Gobstones

Programas:

program{

}

Comandos: Mover, Poner, Sacar

Procedimientos

procedure xxxxxxx(){

}

Los nombres de los procedure se escriben todos juntos con mayúscula cada palabra. Ejemplo: PonerVerdeYAzul, Poner3Rojas. Luego va el ( ) cerrado.

Y le das program para que se ejecute

Sentencias: **repeat** → van todo en minúsculas

Se le pueden pasar parámetros que van entre paréntesis y separados por , coma

Cada parámetro puede estar compuesto por expresiones por ejemplo decir PonerN(5,color) o se puede decir PonerN(3+1+1, color) ósea que dentro de los parámetros puede haber cuentas.

La alternativa condicional de gobstones

if (CONDICIÓN de que pase tal cosa){

ACCIÓN que va a suceder

}

→ LA CONDICIÓN TIENE QUE SER VERDADERA

Esta es la sintaxis de if en Gobstones

Si se le agrega un **else**, también se abre corchete y se pone la otra ACCIÓN que tiene que realizar. La acción a realizar si lo de if no es verdadero

Funciones

Son la herramienta para definir expresiones complejas. Parecido a los procedure pero sirve para encapsular expresiones. Se escribe:

function nombreDeLaFuncion () {

return (lo que tiene que hacer)

}

son un caso particular de las expresiones, y por lo tanto siguen las mismas reglas que ellas: se escriben con la primera letra minúscula y siempre denotan algún valor

en la última línea de su definición siempre va un return, seguido de una expresión entre paréntesis: el valor que la función va a retornar.

Java Script

FUNCIONES

function cosa_para_hacer(parametro){

return loquehayquehacer; ⟶ El return sin **()** y con **;** al final

}

LOS PROCEDIMIENTOS NO RETORNAN NADA. NO VA RETURN

Operadores de java **===, >=** , **>** , **<**, **<=**

Operadores para todo tipo de datos **!==** , **===**

Negación en Java: **!**

Adición **&&** ambas condiciones verdaderas Amigas Analía && Silvana

Disyunción **||** se cumple si alguna es verdadera Amigas Analía|| Silvana||Sole

Strings es cuando queremos que el programa tome la palabra literal. Se pone a la palabra **“ ”**

Variables: por ejemplo let. Van antes de la function.

Cuando declarás una variable tenés que darle un valor inicial, lo cual se conoce como **inicializar** la variable.

```
Atajos:  x += y; //equivalente a x = x + y;
x *= y; //equivalente a x = x * y;
x -= y; //equivalente a x = x - y;
x++;   //equivalente a x = x + 1;
```

Alternativa condicional

```
if (unNumero >= 0) {
    return unNumero;
  } else {
    return -unNumero;
  }
}
```

En Java la alternativa condicional si tiene que retornar algo se escribre

IF (CONDICIÓN){

RETURN ACCIÓN SI SE CUMPLE LA CONDICION

}

ELSE {

ACCIÓN SI NO SE CUMPLE

}

Listas

Una lista se escribe de la siguiente manera en JS

let [elemento, elemento, elemento] ejemplo let pertenencias = [capa, espada, barra] Lista pertenencias

Algunas cosas para hacer con las listas: **longitud, agregar, remover**, **posicion**. Son ordenes que se le da a lista. Va la orden y el nombre de la lista () Ejemplo: agregar(pertenencias , cobre)

Así como existe una función para averiguar en qué posición está un elemento, también puede ocurrir que queramos saber lo contrario: qué elemento está en una cierta posición.

Para averiguarlo podemos usar el operador de indexación, escribiendo después de la colección y entre corchetes [] la posición que queremos para averiguar:

> mesesDelAnio[0]

"enero"

> ["ese", "perro", "tiene", "la", "cola", "peluda"][1]

"perro"

For es la orden para recorrer una lista.

For (let [elemento] of [lista]){

}

podemos guardar muchos elementos de un mismo tipo que representen una misma cosa

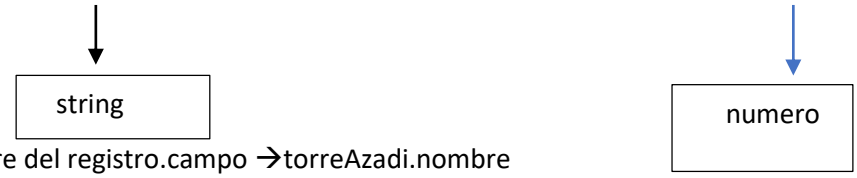
REGISTROS

Los registros en JS son como listas que adentro tienen campos

Por ej: let torreAzadi = { nombre: "Torre Azadi", locacion: "Teherán, Irán", anioDeConstruccion: 1971 };

registro vamos a guardar información relacionada a una única cosa (por ejemplo un **monumento o** una **persona), pero los tipos de los campos pueden cambiar.**

let torreAzadi = { nombre: "Torre Azadi", locacion: "Teherán, Irán", anioDeConstruccion: 1971 };

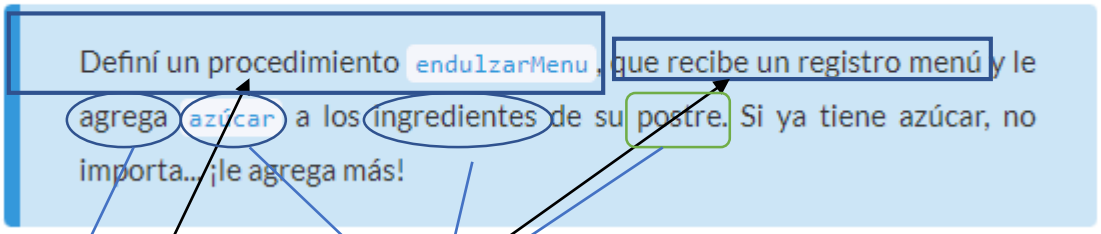


Se accede a los campos poniendo el nombre del registro.campo →torreAzadi.nombre

Para cambiar los campos ponemos registro.campo = lo que cambiamos. → torreAzadi.nombre = “casa babel”

Se pueden tener registros que tengan como campos listas y listas que tengan como campos registros

Para terminar, trabajemos una vez más con los menúes.



Function endulzarMenu (registroMenu){

Agregar(registromenu.postre.ingrediente, “azucar”)

}

Ruby

Crear objetos:

module Objeto -- el objeto se escribe con mayuscula

end

Los mensajes que entiende el objeto pueden escribirse de diferente forma:

`comer_lombriz!` Puede hacer algo y modifica su estado

`energía` nos muestra una característica del objeto

`débil?` Los mensajes que devuelven booleanos terminan con `?`

Para definir un método, se escribe asi:

def self. Metodo

end

Por ejemplo para que Pepita baile y cante:

module Pepita

end

def self. bailar!

end

def self.cantar!

end

Por ahora esos métodos están vacíos. Solo están definidos pero no hacen nada

Se le pone end a las definiciones de método y a los objetos.

Para establecer una característica del objeto se pone asi:

module Objeto

@energia = 100

```
end
```

Luego algúna orden que le demos al objeto puede hacer que este haga algo en su característica. Por ejemplo si pepita vuela su energía baja en 10 puntos:

```
module Pepita
  @energia = 100

  def self.volar_en_circulos!
    @energia = @energia - 10
  end
end
```

Si las ordenes al objeto pueden realizar mas de un cambio en sus características las ordenes se ponen una debajo de la otra:

```
module Pepita

   @energia = 100

   @ciudad = Iruya

def self.volar_hacia!(destino)

   @ciudad= destino

   @energia -=100

end

end
```

Si queremos consultar como es el atributo del Objeto no basta solo con inicializar el atributo, hay que definir un método para que cuando lo consultemos el objeto nos devuelva la cantidad que tiene de eso.

Por ejemplo para consultar la energía o la ciudad no basta con poner @propiedad sino que también hay que hacer un método que defina que el objeto nos la dice:

```
module Pepita
  def self.energia
    @energia
  end
end
```

Estos métodos se llaman ACCESSORS, son métodos de acceso a los atributos.

ALTERNARIVA CONDICIONAL

La alternativa condicional en Ruby se escribe.:

```
if self. Pregunta que le hacemos al objeto?

  self. Orden que le damos

end
```

ejemplo

```
if self. Débil?

self.comer_alpiste! 10

end
```

Si se le agrega else

if self. Pregunta

self. Lo que hace si es verdadero

else

self. Lo que hace si es falso

end del else

end del if

Pueden tener if encadenados, en ese caso se usa **elsif**

if self. Condición que debe cumplir?

self. Lo que hace si es verdadero

elsif Otra condición que puede cumplir

self lo que hace si es verdadera la segunda condición

else

self.lo que hace si todo es falso

end del else

end del elsif

end del if

Cuando modelamos un objeto le damos atributos. Por ejemplo

Module Pepita

@energia = 100 → este es el atributo

end

Estos atributos pueden ser consultados, definiéndoles un método:

module Pepita

@energia =100

end

def self.energia

@energia → esto es el metodo

end

end

HAY MENSAJES QUE SOLO MODIFICAN LOS ATRIBUTOS SON LOS **SETTERS**

Si por ejemplo el atributo es energía y queremos modificarlo se tiene que definir un método que lo modifique. Con está sintaxis:

```
def self.energia=(nueva_energia)
@energia= nueva_energia
End
```

Entonces cuando en un método usemos @enegia, estaremos usando la nueva_energia.

Para los getters, que sirven para obtener el valor de un atributo, usamos el mismo nombre que este.

Para los setters, que sirven para fijar el valor de un atributo, usamos el mismo nombre que este pero con un = al final.

Para enviar un mensaje a un objeto se hace poniendo **. mensaje**

Ejemplo**:** saludo.upcase

Equal? Nos dice si dos objetos son el mismo

== nos dice si dos objetos son equivalentes es decir si representan la misma cosa

Por ejemplo “hola”== “hola” son dos objetos distintos pero que representan la misma cosa

LISTAS

Son objetos que contienen otros objetos adentro.

Se definen dándole el nombre con @nombre [elemento1, elemento2, elemento3]

Para poder acceder a ella se define como cualquier objeto

```
Def self. Nombre
@nombre
End
```

Mensajes que se le pueden enviar a las colecciones

Se escribe: Nombre de la lista . mensaje variable

Por ejemplo teniendo la lista se escribiría

```
numeros=[4, 5,6,7]
numeros.push 8
```

Push agregar

Delete quita un elemento

Include? Consulta si un elemento esta incluido devuelve booleano

Size consulta la cantidad de elementos que hay en la lista. Devuelve un nro.

SETS

Otro tipo muy común de colecciones son los sets (conjuntos), los cuales tienen algunas diferencias con las listas:

no admiten elementos repetidos;

sus elementos no tienen un orden determinado.

Una lista se transforma en set poniendo el mensaje to.set

Bloques

Los bloques son objetos que representan un mensaje o una secuencia de envíos de mensajes, sin ejecutar, lista para ser evaluada cuando corresponda

La palabra que los define es proc.

Sintaxis de los bloques:

Nombre del bloque = proc {referencia = referencia y lo que querramos hacer}

Ejemplo del bloque incrementador.

un_numero = 7

incrementador = proc { un_numero = un_numero + 1 }  en este bloque la función es la de sumar 1 nro al numero referenciado

Los bloques no devuelven nada sino se les pasa el mensaje final .call

incrementador = proc { un_numero = un_numero + 1 }.call

LOS BLOQUES PUEDEN RECIBIR ARGUMENTOS. LOS ARGUMENTOS VAN || Y PUEDEN SER MAS DE UNO SEPARADOS POR COMAS.

EJ

jugar_a_timba = proc{|minutos| TimbaElLeon.jugar!(minutos/60)}

Mensajes que reciben los bloques

Select. select recibe un bloque con un parámetro que representa un elemento de la colección y una condición booleana como código, y lo que devuelve es una nueva colección con los elementos que la cumplen

ejemplo

algunos_numeros = [1, 2, 3, 4, 5]

mayores_a_3 = algunos_numeros.select { |un_numero| un_numero > 3 }

find encuentra un elemento que cumpla la condición

all? Devuelve un bool si todos los elementos cumplen

map El mensaje map nos permite, a partir de una colección, obtener otra colección con cada uno de los resultados que retorna un envío de mensaje a cada elemento.

Count nos dice cuántos elementos de una colección cumplen la condición.

Sum nos suma la cantidad de elementos que cumplen la condición

Each hace que cada elemento cumpla con la condición que le mandamos. Es el único msj que produce efectos en el bloque.

Cuando tenemos muchos objetos que se comportan igual lo que podemos hacer en vez de escribirlos 1 x 1 es generalizar sus comportamientos en una **CLASE**

En las clases se generalizan los comportamientos de los objetos. Los atributos iniciales se escriben así:

def. Initialize →

@atributo

end

initialize. Al trabajar con clases tenemos que inicializar los atributos en algún lugar. El mensaje initialize nos permite especificar cómo queremos que se inicialice la instancia

No va SELF. ya que la clase no hace las cosas solo modela lo que sus integrantes puede hacer

Instanciar objetos: se le dice así a crear nuevo objetos para las clases. La sintaxis es:

Nombre_de_objeto =nombre de la clase.new

Ejemplo: un nuevo celular para la clase celular:

celular_de_maria=Celular.new

initialize puede recibir parámetros que especifiquen con qué valores deseamos inicializar los atributos al construir nuestros objetos. ¡Suena ideal para nuestro problema! Initialize @salud=una_salud

Tanto los objetos bien conocidos (Pepita, Norita, Bouba) como las clases (Zombi, Ave) van con mayúsculas. Cuando las instanciamos pasan a estar en minúsculas, porque ahora son variables de la clase.

pepita= Ave.new

Este es el método para agregar mas cantidad de variables(integrantes) a la clase, por ejemplo la clase caminantes

caminantes = [ ]

20.times{caminantes.push (Zombi.new)}

Los casos en los que un objeto puede conocer a otro son:

Cuando es un objeto bien conocido, como con los que veníamos trabajando hasta ahora

Cuando el objeto se pasa por parámetro en un mensaje (Juliana.atacar bouba, 4)

Cuando un objeto crea otro mediante el envío del mensaje new

CONSTRUCTORES:

Si algunos de los atributos iniciales de nuestros objetos de la clase cambia entre ellos, (por ej la salud de los zombies) no hay problema porque se pueden iniciar con un parámetro. Ej

class Zombi

def initialize(puntos_de_salud) → esto especifica como se construye esta instancia. Es un constructor

@salud = puntos_de_salud

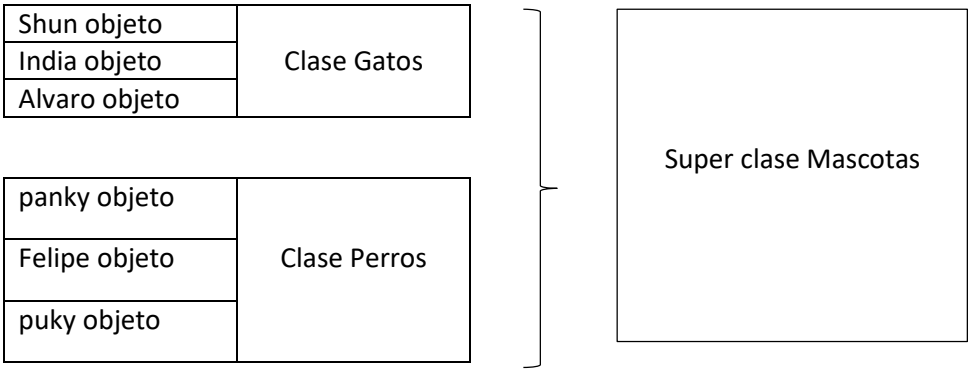
End

Los constructores pueden recibir más de un parámetro. Por ejemplo, si de una Planta no sólo pudiéramos especificar su altura, sino también su especie y si da o no frutos...

jazmin = Planta.new 70, "Jasminum fruticans", true

JERARQUÍA Y HERENCIAS

Es probable que querramos meter dentro de una misma clase algunos objetos que tienen algunos atributos parecidos pero no todos. En la vida real se ve como por ejemplo



Las clases heredan de su clase de mayor jerarquía mediante este símbolo < y con esta sintaxis

class Condor < Ave

Hereda lo que tiene la jerarquía mas alta y también puede **def**inir sus propios métodos

A las superclases nunca las instanciamos. En otras palabras, no creamos objetos con esa clase, solo nos sirven para proveer comportamiento a sus subclases.

Mensaje **super**

```
class Saludo
  def saludar
    "Buen día"
  end
end


class SaludoDocente < Saludo
  def saludar
    super + " estudiantes"
  end
```

endl utilizar **super** en el método de una subclase, se evalúa el método con el mismo nombre de su superclase.

En la consola vemos “Buen día estudiantes”