Language Tutorial

This section serves as an introduction to the *Words* programming language. Our aim in this tutorial is to show you the essential parts of the programming language in real programs to get you started as quickly as possible. From the information in this tutorial, you should be able to design your own interactive animated worlds.

As you dive deeper into *Words* programming, you may find it useful to read the Reference Manual to learn more details about the language.

1. Setup

1.1. System Requirements

The *Words* interpreter (which runs your *Words* programs) was itself created in Java. So, to run *Words* programs, you'll need the Java Runtime Environment (JRE) on your computer. You can download and install the latest JRE from Oracle at http://www.oracle.com/technetwork/java/javase/downloads/index.html. Select JRE, then download and run right installer for your computer.

1.2. Downloading and Building *Words*

Currently, the *Words* interpreter is provided as source code hosted on GitHub. To run *Words* programs, you will have to download the *Words* source code and build it on your computer. To download the source code, you will need git, available for download at http://git-scm.com/downloads. To build the source code, you will need ant, available for download at https://ant.apache.org/bindownload.cgi.

You can download the *Words* source code by opening a command prompt (sometimes called the terminal or console) in the directory (folder) where you would like to store *Words*, then executing the following command:

```
git clone https://github.com/Aalk4308/Words.git
```

This will create a new directory on your computer called Words. To build *Words*, enter that directory and then execute the following command:

```
ant
```

Congratulations! You have successfully downloaded and built *Words*. You're now ready to run and write your first programs.

1.3. Running *Words*

To run *Words*, enter the Words directory you created and execute the following command:

```
java -jar jar/Words.jar
```

You will see a text prompt (described below) where you can write statements in the *Words* language and see them executed, as well as a blank grid called the “board” or “world”.

Words also comes bundled with full sample programs in the samples directory. You can run one of these programs with the following command:

```
java -jar jar/Words.jar samples/HelloWorld.words
```

When you run a program, *Words* processes the entire file (in this case, HelloWorld.words), then presents you with the console to allow you to further interact with the program.

You can follow along the rest of this tutorial by writing *Words* statements directly into the *Words* console. But remember, you can also create a *Words* programs as a separate file like the ones in the sample directory using a text editor. Once you gain experience with *Words*, you will probably find it easier to write programs in a file so that you can edit them more easily.

2. Basic *Words* Features

2.1. Objects

Start up the *Words* environment. In front of you is a blank world and a text prompt called the console. In this world, you will create **objects**. Each object can **move** around the world, **say** statements, and hold **properties**.

Let’s begin by creating two objects, Bob and Alice, on the board. Type into your console

```
Bob is a thing at 0,0.
Alice is a thing at 1,0.
```

and hit enter. Soon, you should see Bob appear in the middle of the world and Alice appear to his right. Try moving Bob around by typing in the console

```
Make Bob move left 3.
```

and you should soon see Bob move left 3 spaces on the board. Try changing the direction he moves, or try changing how far he moves — each **frame**, when moves are performed, you should see him move around the board.

In addition to moving around the board, you can also make Bob say things. Type into the console

```
Make Bob say "Hello World".
```

and watch Bob's text prompt change to say "Hello World."

In addition to performing actions, objects can also hold properties. These properties can be used in later calculations or directly in an object's actions. For instance, try typing into your console:

```
Bob's catchphrase is "heyyyyyyyy".
Make Bob say Bob's catchphrase.
```

You should now see Bob say "heyyyyyyyy." In the first line, you set the name of Bob's "catchphrase" property to be "heyyyyyyyy." In the second line, you recalled the property by requesting "Bob's catchphrase."

Congratulations! You've created your very first object, you've made it move, you've given it properties, and you've made it say things. These are the most basic features that an object can perform.

2.2. Object Action Queue

Each object contains an **action queue** which contains the actions the object will perform in the future. You can think of an object's action queue as its personal "to-do list". The *Words* environment keeps track of **frames**, which are set amounts of time (defaulted to be 1 second per frame). During each frame, you can enqueue actions onto an object (add to their to-do list) by typing statements at the console or by loading a program from a file. At the end of each frame, the *Words* environment dequeues the first action from each object's action queue, and executes these actions simultaneously (checks off the first item on every object's to-do list).

To enqueue a statement on an object's action queue, we use the `Make` command. For an example, try running:

```
Make Bob move left 4.
Make Bob say "hey".
Make Alice move left 2.
Make Alice say "hey".
Make Bob move right 4.
```

Nine actions are enqueued onto Bob's action queue, and two actions are enqueued onto Alice's action queue. For the next two frames, both Bob and Alice will move left. In the third frame, Alice will speak while Bob continues moving left. In the fourth frame, Bob continues moving left, and In the fifth frame, he returns Alice's greeting. For the next four frames, Bob moves right to return to his starting position.

Each time an action is enqueued, it goes to the back of the queue, where it will be executed after all other actions in the queue. However, actions can be forced to the front of the queue by using the keyword `now`. For instance, let's give Bob a long list of actions. Type into your console:

```
Make Bob move left 1000.
```

He should now be moving left for the next thousand frames. However, say you suddenly want to make him move right for two frames. You can type into your console:

```
Make Bob move right 2 now.
```

For two frames, you should see Bob move right before he continues moving left for however many frames he has left in his thousand-frame move. You can also pause Bob by using the `wait` action, which halts the execution of an object's action queue for a set number of frames. Try typing:

```
Make Bob wait 10 now.
```

Bob will now pause for 10 frames before continuing to execute his action queue. Once again, by using the `now` command, the waiting was put at the front of the command. If the `now` keyword was not used, Bob would have finished his current action queue (the 1,000 moves to the left), and then paused for 10 seconds.

An object's item queue can also be cleared by using the `Stop` command. Depending on your reading speed, Bob may still be moving left on the board, but you're really tired of watching him move left. To stop him and completely clear his action queue, just type:

```
Stop Bob.
```

Basic actions, custom actions (discussed in the next section), and property modifications can all be enqueued onto an object's action queue. Other commands which do not use the `Make` command are executed before action queues are dequeued. For instance, try running the following:

```
Bob's catchphrase is "yo".
Make Bob say Bob's catchphrase.
Make Bob's catchphrase be "Hello".
Make Bob say Bob's catchphrase.
```

Before Bob says his catchphrase at the next frame's execution, it is set to be "yo". Then, three actions are placed in Bob's action queue and the first one is executed. Therefore, Bob will say

“yo” in the first frame. Another frame will pass where Bob’s catchphrase is set to be “Hello”. Then, in the third frame, he will say “Hello”, which had been updated in the second frame.

2.3. Classes

So far, you created Bob and Alice by defining them to each be a `thing`. However, they aren’t only things — they’re persons! Persons, like any class of things, can have special properties and actions that apply to them only and not all things. This is why *Words* has the concept of classes. When an object is created, it is assigned a **class**. Each class can contain a number of properties and **custom actions** that get inherited to the object.

Let’s create a person class for Bob and Alice. Type into your console:

```
A person is a thing which {
      has a height of 5.9.
      has a weight of 150.
      can jump which means {
            Make them move up 2.
            Make them move down 2.
      }
}
```

First, we defined the class `person` as a subclass of `thing`. This means that `person` automatically inherits all properties and actions of `thing`. Later on if you’d like, you can make other classes (like `adult` or `child`) that are subclasses of person, and they would inherit the properties and actions of `person`. In this way, you create a *hierarchy* of classes that eventually lead back to `thing`.

To create new properties and actions in the `person` class we used the `which` keyword and opened a block of statements using a curly brace. You can see that each person will start off with a height of 5.9 and a weight of 150. Any new person will have these properties by default. Additionally, each person will be able to `jump`, which we have defined in the inner block.

You can’t change an object’s class once it’s been created, so to make Bob and Alice persons, we’ll have to remove and recreate them. Let’s make Bob and Alice be persons. Type into your console:

```
Remove Bob.
Remove Alice.
Bob is a person at 0,0.
Alice is a person at 2,0.
Make Bob jump.
Make Bob say "My height is " + Bob's height.
```

```
Make Bob's height be 6.2.
Make Bob say "Wow, I just grew to " + Bob's height.
Make Alice say "My height is " + Alice's height.
```

After we remove Bob as a `thing` and create Bob as a `person`, we watch him jump. We then watch him say his height, which is the default of the person class. Then we change his height and watch him say his new height. Notice that Alice's height has not changed — only Bob's height did. The changes to one object's properties don't affect the properties of other objects of the same class.

2.4. Looping

It's possible to make objects repeat a sequence of actions multiple times by **looping**. There are two separate types of loops. Try typing into your console:

```
Repeat 5 times {
      Make Bob say "I'm going to jump".
      Make Bob jump.
      Make Bob say "I just finished jumping".
}
```

Watch and see Bob repeat these actions 5 times.

It is also possible to make Bob do something until a certain condition is met. For instance, try typing:

```
Bob's jump_counter is 0.
While Bob's jump_counter < 3 {
      Make Bob say "I'm going to jump".
      Make Bob jump.
      Bob's jump_counter is Bob's jump_counter + 1.
}
```

A `while` loop executes the list of statements enclosed in braces until its **condition** (sometimes called its **predicate**) is no longer true. In this condition, we test if Bob's jump_counter (one of his properties) is greater than 0. In the loop, we decrement Bob's jump_counter so that it will eventually equal zero, at which point the condition will no longer be true.

Conditions can also be significantly more complicated than this one. You can use boolean logic, arithmetic, and more than one property from more than one object. For details and example on these more complicated conditionals and their rules, you can consult the Reference Manual.

2.5. Event Listeners and Locally Scoped Objects

Objects can also interact with each other through event listeners. An event listener, like a `while` loop, has a condition and a block of statements. At the end of each frame, the condition will be checked, and, if the condition is true, the statements will be executed. For instance, let's make Bob and Alice play a round of Marco Polo.

```
Whenever Bob says "Marco" {
      Stop Alice.
      Make Alice say "Polo".
      Repeat 10 times {
            Make Alice move anywhere.
      }
}

Whenever Alice says "Polo" {
      Stop Bob.
      Bob's xMove is Alice's column - Bob's column.
      Bob's yMove is Alice's row - Bob's row.
      Make Bob move right Bob's xMove.
      Make Bob move up Bob's yMove.
      Make Bob say "Marco".
}
```

We have created two event listeners. The first event listener causes Alice to respond and say "Polo" whenever Bob calls says "Marco", then move a few cells in the hopes that Bob won't catch her. The second event listener causes Bob to pursue Alice at her last known position and then say "Marco" to continue his search.

This version of the game will unfortunately last forever, since there is no condition to stop them from playing when Bob catches Alice!. A more complicated version of the game might involve Alice and Bob switching off hiding and seeking. Try adding the following properties and event listeners:

```
Alice's role is "seeker".
Bob's role is "hider".

Whenever Alice says "Marco" {
      Stop Bob.
      Make Bob say "Polo".
      Repeat 10 times {
            Make Bob move anywhere.
      }
```

```
}

Whenever Bob says "Polo" {
      Stop Alice.
      Make Alice move left Bob's column - Alice's column.
      Make Alice move up Bob's row - Alice's row.
      Make Alice say "Marco".
}

Whenever Alice's row = Bob's row and Alice's column = Bob's column {
      Stop Alice.
      Stop Bob.
      If Alice's role = "seeker" then {
            Alice's role is "hider".
            Repeat 10 times {
                  Make Alice move anywhere.
            }
            Bob's role is "seeker".
            Make Bob wait 5.
            Make Bob say "Marco".
      }
      If Bob's role = "seeker" then {
            Bob's role = "hider".
            Repeat 10 times {
                  Make Bob move anywhere.
            }
            Alice's role is "seeker".
            Make Alice wait 5.
            Make Alice say "Marco".
      }
}
```

Now, since Alice and Bob each hold their roles as hider and seeker, and each know how to perform both roles, they can switch off. Whenever they are on the same cell, they will switch roles. The new hider can move anywhere for 5 frames while the seeker waits 5 frames. After the seeker waits 5 frames, the person will start calling “Marco” and the game will once again be on.

The conditions for an event listener can use the same logic that we used in the `while` loop and can also look at actions that happened in the past turn. Event Listener conditions can also look for conditions that affect any member of a class. For instance, let’s assume that while Bob and Alice are playing Marco Polo, they also eat any food they stumble across. We

create a food class, and then define what happens when a person intersects food. We also use the `touches` keyword, which tests if two objects are in the same position.

```
A food is a thing.

Food_maker is a thing at -50, -50.

Repeat 100 times {
	Pizza is a food at Food_maker's column, Food_maker's row.
	Food_maker's row is Food_maker's row + 1.
	Food_maker's column is Food_maker's column + 1.
}

Remove Food_maker.

Whenever a person [p] touches a food [f] {
	Make p say "Yum!" now.
	Remove f.
}
```

First we defined a food class, and then we placed 100 slices of pizza on a linear path through the board. In the repeat loop, we used a **locally scoped** object for pizza. A “Pizza” object is created within the loop at a location defined by the item generator. While you are in the repeat loop, you can modify the Pizza object as much as you want, but after we leave the loop, we have no way of reaching the Pizza variable any more. If you try writing “`Remove Pizza.`” you will receive an error as Pizza is not defined outside of the loop. The only way to interact with the Pizza is now through a class event listener, such as the one presented.

The event listener will be called whenever a person and food are in the same location. A person will “eat” the food by saying yum, and the food will go missing. The bracket notation defined a local name for the person and food: `p` and `f` can therefore be used to modify the person and food that are on the same space. If multiple people and food end up touching in the same frame, the event listener will be called on all qualifying pairs.

3. Where to Go from Here

Congratulations! You’ve completed the *Words* tutorial. With your new knowledge, you should be able to start creating your own interesting worlds, and you can have fun making objects interact. However, *Words* contains many more features that were not covered in this tutorial. You can find this and more in the Reference Manual.

Eventually, you may start to feel limited by the *Words* environment. That is okay. The environment is not designed to be your last programming language — merely your first. When the time comes that you cannot express your ideas in *Words*, you may want to look for

a more advanced programming language, using the experience you gained in *Words* as a stepping stone.