteram cada letra contida na String entre letras minúsculas e maiúsculas. Downcase e Upcase são métodos que alterarão o valor da String somente no momento em que forem utilizados, para alterar o valor da String permanentemente, são utilizados uma exclamação ("!") após as palavras downcase e upcase.

Exemplo:

```
string = "Pipoca Salgada"

string.upcase     # PIPOCA SALGADA
string.downcase     # pipoca salgada
string     # Pipoca Salgada

string.upcase!
string     # PIPOCA SALGADA
string.downcase!
string     # pipoca salgada
```

Sub e Gsub

Sub e Gsub são comandos utilizados para uma substituição de determinadas palavras da String por outras palavras. Para utilizar os comandos Sub e Gsub é necessário após chamar o método, um valor a ser substituído (será colocado entre parênteses) e um valor a substituir (será colocado entre chaves). A única diferença entre Sub e Gsub é que Gsub irá substituir todos os trechos da String que conhecidirem com o valor entre parênteses, e Sub irá substituir somente o primeiro valor encontrado, como mostra o exemplo a seguir:

```
pipoca = "Pipoca Salgada"
pipoca.sub("Salgada"){"Doce"}
pipoca          # "Pipoca Doce"

texto = "o rato roeu a ropa do rei de roma"
texto.gsub("r"){"l"}
text          # "o lato loeu a lopa do lei de loma"
```

Scan

Scan é um método utilizado para recolher uma array com todos os valores do texto desejado.

Exemplo:

```
texto = "o rato roeu a ropa do rei de roma"
text_escaneado = texto.scan("ro")
texto_escaneado          # ["ro", "ro", "ro"]
```

Tópico 2.3 – Array

Arrays são tipos de valores que podem guardar vários valores diferentes, como mostra o exemplo abaixo:

```
array = [4, [16, [0, 82], 39.4], "texto", 1_000_000, ["variável:
#{variável}", 8] ]

# Cuidado para não confundir um valor com o outro, eles são sempre
separados por vírgulas
```

Cada valor contido em uma Array tem um id, que corresponde a sua posição, dentro da Array, começando a contagem do 0 (zero).

Exemplo:

```
array = [4, [16, [0, 82], 39.4], "texto", 1_000_000, ["variável:
#{variável}", 8] ]

=begin
id 0: 4
id 1: [16, [0, 82], 39.4]
id 2: "texto"
id 3: 1_000_000
id 4: ["variável: #{variável}", 8] ]
=end
```

Você pode alterar diretamente um único valor de uma array, colocando seu id entre chaves logo após o nome da array.

Exemplo:

```
array = []          # Declara a variável 'array'
array[0] = 4
array[1] = [16, [0, 82], 39.4]
array[2] = "texto"
array[3] = 1_000_000
variável = 10
array[4] = ["variável: #{variável}", 8] ]

# a variável 'array' passará a ter o seguinte valor:
# array = [4, [16, [0, 82], 39.4], "texto", 1_000_000, ["variável: 10",
8] ]
```

Você pode utilizar os valores dentro da array da mesma forma que outra variável qualquer.

Exemplo:

```
array = [1, 2, 3]
array[1] = 3
array[2] = 9
array[0] -= 1

# array agora esta assim: [0, 3, 9]
```

OBS: Ao tentar selecionar um valor ainda não declarado pela array, ele será nil, exemplo:

```
array = [1, 2]
# array[2] é nil, array[3] é nil, array[4] é nil...

array[3] = 4

# array agora esta assim: [1, 2, nil, 4]
```

Você também pode adicionar determinados novos valores à array, simplesmente somando uma array com outra.

Exemplo:
```
array = [1, 2, 3]
array += [4, 5, 6]
# array agora está assim: [1, 2, 3, 4, 5, 6]
```

Você pode selecionar também uma determinada quantidade de valores de uma única vez, utilizado dois valores, dentro das chaves. O primeiro determinará o id do primeiro valor a ser alterado, o segundo determinará a quantidade de valores que deverão ser selecionados.

Exemplo:
```
array = [0, 0, 0, 0, 0, 0, 0, 0]

array[1, 6] = [1, 1, 1, 1, 1, 1]
# array agora está assim: [0, 1, 1, 1, 1, 1, 1, 0]

array[1, 6] = 1
# array agora está assim: [0, 1, 0]
```

Métodos

Por ser uma classe, Array tem alguns métodos que deverão ser utilizados de acordo com a situação. Os principais métodos de Array são: clear, compact, delete/delete_at, empty?, include?, index, length/size, nitems, shift/unshift, push/pop, reverse/sort.

Clear

Clear é um comando muito simples, que apaga a array, é o mesmo que redeclarar a array.

Exemplo:
```
array = []          # Declara a variável 'array'


array[0] = 4
array[1] = [16, [0, 82], 39.4]
array[2] = "texto"
array[3] = 1_000_000
array[4] = ["variável: 10", 8]
array += [4, 5, 6]
array[1, 3] = [1, 2, 1]
# array agora está assim: [4, 1, 2, 1, ["variável: 10", 8], 4, 5, 6]

array.clear
# array agora está assim: []
```

Compact

Compact é um simples comando que elimina todos os valores 'nil' da array.

Exemplo:

```
array = [1, 2]
array[6] = 0
# array agora esta assim: [1, 2, nil, nil, nil, nil, 0]

=begin
array.compact será: [1, 2, 0]
note que o comando 'compact' não altera o valor verdadeiro da array.
(assim como downcase/upcase, das strings)
assim, como downcase/upcase, compact também poderá ser utilizado com
uma exclamação para alterar definitivamente o valor da array
=end

array2 = array.compact
# array ainda está assim: [1, 2, nil, nil, nil, nil, 0]
# array2, está assim: [1, 2, 0]

array.compact!
# agora, array está assim: [1, 2, 0]
```

Delete e Delete_at

Delete e Delete_at são commandos extremamente parecidos, porém utilizados de forma diferente. Ambos têm a função de eliminar um determinado valor da array, alterando assim toda a estrutura, e conseqüentemente, os ids de cada um dos valores que estiverem posicionados após o valor eliminado.

Para utilizar o 'Delete' é necessário especificar o valor a ser deletado na array, logo após o comando. O programa irá procurar por todos os valores especificados, que a array guardar, e apagá-los.

Exemplo:

```
array = [1, 2, 3, 2, 1, 2, 3, 1, [1, 2, 3]]

array.delete(2)
# array agora está assim: [1, 3, 1, 3, 1, [1, 2, 3]]

array.delete(1)
# array agora está assim: [3, 3, [1, 2, 3]]

array.delete(3)
# array agora está assim: [[1, 2, 3]]

array.delete([1, 2, 3])
# array agora está assim: []
```

Já o comando 'Delete_at', apaga apenas o valor que estiver na posição definida ao chamar o comando.

Exemplo:

```
array = [1, 2, 3, 2, 1, 2, 3, 1]

array.delete_at(2)
# array agora está assim: [1, 2, 2, 1, 2, 3, 1]

array.delete_at(0)
# array agora está assim: [2, 2, 1, 2, 3, 1]

array.delete_at(4)
# array agora está assim: [2, 2, 1, 2, 1]
```

Empty?

O comando empty, é utilizado em condições, ele será substituído por true, ou false sempre que for chamado, sendo true se a array tiver valor '[]' (nenhum valor), e false se ela tiver ao menos um único valor.

Exemplo:

```
if array.empty?
  # a array está vasia
else
  # a array tem algum valor
end
```

Exemplo 2:

```
array.clear

if array.empty?
  print "a array está vasia"
end

=begin
Não importa qual valor array tinha antes do comando 'array.clear', a
condição será cumprida, pôs todos os valores da array foram apagados
no comando 'array.clear'.
=end
```

Include?

A forma de utilizar o comando 'Include?' é extremamente semelhante ao comando 'empty' pelo fato de ambos se referirem a uma condição. O comando 'Include?' verifica se a array têm algum valor específico.

Exemplo:

```
if array.include?(5)
  # dentro de array, existe um 5
else
 # dentro de array, não existe nenhum 5
end
```

Index

Index seleciona o id de um determinado valor. Se a Array tiver mais do que um valor especificado, será selecionado o id do primeiro valor, se a Array não possuir nenhum valor igual ao especificado, o comando será substituído por nil.

Exemplo 1:

```
array = [1, 2, 3]
id = array.index(2)
```

```
# neste caso, id é 1
```

Exemplo 2:

```
array = [1, 2, 3, 2, 1]

id = array.index(2)
# neste caso, id é 1

id = array.index(1)
# neste caso, id é 0
```

Exemplo 3:

```
array = [1, 3, 1, 3]
id = array.index(2)
# neste caso, id é nil
```

length/size

Este comando é bem simples, ele seleciona o tamanho da Array, ou seja, o número de valores contidos na mesma.

Exemplo:

```
array = [1, 2, 3]
array.length          # neste caso, será 3

array = [0, 0, 0, 0, 0, 0]
array.size            # neste caso, será 6

array = []
array.length          # neste caso, será 0

array = [“texto 01”, “texto 02”, “texto 03”]
array.size            # neste caso, será 3
```

OBS: Observe que não há qualquer diferença entre os comandos ‘length’ e ‘size’, é apenas a palavra.

Nitems

Seleciona o número de valores não ‘nil’ existentes na array.

Exemplo:

```
array = [1, 2, 3, 4, 5]
array.nitems          # neste caso, será 5

array = [1, nil, 3, nil, 5]
array.nitems          # neste caso, será 3

array = []
array.nitems          # neste caso, será 0

array = [nil, nil, nil]
array.nitems          # neste caso, será 3

array = [0, nil, 1, nil, nil, 2, nil]
array.nitems          # neste caso, será 3
```

OBS: Note que nitems será igual a size/length sempre que não existir nenhum nil, na array.

Shift/Unshift

Shift e Unshift são comandos utilizados para adicionar ou remover elementos, de id 0 da array.
O comando Shift irá remover o primeiro elemento da array, e Unshift adiciona um novo elemento a array, que virá como id 0, mudando a posição dos outros valores da array.

Exemplo:

```
array = [1, 2, 3]

array.shift
# array, agora está assim: [2, 3]

array.unshift(1)
# array, agora está assim: [1, 2, 3]
```

Push/Pop

Push e Pop são comandos extremamente parecidos com Shift e Unshift, a única diferença é que os comandos Push e Pop trabalham com o último valor da array, diferente de Shift e Unshift, que se referem ao primeiro valor da array.

Exemplo:

```
array = [1, 2, 3]

array.pop
# array, agora está assim: [1, 2]

array.push(3)
# array, agora está assim: [1, 2, 3]
```

Sort e Reverse

Sort é um Comando que organizará a array em uma ordem fixa, e reverse inverterá a array para que sua ordem de valores fique na ordem inversa. Uma array não pode conter tipos diferentes de valores quando utilizar o comando 'sort', pois neste caso ocorrerá um erro. Se a array conter somente valores numéricos, ela será organizada por ordem de valores do menor para o maior, se ela conter somente textos (strings) ela será organizada em ordem alfabética, e se ela conter somente arrays, seus valores devem conter somente um mesmo tipo de valor, e assim sucessivamente.

Exemplo:

```
array = [5, 8, 9, 7, 4, 2, 6, 8]

array.sort
# array, agora está assim: [2, 4, 5, 6, 7, 8, 8, 9]

array.reverse
# array, agora está assim: [9, 8, 8, 7, 6, 5, 4, 2]

array = ["Dark Chocobo", "Jão Rafael", "Cabana Ruby", "Reino RPG"]

array.sort
# array, agora está assim: ["Cabana Ruby", "Dark Chocobo", "Jão
Rafael", "Reino RPG"]

array = ["Dark", "Chocobo", "Jão", "Rafael", "Cabana", "Ruby", "Reino",
"RPG"]
```

```
array.sort
# array, agora está assim: [“Cabana”, “Chocobo”, “Dark”, “Jão”,
“Rafael”, “Reino”, “RPG”, “Ruby”]
```

Tópico 2.4 – Hash

Assim como as Arrays, Hash é um tipo de valor utilizado como container para guardar outros tipos de valores. Para trabalhar com hashes, são utilizados chaves (‘{‘ e ‘}’) ao invés de colchetes, como nas arrays.

Diferente das Arrays, nas Hashes, não importa a posição em que cada valor se encontra, pois quando trabalhamos com Hashes, não mais utilizaremos um sistema de ‘ids’ dos valores. Ao invés disso, os valores estarão “soltos” dentro da Hash, podendo assim chamar qualquer um de seus valores, através de númetos ou textos determinados, estes números textos representarão cada valor da hash.

Para declarar uma hash, deve-se colocar cada um de seus valores separados por vírgulas (assim como na Array), porém, para cada valor introduzido na hash, você deve atribuir um número ou um texto como um “nome”(ou um núemero) para aquele valor; que deverá ser colocado logo antes do valor, separado por um ‘=>’. (Não representa ‘maior que’.)

Exemplo de Declaração de uma Hash:
```
hash = { “valor 01” => 5, “valor 02” => 10, “valor03” => 15 }
```

No Exemplo acima, os nomes “valor 01”, “valor 02” e “valor 03” representarão os “nomes” dos valores 5, 10 e 15. Note que, como nas hashes não importa a ordem com que os valores estão na Hash, o exemplo poderia ser realizado também das seguintes formas:
```
hash = { “valor 01” => 5, “valor 02” => 10, “valor03” => 15 }

ou

hash = { “valor02” => 10 , “valor 01” => 5, “valor 03” => 15 }

ou

hash = { “valor 03” => 15 , “valor02” => 10 , “valor 01” => 5

# Em outras palavras, não importa a ordem utilizada.
```

Como eu determinei certos valores para representarem cada valor, eu deverei colocá-los no lugar do id do valor, e chamá-lo exatamente igual as Arrays.

Exemplo:
```
hash = {}
hash[“valor 01”] = 5
hash[“valor 02”] = 10
hash[“valor 03”] = 15

# Desta forma, resultaria uma hash exatamente igual a hash do exemplo
anterior.
```

Como foi dito, cada valor da hash deverá ser representado por um nome ou um número, logo, eu poderia criar uma hash desta forma também:

```
hash = {}
hash[0] = 5
hash[1] = 10
hash[2] = 15

# Lembrando que 0, 1 e 2 são os nomes que representam cada valor 5, 10
e 15 da hash,
# e não os ids dos mesmos, pois hashes não têm ids.
```

Métodos

Por ser uma classe, Hash também tem seus próprios métodos, que deverão ser utilizados de acordo com a situação, seus principais métodos são: clear, delete, empty, has_key?/include?, has_value?, index, keys, length/size, values.

Clear, Delete, Empty?, Length e Size

Estes comandos, também estão presentes nas Arrays, por este motivo, suas explicações serão bem simples:
Clear apaga cada valor da Hash, como se a apagasse e a recriasse sem nenhum valor.

Exemplo:

```
hash = { "valor" => 0 }
hash.clear
# hash agora está assim: {}
```

Delete deve ser utilisado com um valor entre parênteses. Delete apaga qualquer valor que tiver como representante o valor entre parênteses.

Exemplo:

```
hash = { "valor 01" => 5, "valor 02" => 10, "valor03" => 15 }
hash.delete("valor 01")
# hash agora está assim: { "valor 02" => 10, "valor03" => 15 }
```

'Empty?' deve ser colocado numa condição, ele será substituído por true se a hash não tiver qualquer valor.

Exemplo:

```
if hash.empty?
  # significa que a hash está assim: {}
else
  # significa que a hash não está assim: {}
End
```

Length e Size representam o número de valores contidos na hash.

Exemplo:

```
hash = { "valor 01" => 5, "valor 02" => 10, "valor03" => 15 }
# neste caso, hash.size será 3

hash.clear
# agora, hash está assim: '{}', logo, hash.size será 0.
```

Index

O comando Index será substituído pelo número ou texto que representa o valor especificado entre parênteses ao chamar o comando.

```
hash = { "valor 01" => 0, "valor 02" => 1, 0 => 42, 1 => 56 }

nome1 = hash.index("valor 02")
nome2 = hash.index(0)

 # neste caso, 'nome1' será '1' e 'nome2' será '42'.
```

Values

O comando será substituído por uma Array que irá conter todos os valores da Hash.

Exemplo:

```
hash = { "valor 01" => 0, "valor 02" => 1, 0 => 42, 1 => 56 }

 # neste caso, hash.values será:
 # [0, 1, 42, 56]
```

Keys

Keys é um comando extremamente semelhante ao Values, porém, ao contrário de Values, em Keys você obterá uma Array com os valores referentes aos textos e números que representam cada um dos valores, na Hash.

Exemplo:

```
hash = { "valor 01" => 0, "valor 02" => 1, 0 => 42, 1 => 56 }

 # neste caso, hash.keys será:
 # ["valor 01", "valor 02", 0, 1]
```

Has_value?

Este, é um comando que deverá ser utilizado como condição. Ele determina se a Hash possue algum determinado valor.

Exemplo 1:

```
if hash.has_value?(10)
  # algum dos valores da Hash é '10'
else
  # nenhum valor da Hash é '10'
end
```

Exemplo 2:

```
hash = { "valor 01" => 0, "valor 02" => 1, 0 => 42, 1 => 56 }

 # neste caso, 'hash.has_value?(0)' será true
 # neste caso, 'hash.has_value?(1)' será true
 # neste caso, 'hash.has_value?(2)' será false
 # neste caso, 'hash.has_value?(42)' será true
 # neste caso, 'hash.has_value?(56)' será true
 # neste caso, 'hash.has_value?("texto 01")' será false
```

Has_key? / Include?

Has_Value e Include deverão ser utilizados em condições, e não há diferença entre eles. Eles determinam se a Hash tem como representante de um valor, o texto ou número especificado.

Exemplo:

```
if hash.has_key?("texto 01")
  # significa que algum valor tem como representante: "texto 01"
```

```
esle
  # significa que nenhum dos valores da Hash têm "texto 01" como
representante
  # em outras palavras, 'hash["texto 01"]' não existe
end
```

Tópico 2.5 – Ragexp

Ragexp foi previamente explicado na aula 02 como "Expressões Regulares". Ragexp é um tipo de valor semelhante às strings, porém são mais complexas. Ragexp é usado principalmente nos scripts de mensagens, para diferenciar os comandos de mensagens do resto da mensagem (ex: \c[2], \g, etc.). Diferente das strings, para utilizar Ragexp é utilizado barras inclinadas para direita ('/') entre o texto (ex: '/Meu nome é DarkChocobo/'). O editor de scripts do RPGMaker XP não diferencia Ragexp em uma cor diferente (provavelmente bug =P) , porém, o editor do RPGMaker VX diferencia, facilitando (para alguns) seu uso. Uma letra 'i' ou 'm' logo após uma Ragexp pode determinar uma opção a mais, alterando assim sua função.

Exemplo:

```
# apesar das 3 expressões abaixo terem o mesmo texto escrito,
# serão interpretados de forma diferente

/Meu nome é DarkChocobo/
/Meu nome é DarkChocobo/i
/Meu nome é DarkChocobo/m
```

OBS: Ao utilizar uma destas letras ('i' ou 'm') logo após uma Ragexp no editor do RMVX você perceberá que esta letra também ficou da cor roxa.

A letra 'i' torna a Ragexp uma expressão que não diferencia letras maiúsculas de minúsculas.
A letra 'm' ativa o modo "multiple-run", que faz com que novas linhas sejam tratadas como caracteres normais, como um ponto. ('.')

Tópico 2.6 – For

For é uma estrutura de repetição muito usado principalmente em Windows, através dela é possível fazer uma repetição alterando um determinado valor quantas vezes forem necessárias.
Um bom exemplo é a janela de status dos heróis no menu do jogo. Para que apareça cada um dos heróis na janela é feita uma repetição, é desenhado os atributos de cada herói (ou seja, a repetição é feita 4 vezes), mas a dava vez que é repetido, um valor é alterado, para que não seja desenhado o mesmo herói as 4 vezes.
Para iniciar uma estrutura for utilize a palavra 'for', o nome de uma variável local qualquer (será declarada automaticamente), a palavra 'in', e por último, uma range ou uma array já declarada. A cada vez que a estrutura se repetir, a variável local que você escolheu, terá o valor de um dos valores da array ou da range. A estrutura é automaticamente quebrada quando a variável local já tiver se equalizado a cada um dos valores da array ou da range.
Para um melhor compreendimento desta estrutura, crie um novo script acima de todos

os outros (para que ocorra primeiro), e então crie uma estrutura for como a do exemplo abaixo:

```
for i in 0 ... 3
  print i
end
```

Agora teste o jogo, e veja o resultado, tendo em mente a função do comando 'print' (abrir uma janela padrão do Windows mostrando o valor de 'i'). Aparecerá 4 vezes a janela, cada vez será mostrado um valor diferente, sendo eles: 0, 1, 2 e 3. Isso aconteceu porque a cada vez que a estrutura se repete, 'i' tem um valor diferente, neste caso, os valores 0, 1, 2 e 3. Com já foi dito, também podemos utilizar uma array no lugar da range, se ao invés de '0 ... 3' fosse utilizado a array.

Tópico 3 – Funções Exclusivas do RGSS

Ao implantar a linguagem Ruby no RPGMaker XP, a Enterbrain incluiu algumas funções extras para facilitar a criação dos jogos, assim nasceu o RGSS. Neste tópico, estudaremos estas funções extras, adicionadas à biblioteca do RGSS (RMXP). Como sabemos, o RGSS (RMXP) tem algumas diferenças quando comparado ao RGSS2 (RMVX), porém as funções extras do RGSS foram mantidas na criação do RGSS2, logo, esta lição vale para ambos RGSS e RGSS2.
Estas funções do RGSS poderão ser chamadas a qualquer momento em qualquer script de todo o jogo, podendo ser comparados com métodos globais (aqueles que são declarados fora de qualquer classe e/ou módulo, podendo assim ser chamados de qualquer parte do jogo).
As principais funções do RGSS são as seguintes: return, eval, exit, open, p/print, rand, printf e save_data/load_data.

Return

Return é o comando utilizado para sair de um método, da mesma forma que se usa o comando "Parar Evento" para mais nenhum de seus comandos serem executados.

Exemplo:

```
def update_buy_selection
  @status_window.item = @buy_window.item
  if Input.trigger?(Input::B)     # Se apertar o botão 'B'
    Sound.play_cancel     # Executa o som de cancelar
    @command_window.active = true     # Altera o valor de várias
variáveis
    @dummy_window.visible = true
    @buy_window.active = false
    @buy_window.visible = false
    @status_window.visible = false
    @status_window.item = nil
    @help_window.set_text("")
    return     # E finalmente é executado o retorno, significa que
mais nenhum dos commandos será utilizado
  end

[...]          # o método é muito grande, não coloquei todo ele aqui
```

O comando return também pode ser utilizado seguido de um valor, neste caso, o valor determinado irá retornar ao local de onde foi chamado o método.

Exemplo:

```
def método
  return 5
end

variável = método

# variável terá o valor de '5', pois 'método' foi subtituido pelo
valor de retorno 5
```

OBS: Note que o comando 'return', quando vem seguido de um valor de retorno, faz com que o comando de chamar o método seja substituído pelo valor do retorno.

Note que eu posso colocar qualquer valor no retorno:

Exemplo:

```
def método
  retorno_1 = 5
  retorno_2 = 20
  return retorno_1 + retorno_2
end

variável = método + 50
# variável terá o valor de '150'
```

Você também pode retornar mais de um valor, utilizando vírgulas:

Exemplo:

```
def método
  return 12, 64
end

valor_1, valor_2 = método

# valor_1 será 12
# valor_2 será 64

def método
  valor = 0
  valor += 5
  valor *= 4
  return valor
  valor *= 5
  valor -= 50
end

variável = método

# variável será 20
```

Eval

Comando pouco utilizado, deve ser utilizado sempre seguido de uma string. Faz com que o conteúdo da string se transforme em um comando (como se tivesse apenas retirado as aspas).

Exemplo 1:
```
def método
  return 5
end

variável = eval "método"

# variável terá o valor de 5
```

Exemplo 2:
```
$scene = eval ("Scene_Title" + ".new")
```

Exemplo 3:
```
def método
  eval "return"
end
```

Exit

Este é um comando extremamente simples, não tem muito que explicar. Ele simplesmente feicha o jogo. Como teste, crie um novo script acima de todos os outros e coloque simplesmente um 'exit'. Ao testar o jogo, o jogo irá feichar automaticamente, pois o exit está no começo dos scripts do jogo. Este é um comando utilizado eventualmente no caso de erros no jogo.

Open

O comando 'open' servirá para carregar arquivos da pasta do jogo, como arquivos gráficos (pictures) e de Áudio. Este comando pode não ser muito útil pelo fato de o próprio RGSS já conter comandos padrões para este tipo de uso (para isso serve o módulo Cache, mas vocês vão aprender isso somente em aulas futuras).
Para utilizar o comando 'open' chame-o com o respectivo diretório do arquivo numa string entre parênteses logo após do nome do comando. (extensões de nome dos arquivos são necessários)

Exemplo:
```
@picture = open("Graphics\\Pictures\\Arquivo.png")

ou

@picture = open("Graphics/Pictures/Arquivo.png")
```

OBS: Note que tanto '/' quanto '\' podem ser utilizados para separar as pastas do diretório, porém, quando é utilizada a barra inclinada para esquerda, é necessário duas barras, pôs '\' é um dígito utilizado para comandos pelos RPGMakers, logo, é necessário duas barras para o reconhecimento da mesma.

p/print

Vocês já devem saber para que servem os comandos 'p' e 'print', mas mesmo assim vou explicar:
Ambos os comandos 'p' e 'print' criam uma janela de pop up padrão do Windows para exibir um determinado valor.
A única diferença entre eles é que o comando 'p', exibe o valor exatamente como ele é,

diferente do comando ‘print’, que exibe o valor na forma de string (ou se preferir: ‘exibe o valor na forma que ele seria exibido no jogo’).

Exemplo:
```
print “String”
p “String”

print 1_000_000
p 1_000_000

print [1, 2, 3]
p [1, 2, 3]

print /Ragexp/
p /Ragexp/
```

OBS: Crie um novo script acima de todos os outros e coloque o exemplo acima depois dê um teste no jogo, e veja na prática a diferença dos comandos ‘p’ para ‘print’.

Rand

Este é um comando bem simples, e muito utilizado em RGSS. Ele cria um número aleatório qualquer em forma de integer entre 0 e um número máximo definido entre parênteses logo após comando.

Exemplo:
```
variável_aleatória = rand(255)

# o valor de ‘variável_aleatória’ poderá ser qualquer um entre 0 e 255.
```
Se o valor máximo for definido como ‘0’ ou não for definido, o comando retorna um número float (decimal) aleatório de 15 dígitos.

Printf

Este é um comando extremamente complexo, e confesso que nem eu sei exatamente usá-lo. Sua função é facilitar a substituição de valores contidos dentro de uma string por outros valores também em string. Um bom exemplo é o chat da batalha do RPGMaker VX, é utilizado sempre as mesmas frases (ex: “Fulano atacou o monstro!!”) porém, algumas palavras podem ser adicionadas/alteradas, como o número do dano e o autor da ação, por exemplo. Estes diálogos da batalha do RPGMaker VX são feitos através do comando ‘printf’, que permite alterar algumas palavras dentro das strings.
Para utilizar o comando é preciso colocar determinados valores entre parênteses, o primeiro valor é a mensagem principal, e os outros são os valores que serão substituídos. Você deverá colocar caracteres como ‘%’ e ‘#’ por exemplo, para serem substituídos pelos valores entre parênteses, porém não entendo exatamente como funcionam.

Exemplo:
```
sprintf("%#b", 10) # => "0b1010"
sprintf("%#010x", 10)  # => "0x0000000a"
sprintf("%#05x", 10) # => "0x00a"

sprintf("%.*s", 5, "foobar") # => "fooba"

sprintf("%s", [1, 2, 3])      # => "123"
sprintf("%p", [1, 2, 3])      # => "[1, 2, 3]"

printf("%d", n & ~(-1 << n.size*8))
```

Outro Exemplo:

```
# A linha abaixo foi retirada da classe Window_PlayTime do RMXP, linha
33

text = sprintf("%02d:%02d:%02d", hour, min, sec)
```

save_data/load_data

Estes dois comandos são utilizados especialmente para salvar e carregar jogos salvos. O comando 'save_data' salva um determinado valor em um arquivo em qualquer diretório dentro da pasta do jogo. Para usá-lo utilize dois valores entre parênteses: o valor que deseja salvar, e o diretório a salvar.

Exemplo:

```
save_data(5, "Arquivo")
```

Crie um novo script acima de todos os outros, coloque o exemplo acima e teste o jogo, logo depois saia do jogo e olha na pasta do jogo, haverá um novo arquivo chamado "Arquivo". Note que este novo arquivo não têm uma extensão de arquivo. Agora altere o comando para 'save_data(5, "Arquivo.rxdata")' ou 'save_data(5, "Arquivo.rvdata")' dependendo do amker que estiver utilizando. Após o teste de jogo haverá um novo arquivo na pasta, com nome "Arquivo", mas agora com a extensão do maker.

OBS: Não é obrigatório o uso da extensão rxdata/rvdata para a leitura do arquivo.
O comando 'load_data' carrega um arquivo salvo por meio do comando 'save_data', para usá-lo basta colocar o diretório do arquivo entre parênteses.

Exemplo:

```
load_data("Arquivo")

ou, no caso do teste anterior:

load_data("Arquivo.rxdata")
load_data("Arquivo.rvdata")
```

Você pode observar, que nas classes Scene_Save(RMXP) e Scene_File(RMVX) os arquivos são salvos e carregados de forma diferente, é utilizado o módulo Marshal (que será estudado em aulas futuras, assim como Cache).

Tópico 3.1 Classes Internas do RGSS

Além dos scripts que aparecem na lista de scripts do editor, também existem mais scripts internos do RGSS, por exemplo: Cache, Bitmap, Window, Sprite, entre outros. Alguns deles estão expostos em suas respectivas páginas no arquivo help, outros não podem ser consultados, porém eles podem ser editados. Para editá-los, é utilizado o procedimento padrão para mexer em qualquer classe ou módulo ('class Nome_da_Classe' ou 'module Nome_do_Módulo').
Cada uma destas classes são essenciais para que o jogo rode corretamente, por exemplo: 'Input' permite a entrada de teclas do teclado/joystick, sem ele, os scripts não iriam saber quando o jogador aperta uma tecla; outro bom exemplo é o da classe 'Bitmap', que permite exibir imagems, sem esta classe, não seria possível a exibição de nenhum tipo de imagem. As funções de cada uma das classes internas do RGSS serão explicadas

mais detalhadamente em aulas futuras.
Segue abaixo a função básica das principais classes internas:

```
Classes Padrões do Ruby:
Array: Permite o armazenamento de vários valores em uma única variável.
Exception: Possui todas as informações necessárias para utilização de
uma exceção.
FalseClass: Pseudo-valor 'false'. (visto na aula 02)
Hash: Permite o armazenamento de vários valores em uma única variável.
File: Classe que permite o acesso aos conteúdos externos do jogo.
(arquivos da pasta)
MatchData: Gerencia o uso de Expressões Regulares. (Permite o uso de
Ragexp)
Class: Superclasse de todas as classes. Possui as informações
necessárias para o uso de uma classe.
NilClass: Pseudo-valor 'nil', utilizado para quando uma variável não
tem qualquer valor.
Numeric: Superclasse de todas as classes que possuem um valor numérico.
Integer: Classe que permite o uso de Números Inteiros. (..., -3, -2, -
1, 0, 1, 2, 3, ...)
Bignum: Integers (números inteiros) de valor extremamente grandes.
Fixnum: Integers (números inteiros) que não são Bignums.
Float: Classe que permite o uso de Números Racionais. (..., 0.8, 0.9,
1.0, 1.1, 1.2, ...)
Range: Valor utilizado geralmente em estruturas 'for'.  Armazena
vários valores em uma ordem.
Proc: Utilizado para salvar uma variável, e chamá-la quando for
necessário.
Ragexp: A expressão regular utilizada em Ruby.
String: Permite o uso de valores escritos (texto).
Symbol: Permite o uso de expressões símbolo.
Time: Controla o tempo (tempo real) utilizado em scripts como "Tempo
Real".
TrueClass: Pseudo-valor 'true'. (visto na aula 02)

Classes Internas do RGSS:
Bitmap: Exibe uma imagem, geralmente utilizado pelas classes Sprite e
Plane.
Color: Classe utilizada para mudar as cores principalmente de Bitmaps
de Textos.
Font: Permite o acesso às fontes do Windows para poder exibir textos.
Plane: Semelhante à Sprite, com poucas diferenças.
Rect: Utilizado para selecionar uma área do bitmap.
Sprite: Gerencia as imagens em geral exibidas pelo jogo.
Table: Utilizado como arrays multidimensionais (3D), arrays comuns não
suportam tantos dados.
Tilemap: Gerencia os tiles do mapa.
Tone: Permite a mudança de tom de cor da imagem. (geralmente utilizado
em battlers e characters)
Viewport: Permite uma imagem ficar exibida por cima de uma outra.
Window: Possui todas as informações necessárias para exibir uma
janelinha padrão do jogo.
RGSSError: Gerencia os erros padrões que possam ocorrer nas classes
internas do RGSS.
RPG::Sprite: Add-on de Sprite. Tem a mesma função que Sprite, porém,
mais efeitos. (somente RMXP)
RPG::Weather: Permite o uso do comando "opções de clima" do RPGMaker.
(somente RMXP)

Módulos Padrões do Ruby:
Comparable: Permite o uso de condições de comparações. (==, >, <,
```

etc.)
Enumerable: Contém informações necessárias para o desempenho correto de algumas classes.
Errno: Contém informações sobre os erros que podem ocorrer nos scripts.
FileTest: Permite verificar se um determinado arquivo realmente existe.
Kernel: Superclasse de Object, que é a superclasse de todas as classes.
Marshal: Facilita a leitura de arquivos do jogo. (utilizado principalmente em 'load' e 'save')
Math: Permite a utilização de Floats.

Módulos Internos do RGSS:
Audio: Gerencia os arquivos de áudio.
Graphics: Gerencia tudo que é exibido na tela, além da contagem de tempo.
Input: Gerencia a entrada de teclas do teclado/joystick.
RPG: Gerencia todas as outras informações do jogo, que não são scripts. (mapas, database, etc.)
RPG::Cache: Facilita a leitura de arquivos gráficos do jogo. (no RMVX, se chama apenas 'Cache')

Aula 6 – Métodos e Classes

Índice de Tópicos:

Tópico 1.0 - Classes

O que são Classes?
Como foi dito no início das aulas, Ruby é uma linguagem interpretada por objetos, as Classes são estes Objetos. A função das variáveis é alojar determinados valores, qualquer valor que possa ser representado por uma variável é uma classe.

Definido uma Classe
Classes são na verdade Constantes, variáveis fixas com determinados dados em si, portanto cada classe deve ter seu nome começando com uma letra maiúscula. Assim como as variáveis, as classes também devem ser declaradas, para criar ou alterar uma classe é utilizado o comando 'class' seguido do nome da classe e logo depois das alterações feitas naquela classe utiliza-se um 'end' para definir o fim das alterações feitas naquela classe.

Exemplo:
```
classe Nova
end
```

Como não existe nenhuma classe chamada Nova, eu estou declarando-a. Note que se for utilizando o método 'p' em 'Nova' antes de sua declaração (desta forma: 'p Nova') correrá um erro, já que a constante 'Nova' ainda não foi declarada, porém, se for utilizado este mesmo comando após a declaração da classe, o programa irá detectar a constante e irá exibir "Nova" na janelinha, pois o que foi enviado ao método Nova foi apenas a constante Nova.
Somente com a classe Nova não iremos conseguir muita coisa, vamos criar uma nova variável logo após a declaração da classe nova, chamarei-a de 'classe_nova', mas você pode nomeá-la como bem entender.

```
class Nova
end
classe_nova = Nova.new
```

Utilizando o método 'new' eu fiz com que 'classe_nova' se transformasse em um novo objeto da classe Nova, ou uma outra Nova.

OBS1: Métodos serão vistos ainda nesta aula. OBS2: O método 'new'(assim como alguns outros métodos básicos) é provido à todas as classes automaticamente.

Para que fique mais claro o que eu fiz acima vou dar um exemplo:
```
ary = [2, 4]
str = "texto"
```

Acima eu criei duas variáveis, cada um como um novo objeto das classes Array e String; ou seja, eu defini as duas novas variáveis como novas classes Array e String sem alterar as constantes. Foi o mesmo que eu fiz em 'classe_nova = Nova.new'. A única diferença é que Array e String são classes padrões do Ruby e por isso podem ser criadas desta forma mais fácil, porém, agora não à diferença alguma entre Array, String e Nova. Abaixo, mais um exemplo:

```
ary = Array.new
str = String.new
```

Acima eu criei uma nova Array e uma nova String, se você utilizar o método 'p' para verificar os valores de 'ary' e 'str' os retornos serão: '[]' e ' "" '; pois serão uma Array e uma String recém criadas, sem nenhum valor nelas, logo, o exemplo acima seria o mesmo que utilizar o seguinte código:

```
ary = []
str = ""
```

OBS: Note que, nestes dois exemplos acima, se você utilizar o método 'print' ao envéz de 'p' não será exibido nada, pois 'print' lê os dados da forma como o programa irá exibir no jogo, e um texto vazio (String vazia, "") e um grupo de valores sem nenhum valor (Array vazia, []) não são nada.

Agora que já expliquei a função do método 'new' e a diferença entre a constante e sua utilização, podemos voltar à nossa nova classe, a 'Nova'. Vou citar novamente o exemplo para recuperarmos a concentração no mesmo:

```
class Nova
end
classe_nova = Nova.new
```

Se você utilizar o método 'p' em 'classe_nova' após sua declaração irá obter o seguinte resultado: '#'(ou algum semelhante), isto ocorre porque tudo que nós fisemos até agora foi criar uma nova variável com o valor de Nova, porém, a classe Nova está totalmente vazia, por isto o programa lê a classe desta forma.

Tópico 2.0 - Métodos

Métodos são o que fasem as classes funcionarem, sem eles a programação não seria possível. Basicamente os métodos são como comandos que fasem as classes funcionarem, sem eles as classes seriam inúteis, assim como sem as classes os métodos perderiam grande parte de seu "poder". Cada método tem sua função específica, que pode ter um grande e importante papel no seu jogo, assim como pode também ser algo bem simples. Assim como as variáveis, os métodos precisam ser declarados, geralmente se utilizam métodos dentro das classes, mas também podem ser utilizados fora das classes. Métodos declarados dentro das classes pertencem única e exclusivamente àquela classe, se você deseja utilizar este método a partir de outra classe deverá copiar o método e colar nesta outra classe.
Para declarar um método você deve utilizar a palavra 'def', seguida do nome do método que você está criando, e depois do conteúdo do método um 'end' para definir o fim do

método. O nome do método deve ser constituído de apenas letras minúsculas e underlines(_).

Exemplo:

```
def nome_do_método
end
```

Para chamar um método já declarado, deve-se chamar simplesmente pelo nome do mesmo, no caso do exemplo acima, para chamar o método declarado deveria ser utilizado a simples expressão "nome_do_método". Quando um método é chamado, ele irá executar todos os comandos que estiverem dentrod ele.
Exemplo de uso de um método: (Você copiar o exemplo abaixo para seu editor e testar o jogo para ver o que acontecerá)

```
def aumentar_variavel
variavel += 20
end

variavel = 10
aumentar_variavel
p variavel # => 30
```

No exemplo acima, foi executado o comando 'variavel += 20' quando foi chamado o código 'aumentar_variavel', logo, subentende-se que quando um método é chamado todo o código que estava dentro dele fosse para o local onde o método foi chamado. Métodos também podem chamar outros métodos dentro de seu código, mas nunca podem chamar a si mesmos, neste caso ocorreria um erro.
Exemplo:

```
def aumentar_variavel
variavel += 20
aumentar_mais_ainda
end
def aumentar_mais_ainda
variavel += 30
end

variavel = 10
aumentar_variavel
aumentar_mais_ainda
p variavel # => 90
```

Se você não entendeu o porquê de 'variavel' ter o valor de 90 no final do código, vou esplicar mais detalhadamente o que aconteceu: (Se entendeu, pode pular esta parte de verde)

Primeiro, foram declarados os métodos 'aumentar_variavel' e 'aumentar_mais_ainda', 'aumentar_variavel' irá aumentar o valor de 'variavel' em 20 e depois irá chamar o método 'aumentar_mais_ainda' que, por sua vez, aumenta em 30 a 'variavel', logo, o método 'aumentar_variável' terá a função de aumentar 'variavel' em 50. Depois, 'variavel' foi declarado como '10' (Se esta linha não existir, haverá um erro ao chamar um dos métodos, pois 'variável' não estará declarada ainda) Então é chamado o método 'aumentar_variavel', que aumenta o valor de 'variavel' em 50, passando seu valor para

60 (porque 'variavel' já tinha o valor de 10, e 10 + 50 = 60). E em seguida é chamado o método 'aumentar_mais_ainda', que aumenta o valor de 'variavel' em 30, passando seu valor para 90.

Substituição de Métodos
Se você tantar declarar um método já declarado, o método anterior será apagado, e o novo ficará no lugar.

Exemplo:
```
def aumentar_variavel
variavel += 20
aumentar_mais_ainda
end
def aumentar_mais_ainda
variavel += 30
end

def aumentar_variavel
variavel -= 20
end

variavel = 10
aumentar_variavel
aumentar_mais_ainda
p variavel # => 20
```

Quebra de Métodos/Retorno
Você pode "quebrar" a execução de um método utilizando o comando "return". Quando "return" for executado, nenhum código que vir depois dele num mesmo método será executado.

Exemplo:
```
def aumentar_variavel
variavel += 20
if variavel == 30
return
end
variavel += 40
end

variavel = 10
aumentar_variavel
p variavel # => 30
```

No exemplo acima, o comando "return" que está no método 'aumentar_variavel' será executado se o valor de 'variavel' for igual a 30, como 'variavel' foi declarada como 10 e antes da condição o valor é aumentado em 20, a expressão condicional retornará true e assim a condição será executada, então o comando 'return' fará com que a execução do método seja "quebrada" e assim, o comando 'variavel += 40' não será executado. Se você alterar o valor da declaração de 'variavel', a condição retornará true, e assim, o comando 'return' não será executado, e sim o comando 'variavel += 40'.
O comando "return" não serve somente para quebrar a execução do método, mas também servirá para retornar um determinado valor que for utilizado juntamente com ele. Como você já aprendeu sobre retorno na lição sobre condições, retorno é quando o

comando é substituido pelo valor de retorno, no caos dos métodos, o comando de chamar o método será substituido pelo valor de retorno.

Exemplo:
```
def vinte return 20 end variavel = vinte p vatiavel # => 20 p vinte # => 20
```

É claro que, como o método vinte será substituído por '20', eu poderia utilizá-lo de várias formas, como por exemplo, esta:

```
def vinte
return 20
end
variavel = 50 + vinte
```

Outro exemplo de retorno:
```
def metodo1
return "Dark"
end

def metodo2
return "Chocobo"
end

var1 = metodo1 + metodo2
var2 = "Dark" + metodo2
var3 = metodo1 + "Chocobo"
var4 = "Dark" + "Chocobo"
var5 = "DarkChocobo"
```

Todas as 5 variáveis(var1, var2, var3, var4 e var5) terão um mesmo valor em comum, a string "DarkChocobo".

Eviando Valores aos Métodos
Também é possível enviar determinados valores ao chamar os métodos. Alguns métodos necessitam de alguns valores para serem executados corretamente. Para que um o necessite de um valor você deve, ao declará-lo, declarar também variáveis locais dentro de um parênteses, logo após o nome do método(as para isto apenas nomeie as variáveis separando-as por vírgulas).

Exemplos de métodos que necessitam de um ou mais valores:
```
def metodo1(valor)
end

def metodo2(valor1, valor2, valor3)
end
```

OBS: Cada método pode necessitar de quantos valores for necessário.

Para chamar um método que precisa de algum valor, você deve determinar este valor logo após chamar o método (separados por um espaço); você pode usar qualquer tipo de valor. Quando um método necessitar de mais de um valor, você deve usar vírgulas para selará-los.

Exemplos:

```
metodo1 1
metodo1 "string"
metodo1 [1, "array"]
metodo2(1, 2, 3)
metodo2(0, "texto", ["array"])
```

OBS: Note que em alguns exemplos foram usados parênteses e em outros não, não é obrigatório o uso de parênteses ao enviar os valores aos métodos, porém, usando-os, você deixa o seu script mais organizado, por isto é muito comum encontrá-los em exemplos como este.

Quando um valor é enviado para um método desta forma, o método faz com que as variáveis locais que você declarou tomem o valor dos valores enviados ao método.

Exemplo:

```
def meu_nome(str)
p str + "Chocobo"
end

meu_nome("Dark")
```

No exemplo acima, a mensagem que será exibida é "DarkChocobo", pois ao chamar o método 'meu_nome' enviando a string "Dark", você fará com que o método 'meu_nome' seja executado considerando que 'str' seja "Dark", e assim 'str + "Chocobo"' será "DarkChocobo".

Outro exemplo de envio de valores aos métodos:

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

ary1 = [1, 2, 3, 4]
ary2 = [5, 6, 7, 8]
p juntar_arrays(ary1, ary2) # => [1, 2, 3, 4, 5, 6, 7, 8]
```

Aula 7 - Sprites

A classe Sprite representa uma imagem, ela possui vários métodos e propriedades.

```
@var = Sprite.new(viewport)
@var.bitmap = RPG::Cache.picture("nome") # RPG Maker XP
@var.bitmap = Cache.picture("nome") # RPG Maker VX
```

No exemplo acima foi considerado o sprite como uma picture, porém ele pode ser um ícone, um character, etc. Tudo que for gráfico e estiver no jogo.
O argumento viewport não é obrigratório, ele está implícito.

Tópico 1.0 - Métodos

```
bitmap => Cria a imagem.
disposed? => Retorna true se a imagem foi deletada.
update => Atualiza a imagem.
widht/height => Respectivamente, largura e altura da imagem.
```

Tópico 2.0 - Propriedades

```
viewport(view) => Indica o viewport que a imagem será criada.
visible => true/false para exibir a imagem.
x/y/z => Respectivamente, posição X, Y e prioridade.
zoom_x/zoom_y => Respectivamente zoom na coordenada X e Y.
angle = 0~360 => Angulo da imagem.
mirror => true para inverter a imagem.
opacity = 0~255 => Transparencia da imagem.
color/tone => Respectivamente a tonalidade e cor.
blend_type = 0/1/2 => Tipo de imagem, normal, multiplicar e inverter
```

Há também os comandos:

```
wave_amp (RGSS2)
wave_length (RGSS2)
wave_speed (RGSS2)
wave_phase (RGSS2)
```

Estes comandos dão uma espécie de onda na imagem, a imagem fica ondulada, como é o caso nas batalhas.

```
wave_amp = n => Amplitude da onda.
wave_lenght = n => Frequencia da onda.
wave_speed = n => Velocidade do movimento.
wave_phase = 0~360
```

A classe Sprite é bem útil, ele é usada em menus,batalhas, windows, etc. Esta classe é uma das que voce pode usar de qualquer forma, além de possuir uma grande quantidade de métodos, teste modificar no Scene_Title a sprite do título, colocando esses métodos e efeitos de onda, voce pode consultar outros scripts do maker.

Aula 8 - Plane

A classe Plane é capaz de realizar movimentos em imagens, tanto vertical como horizontal. Esta classe, assim como Sprite dependem da Bitmap, portanto para criar a imagem, usamos o método Bitmap, e depois definimos como quisermos, Sprite, Plane, etc.

```
@var = Plane.new
@var.bitmap = Cache.picture("nome")
```

Tópico 1.0 - Métodos

dispose => Apaga a imagem.
disposed? => Verifica se a imagem está apagada, retorna true se ela estiver.
bitmap => Criar imagem.
viewport => Refere-se ao viewport da imagem, a camada.
ox -= 1 => O comando principal, ele move a imagem na coordenada X.
oy += 3=> o comando principal, ele move a imagem na coordenada Y.
z = 55 => Prioridade.
opacity = 90 => Opacidade da imagem.
zoom_x = 1.2 => Zoom na coordenada X.
zoom_y 0.5 => Zoom na coordenada Y.
blend_type => 0: normal - 1: Multiplicar - 2: Inverter
tone => Refere-se a tonalidade da imagem.
color => Refere-se a cor da imagem.

OBS: Note que estes exemplos que usam números, são só exemplos!

Tópico 2.0 - Exemplos

Você pode tomar como exemplo o da aula do Bitmap. Eu vou citar outro bem simples. Vá no Scene_Title e procure no def create_title_graphic e na linha:

```
@sprite = Sprite.new
```

Esta linha indica que a variável @sprite é um Sprite, porém bastar trocar por isto:

```
@sprite = Plane.new
```

Teste o jogo e voce verá que nada mudou...
Bem, o legal vem agora! Vá no def update e depois do super escreva:

```
@sprite.ox -=2
@sprite.oy += 2
```

Agora teste e veja que a imagem começa a subir na direção direita, a velocidade é o 2, significa que a cada frame a imagem sobe 2 pixels e vai para direita 2 pixels.

Basta usar sua criatividade, este comando é bem útil, e dá um efeito muito bom em menus, por exemplo.

Aula 9 - Bitmap

A classe **Bitmap** serve para nós criamos coisas relacionadas a imagens e gráficos, não necessariamente imagens, esta aula fala sobre o comando Bitmap, porem usa como exemplos Sprite e Plane, que serão estudados mais a frente.

OBS 1: O Sprite e Plane, podem ser classificados como parentes proximos da Bitmap, porque eles são digamos que, filhos dela, herdam alguns métodos.

Basicamente, uma imagem pode ser carregada, editada, e deletada com códigos simples e fáceis, é só saber usar os métodos.

Vamos começar com o sprite. O que é um Sprite? São 'filhas' da classe Bitmap, elas são cópias dela, a imagem propriamente dita.

Tópico 1.0 - Sintaxe

```
@var = Sprite.new
@var.bitmap = Cache.picture("nome_do_arquivo")
```

OBS 1: A ultima parte só é válida para o **VX**, que usa o método **Cache.picture**.
OBS 2: Cache.picture é apenas um exemplo, porque além dele, existe também: Cache.animation, Cache.battler, Cache.face.... Para ver é só ir ao módulo 'Cache'.
OBS 3: Não precisa informar o formato da imagem.

Tópico 2.0 - Métodos

z = integer => prioridade da imagem, quanto maior, mais visivel ficará.
dispose => apaga a imagem.
width e **height** = integer => muda o comprimento e altura respectivamente.
blur e **radial_blur** => dão uma distorçao na imagem.
clear => limpa a imagem.
draw_text(x, y, largura, altura, text) => escreve um texto. (text = string)

Esses são os mais usados, há outros também, como o rect, disposed?, blt... Consule-os no arquivo anexado do RGSS.

Aula 10 - Rect

Nesta aula, vou falar sobre o comando Rect, que serve para criar retângulos. Vamos lá! Lembrando que vamos usar uma window como exemplo.

Para criar um retângulo, use o código:
```
@variavel = Rect.new(x, y, largura, altura)
```

Para preencher o retângulo, usamos:
```
self.contents.fill_rect(rect, cor)
```

Ou seja, rect significa a variável em que esta o rect, que no caso é @variável. E cor significa a cor do rect, pode ser Color.new(red,green,blue) ou pelas cores do Window_Base: normal_color, system_color, crisis_color,etc...

Ou, desta forma:
```
self.contents.fill_rect(x, y, largura, altura, cor)
```

Neste método, você não precisa criar uma variável pro rect, o comando já cria um rect com os dados que você fornece.

Podemos também usar o **gradient_fill_rect**, que da um efeito bem legal de gradiente, ele mistura duas cores e forma um gráfico:
```
self.contents.gradient_fill_rect(rect, cor1, cor2)
```

OBS: Desta forma, nós precisaremos da variável que representa o Rect.

Ou, da outra forma, que não precisa de variáveis:
```
self.contents.gradient_fill_rect(x, y, largura, altura, cor1, cor2)
```

Vou dar um exemplo de um rect com gradiente:

```
1 class Window_Jao < Window_Base
2   def initialize
3     super(10, 10, 100, 90)
4     self.contents.draw_text(0,30,50,29,"Barra")
5     self.contents.fill_rect(0,0,80,30,Color.new(0,0,0))
6     self.contents.gradient_fill_rect(0,0,80,30,crisis_color,system_color)
7   end
8   def refresh
9     self.contents.clear
10  end
11 end
```

Neste exemplo, o código cria a janela e o rect. Apenas faça isso e crie um evento com o Código: `Window_Jao.new`

Tentem atribuir um valor para o rect, por exemplo: @largura = 50 * @width / 100.
Onde 50 e um valor qualquer e width é a largura do rect, e / 100 significa porcentagem.
É assim que funcionam as barras de HP/SP, basta ver o cálculo na Window_Base.

Aula 11 - Viewport

Índice de Tópicos:

A classe Viewport é muito importante, ela "fatia" a imagem, ou seja, definimogs qual o tamanho da imagem, altura, largura e posição x/y. Com ela, podemos fazer muitas coisas interessantes, principalmente com sprites e planes.

Tópico 1.0 - Sintaxe

```
@viewport = Viewport.new(X, Y, Largura, Altura) # assim
@viewport = Viewport.new(rect) # ou assim
```

Todas as formas são iguais, só muda que a segunda precisa de uma variável que representa Rect.

Tópico 2.0 - Métodos

dispose => Apaga o viewport.
disposed? => Se a imagem estiver apagada, retorna true.
flash(cor, tempo) => Define o flash e o tempo que ocorrerá no viewport, se cor for nil, o viewport irá desaparecer no flash.
visible => Mostrar o viewport (true) e não mostrar (false).
tone/color => Tom e cor do viewport.
z => Prioridade do viewport, se múltiplos objetos usam a mesma prioridade, o mais recente terá mais prioridade.

Tópico 3.0 - Exemplo

Vamos agora falar do que interessa, na verdade há várias formas de usar este comando, eu vou usar no Title, por isso, crie um novo jogo e vá ao Scene_Title. Vá no "**def create_title_graphic**" e substitua-o por isto:

```
def create_title_graphic
@v1 = Viewport.new(0, 0, 260, 416)
@v2 = Viewport.new(284, 0, 260, 416)

@sprite2 = Plane.new(@v2)
@sprite2.bitmap = Cache.system("Title")

@sprite = Plane.new(@v1)
@sprite.bitmap = Cache.system("Title")
end
```

Agora vá no def update e logo abaixo do super, escreva:

```
@sprite.ox += 2
@sprite2.ox -= 2
```

Se você testar, vai ver que ficou uma coisa bem louca, basicamente viewports fazem isso. Porém, eu fiz uma demonstração bem básica do poder que eles têm. Os viewports fazem basicamente isso: Cortam o objeto com o tamanho que voce decidiu altura e largura, e colocam-no na posição x e y. Para atribuir um viewport a o objeto, é só fazer como eu disse:

```
@objeto = Plane.new(@var_do_viewport) # Lembrando que Plane é um
exemplo.
```

O interessante seria misturar tudo como eu fiz, por exemplo, coloquei viewports e Planes no script, que dá um efeito muito interessante, não acha? Principalmente se voce mesmo fizer uma imagem bem legal, imagina os efeitos que o viewport e plane podem fazer!

Aula 12 - Windows

Começaremos a estudar sobre as Windows, também chamadas de janelas, praticamente são a base dos scripts visuais. Ou seja, sua importância é muito grande. Na figura abaixo, vamos identificar janelas. A partir daqui começaremos a fazer nossos scripts, por mais simples que sejam, mostram que vocês evoluíram.

Vamos aprender como criar uma janela básica.

Primeiramente devemos saber o que é herança. Em uma de nossas aulas falamos um pouco sobre isso. Para criar uma janela devemos fazer herança. Neste caso será com a Window_Base.

```
class Teste < Window_Base
def initialize
super(x,y,altura,largura)
refresh
end
def refresh
self.contents.draw_text(0,0,self.width,self.height,”texto”)
end
end
```

Note que há dois métodos na classe, ou seja, as janelas normalmente utilizam um, mas para organizar usamos dois. Por exemplo, o def initialize serve para nós organizarmos as variáveis e opções da janela, e o refresh serve para digitarmos o texto.

Toda vez que quisermos criar uma janela, temos que usar herança, sem ela não dá! Portanto geralmente usamos a Window_Base como pai da nova classe.
Toda window tem que ter a seguinte base:

```
class Seu_Nome < Window_Base
def initialize
super(10,10,100,200)
end
end
```

No super, apenas substitui as variáveis de posição x, posição y, largura e altura por números quaisquer. A base da janela são textos, ou seja, usaremos muito o comando draw_text.

```
self.contents.draw_text(4, WLH * 0, 92, WLH, Vocab::weapon1)
```
(Exemplo extraído da classe Window_Equip do VX.)

Em outras aulas, vimos que alguns métodos necessitam de valores, também chamados de argumentos, devemos seguir esta regra, o método draw_text requer 5 argumentos (que são separados por virgula); posição x, posição y, largura do texto, altura do texto e texto. Onde texto deve estar dentro de aspas, ou seja , uma string.

Bom, vamos começar então a usá-las, de uma forma bem simples, mas que dá um efeito bem legal de início. Crie uma classe e escreva o código:

```
class Window_Jão < Window_Base
  def initialize
    super(10,10,300,60)
    self.contents.draw_text(0,0,300,30,"Digite seu texto aqui!")
  end
end
```

Logo após, vá ao script Scene_Map e no def start coloque antes do end:

```
@jao = Window_Jão.new
```

No def terminate coloque depois do @message_window.dispose:

```
@jao.dispose
```

Finalmente, no def update depois do @message_window.update:

```
@jao.update
```

Isso faz com que automaticamente a janela entre no mapa sem ter que chamarmos, e fica infinitamente lá. Teste o jogo e veja a window.

Antes de prosseguir, um detalhe, como vimos, na CLASSE da janela, usamos o self. Mas se usarmos em outro script, como no scene map, veremos que vai dar erro, porque o self indica a classe em si, e o RGSS não vai saber diferenciar, portanto tudo que formos fazer com a janela, fazemos na sua VARIÁVEL.

Agora chegaremos a uma parte interessante, o que dá vida a janela, os métodos. Citarei alguns, pois são diversos, estes métodos são da Window_Base, caso queira ver todos seus métodos, basta consultá-la e ver os seus def's.

No def initialize, logo abaixo do super, coloque os códigos.

```
self.opacity = 150           # opacidade da janela - 0~255
self.visible = true          # visibilidade da janela
self.back_opacity = 10       # opacidade do interior da janela
self.contents_opacity = 150  # conteúdo da janela (texto por exemplo)
```

Esses métodos, são os responsáveis por alterar a estética da janela, e se você ver no Window_Base, ela é filha da Window que é uma classe interna do ruby, ou seja ela está embutida lá. Portanto, toda janela criada que possua herança da base poderá ter estes métodos. Em suma:

```
self.opacity = 0-255 indica a opacidade total da janela (exceto o seu
conteúdo)
self.visible = true/false indica se a janela esta ou não visível
self.back_opacity = 0-255 indica a opacidade do interior, e não da
borda da windowskin
self.contents_opacity = 0-255 indica a opacidade de todo o conteúdo da
janela
```

Lembrem-se de que você pode pegar qualquer def (método) da Window_Base e jogar nessa janela, é muito aconselhável você fazer isso.

Podemos concluir que:

1. self.contents_opacity = 0 é igual a: self.visible = false porque ambos estão nulos. A diferença é que a opacidade total pode-se alterar entre 0 e 255 e a visible é totalmente visível, ou não.
2. Vimos que para criar janelas, precisamos de um super, que indica sua posição x, posição y, largura e altura;
3. Precisamos de herança para que a janela seja criada;
4. Colocar métodos do script Window e Window_Base;
5. Não usar o self num outro script, e sim criar uma variável;
6. E aprenderemos mais sobre elas na próxima aula, que vai ser sobre Scenes, que envolverá as janelas.

Aula 13 - Scenes

O que são Scenes?

Scenes são na tradução bruta, um conjunto de windows, em que há interação entre elas.

Para que eu uso elas?

Bom, por exemplo, a Scene_Menu é um conjunto de janelas, que podemos interagir para escolher o que fazer, acessar itens,habilidades,status,salvar,etc... As scenes servem para organizar e facilitar o trabalho dos scripters.

Tá, mas o que eu tenho que fazer?

Na verdade você tem que criar uma Window, e na Scene chamá-la através de uma variável correspondente a Window.

Bom, para iniciar é preciso dizer que Scenes é o que chamamos de classe artificial, porque não muda em nada o nome do script ser Scene_Menu ou Cena_Menu (lembrando que se o nome for Cena_Menu teremos que chamá-lo sempre com este nome), Scene suponhamos que seja só para orientar.

Vamos usar Scenes, ou seja, quase todos scripts que usam interação com Windows, animações, etc são Scenes, lembrando que há outros nomes – no RGSS o nome da classe não importa -, como você já deve saber, nos orientamos pelo script, pelo seu nome – ou nome da class -, portanto, tenha muito cuidado com isso, pode ser que, ele não tenha nome de Scene, mas seja uma Scene ou vice-versa.

Como disse, Scenes é um nome que só serve para diferenciar ou classificar scripts.

Para usar uma Scene, tomarei base o RGSS2 que usa uma classe Base, a Scene_Base, uma coisa que no XP não existia. Se você estiver usando o VX e quiser criar uma Scene, use a herança < Scene_Base.

Depois crie um def initialize para organizar dados parciais, como declarar uma variável para Windows, depois vem o método main, onde tudo ocorre na Scene, há vários outros, como o update, não sei dizer ao certo se os métodos 'main,initialize e update' são "oficiais" ou são inventados para classificar como o nome Scene e Window.

Há certos casos que precisamos usar o update para por exemplo, animar uma Scene ou atualizar após algum método ser chamado.
Eu costumo fazer em meus menus uma animação, que servirá de exemplo para esta aula...

Vá no def update, tomemos como exemplo a variável time, que se refere a janela de tempo;

```
@time.y -= 5
    if @time.y <= 366
      @time.y = 366
    end
```

Nela, o código `@time.y -= 5` significa que ela subirá na posição y em velocidade 5, e se a posição y for menor ou igual a 366, ela para em 366, assim como todas outras, isso serve para que se ela for maior/menor que 366, ela não continue subindo eternamente pela tela, e pare na posição 366y.

Aula 14 - Comandos de Windows

Para começar, toda Window herda métodos das windows: Command,Selectable ou Base. Segue a ordem:
Window > Window_Base > Window_Selectable > Window_Command

A classe "Window" esta oculta no RGSS. A window_base serve para qualquer uso de windows, tanto para textos, imagens,etc, exceto comandos e seleções, por isso existe a Selectable para selecionar itens, como os scripts Scene_Item e Skill. E a Command, serve logicamente para criar comandos pelas janelas, como é o caso da Scene_Title, End e a janela de comandos do Menu.

Como diferenciá-las?
Simples, a **Window_Base** tem tudo, menos comandos e seleções, a **Selectable** geralmente tem muitas seleções, como é o caso dos itens, que são diversos, já a **Command** apresenta comandos específicos, por exemplo, você cria a janela pelo código:

```
@var = Window_Command.new(largura,["comando1","..."])
```

Já as outras, geralmente são criadas por classe, mas podem ser criadas por variáveis (isso pode ser um pouco complexo, mas não é, basta saber usar argumentos).

Não vou comentar nas minhas aulas sobre a Selectable, pois não tem muito pra se falar. Nesta vou começar falando da **Base**, mostrando os comandos. Vamos lá!

Logo de cara, no inicio do script, você vê o **def initialize**, e um código dentro dos parênteses, isso é chamado argumento, são códigos que precisam ser especificados quando chamamos uma classe. Então deduzimos que para chamar a Window_Base precisamos especificar a posição X e Y e a largura e altura da janela.

Há também comandos como '**self**.', isto indica que a ação será feita na própria classe, neste caso a Window. O exemplo: **self.back_opacity = 200** indica que a opacidade da janela será 200.

Uma pergunta, por que especificar com o self e não usando variáveis? Simples, porque se usássemos uma variável, teríamos que usar em toda classe filha da base, o self indica que a ação ocorrerá com a classe em si, e não com a classe em um devido momento, visto que podemos usar a mesma Window em várias Scenes, como no caso a **Window_Help**.

Tá, chega de enrolação, vamos para os métodos!

O método draw_icon e novo no VX, infelizmente no XP não havia. Para desenhar um ícone na Window, basta usar: `draw_icon(numero do ícone, x, y)`. O numero do ícone significa que você deve especificar o ícone, porque como vimos, o ícone é uma imagem só, feita com vários ícones, o RPG Maker automaticamente divide os ícones, então teremos que contar o ícone.

O método draw_face, também novo no VX, não existe no XP, ele desenha uma face

qualquer, desde que especificada: `draw_face(face_name, face_index, x, y)` face_name indica o nome do arquivo, que deve ser expresso em aspas,face_index é o id da face, vide o numero do ícone, e o X e Y nem precisa explicar.

O método draw_character desenha o gráfico do personagem, `draw_character(character_name, character_index, x, y)`. A mesma explicação do draw_face.

O método `draw_actor_hp(actor, x, y)` Desenha o gráfico do HP do herói, lembrando que a variável actor deve ser especificada (eu não sei como é no VX, mas no XP é: `actor = $game_actors[id]`).

O método `draw_currency_value(value, x, y, width)` serve para você escrever o valor exato e atual de uma certa coisa, no caso é para Gold, veja que o Gold se refere a `$game_party.gold`

Por fim, há o mais conhecido, o draw_text, ele deve ser feito com o self.contents: `self.contents.draw_text(x, y, largura, altura,"texto")`

Há também o **opacity**, que indica a transparência da janela, podemos usar tanto o self como uma variável.

OBS: self é uma pseudo-variável, você não pode usar self em uma Scene, porque não esta especificando qual janela você quer alterar. Em uma Scene, devemos usar o **opacity** na variável que retrata a janela.

O **z** indica a prioridade, quanto maior, mais prioridade terá.

E por fim, o **visible = true/false** indica que quando false, a janela não será visível, e quando true, será.