

Learning to program: **by example**
Jean Lazarou

Table of Content

Table of Content	2
Preface	3
Introduction	4
Step 1: Hide a word and ask for a letter	5
Step 2: Wait for a proposal and check it	9
Step 3: Improve the proposal check	16
Step 4: Ask for more letters	21
Step 5: Ignore empty proposals	25
Step 6: Show the letters that the player finds	28
Step 7: Temporarily put a problem aside	38
Step 8: Split the source code	40
Step 9: Show all the letters that the player has already found	44
Step 10: Recall the wrong proposals	48
Step 11: Declare that the player has won	52
Step 12: Declare that the player has lost	56
Step 13: Draw the gallows	60
Step 14: Support the repeated letters	70
Step 15: Choose a random word	74
Conclusion	85

Preface

This book is an introduction to programming by example. It explains how to create, step by step, a *hangman game* played with a computer.

The hangman game is a game where a player chooses a (hidden) word while a second player tries to find the word.

The second player knows how many letters the word contains. He/she makes guesses of possible letters, one by one. If the letter is part of the hidden word, the first player reveals the position of the letter (if the letter appears several times, all positions are revealed). If the letter is not part of the word, the second player loses one chance.

The first player draws elements of the hanged man, each time the second player proposes a wrong letter, starting with the head, then the torso, then the arms and legs. If the hanged man is complete, the second player loses the game, the man is hanged...

Préface

Ce livre est une introduction à la programmation par l'exemple. Le livre montre comment créer, pas à pas, un jeu du *bonhomme pendu*, joué avec un ordinateur.

Le jeu du bonhomme pendu est un jeu où un joueur choisit un mot caché alors qu'un deuxième joueur essaie de trouver le mot.

Le deuxième joueur sait combien le mot contient de lettres. Il fait des hypothèses sur les lettres, une à une. Si la lettre fait partie du mot caché, le premier joueur montre la position de la lettre (si la lettre apparaît plusieurs fois, il montre toutes les positions). Si la lettre ne fait pas partie du mot, le deuxième joueur perd une chance.

Le premier joueur dessine les éléments d'un bonhomme pendu, chaque fois que le second joueur propose une mauvaise lettre. On commence par la tête, puis le torse, ensuite les bras et les jambes. Si le bonhomme est complètement pendu, le deuxième joueur perd.

About the illustrations

The illustrations showing a *P*, the first letter of programming, and a character hanging from the letter, recalling the hangman, were made by Constantin Lazarou.

À propos des illustrations

Les illustrations montrant la lettre *P* de programmation et un personnage qui y serait pendu, rappelant le bonhomme pendu, ont été faites par Constantin Lazarou.

Introduction

Programming is a way to teach a computer how to do something the way the programmer thinks it should be done. Put another way, programming is the description of how to perform some processing.

The programmer describes the processing by writing some text called source code. The computer reads and analyzes the text to execute what the code says.

The code is made of words, the sequence of words must follow some rules. Some words are special. We say that the text is written in some language, some programming language.

Both the words and the rules may differ depending on the programming language, like spoken languages.

The text follows basic grammar rules, usually much more strict than spoken languages so that computers can easily understand them. The rules are strict and breaking the rules leads to invalid text, up to the point that the computer cannot understand anything. No poetry allowed.

A computer reads the code and executes what the code asks it to do. While reading, it understands some words and word sequences because the code applies the rules defined by the programming language, it recognizes some predefined words and the structure. Usually, programming languages allow to define new words that help writing the code.

Here, we are going to talk to the computer in *Ruby*, one of the numerous programming languages¹. One tool available on the computer, named the *Ruby* interpreter, is understanding the language and handles the code.

In the following sections, we are going to write, step by step, the code that is going to explain to the computer how we want it to play the hangman game.

Introduction

La programmation est un moyen d'apprendre à un ordinateur à faire quelque chose selon la méthode imaginée par le programmeur. En d'autres termes, c'est la description du comment accomplir un certain traitement.

Le programmeur décrit le traitement en écrivant un texte appelé code source. L'ordinateur lit et analyse le texte pour exécuter ce que le code lui *dit*.

Le code est formé de mots, la séquence de mots doit respecter certaines règles. Certains mots sont spéciaux. On dit que le texte est écrit dans un langage, un langage de programmation.

Les mots et les règles peuvent différer d'un langage de programmation à un autre, comme les langues parlées.

Le texte suit des règles grammaticales de base, habituellement beaucoup plus strictes que les langues parlées ainsi les ordinateurs les comprennent facilement. Les règles sont strictes et les briser mène à un texte invalide au point que l'ordinateur ne comprend rien. Pas de poésie.

Un ordinateur lit le code et exécute ce que le code demande de faire. En lisant, il comprend des mots et les séquences de mots parce que le code applique les règles définies par le langage de programmation, il reconnaît certains mots prédéfinis et la structure. Habituellement, les langages de programmation permettent de définir de nouveaux mots pour aider à écrire le code.

Nous allons parler *Ruby* à l'ordinateur, un des nombreux langages de programmation². Un outil disponible sur l'ordinateur, qu'on appelle un *interpréteur Ruby*, est capable de comprendre le langage et pourra traiter le code.

Dans la suite du document, nous allons écrire, pas à pas, le code qui expliquera à l'ordinateur comment nous voulons qu'il joue le jeu du pendu.

¹ In spoken languages, we could say that *Ruby* is Japanese (because the *Ruby* language was invented by a Japanese guy: Yukihiro Matsumoto alias Matz).

² Dans le cas de langues parlées, on pourrait dire que *Ruby* est du japonais (parce que le langage *Ruby* a été inventé par un Japonais : Yukihiro Matsumoto alias Matz).

1

Hide a word and ask for a letter

Cacher un mot et demander une lettre



Step 1: Hide a word and ask for a letter

We start with a few lines of code, each line forms an instruction. The tools we use to write the code highlights different elements of the code, that relates to the language rules, using colors.

 hangman.rb

```
1  hidden_word = "hello"
2
3  puts "_ _ _ _ _"
4  puts
5
6  print "Give me a letter: "
7
8  gets
```

Line 1

The first line contains three elements:

- the word `hidden_word` (two English words seen as one word because of the underscore)
- the equal sign
- a sequence of characters enclosed in double quotes (`"hello"`), called *string*

What does *Ruby* understand?

- the double quotes (the *string*) means that we need a sequence of characters (`"hello"`) that has no meaning for *Ruby* even if it is meaningful for us
- the equal sign means that we want to give a name to the string so that we can later use the name to refer to that string¹
- the name we give to the string is `hidden_word`

What does line 1 mean to us? We decide that the hidden word is *hello* and we give it the name `hidden_word`. Let us repeat the above explanation in another way using a diagram that shows:

- the three types of elements that *Ruby* sees,
- the function of the elements using different colors,
- the naming of the string (`"hello"`) with an arrow,
- that naming something requires the equal sign by using the same color for the arrow and the equal sign.

Étape 1 : Cacher un mot et demander une lettre

Nous commençons par quelques lignes de code. Chaque ligne de code forme une instruction. Les outils que nous utilisons pour écrire le code mettent en évidence différents éléments en utilisant des couleurs selon les règles du langage de programmation.

Mots du code suivant : `hidden` signifie caché, `word` signifie mot, `print` signifie imprimer, `give` signifie donnez, `letter` signifie lettre.

Ligne 1

La première ligne contient trois éléments :

- le mot `hidden_word` (deux mots anglais forment un seul mot grâce au caractère souligné)
- le signe égal
- une séquence de caractères entre guillemets (`"hello"`), appelé *chaîne*

Que comprend *Ruby* ?

- les guillemets signifient que nous avons besoin d'une séquence de caractères (`"hello"`) qui n'a pas de sens pour *Ruby* alors que pour nous il s'agit d'un mot
- le signe égal signifie que l'on veut donner un nom à la chaîne afin que l'on puisse utiliser ce nom plus tard pour y faire référence²
- le nom que nous attribuons à la chaîne est `hidden_word`

Que signifie la ligne 1 pour nous? Nous décidons que le mot caché est *hello* et nous lui donnons le nom `hidden_word`. Reprenons l'explication en utilisant un diagramme qui montre :

- les trois type d'éléments que voit *Ruby*,
- la fonction des éléments par des couleurs différentes,
- l'attribution d'un nom à la chaîne avec une flèche,
- que cette attribution exige le signe égal en utilisant la même couleur pour la flèche et le signe égal.



Line 3

The third line has two elements:

- the word `puts`
- another sequence of characters : `" _ _ _ _ "`

What does *Ruby* understand?

- it recognizes the `puts` word in the *Ruby* parlance, it means: show something to the screen
- the string is for `puts`

We can illustrate it using again a diagram that shows:

- the two elements identified by *Ruby*,
- that the second element is for the word `puts` .

Ligne 3

La troisième ligne a deux éléments:

- le mot `puts`
- une autre séquence de caractères `" _ _ _ _ "`

Que comprend *Ruby* ?

- il reconnaît le mot `puts` , dans le parlé *Ruby*, il signifie: montre quelque chose à l'écran
- la chaîne est destinée à `puts`

Illustrons tout ça avec un diagramme qui montre :

- les deux éléments repérés par *Ruby*,
- que le deuxième élément est destiné au mot `puts` avec une flèche.



The effect is to show on the screen the `" _ _ _ _ "` string that gives information to the player about the hidden word (the number of letters).

L'effet est de montrer à l'écran la chaîne `" _ _ _ _ "` qui donne des indices au joueur à propos du mot caché (le nombre de lettres).

Line 4

The line is very similar to the previous one, the difference is that we give nothing to `puts` in which case the effect is to skip a line.

Ligne 4

La ligne ressemble à la précédente, la seule différence est qu'on ne donne rien à montrer à `puts` ce qui revient à demander de passer une ligne.

Line 6

The 6th line contains:

- the word `print`
- the string `"Give me a letter: "`

Here, instead of `puts` we see `print` followed by a new string (the prompt for a letter). The effect of `print` is similar to `puts` except that, while `puts` shows something and moves to the next line, `print` shows something and stays at the end of *something*.

Ligne 6

La 6^{ème} ligne contient:

- le mot `print`
- la chaîne `"Give me a letter: "` ³

Ici, au lieu de `puts` on a `print` suivi d'une nouvelle chaîne (l'invitation à donner une lettre). L'effet de `print` ressemble à celui de `puts` mais, au lieu de passer à la ligne comme le fait `puts` , on reste juste après ce qui a été montré.

Line 8

The line contains only the word `gets`. The word is part of the *Ruby*'s words. Its effect is to wait for the person, in the front of the screen, to type letters on the keyboard and end with the *enter* key (carriage return). *Ruby* displays on the screen every character that the user types.

As line 8 is the last line there is nothing left to do.

Ligne 8

La ligne contient juste le mot `gets`. Ce mot fait partie des mots *Ruby*. Il a pour effet d'attendre que la personne en face de l'écran utilise le clavier pour introduire des lettres et marquer la fin avec la touche retour charriot ou *entrée*. *Ruby* montre à l'écran chaque caractère tapé au clavier.

Comme c'est la dernière ligne, il ne reste rien d'autre à faire.

Old typewriters had a *carriage*, moving and pulling the paper because the characters hit the paper at a specific place. When the operator wanted to move back on the left of the page, he/she could press a button that returned the carriage back on the leftmost position.

Les vieilles machines à écrire avaient un *chariot* qui bougeait en entraînant le papier, les caractères venaient frapper toujours au même endroit. Pour revenir à l'extrême gauche, on faisait revenir le chariot en appuyant sur une touche, on provoquait un retour charriot.

Let us test

The source code is saved in a file named `hangman.rb`, if we want the computer to read the file and apply what the file asks, we must request: `ruby hangman.rb`.

We see here-after the result, the letter `A` is what we enter after receiving the prompt for a letter.

Testons

Le code source se trouve dans un fichier appelé `hangman.rb`, pour que l'ordinateur le lise et applique ce qu'il demande, on donne l'ordre : `ruby hangman.rb`.

On voit le résultat ci-après, la lettre `A` est ce qu'on lui donne en réponse à la demande de donner une lettre.

```
$> ruby hangman.rb
- - - - -
Give me a letter: A
```

¹ Likewise, when we need to talk about a person we use the name he/she was given.

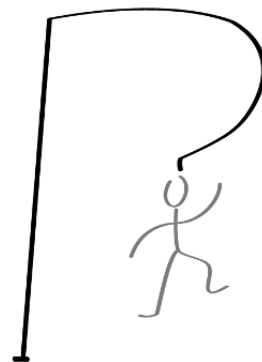
² Nous faisons de même lorsque nous voulons parler d'une personne, nous utilisons le nom qu'il lui a été donné.

³ ce qui signifie "Donne moi une lettre"

2

Attendre une proposition et la vérifier

Wait for a proposal and check it



Step 2: Wait for a proposal and check it

At the previous step, the code ended with `gets` meaning: *wait for some input from the user*, nothing more. Now, we are going to use the input and check if the given letter is part of the hidden word.

Étape 2 : Attendre une proposition et la vérifier

L'étape précédente se terminait par `gets`, c'est-à-dire : *attendre une entrée de caractères de l'utilisateur*, rien d'autre. Nous allons maintenant exploiter cette entrée et vérifier si la lettre donnée appartient au mot caché.

Mots du code suivant : answer signifie *réponse*, if signifie *si*, include? signifie *inclu?*, chomp signifie *mâcher*, is part of signifie *fait partie de*, else signifie *sinon*, invalid signifie *non valable*, end signifie *fin*.

 hangman.rb

```

1  hidden_word = "hello"
2
3  puts "_ _ _ _ _"
4  puts
5
6  print "Give me a letter: "
7
>8  answer = gets
9
>10 if hidden_word.include?(answer.chomp)
>11   puts "The letter is part of the hidden word"
>12 else
>13   puts "Invalid letter"
>14 end

```

Lines 1-6

The code is the same as above. Last line, line 6, is asking to show the prompt message.

Lignes 1 à 6

Ces lignes sont les mêmes que précédemment. La ligne 6 a pour effet de montrer le message invitant à donner une lettre.

Line 8

Here we find something similar to what we saw earlier:

- the word `answer`
- the equal sign
- the word `gets`

Previously, the code effect was to attach a name to a string. Here we give the name `answer` to the result produced by `gets`.

What does it mean?

Because we are using *things* and want somehow to make our life easier, we name them, so that we can recall them (as we would do with people).

When a language interpreter picks a word it tries to do something. What it does depends on the programming language. In *Ruby*, it proceeds as follows:

- when the word is on the left of an equal sign, it decides to use it as a name given to what is on the right-side of the equal sign,
- when the word is isolated (as `gets` here) it searches:
 - if the word is a name that was given to something and *replaces* the name by the value,
 - if it knows the name as something able to perform some action, like `gets` that waits for user's input, it performs the action which usually produces a result. Again, it *replaces* the name with the produced value.

Ligne 8

Il s'agit de code similaire à ce qu'on a déjà vu:

- le mot `answer`
- le signe égal
- le mot `gets`

L'effet précédent était d'attacher un nom à une chaîne. Ici, il s'agit de donner le nom `answer` au résultat produit par `gets`.

Qu'est-ce que cela signifie?

Lorsque nous utilisons des *choses*, nous leur donnons des noms afin de pouvoir les rappeler (comme nous le ferions avec des gens) et ce pour nous faciliter la vie.

Lorsqu'un interpréteur de langage capte un mot il essaie de faire quelque chose. Ce qu'il fait dépend du langage de programmation. Pour *Ruby* :

- lorsque le mot est à gauche d'un signe égal, il décide de donner un nom pour ce qui se trouve à la droite du signe égal,
- lorsque le mot est isolé (comme `gets`, ici) il recherche :
 - si il s'agit d'un nom donné à quelque chose et *remplace* le nom par la valeur,
 - si il connaît le mot comme quelque chose capable d'accomplir une action, comme `gets` qui attend une entrée de l'utilisateur. Si c'est le cas, il accomplit l'action qui produit en général un résultat. Il *remplace* alors le nom par la valeur produite.

Lines 10-14

The elements at line 10 are new:

- the word `if`, a special word in *Ruby*
- the word `hidden_word`
- a dot
- the word `include?`
- a left parenthesis
- the word `answer`
- a dot
- the word `chomp`
- a right parenthesis

Take a breath, it gets complicated...

The `if` word marks a condition, the result of what appears after it is used to take a decision. If the result is right (or `true`) then *Ruby* must use the following lines, all the lines until the `else` word (line 12). If the result is wrong (or `false`) then it skips the lines until the `else` and takes into account everything after the `else` until the `end` word (line 14).

In other words, `if` marks the beginning of a choice between two branches. The branch to follow depends on the truthness of a condition, that is the result of everything after the `if`. The diagram hereafter shows the two branches with the arrows departing from the `if` box.

Lignes 10 à 14

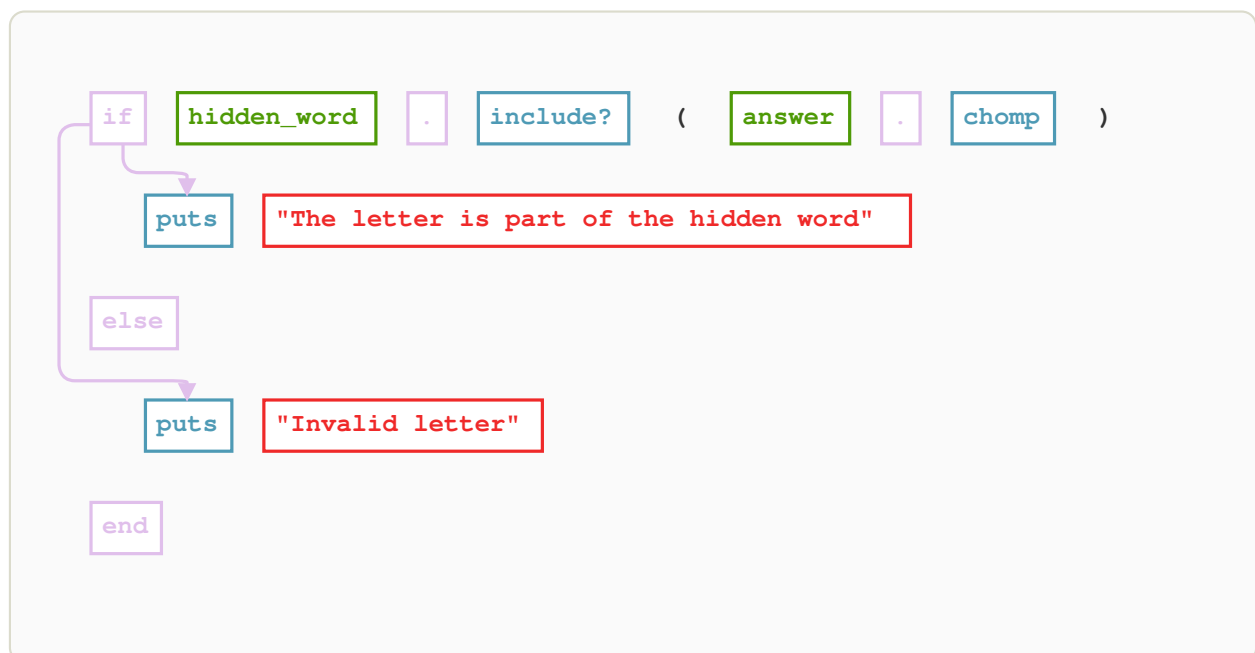
Les éléments de la ligne 10 sont nouveaux :

- le mot `if` est un mot spécial en *Ruby*
- le mot `hidden_word`
- un point
- le mot `include?`
- une parenthèse ouvrante
- le mot `answer`
- un point
- le mot `chomp`
- une parenthèse fermante

Bon. Là, ça se complique...

Le mot `if` marque une condition, le résultat de ce qui se trouve après lui est utilisé pour prendre une décision. Si le résultat est juste (ou vrai), alors *Ruby* doit utiliser les lignes qui suivent jusqu'au mot `else` (ligne 12). Si le résultat est inexact (ou faux), il passe au dessus des lignes qui suivent jusqu'au `else` et traite les lignes après le `else` jusqu'au mot `end` (ligne 14).

En d'autres termes, `if` marque le choix entre deux branches. La branche à suivre va dépendre de la valeur de la condition, c'est-à-dire le résultat de ce qui vient après le mot `if`. Ces deux branches sont mises en évidence dans le diagramme suivant par les flèches qui partent de la boîte `if`.



So, the first thing that appears after the `if` is the word `hidden_word`, *Ruby* finds that the word is the **name** given to a string associated above and uses it.

A dot follows the name. The dot means: send a message to the string (here). The word after the dot, `include?`, is the name of the message. Everything after the message is attached to the message.

Before going further, let us talk about the sending of the message. Say `hidden_word` is someone's name. We send a message named `include?` to that person. When he/she receives the message, he/she is going to treat the message unless he/she does not understand the meaning of the message. The same is true with the string in our example. If the string understands the `include?` message, it is going to perform some action, otherwise the system is going to fail, *Ruby* will complain and stop executing the code.

Let us try to visualize all this with a diagram showing:

- the messages and the receivers in specific colors,
- the arrows symbolizing the sending,
- that *Ruby* spots the message using the dot, by using the same colors.



Afterwards, *Ruby* meets the parentheses, they are optional but make it clear that everything inside the parentheses is the content we want to attach to the message.

Before sending the `include?` message, *Ruby* must evaluate the attachment. The attachment is made of the word `answer`, the name given to user's input (remember the string produced by `gets`). As a dot follows `answer`, the attachment is not the answer, *Ruby* must first send the `chomp` message (the word right after the dot). This time the message has no attachment.

Does a string understand `chomp`? Yes.

When a string receives `chomp` it removes trailing characters provided they are the `enter` characters and produces a resulting string.

Having sent `chomp`, *Ruby* has the attachment to the `include?` message (as shown in the following diagram).

La première chose qui vient après le mot `if` est le mot `hidden_name`, *Ruby* retrouve qu'il s'agit du **nom** associé précédemment à une chaîne et l'utilise.

Le nom est suivi d'un point. Un point signifie : envoi un message à la chaîne (dans ce cas). Le mot après le point, `include?`, est le nom du message. Tout ce qui suit le message est attaché au message lors de l'envoi.

Avant d'aller plus loin, parlons de l'envoi du message. Supposons que `hidden_word` est le nom de quelqu'un. Nous envoyons un message appelé `include?` à cette personne. Lorsqu'elle reçoit ce message elle va le traiter à moins qu'elle ne comprenne pas le sens du message. Il en est de même pour la chaîne dans notre exemple. Si la chaîne comprend le message `include?`, elle va accomplir une certaine action, autrement le système va échouer, *Ruby* va se plaindre et arrêter de traiter le code.

Reprenons tout ça dans un diagramme qui montre:

- les messages et les receveurs par des couleurs,
- les flèches qui symbolisent l'envoi du message,
- que *Ruby* repère les messages grâce au point en utilisant les mêmes couleurs.

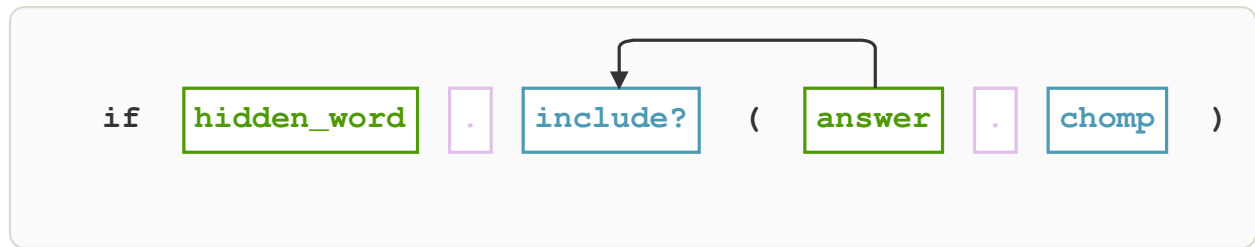
Ensuite, *Ruby* tombe sur les parenthèses, elles sont optionnelles mais permettent de bien baliser le contenu à joindre au message.

Avant d'envoyer `include?`, *Ruby* doit évaluer le contenu à joindre. La *pièce jointe* est faite du mot `answer`, le nom donné à l'entrée de l'utilisateur (souvenez-vous de la chaîne produite par `gets`). Vu qu'un point suit `answer`, la pièce jointe n'est pas cette réponse, *Ruby* doit d'abord envoyer le message `chomp` (le mot juste après le point). Cette fois le message n'a pas de pièce jointe.

Une chaîne comprend-elle `chomp`? Oui.

Lorsqu'une chaîne reçoit `chomp` elle retire les derniers caractères s'ils sont des caractères `enter` (retour chariot) et produit une nouvelle chaîne.

Après l'envoi de `chomp`, *Ruby* a la pièce jointe pour le message `include?` (comme le montre le diagramme suivant).



On receiving the message the hidden-word string checks if the character sequence it receives as attachment appears in itself. It produces a positive or negative result consumed by `if` (see the diagram hereafter). Based on the produced response, *Ruby* moves to line 11 or line 13.

The two possible choices lead to very similar lines, they both show a message...

En recevant le message la chaîne (*mot caché*) vérifie si la séquence de caractères qu'elle reçoit en pièce jointe apparaît dans la sienne. Elle produit un résultat positif ou négatif exploité par le mot `if` (voir le diagramme ci-après). Selon la réponse, *Ruby* passe à la ligne 11 ou 13.

Les deux options possibles mènent à des lignes très proches, toutes deux montrent un message...



Let us test

First, try giving the letter `h` that belongs to the hidden word.

Testons

Commençons par essayer la lettre `h` qui se trouve dans le mot caché.

```

$> ruby hangman.rb
- - - - -

Give me a letter: h
The letter is part of the hidden word

```

We get a positive message as expected. We typed a lowercase letter. What happens if we type an uppercase letter?

Nous recevons un message positif comme prévu. Nous avons donné une lettre minuscule. Que se passe-t-il si on donne une majuscule?

```

$> ruby hangman.rb
- - - - -

Give me a letter: H
Invalid letter

```

Oops! Too bad, it says the letter is not part of the hidden word.

Domage, il répond que la lettre ne fait pas partie du mot caché.

Indeed, it is strict. Did you notice that all the letters in the hidden word are lowercase?

En effet, il est strict. Aviez-vous remarqué que toutes les lettres du mot caché étaient en minuscule?

Another interesting case to test is typing two letters instead of one.

Un autre cas intéressant à tester est de donner deux lettres au lieu d'une.

```
$> ruby hangman.rb
- - - - -

Give me a letter: he
The letter is part of the hidden word
```

It passed, because the effect of `include?` is to check that matching sequences does exist, as does `he` with the beginning of the hidden word.

Of course, typing two letters that are not a subsequence of the hidden word, results in *invalid letter*.

Ça passe parce que l'effet de `include?` est de vérifier qu'il existe une séquence correspondante, ce qui est le cas avec `he` en début du mot caché.

Bien sûr, donner deux lettres qui ne sont pas une séquence du mot caché aboutit à *lettre incorrecte*.

```
$> ruby hangman.rb
- - - - -

Give me a letter: hz
Invalid letter
```

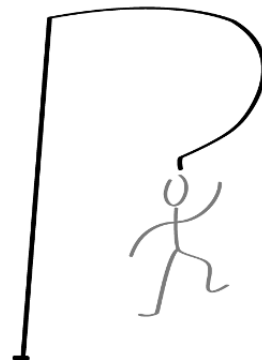
Before moving on, we should first improve the current behavior.

Avant d'aller plus loin, nous allons essayer d'améliorer ce que nous avons pour le moment.

3

Improve the proposal check

Améliorer la vérification des propositions



Step 3: Improve the proposal check

In the previous step we pointed out some unwanted behaviors:

- giving an uppercase letter does not work,
- giving two letters instead of one results in a two-letter comparison.

Let us try to fix the two cases.

From now on, we are not going to give as much details as before, with the hope that the explanation given so far are clear enough...

 hangman.rb

```

1  hidden_word = "hello"
2
3  puts " _ _ _ _ _ "
4  puts
5
6  print "Give me a letter: "
7
8  answer = gets
9  answer = answer.chomp.downcase
10
11 if hidden_word.include?(answer.chomp)
12   puts "The letter is part of the hidden word"
13 else
14   puts "Invalid letter"
15 end

```

We add one more processing at line 9, right after giving a name (`answer`) to the letter that the user proposed.

We send the `chomp` message to remove the *carriage return* character and send the `downcase` message to the produced result. As we said before, every action produces a result to which we can send another message. Here, `chomp` produces a new string to which we send the `downcase` message. The string understands the message and responds by changing all the characters to lowercase letters, producing a new equivalent string with all lowercase letters.

We give the same name `answer` to the received string (we cannot recall the old string anymore).

Now, at line 11, we compare the lowercase letters with the lowercase letters of the hidden word. If we test it again...

Étape 3 : Améliorer la vérification des propositions

A l'étape précédente nous avons constaté quelques comportements indésirables :

- donner une lettre en majuscule ne convient pas,
- donner deux lettres au lieu d'une seule donne lieu à une comparaison des deux lettres.

Essayons de rectifier ces deux cas.

Nous n'allons cependant plus détailler le code de la même façon, en espérant que les explications données jusqu'ici ont été assez claires...

Nous ajoutons un traitement supplémentaire à la ligne 9, après avoir donné un nom à la lettre proposée par le joueur.

Nous envoyons le message `chomp` pour retirer le caractère *retour chariot* et on envoie le message `downcase` au résultat obtenu. Comme nous l'avons dit, chaque action produit un résultat auquel on peut envoyer un message. Ici, `chomp` produit une nouvelle chaîne de caractères à laquelle nous envoyons le message `downcase`. La chaîne comprend ce message et agit en transformant tous les caractères en minuscule, produisant une autre chaîne équivalente avec tous les caractères en minuscule.

Nous donnons le même nom `answer` à cette chaîne (nous ne pouvons plus rappeler l'ancienne chaîne).

Maintenant, à la ligne 11, nous comparons des lettres en minuscules avec les lettres en minuscules du mot caché. Si on essaie à nouveau...

```
$> ruby hangman.rb
_ _ _ _ _

Give me a letter: H
The letter is part of the hidden word
```

The uppercase `H` letter is accepted.

Let us turn to the two-letter case. We must check if the proposal does not contain two letters and reject the proposal.

Cette fois le `H` majuscule est accepté.

Regardons maintenant le cas des deux lettres. Nous devons vérifier que la proposition ne comporte pas deux lettres, pour refuser la proposition.

Mots du code suivant : length signifie longueur, you must signifie vous devez, only signifie uniquement, give signifie donner, one signifie un.

 hangman.rb

```
1  hidden_word = "hello"
2
3  puts "_ _ _ _ _"
4  puts
5
6  print "Give me a letter: "
7
8  answer = gets
9  answer = answer.chomp.downcase
10
11 if answer.length > 1
12   puts "You must give only one letter!"
13 elsif hidden_word.include?(answer)
14   puts "The letter is part of the hidden word"
15 else
16   puts "Invalid letter"
17 end
```

We added a new choice at line 11 that tests the number of characters that the player typed.

After the `if` word, *Ruby* sees `answer` that refers to the proposal. We just turned the proposal to lowercase letters, the proposal is a string to which we send the `length` message. The string processes the message and produces a value equal to the number of character it contains, we also say the *length* of the string.

Afterwards, *Ruby* sees the `>` mathematical sign meaning *is greater than* the value of what follows `>`, namely the value `1`. If the length of the string is greater than 1, the execution path is line 12 (that outputs a message saying that we must give only one letter). Otherwise we must skip line 12.

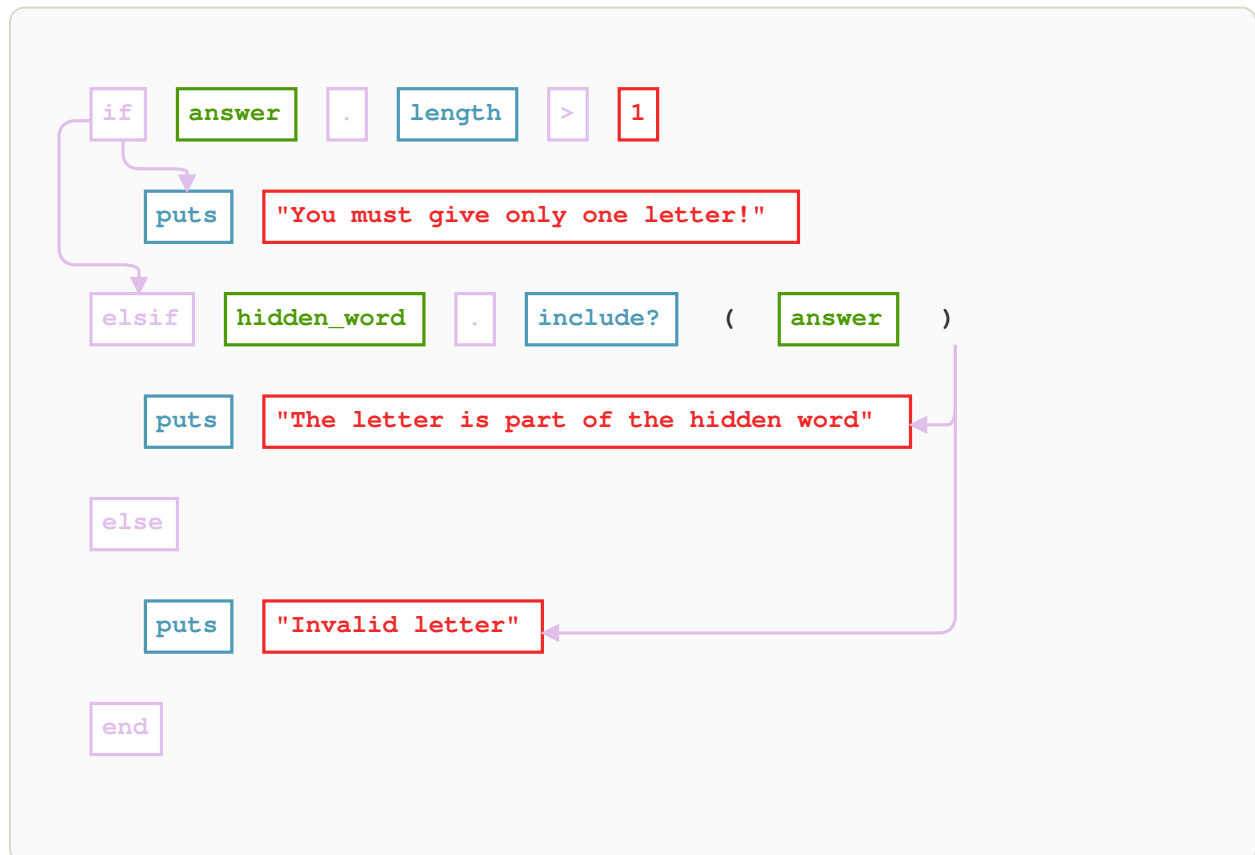
Ruby meets a new word `elsif` (line 13), a word build from *else* and *if*, it means that we specify another choice of what to do next that depends on the result of `hidden_word.include?(answer)`. Follow the arrows in the diagram herafter to see the possible flows.

Nous avons ajouté, à la ligne 11, un nouveau choix qui teste le nombre de caractères donnés par le joueur.

Après le mot `if`, *Ruby* rencontre `answer` qui renvoi à la proposition. Nous venons de la mettre en minuscules, il s'agit d'une chaîne à laquelle nous envoyons le message `length`. La chaîne traite ce message en produisant une valeur donnant le nombre de caractères qu'elle contient, on parle également de *longueur* de la chaîne.

Ensuite, *Ruby* voit le symbole mathématique `>` signifiant *est-ce plus grand* que ce qui se trouve après le `>`, c'est-à-dire le nombre `1`. Si la longueur de la chaîne est supérieur à 1, il faut poursuivre à la ligne 12 (qui donne le message indiquant qu'on ne peut proposer qu'un caractère). Dans le cas contraire il faut passer au-dessus de la ligne 12.

Ruby tombe sur un nouveau mot `elsif` (ligne 13), contraction des mots `else` et `if`, cela signifie qu'on donne un nouveau choix, selon le résultat de `hidden_word.include?(answer)`. Les flèches dans le diagramme suivant montrent les alternatives possibles depuis la boîte `if`.



The expression after the `elsif` word looks like the expression used earlier that checks if the hidden word does contain the proposed letter, without the carriage return character that we just removed.

What happens if we enter two letters?

L'expression après `elsif` ressemble à l'expression utilisée plus haut, qui vérifie si le mot caché contient la lettre proposée, sans le caractère *retour chariot* que nous venons de retirer.

Que se passe-t-il, maintenant, si on donne deux lettres?

```
$> ruby hangman.rb  
- - - - -  
Give me a letter: he  
You must give only one letter!
```

We get what we wanted.

Both annoying cases are solved, let us improve the game.

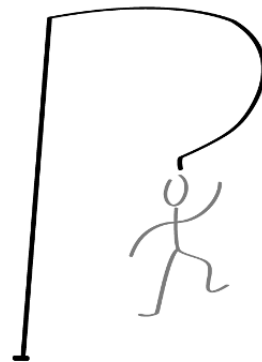
Nous obtenons bien ce que nous voulions.

Les deux cas ennuyeux étant résolus, nous allons perfectionner le jeu.

4

Demander plusieurs lettres

Ask for more letters



Step 4: Ask for more letters

So far, after giving a letter, the game stops. We are going now:

- ask for a letter,
- give a feedback on the proposal,
- ask a letter again,
- and so on...

The player can enter the word *stop* to end the cycle.

Étape 4 : Demander plusieurs lettres

Jusque là, après avoir donné une lettre le jeu s'arrête. Nous allons maintenant faire en sorte de :

- demander une lettre,
- donner une information sur la proposition,
- demander à nouveau une lettre,
- et ainsi de suite...

Pour pouvoir arrêter ce cycle le joueur pourra taper le mot *stop*.

Mots du code suivant : loop signifie faire une boucle, exit signifie sortir.

 hangman.rb

```

1  hidden_word = "hello"
2
3  puts "_ _ _ _ _"
4  puts
5
>6  loop do
7
8      print "Give me a letter: "
9
10     answer = gets
11     answer = answer.chomp.downcase
12
>13     if answer == 'stop'
>14         exit
>15     end
16
17     if answer.length > 1
18         puts "You must give only one letter!"
19     elsif hidden_word.include?(answer)
20         puts "The letter is part of the hidden word"
21     else
22         puts "Invalid letter"
23     end
24
>25 end

```

We added the words `loop do` at line 6 and `end` at line 25.

The `do` and `end` words, that *Ruby* recognizes, surround a sequence of instructions called a *block*.

The effect of the `loop` word is to repeat the execution of all the instructions inside the block¹, without ever stopping.

Next diagram shows a simple snippet of code that would indefinitely display `hello`.

The arrows shows where the block starts, from the `do` word, until the `end` word and `loop` that causes the endless repetition.

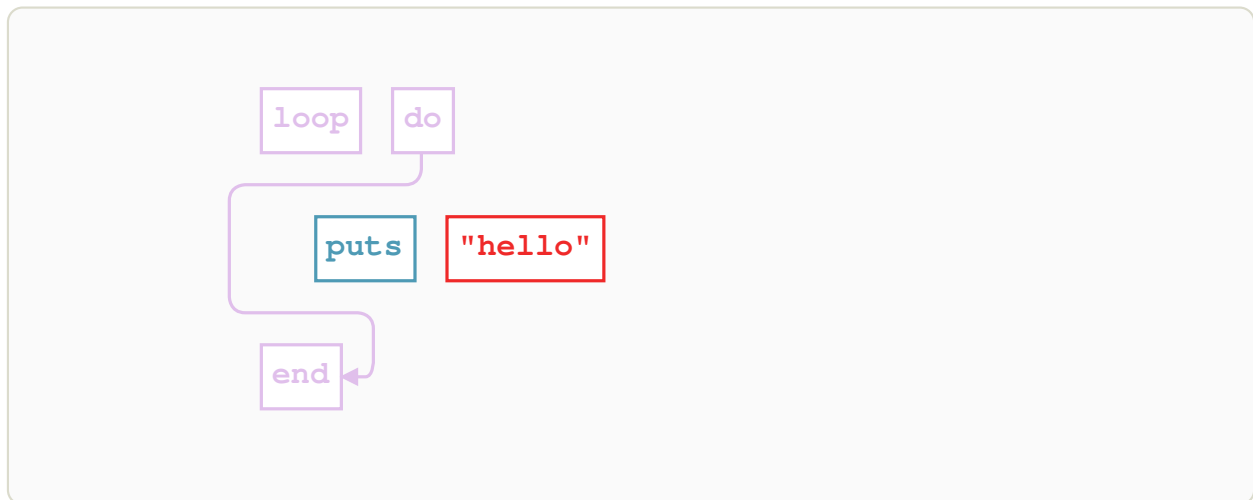
Nous avons ajouté `loop do` à la ligne 6 et `end` à la ligne 25.

Les mots `do` et `end`, reconnus par *Ruby*, encadrent une séquence d'instructions qu'on appelle *bloc*.

Le mot `loop` a pour effet de répéter l'exécution de tout ce qui se trouve à l'intérieur du bloc², sans jamais s'arrêter.

Le diagramme suivant montre un exemple simplifié de code qui afficherait indéfiniment `hello`.

La flèche montre d'où part le bloc depuis le mot `do` jusqu'au mot `end`, `loop` faisant en sorte de répéter le contenu du bloc indéfiniment.



To prevent the game from never stopping, we added the lines 13 to 15.

Before checking if the proposal is valid, the lines check if the player typed the *stop* word, if so, we fall on `exit`. *Ruby* knows the *exit* word that implies to stop everything, we say *exit the program*.

Line 13 shows an ease of writing of *Ruby*, the double equal sign (`==`) is a message sent to the value named `answer`, a string, the effect of the message is to check if its content is the same as the value of what comes after the double equal sign (`'stop'`). Therefore we can write `answer == 'stop'` instead of `answer.== 'stop'`.

Let us see what happens.

Pour éviter que le jeu ne s'arrête jamais, nous avons ajouté les lignes 13 à 15.

Avant de vérifier si la proposition est correcte, ces lignes vérifient si le joueur n'a pas tapé le mot *stop*, si c'est le cas on tombe sur `exit`. *Ruby* reconnaît ce mot qui a pour effet de tout arrêter, on dit *sortir du programme*.

La ligne 13 montre une facilité d'écriture de *Ruby*, le double égale (`==`) est un message envoyé à ce que représente le nom `answer`, une chaîne de caractères, dont l'effet est de vérifier si son contenu est le même que ce qui vient après le double égale (`'stop'`). Ce qui nous permet d'écrire `answer == 'stop'` plutôt que `answer.== 'stop'`.

Voyons comment ça se passe.

```
$> ruby hangman.rb
- - - - -

Give me a letter: H
The letter is part of the hidden word
Give me a letter: L
The letter is part of the hidden word
Give me a letter: stop
```

The program did effectively prompt to give a letter and we typed next sequence: `H`, `L` and `stop`. Typing `stop` did stop the game as expected.

But what happens if we type no letter?

Le programme a bien répété les demandes, auxquelles nous avons donné tour à tour: `H`, `L` et `stop`. Et donner `stop` permet bien d'arrêter le jeu.

Mais que se passe-t-il si on ne donne aucune lettre?

```
$> ruby hangman.rb
-- -- -- --
Give me a letter:
The letter is part of the hidden word
Give me a letter: stop
```

Strangely, when we type no letter, we receive a positive feedback. Of course, it makes no sense.

Why do we get such behavior? What should we do in such a case?

The reason for this behavior is:

1. the check that rejects the proposal of more than one letter (line 17) accepts the entry because no letter is less than two letters or more,
2. the check that accepts that the proposed letter is part of the word (line 19) does accept an empty proposal (to the question *is nothing part of the word?* the answer is yes).

The right behavior should be to force exactly one letter.

Curieusement lorsqu'on ne donne aucune lettre, on reçoit une réponse positive. Cela n'a pas de sens bien-sûr.

Pourquoi avons-nous ce comportement? Que devrions-nous faire dans ce cas?

La raison de ce comportement est :

1. la vérification qui rejette les propositions de plus d'une lettre (ligne 17) laisse passer vu que aucune lettre est bien moins que deux lettres ou plus,
2. la vérification qui accepte que la lettre proposée fait partie du mot caché (ligne 19) va malheureusement accepter la proposition vide (à la question *est-ce que rien fait partie du mot?* la réponse est oui).

Le comportement que nous devrions appliquer est de demander immédiatement une lettre sans plus.

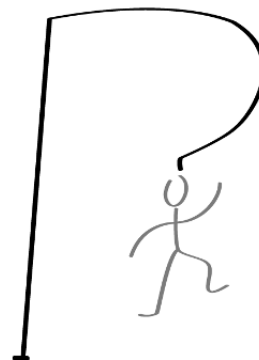
¹ We shift the lines to make the code more readable between the `do` and `end`, we say *indent* the code.

² Pour rendre le code plus lisible, on décale toutes les lignes comprises entre `do` et `end` pour les mettre en évidence visuellement, on appelle cette manière de faire *indenter* le code.

5

Ignore empty proposals

Ignorer les propositions vides



Step 5: Ignore empty proposals

If we want to ignore the empty proposals, we must do it before checking the occurrence of the proposed letter.

 hangman.rb

```

1  hidden_word = "hello"
2
3  puts "_ _ _ _ _"
4  puts
5
6  loop do
7
8      print "Give me a letter: "
9
10     answer = gets
11     answer = answer.chomp.downcase
12
13     if answer == 'stop'
14         exit
15     end
16
17     if answer.length > 1
18         puts "You must give only one letter!"
19 elsif answer.empty?
20 elsif hidden_word.include?(answer)
21     puts "The letter is part of the hidden word"
22 else
23     puts "Invalid letter"
24 end
25
26 end

```

We added to the line 19 a new alternative: `answer.empty?`.

The code means “send the `empty?` message” to `answer`, the string containing the proposal. The strings understand the message which implies that the strings check if their content has or does not have characters (namely *are you empty?*). If the string is empty, nothing happens since we wrote nothing between lines 19 and 20.

Let us give a try.

Étape 5 : Ignorer les propositions vides

Pour passer les propositions vides sous silence, il faut le faire avant la vérification de l'occurrence de la lettre proposée.

Mots du code suivant : empty signifie vide.

Nous avons ajouté à la ligne 19 une nouvelle alternative : `answer.empty?`.

Il s'agit d'envoyer le message `empty?` à la chaîne de caractères contenant la proposition. Ce message est compris par les chaînes de caractères et a pour effet de vérifier si une chaîne contient ou non des caractères (ce qui veut dire *es-tu vide?*). Si la chaîne est vide, il ne se passe rien puisqu'il n'y a rien entre les lignes 19 et 20.

Faisons un essai.

```

$> ruby hangman.rb
_ _ _ _ _

Give me a letter:
Give me a letter: stop

```

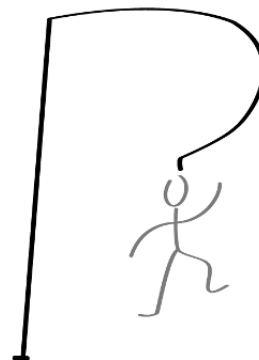
We do have the behavior we want, to the first question we give no letter and we immediately receive the same question. We do not go any further and we stop the game, as we only wanted to check the behavior.

Nous avons bien le comportement désiré, à la première question nous ne donnons aucune lettre et nous recevons immédiatement la même question. Nous n'allons pas plus loin et arrêtons le jeu vu que nous voulions juste faire cette vérification.

6

Montrer les lettres que le joueur trouve

Show the letters that the player finds



Step 6: Show the letters that the player finds

Until now we show the number of letters by explicitly setting the display (we are using `puts " _ _ _ _ _ "`) since we set the hidden word, the current game only offers one hidden word.

We will of course consider having other words, therefore we must be able to show the number of letters depending on the word we hide, we say: do it **dynamically**.

We introduce a new concept offered by the *Ruby* language called function, method or routine. The name *routine* is the one that is best suited.

When we do something that we repeat over and over, we say that we accomplish a *routine* task. The task is broken down into a sequence of actions or operations that we perform in the same order. The programming languages allow to define a sequence of operations or instructions, to group them and to give a name to the group. Such a group of operations with its name is what we call a *routine* (or *function*).

In *Ruby* you define such a group by using the `def` word (from *define*), followed by a word that gives the name. You then give the sequence of instructions and you end the group with the `end` word.

The name is very important because you will later use it to refer to the group of instructions performing a routine task. Actually, the name is simply the name of the message that you send. When *Ruby* manages the sending of the message, it searches what action is to be executed in particular among the names of the routines that have been defined.

As we already mentioned, in *Ruby*, the processing of a message always produces a result, The result is what the last instruction of the sequence produces.

Ok, let us add a routine that builds the information showing the number of letters.

 `hangman.rb`

```

>1  def word_info word
>2    "_ " * word.length
>3  end
4
5  hidden_word = "hello"
6
7  puts " _ _ _ _ _ "
8  puts
9
10 loop do
11
12   print "Give me a letter: "
13

```

Étape 6 : Montrer les lettres que le joueur trouve

Jusqu'à présent nous montrons le nombre de lettres en ayant mis explicitement l'affichage (en utilisant `puts " _ _ _ _ _ "`) vu que nous avons fixé le mot caché, le jeu n'offre qu'un seul mot caché.

Nous allons bien sûr envisager d'avoir d'autres mots, il faut donc arriver à montrer le nombre de lettres en fonction du mot qu'on cache, on dira : le faire de manière **dynamique**.

Nous allons introduire un nouveau concept offert par le langage *Ruby* appelé fonction, méthode ou routine. Le nom *routine* est celui qui convient le mieux.

Lorsqu'on fait quelque chose qu'on répète encore et encore, on dit qu'on accomplit une tâche *routinière*. La tâche se décompose en une série d'actions ou d'opérations que l'on effectue dans le même ordre. Les langages de programmation permettent de définir une séquence d'opérations ou d'instructions, de les grouper et de donner un nom à ce groupe. C'est ce groupe et son nom qu'on appelle *routine* (ou *fonction*).

En *Ruby* on crée un tel groupe en le marquant avec le mot `def` (de *define*, définir), suivi par un mot qui donne le nom. On donne après la série d'instructions et on marque la fin de ce groupe par le mot `end`.

Le nom est très important car c'est par ce nom qu'on pourra faire référence au groupe d'instructions accomplissant une tâche routinière. Le nom n'est autre que le nom d'un message qu'on envoie. Lorsque *Ruby* gère l'envoi d'un message, il cherche quelle est l'action à accomplir notamment parmi les noms de routines qui ont été définies.

Comme nous l'avons déjà mentionné, en *Ruby* le traitement d'un message produit toujours un résultat, le résultat est ce que produit la dernière instruction dans la séquence.

Ajoutons donc une routine qui va construire l'information donnant le nombre de lettres.

```

14  answer = gets
15  answer = answer.chomp.downcase
16
17  if answer == 'stop'
18    exit
19  end
20
21  if answer.length > 1
22    puts "You must give only one letter!"
23  elsif answer.empty?
24  elsif hidden_word.include?(answer)
25    puts "The letter is part of the hidden word"
26  else
27    puts "Invalid letter"
28  end
29
30 end

```

Lines 1 to 3 define a routine named `word_info`. After the name we meet another word: `word`. What is this word? As we said before, when you send some message you can attach an attachment, here the attachment receives the name `word` during the execution of the instructions of the routine. (we review later the detail of the processing that takes place in the routine)

The attachment is important because it allows to apply a treatment for different words, in our case, since the attached word will not always be the same.

After creating the routine, we are going to use it. We remove the fixed display, from line 7 and line 8, and replace it with the sending of the new message so that we get back what we want to show. We place the sending among the instructions of the loop to re-display the information each time.

Les lignes 1 à 3 définissent une routine appelée `word_info`. Après le nom nous avons un autre mot : `word`. A quoi sert ce mot? Comme on l'avait dit, lorsqu'on envoi un message on peut attacher une pièce jointe, ici la pièce jointe portera le nom `word` le temps de l'exécution de la routine. (nous reviendrons plus loin sur le détail du traitement que fait cette routine)

Cette pièce jointe est importante car elle permet d'effectuer un traitement pour des mots différents, dans notre cas, puisque le mot attaché ne sera pas toujours le même.

Après avoir créé cette routine, nous allons l'utiliser. Nous allons supprimer l'affichage fixe des lignes 7 et 8 et le remplacer par l'envoi du nouveau message pour récupérer ce que nous allons montrer. Nous plaçons cet envoi parmi les instructions de la boucle pour ré-afficher cette information à chaque fois.

 hangman.rb

```

1  def word_info word
2    "_" * word.length
3  end
4
5  hidden_word = "hello"
6
7  loop do
8
9    info = word_info(hidden_word)
10
11    puts info
12    puts
13
14    print "Give me a letter: "
15
16    answer = gets
17    answer = answer.chomp.downcase
18
19    if answer == 'stop'

```

```

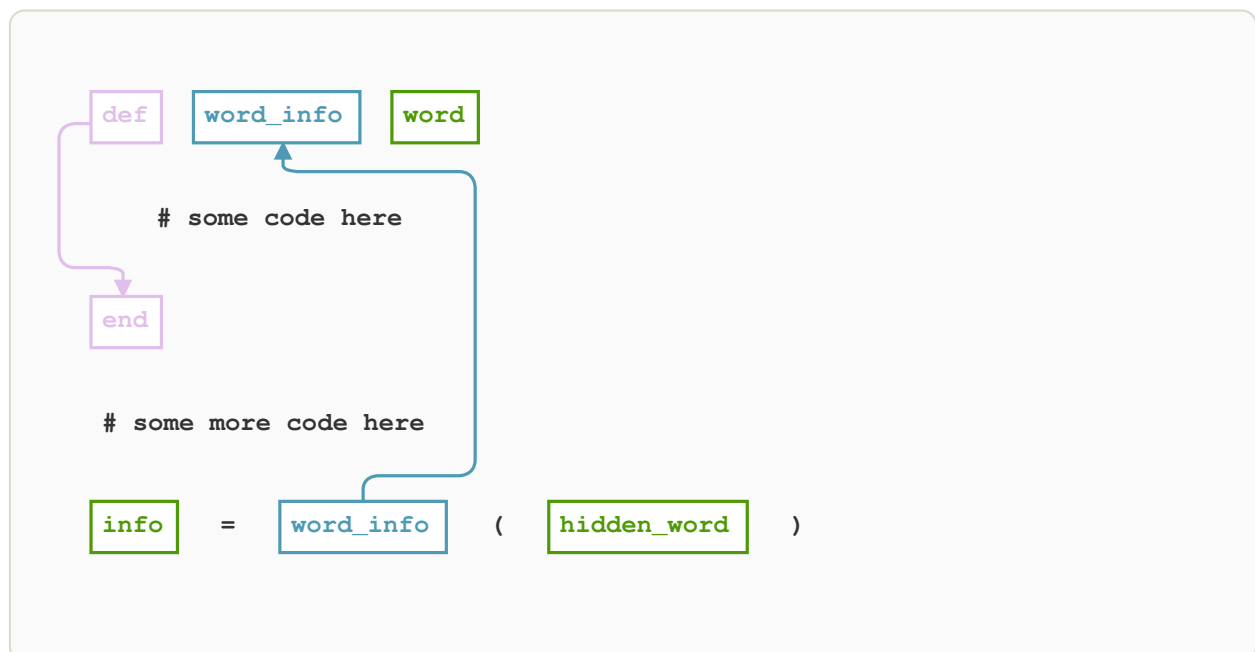
20     exit
21   end
22
23   if answer.length > 1
24     puts "You must give only one letter!"
25   elsif answer.empty?
26   elsif hidden_word.include?(answer)
27     puts "The letter is part of the hidden word"
28   else
29     puts "Invalid letter"
30   end
31
32 end

```

Next diagram shows the behavior that we changed at line 9.

Le diagramme suivant montre le comportement de ce qu'on vient de changer à la ligne 9.

Mots du diagramme suivant : *some more code here* signifie un peu plus de code ici.



By analyzing what the diagram shows, *Ruby*:

1. understands that a new message is being defined (or a new routine) because of the `def` word,
2. registers the definition of the new message named `word_info` (and the associated processing that ends with the `end` word),
3. detects that a message must be sent with an attachment being what the `hidden_word` stands for (our hidden word),
4. detects that it must give a name (`info`) to the result produced by the sending of `word_info` because of the `=` sign.

Ruby knows that we want to send the `word_info` message because the only thing it finds, related to `word_info`, is a message named with that name.

En analysant ce que montre le diagramme, *Ruby* va :

1. comprendre qu'il y a une définition d'un nouveau message (ou d'une routine) grâce au mot `def`,
2. enregistrer la définition de ce nouveau message baptisé `word_info` (ainsi que le traitement qui lui est associé qui se termine au mot `end`),
3. détecter qu'il y a un message à envoyer auquel il faut joindre ce que représente `hidden_word` (notre mot caché),
4. détecter qu'il y a un nom (`info`) à associer au résultat de l'envoi du message `word_info` grâce au signe `=`.

Ruby sait que l'on veut envoyer le message `word_info` parce la seule chose qu'il retrouve, concernant ce mot, est la définition d'un message portant ce nom.

What is `word_info` doing?

Let us now have a look at what `word_info` is doing:
`"_ " * word.length`.

We multiply a string by the number of characters of the word attached to the message (transmitted at the sending of the message). The syntax is close to the mathematical notation we are used to, that *Ruby* accepts as a *syntactic sugar*, the code is equivalent to `"_ ".* word.length`.

The aim is to send the message named `*` to the string. The action of the string is to produce a new string made by the repetition of its content as many time as the given number.

Normally, we did not add anything new to the game, we only improved the behavior. However, we must check that we did not break anything...

Que fait `word_info` ?

Voyons maintenant ce que fait `word_info` :
`"_ " * word.length`.

On multiplie une chaîne par le nombre de caractères compris dans le mot reçu en pièce jointe (transmis à l'envoi du message `word_info`). *Ruby* accepte cette écriture qui nous est familière, c'est une facilité d'écriture équivalente à `"_ ".* word.length`.

Il s'agit d'envoyer le message appelé `*` à la chaîne de caractères. L'effet de la chaîne est de produire une nouvelle chaîne faite de la répétition de son contenu autant de fois que le nombre reçu.

En principe, nous n'avons rien de nouveau dans le fonctionnement du jeu, nous avons juste amélioré le comportement. Nous devons toutefois vérifier que nous n'avons rien casser...

```
$> ruby hangman.rb
- - - - -
Give me a letter: H
The letter is part of the hidden word
- - - - -
Give me a letter: L
The letter is part of the hidden word
- - - - -
Give me a letter: stop
```

Just saying if a letter is part of the hidden word is not enough, we need to show the correct letters at the right position. We are going to improve the routine that we just wrote to show the letter. We also need the letter that the player proposed to be able to show it, we must join the letter with the hidden word when we send the `word_info` message.

Notice that we can focus on that need without thinking about the rest of the game: show the number of unknown letters and the letter that the player just found at the right position.

Dire si une lettre fait partie du mot caché ne donne pas assez d'information, il faut montrer les lettres à la bonne position. Nous allons améliorer la routine que nous venons de décrire pour le faire. Il faut pour cela disposer également de la lettre qui a été proposée, nous devons joindre cette lettre avec le mot caché à l'envoi du message `word_info`.

Il est intéressant de noter que nous pouvons nous concentrer sur ce besoin sans penser au reste du jeu: montrer le nombre de lettres inconnues et la lettre qui vient d'être trouvée à sa place.

Mots du code suivant : underscores signifie caractères soulignés, unless signifie sauf si.

 hangman.rb

```
1 def word_info word, letter
2
3   n = word.length
4   underscores = "_ " * n
5
6   unless letter.nil?
7     i = word.index(letter)
```



```

8      underscores[2 * i] = letter.upcase
9      end
10
11     underscores
12
13 end

```

Lines 3 and 4 build the string, that shows the *ghost letters* of the hidden word at their position, to which we assign the name `underscores`.

Then comes the special word `unless`, it plays the same role as `if` as it sets a new choice, but the choice is the opposite of the `if` word. The following instructions are applied *unless* the result that follows (`letter.nil?`) is true. The message named `nil?` is a question: are you nil? Ruby has a notion of *nothing* called `nil`. The question *are you nil* strangely means *are you nothing*. We allow to pass *nothing* as letter to the routine, because we are thinking at the beginning of the game when the game has not yet prompted for any letter but we need to give information on the hidden word to the player.

The behavior we are working out is to produce a string with underscores where underlined characters represent the hidden letters, if no letter is joined the routine ends with the line containing `underscores` (line 11) resulting in producing the built string at line 4.

Otherwise, when a letter is joined, at line 7 we send the `index` message with the letter (`letter`) as attachment. A string then receives the `index` message searches the position of the letter through its letters sequence and produces a number giving the position¹. If the letter does not appear the result is *nothing* (`nil`), but that should not happen we expect that we checked for the presence of the letter before. We give the name `i` to the position.

Afterwards, line 8, all we need to do is to replace the underlined letter that appears at the position of the letter in the hidden word with the actual letter, that we force to uppercase letter with `letter.upcase`. The notation `[i * 2] =`³ is a special notation, meaning: send the `[]=` message attaching the value of `i * 2`. The effect is to replace the value that happens to be at the position given by `i * 2` with what comes after the `=`, `letter.upcase`. Why `i * 2`?

First things first. The hidden word is *hello*, in other words a string containing five letters as shown in the following figure with the number of each position.

Aux lignes 3 et 4 nous construisons la chaîne donnant des *lettres phantômes* du mot caché à laquelle on attribue le nom `underscores`.

Ensuite arrive le mot spécial `unless`, il joue le même rôle que `if` car il marque un choix mais fait le choix dans le sens contraire. Les instructions qui le suivent s'appliquent *sauf si* le résultat qui suit (`letter.nil?`) est vrai. Le message `nil?` est une question: es-tu nil? Ruby a une notion de *rien* appelée `nil`. La question *es-tu nil* veut dire, bizarrement, *es-tu rien*. En permettant de donner *rien* comme lettre à cette routine, nous pensons au début du jeu où on n'a pas encore demandé la moindre lettre au joueur mais nous devons lui donner de l'information sur le mot caché.

Le comportement que nous cherchons à avoir est de produire une chaîne avec des caractères soulignés, représentant les lettres cachées, si aucune lettre n'est transmise la routine termine par `underscores` (ligne 11) qui a pour effet de produire la chaîne construite à la ligne 4.

Si en revanche une lettre a été transmise, à la ligne 7 nous envoyons le message `index` en joignant la lettre (`letter`). Une chaîne recevant le message `index` recherche la position de la lettre dans sa suite de caractères et produit un nombre donnant la position². Si la lettre n'apparaît pas le résultat est *rien* (`nil`), ce qui ne devrait pas arriver car nous supposons qu'on a vérifié sa présence auparavant. Nous attribuons le nom `i` à cette position.

Ensuite, ligne 8, nous n'avons plus qu'à remplacer le souligné se trouvant à la position de la lettre dans le mot caché par la lettre, que nous forçons en majuscule avec `letter.upcase`. La notation `[i * 2] =`⁴ est une notation spéciale, elle signifie: envoyer le message `[]=` en joignant `i * 2`. L'effet est de remplacer (d'où l'utilisation du symbole `=`) ce qui se trouve à la position donnée par `i * 2` par ce qui se trouve après le `=`, `letter.upcase`. Pourquoi `i * 2`?

Procédons par étape. Le mot caché est *hello* c'est-à-dire une chaîne de cinq caractères comme le montre la figure qui suit avec le numéro de chaque position.

h	e	l	l	o
0	1	2	3	4

The routine named `word_info` begins by creating a string with underscores, symbolizing the hidden letters, separated by a space to make them more visible. So, there are five underscores and five spaces, ten characters, shown in the following figure.

La routine `word_info` commence par créer une chaîne avec des soulignés, symbolisant les lettres cachées, séparés par un espace pour les rendre plus visible. Il y a donc cinq caractères soulignés et cinq espaces, soit dix caractères comme le montre la figure suivante.

_		_		_		_		_	
0	1	2	3	4	5	6	7	8	9

Say that the player proposed the letter `o` which is the fifth letter with the number 4.

Supposons que la lettre qu'a proposé le joueur soit le `o` qui se trouve en cinquième position portant le numéro 4.

h	e	l	l	o
0	1	2	3	4

↑

To show this letter, we must replace the last underscore, position 8, with the letter. Since we have two characters per hidden letter, the offset of the letter at position 4 is 2×4 . For the first letter, as the position is zero we would compute 2×0 which is zero.

Pour montrer cette lettre il faut remplacer le dernier souligné, à la position 8, par la lettre. Puisqu'on a deux caractères par lettre cachée le décalage pour la lettre en position 4 est de 2×4 . Pour la première lettre, vu que la position est zéro on aurait 2×0 qui fait zéro.

_		_		_		_		o	
0	1	2	3	4	5	6	7	8	9

↑

This allows us to establish the general rule: to replace an underscore with the found letter, we must replace the character at the position `2 x i` where `i` is the position of the letter in the word.

Using `word_info`

We wrote a routine that builds information for the player, we need now to use it as expected because we added a new attachment (the found letter).

Ce qui nous permet d'établir la règle générale : pour remplacer le souligné par la lettre trouvée, il faut remplacer le caractère à la position `2 x i` où `i` est la position de la lettre dans le mot.

Utilisation de `word_info`

Nous avons écrit une routine qui construit l'information pour le joueur il nous reste à l'utiliser correctement vu que nous avons ajouté une nouvelle pièce jointe (la lettre trouvée).

The term *attachment to the message* is not the usual technical name. We would talk about *parameters* and *arguments*.

The term *parameter* is used to express the attached items that are expected and specified at the definition of a routine (or a message).

The term *argument* expresses the values we associate with the message when we send it.

L'expression *pièce(s) jointe(s)* n'est pas la terminologie technique utilisée. On parlera de *paramètres* et d'*arguments*.

Le terme *paramètre* est utilisé pour parler des pièces jointes attendues que l'on spécifie lors de la définition d'une routine (ou d'un message).

Le terme *argument* exprime les valeurs qu'on associe au message au moment où on l'envoie.

hangman.rb

```

1  def word_info word, letter
2
3      n = word.length
4      underscores = "_" * n
5
6      unless letter.nil?
7          i = word.index(letter)
8          underscores[2 * i] = letter.upcase
9      end
10
11     underscores
12
13 end
14
15 hidden_word = "hello"
16
17 letter = nil
18
19 loop do
20
21     info = word_info(hidden_word, letter)
22
23     puts info
24     puts
25
26     print "Give me a letter: "
27
28     answer = gets
29     answer = answer.chomp.downcase
30
31     if answer == 'stop'
32         exit

```

```

33     end
34
35     if answer.length > 1
36       puts "You must give only one letter!"
37     elsif answer.empty?
38     elsif hidden_word.include?(answer)
39       puts "The letter is part of the hidden word"
➤40       letter = answer
41     else
42       puts "Invalid letter"
43     end
44
45   end

```

First we change to add the sending of the new message (`word_info`, line 21) where we also attach a letter. The letter is named `letter` (easy...). If we only make that change *Ruby* will complain when it sees `letter` because it does know the word `letter` and it stops immediately the execution.

You must ensure that all the used words are known **at the moment they get used**. This is why we did give, at line 17, the name `letter` to *nothing* (that *Ruby* calls `nil`). Because we definitely have no clue what letter we could propose, the best we can do is let *Ruby* know the name even if it is *nothing*.

Notice that the declaration of the name does appear before the `loop` keyword, because the name is needed to name the correct letters that the player gives. As the game begins only when the loop does start (the repetition of the display of the status of the game, the prompt for a proposal, the check of the proposal), the name assignment to *nothing* must happen before and no more when we are inside the body of the loop.

Finally, we must change the letter attached to the word `letter` as soon as the check of the proposed letter appears to be valid. We must do it after line 38, line 40, where we give a second name (`letter`) at the thing that is already named `answer`.

Let us check with a new test

We must check that when the player gives a valid letter, the computer shows the letter at the right position.

Le premier changement est l'envoi du nouveau message `word_info` (ligne 21) où nous joignons une lettre en plus. La lettre en question est appelée `letter`. Si on ne fait rien d'autre, *Ruby* pourrait se montrer intraitable car en tombant sur ce nom `letter` il se plaindrait de ne pas voir ce que c'est et arrêter tout.

Il faut s'assurer que tous les mots utilisés soient connus **au moment de leur utilisation**. C'est pourquoi nous avons ajouté, à la ligne 17, l'attribution du nom `letter` à *rien* que *Ruby* appelle `nil`. N'ayant pas la moindre lettre à proposer à ce stade autant faire en sorte que *Ruby* connaisse le nom quitte à ce que ne ce soit *rien*.

Il est important de voir que cette déclaration de nom se trouve avant le mot `loop`, car ce nom va servir à nommer les lettres correctes données par le joueur. Comme la partie ne commence qu'au moment où démarre la boucle (c'est-à-dire la répétition de l'affichage de l'état de la partie, l'attente d'une proposition, la vérification de la proposition) l'attribution du nom à *rien* doit se faire avant et plus une fois qu'on se trouve dans la boucle.

Il faut enfin changer la lettre associée au mot `letter` dès que la vérification de la lettre proposée s'avère correcte. Nous devons le faire après ligne 38, à la ligne 40, où nous donnons un deuxième nom (`letter`) à ce qui s'appelle déjà `answer`.

Vérifions par un nouveau test

Nous devons vérifier que lorsqu'on donne une bonne lettre, l'ordinateur nous la montre à la bonne position.

```
$> ruby hangman.rb
_ _ _ _ _

Give me a letter: H
The letter is part of the hidden word
H _ _ _ _

Give me a letter: L
The letter is part of the hidden word
_ _ L _ _

Give me a letter: stop
```

The letters are shown at the right place but we notice two problems:

- we only show the last given letter
- if a letter is repeated several times we only show the first occurrence, like we see with the double `L` letter

We did not finish yet.

Les lettres sont au bon endroit mais nous remarquons deux défauts:

- nous ne montrons que la dernière lettre donnée
- si une lettre est répétée, comme le `L`, nous ne la montrons qu'une fois

On a encore du pain sur la planche.

¹ The position numbering of the letters always starts at 0, then 1, ...

² La numérotation de la position des lettres commence toujours à 0. suivit de 1, ...

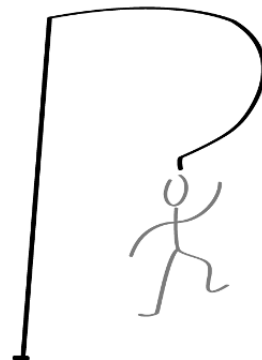
³ The start character (`*`) represents the mathematical multiplication symbol (it replaces the multiplication sign that is too close to the X letter).

⁴ Le caractère étoile (`*`) représente le symbole mathématique de multiplication (il remplace le signe fois qui ressemble trop au caractère X).

7

Temporarily put a problem aside

Mettre un problème de côté, provisoirement



Step 7: Temporarily put a problem aside

We are not going to solve the repeated letters problem, let us replace the *hardcoded* word (line 15) with *fuchsia* instead.

 hangman.rb

```
1  hidden_word = "fuchsia"
```

We will come back to the issue of the repeated letter once we improved the game, as long as we do not declare the game as finished it does not matter.

Étape 7 : Mettre un problème de côté, provisoirement

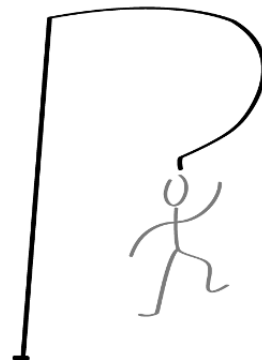
Nous n'allons pas résoudre le problème des lettres qui se répètent tout de suite, nous allons changer le mot *écrit en dur* dans le code (ligne 15) et utiliser le mot *fuchsia*.

Nous reviendrons au problème des lettres répétées quand nous aurons amélioré le jeu, tant que nous ne déclarons pas le jeu terminé cela ne pose pas de problème.

8

Diviser le code source

Split the source code



Step 8: Split the source code

Every time we make a change the number of lines of code increases, we went from a few lines of code, only eight lines (sure they did not offer any interesting functionality) to forty five lines. The more the number of lines increases, the more it is difficult to find where one or the other item is located. However, we did already isolate a set of operations by grouping them in a routine.

One option that the programming languages provide is the ability to isolate parts of the code in different files, it frees us from the need to keep in mind the dozens of lines that are written in other files so that we can focus on the file in which we are working.

Another advantage is that we can reuse the same code doing useful thing in other projects (other games for instance).

We are going to move the `word_info` routine to a new file named `word_info.rb`.

 `word_info.rb`

```
1  # +letter+ should be +nil+ or part of the given +word+ and down-case
2  def word_info word, letter
3
4      n = word.length
5      underscores = "_" * n
6
7      unless letter.nil?
8          i = word.index(letter)
9          underscores[2 * i] = letter.upcase
10     end
11
12     underscores
13
14 end
```

The first line is a comment line, namely text that *Ruby* does not even take into account. The goal of a comment is to give some explanation to people who will possibly read the code. A comment starts with the sharp sign (`#`) and extends to the end of the line.

We also need to express in the initial file that it must use what we moved to another file.

 `hangman.rb`

```
1  require_relative "word_info"
2
3  hidden_word = "fuchsia"
4
```

Étape 8 : Diviser le code source

A chaque changement nous voyons le nombre de lignes de code augmenter, nous sommes passé de huit lignes (qui ne faisait rien d'intéressant bien-sûr) à quarante-cinq lignes. Plus ce nombre de lignes va augmenter plus nous aurons du mal à retrouver où se trouve l'un ou l'autre élément que nous avons ajouté. Cependant, nous avons déjà isolé des opérations en les groupant dans une routine.

Une des possibilités offertes par les langages de programmation est d'isoler des parties du code dans des fichiers différents, cela nous permet de ne pas avoir en tête les dizaines de lignes qui sont dans d'autres fichiers et nous concentrer sur le fichier dans lequel nous travaillons.

Un autre avantage est de pouvoir réutiliser du code faisant des choses utiles dans d'autres projets (d'autres jeux par exemple).

Nous allons déplacer la routine `word_info` dans un nouveau fichier appelé `word_info.rb`.

La première ligne est un commentaire c'est-à-dire du texte que *Ruby* ne prend pas en compte. Le but est de donner une indication aux personnes qui seront amenées à lire le code. Un commentaire commence par un dièse (`#`) et se prolonge jusqu'à la fin de la ligne.

Nous devons à présent signaler dans le fichier de départ qu'il faut utiliser ce que nous venons de déplacer.

Mots du code suivant : require signifie exiger, relative signifie relatif.

```

5  letter = nil
6
7  loop do
8
9    info = word_info(hidden_word, letter)
10
11    puts info
12    puts
13
14    print "Give me a letter: "
15
16    answer = gets
17    answer = answer.chomp.downcase
18
19    if answer == 'stop'
20      exit
21    end
22
23    if answer.length > 1
24      puts "You must give only one letter!"
25    elsif answer.empty?
26    elsif hidden_word.include?(answer)
27      puts "The letter is part of the hidden word"
28      letter = answer
29    else
30      puts "Invalid letter"
31    end
32
33  end

```

We added the `require_relative` word that *Ruby* recognizes followed but the file name without the `.rb`¹ part of the name. `.rb` is the file extension for files of *Ruby* type.

Ruby is going to search for a file with the given name and analyze it before stepping to the next line of the current file. Each scanned file can also do the same. However, *Ruby* keeps track of all the files that it processes and take them into account only once.

With the `require_relative` instruction, *Ruby* searches for the file relatively to the place where the current file is².

Even if we did not make any real change to the game, we could have broken what was fine until now. Therefore, we must **test again** that it **still** behaves correctly.

Nous avons ajouté le mot `require_relative` reconnu par *Ruby* auquel nous ajoutons le nom du fichier sans la partie `.rb`³, extension pour les fichiers de type *Ruby*.

Ruby va essayer de retrouver un fichier portant le nom donné et l'analyser avant de passer à la ligne suivante du fichier en cours. Chaque fichier analysé peut à son tour faire de même. *Ruby* va toutefois se souvenir des fichiers déjà traités et ne les prendre en compte qu'une seule fois.

Avec l'instruction `require_relative` *Ruby* recherche le fichier par rapport à l'endroit où se trouve le fichier en cours⁴.

Nous n'avons apporté aucun changement au jeu à proprement parlé mais nous avons peut-être cassé ce qui fonctionnait jusque là. Nous **devons vérifier** que tout se déroule correctement, **malgré tout**.

```

$> ruby hangman.rb
- - - - -
Give me a letter: F
The letter is part of the hidden word
F _ - - - -
Give me a letter: stop

```

All right, we did not break anything.

Nous n'avons rien cassé.

¹ This part of the file name is called *file extension*, namely what characterizes the type of file contents.

² File are saved in directories. When we say *the place where the current file is* we mean the director, is which it is saved.

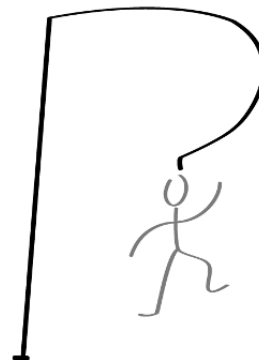
³ Cette partie du nom d'un fichier est appelée *extension du fichier*, c'est-à-dire ce qui caractérise le type de contenu du fichier.

⁴ Les fichiers sont stockés dans des répertoires. Lorsqu'on parle *d'endroit où se trouve un fichier* on parle du répertoire dans lequel il est enregistré.

9

Show all the letters that the player has already found

Montrer toutes les lettres que le joueur a déjà trouvées



Step 9: Show all the letters that the player has already found

We only show one letter at a time, the last one that the player found. Understanding that behavior is rather simple: we assign a name to the proposed letter when it does match, we use again the same name each time, hiding in the same time the previous letter.

We need to accumulate all the found letters. We have what we call *collections*, we could say a *bag* in which we can add things.

Étape 9 : Montrer toutes les lettres que le joueur a déjà trouvées

Nous ne montrons qu'une lettre à la fois, la dernière que le joueur a trouvé. Il est simple de comprendre ce comportement : nous donnons un nom à la lettre au moment où elle s'avère correspondre, nous utilisons à nouveau ce même nom à chaque fois cachant par la même occasion la lettre précédente.

Nous avons besoin d'accumuler les lettres trouvées. Nous disposons de ce qu'on appelle des *collections*, on pourrait dire un *sac* dans laquelle on peut ajouter des choses.

 hangman.rb

```

1  require_relative "word_info"
2
3  hidden_word = "fuchsia"
4
5  found_letters = []
6
7  loop do
8
9      info = word_info(hidden_word, found_letters)
10
11      puts info
12      puts
13
14      print "Give me a letter: "
15
16      answer = gets
17      answer = answer.chomp.downcase
18
19      if answer == 'stop'
20          exit
21      end
22
23      if answer.length > 1
24          puts "You must give only one letter!"
25      elsif answer.empty?
26      elsif hidden_word.include?(answer)
27          puts "The letter is part of the hidden word"
28
29          if found_letters.include?(answer)
30              else
31                  found_letters << answer
32              end
33
34      else
35          puts "Invalid letter"
36      end
37
38  end

```


At line 5, we create an empty collection `[]` to which we give the name `found_letters`.

We send the collection to the routine that builds the information shown instead of the single letter (line 9).

We add each found letter to the collection (line 31), the branch that executes when the proposed letter is part of the hidden word. *Ruby* offers the `<<` sign to add something to the collection.

To prevent from adding the same letter several times to the collection, if the player gives the same letter again, we first check at line 29 (by sending the `include?` message to the collection).

At this stage we are not yet done because the `word_info` routine does not expect to receive a collection, it expects a single letter.

 `word_info.rb`

```

>1  # +letters+ should be part of the given +word+ and down-case
>2  def word_info word, letters
3
4      n = word.length
5      underscores = "_ " * n
6
>7      letters.each do |letter|
8          i = word.index(letter)
9          underscores[2 * i] = letter.upcase
10     end
11
12     underscores
13
14 end

```

Since we now send a collection of letters, instead of a single letter, to the routine we changed the name of the parameter (the attachment), using its plural form (`letters`) to reflect that the name is associated with several letters. Notice that we also had to change the above comment (often people do not review the comments which on the long run become useless, it is better to have no comments rather than keeping outdated information).

The most important change appears at line 7. We are sending the `each` message to the collection of letters and we attach a block (a sequence of instructions). The collection will process the message by applying the block for each letter it contains. The block will receive each letter through the word `letter`, name that we choose, this is the same concept as the attachments for the routines. Each time the instructions will be applied the value associated with the word `letter` is different. Consequently, several underscores are replaced by letters.

Once again, a test.

À la ligne 5 nous créons une collection vide `[]` à laquelle nous donnons le nom `found_letters` (*lettres trouvées*).

Nous passons cette collection à la routine qui construit l'information qu'on montre au lieu de l'unique lettre (ligne 9).

Nous ajoutons chaque lettre trouvée à la collection (ligne 31), dans la partie qui est exécutée lorsque la lettre fait partie du mot caché. *Ruby* nous propose le signe `<<` pour faire l'ajout à la collection.

Pour éviter de mettre dans la collection les mêmes lettres plusieurs fois, si le joueur propose à nouveau une même lettre, nous faisons une vérification à la ligne 29 (en envoyant le message `include?` à la collection).

À ce stade nous n'avons pas encore tout fait car la routine `word_info` ne s'attend pas à recevoir une collection mais une lettre.

Puisque nous passons une collection de lettres, au lieu d'une seule lettre, à la routine nous avons changé le nom donné à ce paramètre (pièce jointe) en le mettant au pluriel (`letters`) pour refléter le fait qu'il y a plusieurs lettres. Nous avons également dû adapter le commentaire (il arrive souvent qu'on n'adapte pas les commentaires qui deviennent inutiles, mieux vaut ne pas avoir de commentaires plutôt que des informations obsolètes).

Le changement le plus important se trouve à la ligne 7. Nous envoyons le message `each` à la collection de lettres et nous joignons un bloc (séquence d'instructions). La collection va traiter le message en appliquant le bloc pour chaque lettre qu'elle contient. Le bloc va recevoir chaque lettre au travers du mot `letter`, nom que nous avons choisi, il s'agit de la même chose que les pièces jointes des routines. À chaque fois que les instructions seront appliquées la valeur associée au mot `letter` sera différente. En conséquence, plusieurs soulignés seront remplacés par des lettres.

Faisons, une nouvelle fois, un test.

```
$> ruby hangman.rb  
_ _ _ _ _  
  
Give me a letter: F  
The letter is part of the hidden word  
F _ _ _ _  
  
Give me a letter: H  
The letter is part of the hidden word  
F _ H _ _  
  
Give me a letter: stop
```

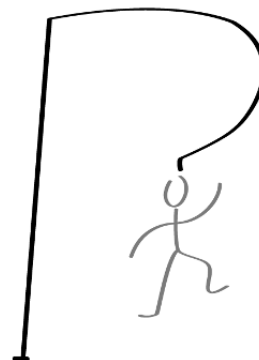
We gave two valid letters (keep in mind we did change the hidden word) and they both are displayed.

Nous avons donné deux bonnes lettres (rappelons que nous avons changé le mot caché) et elles apparaissent bien.

10

Rappeler les mauvaises propositions

Recall the wrong proposals



Step 10: Recall the wrong proposals

Displaying the number of letters of the hidden word and the found letters is not enough to help the player, showing constantly the wrong letters that he/she proposed is a nice addition.

The same way we accumulate the correct letters in a collection, we are going to store the wrong letters.

Étape 10 : Rappeler les mauvaises propositions

Pour aider le joueur, montrer le nombre de lettres du mot caché et les lettres correctes ne suffit pas, il est intéressant de lui montrer en permanence les mauvaises lettres qu'il a proposé.

De la même façon que nous accumulons les lettres correctes dans une collection, nous allons garder les lettres erronées.

Mots du code suivant : wrong signifie incorrect.

 hangman.rb

```

1  require_relative "word_info"
2
3  hidden_word = "fuchsia"
4
5  found_letters = []
>6  wrong_letters = []
7
8  loop do
9
10   info = word_info(hidden_word, found_letters)
11
12   puts info
13   puts
14
>15   print "Wrong letters: "
>16   puts wrong_letters.sort.uniq.join(" ")
>17   puts
18
19   print "Give me a letter: "
20
21   answer = gets
22   answer = answer.chomp.downcase
23
24   if answer == 'stop'
25     exit
26   end
27
28   if answer.length > 1
29     puts "You must give only one letter!"
30   elsif answer.empty?
31   elsif hidden_word.include?(answer)
32     puts "The letter is part of the hidden word"
33
34     if found_letters.include?(answer)
35     else
36       found_letters << answer
37     end
38
39   else
40     puts "Invalid letter"
>41     wrong_letters << answer.upcase
42   end
43

```

44 **end**

We add another collection named `wrong_letters` at line 6.

We add the wrong letters at line 41, right after the display of the message saying that the letter is not in the hidden word. This operation looks like the one that adds the correct letters to the other collection.

Lines 15 to 17 display the message showing all the wrong letters. They are simple enough except the messages `sort`, `uniq` and `join`:

1. we send the `sort` message to the collection that sorts the items of the collection and produces a new collection. The `wrong_letters` collection contains letters and they are going to be ordered alphabetically. Since we accumulate the letters as the player makes proposals, it is *friendly* to show the in an ordered sequence.
2. we send the `uniq` (short for *unique*) message to the collection ordered alphabetically that also produces a new collection in which duplicate letters are removed (because at line 41 we accumulate only the invalid letters, if the player proposes the same letter again, the letter will be duplicated in the collection).
3. we finally send the `join` message with a string containing a space character to transform the collection of letters to a string made of the letters placed one after the other and separated by a space.

Test the game by giving a wrong letter.

Nous ajoutons une nouvelle collection appelée `wrong_letters` (*lettres incorrectes*) à la ligne 6.

Nous ajoutons les lettres incorrectes à la ligne 41, juste après l'affichage du message disant que la lettre n'est pas dans le mot caché. Cette opération ressemble à celle qui ajoute les lettres correctes à l'autre collection.

Les lignes 15 à 17 affichent le message donnant l'ensemble des lettres incorrectes. A priori elles sont assez simple si ce n'est la suite des messages `sort`, `uniq` et `join`:

1. nous envoyons le message `sort` à la collection qui a pour effet d'ordonner les éléments de la collection et produire une nouvelle collection. La collection contient des lettres et elles vont être ordonnées par ordre alphabétique. Vu que nous accumulons les lettres au fur et à mesure que le joueur les propose il est *amical* de les montrer de manière ordonnée.
2. nous envoyons le message `uniq`¹ à la collection triée par ordre alphabétique qui a pour effet de produire une nouvelle collection dans laquelle les lettres qui seraient répétées sont supprimées (nous ne faisons qu'accumuler les lettres incorrectes à la ligne 41, si le joueur donne à nouveau une même lettre elle sera deux fois dans la collection).
3. nous envoyons enfin le message `join` avec une chaîne contenant un espace pour transformer la collection de lettres en une chaîne faites des lettres mises les unes derrières les autres et séparées par un espace.

Faisons le test en donnant une mauvaise lettre.

```
$> ruby hangman.rb
- - - - -
Wrong letters:

Give me a letter: G
Invalid letter
- - - - -
Wrong letters: G

Give me a letter: stop
```

The game shows that the hidden word contains seven letters, that there is not yet any wrong letter and prompts to enter a letter. As we input the letter `G`, which is not part of the hidden word, the incorrect letter message appears followed again by the underscores and the invalid letter we gave.

Let us see if the sequence of invalid letters is displayed ordered alphabetically and that the same letter appears only once.

Le jeu nous montre qu'il y a sept lettres cachées, qu'il n'y a pas encore de lettre erronée et demande de donner une lettre. Comme nous donnons la lettre `G` qui ne fait pas partie du mot caché, nous recevons le message de lettre incorrecte puis à nouveau les soulignés et cette fois la lettre incorrecte.

Essayons de voir si la séquence des lettres est bien montrée par ordre alphabétique et qu'une même lettre n'apparaît qu'une fois.

```
$> ruby hangman.rb
_ _ _ _ _
Wrong letters:

Give me a letter: M
Invalid letter
_ _ _ _ _

Wrong letters: M

Give me a letter: B
Invalid letter
_ _ _ _ _

Wrong letters: B M

Give me a letter: M
Invalid letter
_ _ _ _ _

Wrong letters: B M

Give me a letter: stop
```

Perfect!

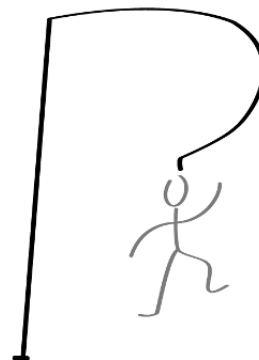
Parfait!

¹ Le mot `uniq` est une contraction de *unique*.

11

Declare that the player has won

Déclarer que le joueur a gagné



Step 11: Declare that the player has won

Once the game is started it only stops if the player enters `stop` when he enters a proposal, even if the player finds all the hidden letters the game goes on and asks for more letters.


We are going to change the code to stop the game as soon as the player has found all the letters.

Étape 11 : Déclarer que le joueur a gagné

Une fois que le jeu est en route il ne s'arrête que si le joueur donne `stop` lorsqu'il doit faire une proposition, même si le joueur trouve toutes les lettres le jeu continue à demander des lettres.

Nous allons modifier le code pour arrêter le jeu dès que le joueur a trouvé toutes les lettres.

Mots du code suivant : `win` signifie gagner, `break` signifie briser.

 hangman.rb

```

1  require_relative "word_info"
2
>3  hidden_word = "a"
4
5  found_letters = []
6  wrong_letters = []
7
8  loop do
9
10     info = word_info(hidden_word, found_letters)
11
12     puts info
13     puts
14
>15     unless info.include?('_')
>16         puts "You win, PATATE!"
>17         break
>18     end
19
20     print "Wrong letters: "
21     puts wrong_letters.sort.uniq.join(" ")
22     puts
23
24     print "Give me a letter: "
25
26     answer = gets
27     answer = answer.chomp.downcase
28
29     if answer == 'stop'
30         exit
31     end
32
33     if answer.length > 1
34         puts "You must give only one letter!"
35     elsif answer.empty?
36     elsif hidden_word.include?(answer)
37         puts "The letter is part of the hidden word"
38
39         if found_letters.include?(answer)
40             else
41                 found_letters << answer
42             end
43

```

```

44     else
45         puts "Invalid letter"
46         wrong_letters << answer.upcase
47     end
48
49 end

```

First, let us replace the hidden word with a word containing only one letter (line 3) so that testing is faster (we only need to enter one letter to win).

At line 15, we re-use the *unless* word that we link to the condition: `info.include?('')`. We send the `include?` message and attach an underscore character, that checks if the underscore is part of the information string. If there are no underscore left then the player has found all the letters.

Line 16 displays the message *you win*¹.

Line 17 contains only the word `break`, known to *Ruby*, it asks *Ruby* that it must immediately stop the current repetition (the loop).

We could have checked if the game is over at the same place as the check of a matching letter (at about line 37), since as soon as the player finds the last letter the game can stop, but we want to show the whole word before ending the game.

We must test the game and check that we can win, very easy to do as the word is very short.

Avant tout nous remplaçons le mot caché par un mot d'une seule lettre (ligne 3) afin de rendre les tests plus rapides (nous n'avons qu'une lettre à donner pour gagner).

À la ligne 15, nous ré-utilisons un *sauf si* que nous soumettons à la condition: `info.include?('')`. Nous envoyons le message `include?`, en attachant un caractère souligné, qui vérifie si le souligné fait partie de la chaîne construite. S'il n'y a plus de soulignés c'est que le joueur a trouvé toutes les lettres.

La ligne 16 affiche le message *vous gagnez*².

La ligne 17 contient juste le mot `break`, reconnu par *Ruby*, il signale à *Ruby* qu'il faut interrompre la répétition (boucle) en cours.

Nous aurions pu vérifier si le jeu est terminé au même endroit que la vérification d'une lettre correcte (vers la ligne 37), vu que dès que le joueur trouve la dernière lettre le jeu peut s'arrêter, mais nous voulons montrer le mot entier avant d'arrêter le jeu.

Il nous reste à vérifier qu'on peut gagner, ce qui en principe est facile avec un mot si court...

```

$> ruby hangman.rb
-
Wrong letters:

Give me a letter: A
The letter is part of the hidden word
A

You win, PATATE!

```

We won!

The test is interesting because it also shows that the display of the underscores succeeds even if the hidden word is only one letter. Such a case is called an *edge case*.

Ça y est, on a gagné!

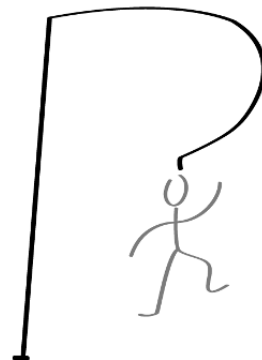
Ce dernier test est intéressant parce qu'il montre également que l'affichage des soulignés réussit même si le mot caché n'a qu'une lettre. On appelle ce genre de cas un *cas limite*.

¹ *PATATE* is the French word for potatoes, it is kind of soft insult, do not take it wrong, we assume that computers hate to lose...

² Ne prenez pas mal le petit ajout affectif du mot *patate*, sans doute que l'ordinateur est mauvais perdant...

12

Déclarer que le joueur a perdu
Declare that the player has lost




Step 12: Declare that the player has lost

Let us face it, after seven tries the player can lose the game if he/she did not find the hidden word.

Étape 12 : Déclarer que le joueur a perdu

Ne nous voilons pas la face, il est possible de perdre si après sept essais le joueur n'a toujours pas trouvé le mot caché.

Mots du code suivant : lost signifie perdu, game signifie jeu, errors signifie erreurs, counter signifie (ici) compteur.

 hangman.rb

```

1  require_relative "word_info"
2
>3  hidden_word = "fuchsia"
4
5  found_letters = []
6  wrong_letters = []
7
>8  errors_counter = 0
9
10 loop do
11
>12   if errors_counter > 6
>13     puts "You lost the game, BANANE!!! The word was #{hidden_word}"
>14     break
>15   end
16
17   info = word_info(hidden_word, found_letters)
18
19   puts info
20   puts
21
22   unless info.include?('_')
23     puts "You win, PATATE!"
24     break
25   end
26
27   print "Wrong letters: "
28   puts wrong_letters.sort.uniq.join(" ")
29   puts
30
31   print "Give me a letter: "
32
33   answer = gets
34   answer = answer.chomp.downcase
35
36   if answer == 'stop'
37     exit
38   end
39
40   if answer.length > 1
41     puts "You must give only one letter!"
42   elsif answer.empty?
43   elsif hidden_word.include?(answer)
44     puts "The letter is part of the hidden word"
45
46     if found_letters.include?(answer)

```

```

47     else
48         found_letters << answer
49     end
50
51     else
52         puts "Invalid letter"
53         wrong_letters << answer.upcase
54     errors_counter = errors_counter + 1
55     end
56
57 end

```

We must count the number of errors, therefore we are assigning the name `errors_counter` to zero (the value) (line 8).

When we conclude that the proposal is wrong, we recall the value associated with `errors_counter`, add 1 (line 54) to it and give the same name to the result. The first time we get `0 + 1`, since we start by associating zero with `errors_counter`. The new value associated is 1. Next time we get an invalid letter, we compute `1 + 1 = 2` that becomes the new value associated with the name `errors_counter`, and so on.

Finally at line 12, before asking for a new proposal, we check if the number of errors is not greater than six (`errors_counter > 6`). If such is the case, we display a message stating the end of the ongoing game and do stop the game (the same way we did before with `break`).

The message (line 13) contains a special feature offered by *Ruby*, the part `#{hidden_word}` in the string, starting with a sharp symbol, followed by an opening brace up to the closing brace. We said before that *Ruby* does not take into account the content of the strings unless they contain the special sequence `#{}`, everything between the braces is processed by *Ruby* and the result of the processing replaces the sequence. In our case, *Ruby* finds that a word named `hidden_word` does exist and uses the associated value. Doing so, we can show the word to the player that he/she did not resolve.

Nous devons compter le nombre d'erreurs, nous associons le nom `errors_counter` (*compteur d'erreurs*) à la valeur zéro (ligne 8).

Lorsqu'on conclue que la proposition est mauvaise, on prend la valeur associée à `errors_counter` lui ajoutons 1 (ligne 54) et donnons le même nom au résultat. La première fois on aura `0 + 1`, puisque nous commençons par associer la valeur zéro à `errors_counter`. La nouvelle valeur associée est donc 1. À la prochaine lettre erronée, on aura `1 + 1 = 2` qui sera la nouvelle valeur associée à `errors_counter`, et ainsi de suite.

Enfin à la ligne 12, avant de demander une nouvelle lettre, nous testons si le nombre d'erreurs n'est pas supérieur à six (`errors_counter > 6`). Si c'est le cas nous affichons un message déclarant la fin du jeu et arrêtons le jeu (de la même manière que précédemment avec `break`).

Le message (ligne 13) contient une particularité proposée par *Ruby*, la partie `#{hidden_word}` dans la chaîne de caractères, marquée par un dièse, suivi par une accolade ouvrante et se terminant par une accolade fermante. Nous avons dit que *Ruby* ne fait aucun cas des chaînes de caractères à moins qu'elles ne contiennent cette séquence spéciale `#{}`, tout ce qui se trouve entre les accolades est traité par *Ruby* et le résultat obtenu apparaîtra à la place de cette séquence. Dans notre cas, *Ruby* va trouver qu'il existe un mot `hidden_word` et en prendre la valeur associée. Cela nous permet de montrer au joueur le mot qu'il n'a pas trouvé.

```
$> ruby hangman.rb  
- - - - -  
Wrong letters:  
  
Give me a letter: Z  
Invalid letter  
- - - - -  
  
Wrong letters: Z  
  
Give me a letter: Z  
Invalid letter  
- - - - -  
  
Wrong letters: Z  
  
Give me a letter: Z  
Invalid letter  
- - - - -  
  
Wrong letters: Z  
  
Give me a letter: Z  
Invalid letter  
- - - - -  
  
Wrong letters: Z  
  
Give me a letter: Z  
Invalid letter  
- - - - -  
  
Wrong letters: Z  
  
Give me a letter: Z  
Invalid letter  
- - - - -  
  
Wrong letters: Z  
  
Give me a letter: Z  
Invalid letter  
You lost the game, BANANE!!! The word was fuchsia
```

Well, we lost now. A bit annoying this computer saying we are *bananas*¹.

Ok, on a perdu. Un peu vexant cet ordinateur qui nous traite de *banane*², non?

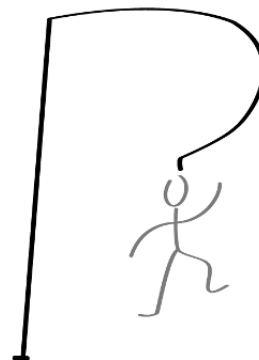
¹ Another soft insult? Yep. I guess the computer is rancorous, because of our past victory.

² L'ordinateur serait-il rancunier?
Se souvient-il de notre victoire
éclatante?

13

Draw the gallows

Dessiner la potence



Step 13: Draw the gallows

A *hangman* game without gallows drawing is not that great. We will be working to draw the gallows and the addition of the parts of a man on every wrong answer.

Our drawing is going to be pretty simple as shown hereafter.

$$\begin{array}{c} \overline{\quad\quad} \\ |/\quad| \\ |\quad0 \\ |/\quad\backslash \\ |\quad| \\ |/\quad\backslash \\ | \\ -\dots \end{array}$$

The gallows only

We are going to write a routine named `draw_gallows` and put the code in a distinct file so that we can focus on our ask.

 gallows.rb

```
1 # draws (writes to the console) a gallows and parts of the hangman, depending on
2 # the number of +errors+
3 def draw_gallows errors
4 end
```

Here is the routine that does not do anything at the moment, it needs the number of errors, we gave the name `errors` to the number that is attached when sending the message `draw_gallows`.

We wrote a comment explaining that the routine displays a gallows and the parts of a hangman based on the number of errors.

gallows.rb

```
1 # draws (writes to the console) a gallows and parts of the hangman, depending on
2 # the number of +errors+
3 def draw_gallows errors
4
5     roof = "      "
```

Étape 13 : Dessiner la potence

Un jeu du *bonhomme pendu* sans dessin de la potence n'est pas folichon. Nous allons nous atteler à dessiner la potence et l'ajout des parties du bonhomme à chaque mauvaise réponse.

Notre dessin sera assez simple comme nous le voyons ci-après.

Juste la puissance

Nous allons écrire une routine appelée `draw_gallows` (*dessine une potence*) et mettre le code dans un fichier distinct nous permettant de nous concentrer sur ce travail.

Mots du code suivant : draw signifie dessine, write signifie écrire, console signifie console, gallows signifie potence, parts signifie parties, depending signifie dépendant, number signifie nombre.

Voici la nouvelle routine qui ne fait rien pour le moment, elle a besoin d'un nombre d'erreurs, nous donnons le nom `errors` à ce nombre joint lors de l'envoi du message `draw gallows`.

Nous avons mis un commentaire expliquant que la routine va afficher une potence et des parties du bonhomme pendu selon le nombre d'erreurs.

Mots du code suivant : roof signifie toit, top signifie sommet, head signifie tête, trunk signifie tonc, hips signifie hanches, legs signifie jambes, bottom signifie bas, foot signifie pied.

```

>6   top    = " | / | "
>7   head   = " |   "
>8   trunk  = " |   "
>9   hips   = " |   "
>10  legs    = " |   "
>11  bottom = " |   "
>12  foot    = " --- "
13
>14  puts roof
>15  puts top
>16  puts head
>17  puts trunk
>18  puts hips
>19  puts legs
>20  puts bottom
>21  puts foot
>22  puts
23
24  end

```

We split the gallows into several parts and give a name to each of them (line 5 to 12). Then, we display them in the right order.

By using several parts, we can place elements, depending on the number of errors, as the head or the legs at their position. Imagine we want to place the head, we use the characters we named `head`, namely:

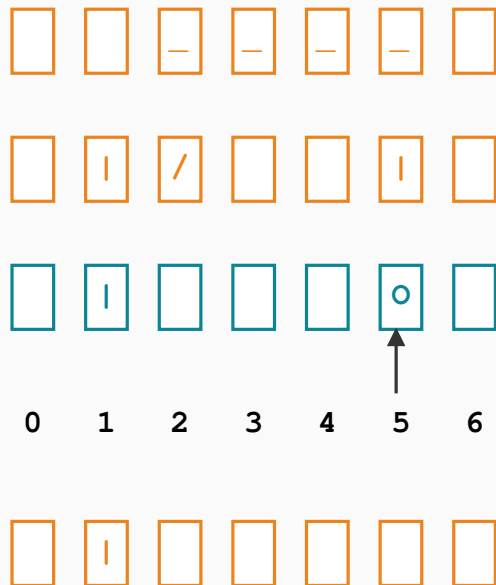
Nous décomposons la potence en plusieurs parties et associons un nom à chacune d'elles (lignes 5 à 12). Ensuite, nous les affichons dans le bon ordre.

En utilisant plusieurs parties, nous pourrions placer, suivant le nombre d'erreurs, des éléments tel que la tête ou les jambes, à la place qui leur est réservée. Ainsi, supposons que nous voulions placer la tête, nous allons prendre la chaîne de caractères appelée `head`, qui est:

0	1	2	3	4	5	6


Set a `0` (capital o) at the 6th position. (it is easier to see the result if we also see the other parts)

Et mettre un `0` à la 6^{ème} position. (il est plus facile de voir le résultat si l'on montre les parties voisines.)



We will do the same for the rest of the body parts.

Nous ferons de même pour toutes les parties du corps.

 gallows.rb

```

1  # draws (writes to the console) a gallows and parts of the hangman, depending on
2  # the number of +errors+
3  def draw_gallows errors
4
5      roof   = "      _"
6      top    = "  /  |  "
7      head   = "  |  "
8      trunk  = "  |  "
9      hips   = "  |  "
10     legs    = "  |  "
11     bottom  = "  |  "
12     foot    = "  ---  "
13
14     head[5] = "0" if errors > 0
15
16     trunk[4] = "/" if errors > 1
17     trunk[5] = "|" if errors > 2
18     trunk[6] = "\\\" if errors > 3
19
20     hips[5] = "|" if errors > 4
21
22     legs[4] = "/" if errors > 5
23     legs[6] = "\\\" if errors > 6
24
25     puts roof
26     puts top
27     puts head
28     puts trunk
29     puts hips
30     puts legs
31     puts bottom
32     puts foot

```

```
33     puts
34
35     end
```

Lines 14 to 23 add the different parts of the body depending on the number of errors. The code uses a syntactic sugar, specific to *Ruby*, that helps in writing code saying: *do next thing if something is true* (it sounds more *natural*). For instance, on line 14, if the number of errors is greater than zero (there are errors) then we need to set the `0` (used to show the head).

To test the new routine, we are not going to clutter all the code of the game but rather focus on the drawing of the gallows.

 test_gallows_drawing.rb

```
1  require_relative 'gallows'
2
3  draw_gallows 0
4  draw_gallows 1
5  draw_gallows 3
6  draw_gallows 6
7  draw_gallows 7
```

We display several versions: no error, one, three, six and seven errors.

Les lignes 14 à 23 ajoutent les différentes parties du corps selon le nombre d'erreurs. Le code utilise une facilité de notation, propre à *Ruby*, qui permet d'écrire du code disant: *fait ceci si ceci est vrai*. Par exemple, à la ligne 14, si le nombre d'erreurs est plus grand que zéro (c'est-à-dire dès qu'il y a une erreur) alors il faut placer le `0` (symbolisant la tête).

Pour tester cette nouvelle routine, nous n'allons pas encombrer tout le jeu mais nous concentrer sur ce dessin de potence.

Nous allons afficher différentes versions : pas d'erreur, une, trois, six et sept erreurs.


```
$> ruby test_gallows_drawing.rb
```

```
  |_____|
  | /   |
  |     |
  |     |
  |     |
  |     |
  |     |
  |_____|
```

```
  |_____|
  | /   |
  |     |
  |     |
  |     |
  |     |
  |     |
  |_____|
```

```
  |_____|
  | /   |
  |     |
  |     |
  |     |
  |     |
  |     |
  |_____|
```

```
  |_____|
  | /   |
  |     |
  |     |
  |     |
  |     |
  |     |
  |_____|
```

```
  |_____|
  | /   |
  |     |
  |     |
  |     |
  |     |
  |     |
  |_____|
```

The drawing of the gallows is ready to use.

Le dessin de la potence en fonction du nombre d'erreur est prêt à l'emploi.

Back to the game

We just have to put the pieces together...

Retour au jeu

Nous n'avons plus qu'à mettre les morceaux ensemble...

 hangman.rb

```
1  require_relative "word_info"
➤2 require_relative "gallows"
```

```
3
4 hidden_word = "fuchsia"
5
6 found_letters = []
7 wrong_letters = []
8
9 errors_counter = 0
10
11 loop do
12
13   draw_gallows errors_counter
14
15   if errors_counter > 6
16     puts "You lost the game, BANANE!!! The word was #{hidden_word}"
17     break
18   end
19
20   info = word_info(hidden_word, found_letters)
21
22   puts info
23   puts
24
25   unless info.include?('_')
26     puts "You win, PATATE!"
27     break
28   end
29
30   print "Wrong letters: "
31   puts wrong_letters.sort.uniq.join(" ")
32   puts
33
34   print "Give me a letter: "
35
36   answer = gets
37   answer = answer.chomp.downcase
38
39   if answer == 'stop'
40     exit
41   end
42
43   if answer.length > 1
44     puts "You must give only one letter!"
45   elsif answer.empty?
46   elsif hidden_word.include?(answer)
47     puts "The letter is part of the hidden word"
48
49     if found_letters.include?(answer)
50       else
51         found_letters << answer
52       end
53
54     else
55       puts "Invalid letter"
56       wrong_letters << answer.upcase
57       errors_counter = errors_counter + 1
58     end
59
60   end
```

We ask *Ruby* to take advantage of the services provided by `gallows.rb`, on line 2. We display the gallows (line 13) before each prompt and the check if the player loses the game, and we show the hanged man and the message saying that he/she has lost.

Even if we are sure that everything is in place, we anyway have to test.

Check first that we still have the behavior we had so far.

Nous demandons à profiter des services rendus par `gallows.rb` à la ligne 2. Nous affichons la potence (ligne 13) avant chaque question et la vérification d'échec du joueur, ainsi nous montrerons le bonhomme pendu et le message lui indiquant qu'il a perdu.

Même si nous sommes sûr que tout est en place cela ne nous dispense pas de devoir tester.

Vérifions d'abord que le déroulement précédent est toujours correcte.

```
$> ruby hangman.rb
```

1/

Wrong letters:

Give me a letter: F

The letter is part of the hidden word

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	-----

F

Wrong letters:

Give me a letter: **H**

The letter is part of the hidden word

	/	

$$F \quad \quad \quad H$$

Wrong letters:

Give me a letter: stop

Let us test some invalid letters.

Testons quelques mauvaises lettres.

```
Give me a letter: Z
Invalid letter
```

$$\begin{array}{c} \hline | / | \\ | 0 \\ | \\ | \\ | \\ | \\ \hline \end{array}$$

```
Give me a letter: Z
Invalid letter
```

$$\begin{array}{r} \hline | / | \\ | 0 \\ | / \\ | \\ | \\ | \\ \hline \end{array}$$

```
Give me a letter: Z
Invalid letter
```

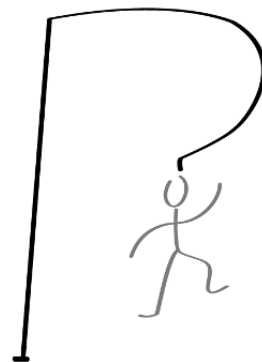
	/	
		0
	/	
- - -		

Wrong letters: Z

14

Gérer les lettres répétées


Support the repeated letters



Step 14: Support the repeated letters

We left aside the issue of the repeated letters, it's time to address the problem.

Once again, we are going to focus on the case of the repeated letters and use a test file (`test_word_info.rb`).

 `test_word_info.rb`

```
1 require_relative 'word_info'
2
3 puts word_info('palladium', ['p'])
4 puts word_info('palladium', ['p', 'l'])
5 puts word_info('palladium', ['p', 'l', 'a'])
```

We used the word “palladium” because it contains two interesting cases: two identical letters following each other and two letters not following each other.

We also see on lines 3, 4, and 5 how we can send a collection (marked by the opening and closing square brackets) containing elements, here the letters `P`, `L` and `A` on line 5.

Let us execute the code to check that it shows the default.

```
$> ruby test_word_info.rb
P _ _ _ _ _
P _ L _ _ _ _
P A L _ _ _ _
```

The second `A` and the second `L` are missing.

Let us see again the current version of the `word_info` routine.

 `word_info.rb`

Étape 14 : Gérer les lettres répétées

Nous avons laissé de côté le cas des mots avec des lettres qui se répètent, il est temps d'y revenir.

Une fois encore, nous allons travailler exclusivement sur le cas des mots avec des lettres qui se répètent et utiliser un fichier de test (`test_word_info.rb`).

Nous avons utilisé le mot “palladium” car il comporte deux cas intéressants: deux lettres identiques qui se suivent et deux lettres qui ne se suivent pas.

Nous voyons également aux lignes 3, 4 et 5 comment on peut donner une collection (marquée par les crochets ouvrant et fermant) avec des éléments, ici les lettres `P`, `L` et `A` à la ligne 5.

Exécutons le pour voir qu'il montre bien le défaut.

Il nous manque le deuxième `A` et le deuxième `L`.

Rappelons également la version actuelle de la routine `word_info`.

```

1  # +letters+ should be part of the given +word+ and down-case
2  def word_info word, letters
3
4      n = word.length
5      underscores = "_" * n
6
7      letters.each do |letter|
8          i = word.index(letter)
9          underscores[2 * i] = letter.upcase
10     end
11
12     underscores
13
14 end

```

The code fails in the part where we replace the underscores with the letters (lines 8 and 9), because we only replace one character.

The downside is that the `index` message only gives the position of the first letter.

Fortunately, we can join two *things* to the `index` message: what we are looking for but also the position from which to start searching for a match.

As `index` does return *nothing* (`nil`) when it does not find what we ask, we can start by searching at the beginning of the word, then from the next position and so on until the searched letter is not found anymore or we past the end of the word.

When we say repeated, we think to the `loop` word and its block of instructions to repeat.

 word_info.rb

```

1  # +letters+ should be part of the given +word+ and down-case
2  def word_info word, letters
3
4      n = word.length
5      underscores = "_" * n
6
7      letters.each do |letter|
8
9          i = -1
10
11         loop do
12
13             i = word.index(letter, i + 1)
14
15             break if i.nil?
16
17             underscores[2 * i] = letter.upcase
18
19         end

```

Le code dérape dans la partie où nous remplaçons les soulignés par les lettres (lignes 8 et 9), car nous ne remplaçons qu'un seul caractère.

L'ennui c'est que le message `index` ne nous donne que la position de la première lettre.

Heureusement on peut joindre deux choses au message `index` : ce qu'on recherche mais également la position à partir de laquelle il faut commencer la recherche.

Comme `index` ne renvoi *rien* (`nil`) s'il ne trouve pas ce qu'on lui demande, nous pouvons commencer par rechercher au début du mot, puis à partir de la position suivante et ainsi de suite jusqu'à ce que la lettre ne soit plus trouvée ou qu'on soit au-delà de la fin du mot.

Quand on parle de répétition, on pense au mot `loop` et au bloc d'instructions qu'il faut répéter.


```

20
21     end
22
23     underscores
24
25     end

```

Let us recall what happens, on line 7 we start a processing on each found letter to replace the underscores by the letter. In the block we have access to all the letters thanks to the word `letter`.

We do, as we said, a loop so that we can find all occurrences of each letter (line 11).

We use the option of `index` of giving a starting position for the search on line 13. As we need to keep track of the previous position each time we want to search a possible repetition, we do so by giving the name `i` to the position of each repetition.

We give the name `i` to a value equal to `-1`, right before the loop

Since we want to search starting at the position following the one where some letter appears (to prevent from finding again the same occurrence), we take the value bound to `i` and add 1 to it. But the first time we make the search for a letter, we must start at the position 0. Therefore, having attached `-1` to the name `i` (line 9) will result, the first time, in computing: `-1 + 1` which equals `0` – the first position.

If we rerun the test.

Rappelons ce qui se passe. À la ligne 7 nous commençons un travail sur chaque lettre trouvée pour remplacer les soulignés par la lettre. Dans le bloc on dispose des différentes lettres grâce au mot `letter`.

Nous faisons, comme nous l'avons dit, une boucle pour retrouver toutes les apparitions de chaque lettre (ligne 11).

Nous exploitons la possibilité de donner un point de départ pour la recherche à la ligne 13. Comme nous devons conserver la position précédente à chaque fois que nous voulons chercher une éventuelle répétition, nous gardons la trace de la position en lui attribuant le nom `i`.

Nous avons donné le nom `i` à la valeur `-1`, juste avant la boucle (l'utilisation du `-1` est une petite astuce).

Puisque nous voulons chercher à partir de la position qui suit la position où se trouve une lettre (afin de ne pas retomber sur la même occurrence), nous prenons la valeur associée à `i` et nous ajoutons 1. Mais la première fois où nous voulons faire une recherche, nous devons partir de la position 0, avoir associé `-1` au nom `i` (ligne 9) fera que la première fois on aura le calcul: `-1 + 1` qui donne `0` – la première position.

Si on relance le test.

```

$> ruby test_word_info.rb
P _ _ _ _ _
P _ L L _ _ _
P A L L A _ _ _

```

And now, we get all the letters.

The game is complete but it has a big flaw: it always hides the same word.

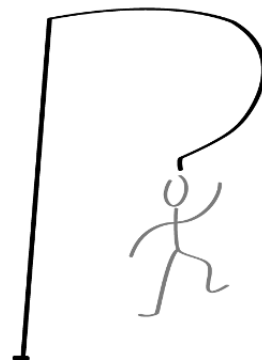
Cette fois nous avons bien toutes les lettres.

Le jeu est complet mais il comporte un gros défaut: il propose toujours le même mot caché.

15

Choose a random word


Choisir un mot au hasard



Step 15: Choose a random word

If we want to offer different words, we need a list of words from which we can draw a word.

A simple way is to use a file containing one word per line¹, we name this file a *dictionary*. Here is an example with several words.

 dictionary_fr.txt

```
a
Bonjour
en
été
```

In our example we used French words and added a suffix (`_fr`) to the file name to show it contains French words.

We must start by reading (loading) all words and fill a collection that we can then use to draw words. But we have some constraints:

- the words must all be lowercase
- we do not want to keep very short words
- we do not want accents (meaning we must *remove the accents*, because our game does nothing to consider that a letter with an accent is equivalent to the same letter without an accent)

Read the dictionary

We are going to isolate this task in a file where we are adding a routine that loads the dictionary. For more flexibility, the routine loads the file received as argument (*attachment*) and only keeps the words having a length greater than a number received as a second argument.

 dictionary_loader.rb

```
1  def load_dictionary file, minimum_length
2
3    words = []
4
5    words
6
7  end
```

Étape 15 : Choisir un mot au hasard

Pour pouvoir proposer des mots différents, il faut disposer d'une liste dans laquelle on peut tirer un mot au sort.

Un moyen simple est d'avoir un fichier contenant un mot par ligne², nous appelons ce fichier un *dictionnaire*. Voici un exemple avec quelques mots.

Nous avons donné un nom contenant la particule `_fr` pour marquer qu'il contient des mots français.

Nous devons commencer par lire (charger) tous les mots et remplir une collection que l'on pourra ensuite utiliser pour tirer des mots. Nous avons quelques autres contraintes :

- les mots doivent être en minuscule
- il ne faut pas garder les mots trop courts
- il ne faut pas de caractère accentué (c'est-à-dire qu'il faut *retirer les accents*, car notre jeu ne fait aucun traitement pour considérer qu'un caractère accentué est équivalent au même caractère sans accent)

Lire le dictionnaire

Nous allons isoler ce travail dans un fichier dans lequel nous allons créer une routine qui charge le dictionnaire. Pour avoir plus de souplesse, la routine charge le fichier qu'on lui passe en argument (*en pièce jointe*) et ne garde que les mots ayant un nombre de caractères supérieurs à un deuxième argument.

Mots du code suivant : `dictionary` signifie *dictionnaire*, `loader` signifie *chargeur*, `load` signifie *charger*, `file` signifie *fichier*.


Our first version creates a new empty collection and names it `words`.

We must read all the file and add every word to the collection.

Cette première version crée une nouvelle collection vide et lui attribue le nom `words`.

Nous devons lire tout le fichier et ajouter tous les mots dans la collection.

Mots du code suivant : `open` signifie ouvrir, `line` signifie ligne.

 dictionary_loader.rb


```
1  def load_dictionary file, minimum_length
2
3    words = []
4
5    open(file, 'r:utf-8').each do |line|
6      end
7
8    words
9
10  end
```

To read a file, we first have to open it (line 5), like we would do to read a book. *Ruby* offers a word `open` that expects a file name (of course, the routine uses the one it receives, `file`) and the type of handling we want to apply, because we could want to read a file, write to a file or both. If we want to open a file to read it, we pass the string `'r'`. We actually give `'r:utf-8'`, to keep a very long story short, the `:utf-8` part indicates how the characters are encoded in the file³.

As a result to the `open` message, *Ruby* gives a *file* to which we can send new messages, specific to it. We use the `each` message that reads all the lines of the file and calls the given block for each line.

Pour lire un fichier, il faut l'ouvrir (ligne 5), comme nous ferions pour lire un livre. *Ruby* offre le mot `open` auquel il faut donner le nom du fichier (nous utilisons le fichier reçu, `file`) et le type de manipulation qu'on veut faire, on pourrait vouloir lire, écrire dans un fichier ou les deux. Si on veut ouvrir le fichier pour le lire on donne la chaîne `'r'`. Dans notre cas nous donnons `'r:utf-8'`, sans rentrer dans trop de détails, la partie `:utf-8` est une indication sur la manière de coder les caractères dans le fichier⁴.

En réponse au message `open`, *Ruby* nous donne un *fichier* auquel nous pouvons envoyer des messages qui lui sont propres. Nous utilisons le message `each` qui a pour effet de lire tout le fichier et de passer au bloc joint chaque ligne.

 dictionary_loader.rb

```

1  def load_dictionary file, minimum_length
2
3      words = []
4
5      open(file, 'r:utf-8').each do |line|
6
7          line = line.chomp.downcase
8
9          if line.length >= minimum_length
10             words << line
11         end
12     end
13
14     words
15
16
17 end

```

On line 7, we start, as we did before, by removing the *carriage return* character (in a simple text file, lines are distinguished/separated by *carriage return* characters). We transform the word in lowercase with the `downcase` message.

Since each line is supposed to contain only one word, we just check if the word has at least the required number of letters (line 9), if so, we can add the word to the collection (line 10). The `>=` symbol, in *Ruby*, means “greater or equal”.

Let us check if everything happens as expected. We create a new test file.

 test_dictionary.rb

```

1  require_relative 'dictionary_loader'
2
3  words = load_dictionary('dictionary_fr.txt', 3)
4
5  puts "*** 3 characters ***"
6  puts words.length
7  puts words

```

We reference the file we want to test on line 1, then we call the new routine to load the file that contains the four words (line 3) and give the name `words` to the produced collection. Then, we display a short text explaining the test, the number of words in the collection and the collection.

À la ligne 7, nous commençons, comme nous l'avons déjà fait plus haut, par retirer le caractère *retour chariot* (dans un fichier de texte simple, les lignes sont différenciées/séparées par un *retour chariot*). Nous transformons le mot en minuscule avec le message `downcase`.

Puisque chaque ligne est censée contenir un seul mot, nous vérifions si le mot a au moins le nombre de lettres demandées (ligne 9), si c'est le cas nous ajoutons le mot à la collection (ligne 10). Le symbole `>=`, en *Ruby*, signifie “plus grand ou égal”.

Vérifions si tout se passe comme prévu jusqu'ici. Nous créons un autre fichier de test.

Nous faisons référence au fichier que nous voulons tester (ligne 1), appelons la nouvelle routine pour charger le fichier contenant les quatre mots (ligne 3) et donnons le nom `words` à la collection produite. Puis, nous affichons un petit texte expliquant le test, le nombre de mots de la collection ainsi que la collection.

```
$> ruby test_dictionary.rb
** 3 characters **
2
bonjour
été
```

Since we set three characters minimum, only three words are kept...


We also see that the capital letter of *bonjour* (*goodmorning* in French) is now a lowercase.

What would happen if we increased the minimum number of letters? Say twenty characters? Let us add such a test.

Puisque nous avons mis trois caractères minimum, seuls deux mots sont conservés...

Nous voyons également que la majuscule du mot *bonjour* est bien devenue une minuscule.

Que se passerait-il si nous augmentions le nombre de caractères minimum? Genre vingt caractères? Ajoutons ce test.

 test_dictionary.rb

```
1 words = load_dictionary('dictionary_fr.txt', 20)
2
3 puts "*** 20 characters ***"
4 puts words.length
5 puts words
```

```
$> ruby test_dictionary.rb
** 3 characters **
2
bonjour
été
** 20 characters **
0
```

Nothing, as the all have less than twenty letters.

Write a third test where we take all the words (zero character minimum).

Il n'y a aucun mot puisqu'ils ont tous moins de vingt lettres.

Faisons un troisième test où nous prenons tous les mots (zéro caractère minimum).

 test_dictionary.rb

```
1 words = load_dictionary('dictionary_fr.txt', 0)
2
3 puts "*** no limit ***"
4 puts words.length
5 puts words
```


```
$> ruby test_dictionary.rb
** 3 characters **
2
bonjour
été
** 20 characters **
0
** no limit **
4
a
bonjour
en
été
```

No word was excluded.

Aucun mot n'a été écarté.

Now, we must address the problem of accented characters (see *été*, summer in French).

Nous devons maintenant nous pencher sur le problème des caractères accentués.

 dictionary_loader.rb

```
1 def load_dictionary file, minimum_length
2
3   words = []
4
5   open(file, 'r:utf-8').each do |line|
6
7     line = line.chomp.downcase
8
9     if line.length >= minimum_length
10      line = line.tr('àéèèèùûîïç', 'aeueeuuic')
11      words << line
12    end
13  end
14
15  words
16
17
18 end
```

In French, we do not consider, for example, the `e` without an accent, the `e` acute accent and the `e` grave accent are different letters. The same goes for some other letters and characters like `ç` equivalent of `c`. We added the line 10 that uses two strings of characters, the first contains a sequence of accented characters and the second the same sequence without the accents. We send the `tr` message with the two strings to each word. The word will process the message by replacing each character matching one letter of the first string with the corresponding letter of the second, we can say that it makes a translation from one word to another.

If we rerun the test, we see that the word `été` has no accents anymore.


Pour les mots français, on ne considère pas que, par exemple, le `e` sans accent, le `e` accent aigu et le `e` accent grave sont des lettres différentes. Il en va de même pour d'autres lettres. Nous avons ajouté la ligne 10 qui utilise deux chaînes de caractères, la première contient la suite des caractères accentués et la deuxième la même suite sans les accents. Nous envoyons le message `tr` avec ces deux chaînes à chaque mot. Le mot va traiter le message `tr` en remplaçant chaque caractère tel qu'il apparaît dans la première chaîne par le caractère correspondant de la deuxième, on peut dire qu'il fait une traduction.

Si nous relançons le test, nous verrons que le mot `été` n'a plus d'accents.

```
$> ruby test_dictionary.rb
** 3 characters **
2
bonjour
ete
** 20 characters **
0
** no limit **
4
a
bonjour
en
ete
```

Back to the game

Load a real file and draw a word makes testing more challenging (how can we know what word was drawn?). We are going to use a trick and put only one word in a dictionary file.

 dictionary_en.txt

addition

We first adapt the game to use the routine that loads the dictionary.

Retour au jeu

Charger un fichier réel et tirer au sort, rend le test plus difficile (comment peut-on savoir quel mot a été choisi?). On va ruser et mettre un fichier avec un seul mot.

Nous commençons par adapter le jeu pour utiliser la routine qui charge le dictionnaire.

Mots du code suivant : sample signifie échantillon.

 hangman.rb

```
1  require_relative "word_info"
2  require_relative "gallows"
>3  require_relative "dictionary_loader"
4
>5  words = load_dictionary('dictionary_en.txt', 3)
>6
>7  hidden_word = words.sample
8
9  found_letters = []
10 wrong_letters = []
11
12 errors_counter = 0
13
14 loop do
15
16   draw_gallows errors_counter
17
18   if errors_counter > 6
19     puts "You lost the game, BANANE!!! The word was #{hidden_word}"
20     break
21   end
22
23   info = word_info(hidden_word, found_letters)
24
25   puts info
```



```

26 puts
27
28 unless info.include?('_')
29   puts "You win, PATATE!"
30   break
31 end
32
33 print "Wrong letters: "
34 puts wrong_letters.sort.uniq.join(" ")
35 puts
36
37 print "Give me a letter: "
38
39 answer = gets
40 answer = answer.chomp.downcase
41
42 if answer == 'stop'
43   exit
44 end
45
46 if answer.length > 1
47   puts "You must give only one letter!"
48 elsif answer.empty?
49 elsif hidden_word.include?(answer)
50   puts "The letter is part of the hidden word"
51
52   if found_letters.include?(answer)
53   else
54     found_letters << answer
55   end
56
57 else
58   puts "Invalid letter"
59   wrong_letters << answer.upcase
60   errors_counter = errors_counter + 1
61 end
62
63 end

```

On line 3 we reference the new *dictionary loader* that we use on line 5. We limit to words having three or more letters. We draw a random word among the collection by sending the `sample` message which returns a random word among the words of the collection (line 7).

Let us check a last time that everything runs fine.

À la ligne 3 nous faisons référence au *chargeur de dictionnaire* que nous utilisons à la ligne 5. Nous limitons les mots à trois lettres. Nous tirons un mot au hasard parmi la collection en envoyant le message `sample` qui produit un mot au hasard parmi les mots de la collection (ligne 7).

Vérifions une dernière fois que tout se déroule correctement.

Figure 1

Wrong letters:

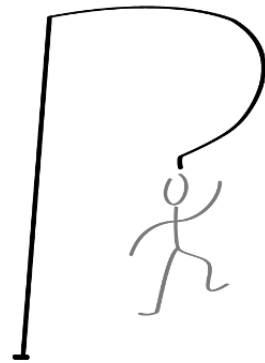
¹ Some systems, like Linux, have such files that we can use. The file `/usr/share/dict/words` contains about hundred thousand words.

² Sur certains systèmes, tel que Linux, il existe des fichiers que l'on peut utiliser. Le fichier `/usr/share/dict/french` contient plus de cent-mille mots.

³ Because we have words with accents we need to use a special encoding, *UTF-8* for instance, a deeply sad story of the IT...

⁴ Dans notre cas le fait qu'on ait des accents nous oblige à avoir un codage spécial, on a choisit celui appelé *UTF-8*, c'est une des histoires sordides de l'informatique...

Conclusion



Conclusion

We presented by creating the game several things about programming. The programming language used to implement the game is also an example of programming language, other languages impose more or less the same rules.

Here are the important points that we covered:

- the machine does nothing by itself, we give all the instructions to follow on how to make the service we want. We must imagine the way to proceed so that we can define the instructions,
- we must test, again and again. Test always. Even the slightest change could break what has been built so far. We can help ourselves by writing programs that test our programs, what we only did partially here for the routine that builds the information about the hidden word and the one that loads the dictionary of words,
- the programming language imposes a set of rules that we must follow strictly because the computer is not able to rectify the ambiguities and the inaccuracies that would appear if we did not carefully apply the rules. Notice that all the tools that process the source code of a programming language were also designed and written by people,
- we must proceed step by step and trying to do everything at the same time is useless, we could get lost in details and we are probably unable to grasp the whole problem,
- likewise, we must break the work into isolated tasks to facilitate the focus on a specific topic and find a solution which will be easier to maintain and to share with other people,
- when we test, we must find unexpected cases, if possible. If we do not find them, they will pop up anyway. To what extent is it important to further search for these *extreme* cases depends on the emergency and the importance of the application. Writing a program that puts people's lives in danger should always be of a flawless precision. Write a little game, moreover for a personal use, can contain inconsistencies without this affecting the universe.

Conclusion

Nous avons montré en réalisant ce jeu différents éléments concernant la programmation. Le langage de programmation utilisé est lui aussi un exemple de langage, d'autres langages imposent des règles plus ou moins identiques.

Voici les points importants que nous avons illustrés :

- la machine ne fait rien d'elle même, nous lui donnons les instructions à suivre pour rendre le service que nous voulons. Nous devons imaginer la manière de procéder afin de pouvoir établir ces instructions,
- il faut tester, toujours tester. Le moindre changement pourrait casser ce qui a été construit jusque là. Nous pouvons nous aider en écrivant des programmes qui testent des programmes, ce que nous n'avons fait que partiellement pour tester la routine qui construit l'information sur les lettres du mot et celle du chargement du dictionnaire,
- le langage de programmation impose des règles qu'il faut suivre car l'ordinateur n'est pas capable de rectifier les ambiguïtés et imprécisions que cela donnerait si on ne suivait pas scrupuleusement les règles. Notons que les outils qui mettent en oeuvre le code d'un langage de programmation ont eux-mêmes été imaginés et écrits par des personnes,
- il faut procéder par étape, il ne sert à rien de vouloir tout faire d'un coup, on se perdrait dans les détails et on n'est probablement pas capable d'appréhender le problème dans sa globalité,
- de même, il faut diviser le travail en tâches isolées pour faciliter la concentration sur un sujet précis et arriver à une solution qui sera plus facile à gérer et à partager avec d'autres personnes,
- lorsqu'on teste, il faut trouver les cas inattendus, si possible. Si on ne les trouve pas tous, ils apparaîtront d'eux-mêmes. Dans quelle mesure il est important de pousser la recherche de ces cas *extrêmes* jusqu'au bout va dépendre de l'urgence du travail et de son importance. Écrire un programme qui met la vie de gens en danger doit impérativement faire l'objet d'une précision sans faille. Écrire un petit jeu, qui plus est à usage privé, peut comporter des lacunes sans que cela n'affecte l'univers.