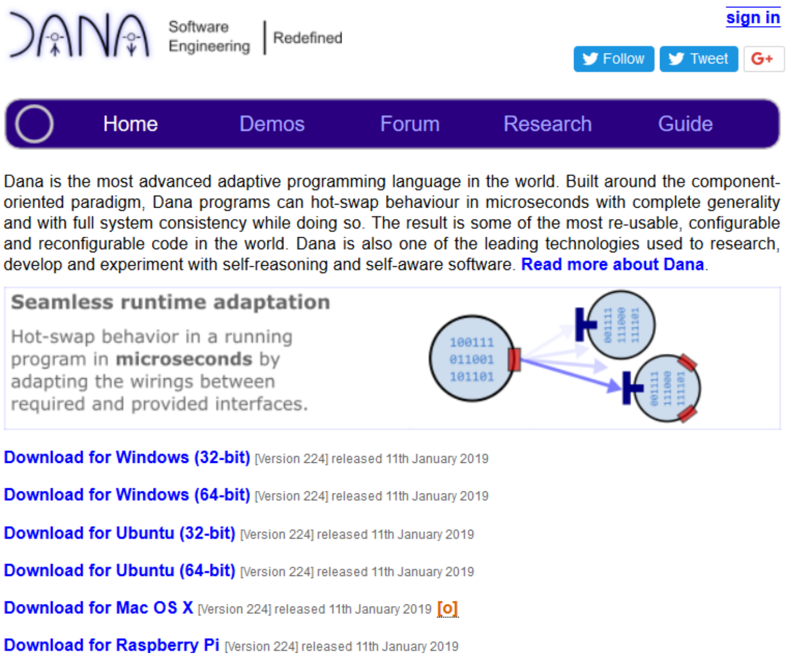
# Practical Session 1 – Fundamentals of Dana Programming

## Download and Install

Start by downloading the latest version of Dana from http://www.projectdana.com

You should select the "Ubuntu 64-bit" version (or whichever one matches your operating system):



Dana is the most advanced adaptive programming language in the world. Built around the component-oriented paradigm, Dana programs can hot-swap behaviour in microseconds with complete generality and with full system consistency while doing so. The result is some of the most re-usable, configurable and reconfigurable code in the world. Dana is also one of the leading technologies used to research, develop and experiment with self-reasoning and self-aware software. Read more about Dana.

Seamless runtime adaptation

Hot-swap behavior in a running program in **microseconds** by adapting the wirings between required and provided interfaces.

Download for Windows (32-bit) [Version 224] released 11th January 2019

Download for Windows (64-bit) [Version 224] released 11th January 2019

Download for Ubuntu (32-bit) [Version 224] released 11th January 2019

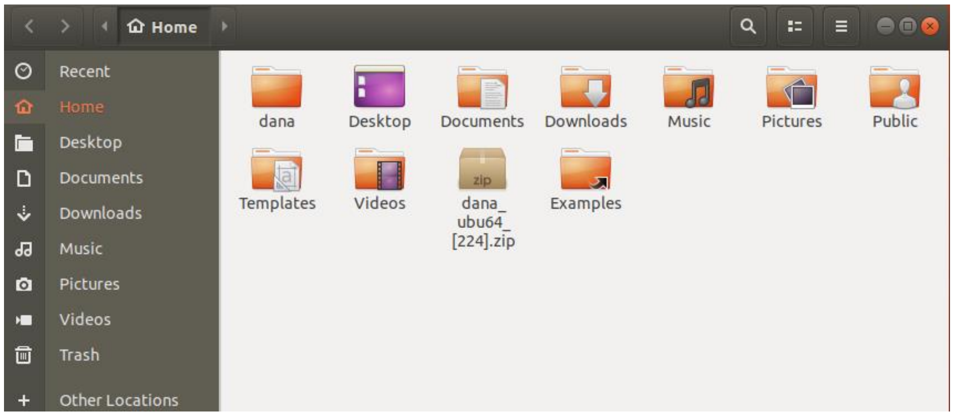
Download for Ubuntu (64-bit) [Version 224] released 11th January 2019

Download for Mac OS X [Version 224] released 11th January 2019 [o]

Download for Raspberry Pi [Version 224] released 11th January 2019

It's possible to install Dana for all users, but for this tutorial we install it locally for one user only.

To do this, extract the .zip file into your home directory in Linux so that you have a folder called "dana":



Inside the dana folder should be the files "dana", "dnc", "License.txt" etc.

Next we need to do three pieces of setup: tell Linux that the Dana interpreter and compiler should be executable; set an environment variable DANA_HOME to point to the directory in which dana is located; and tell Linux where to find the Dana executable programs so they can be executed anywhere.

First, open a terminal window and change into the “dana” directory.

Then run the command:

```
chmod +x dana dnc
```

This tells Linux to allow these two files to be executable.

Next, we set an environment variable DANA_HOME for the local user and also tell Linux where to find the Dana compiler and runtime executables:

```
echo “export DANA_HOME=/home/your_username/dana/” >> ~/.bashrc
```

```
echo “PATH=$PATH:$DANA_HOME” >> ~/.bashrc
```

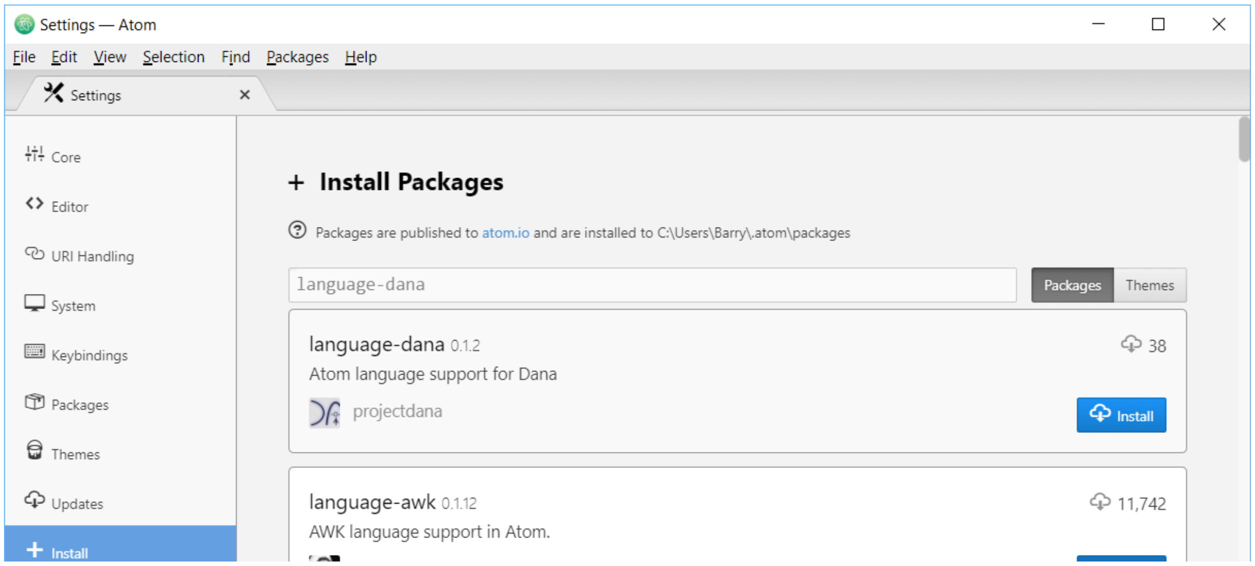
Note that you need to replace your_username with the local username in your Linux environment.

This completes your Dana installation. To allow the new environment variables to take effect you’ll need to log out of your session and log back in.

## Configuring Atom

For this course we assume that you’re using the Atom text editor. This should already be installed, but if it isn’t go ahead and install it as per the instructions on atom.io

Open Atom, then go to File → Settings → Install, search for language-dana then click Install:



+ Install Packages

Packages are published to atom.io and are installed to C:\Users\Barry\.atom\packages

language-dana

language-dana 0.1.2 — 38
Atom language support for Dana
projectdana

language-awk 0.1.12 — 11,742
AWK language support in Atom.

## Your first program

We start by creating a new directory for our first project. Make a new directory (anywhere you like) with any name. Now open Atom and create a new file. Save the file in your new directory as `Main.dn`.

Now we're going to write our first Dana program. We'll write a console application which reads some user input and writes it back out.

Type this program into your open file in Atom, then save your changes:

```
component provides App requires io.Output out, io.Input in {

       int App:main(AppParam params[])
               {
               out.print("Type something: ")

               char txt[] = in.readln()

               out.println("You typed '$txt'")

               return 0
               }
}
```

To compile the program, open a terminal window in your new directory and type the command:

```
dnc Main.dn
```

Assuming that there were no compile errors, this will generate a file Main.o

You can now run the program with the command:

```
dana Main.o
```

You should now be asked to enter some text and have it printed back to you on the terminal.

This program demonstrates the use of standard library APIs via `requires` directives, and also shows the way in which strings can be printed out within other strings using the $ notation. In Dana a string is just a character array and is an example of the principle of separating *pure data* from *behaviour*, such as an API to tokenise the string into separate fragments (see the `data.StringUtil` API for this capability).

## Your first multi-component program

Next we'll add more components to our system. This will allow you to become familiar with the standard directory layout used by Dana for all programs.

Add a new sub-directory to your project called `resources` and add a further sub-directory called `text`.

Create a new file in Atom and save it into the text directory with the name `Analysis.dn`.

Now add the following code to this file in Atom, and save the changes:

```
interface Analysis {

	int countWords(char text[])

	}
```

In Dana, all interfaces and other public types are stored within a `resources` directory. Any sub-directories within `resources` indicate package paths to the types. We make this separation because interfaces are strongly separated from specific *implementations* of those interfaces.

Next we'll write an implementation of this interface. Back in your project's root directory (where Main.dn is), create a new folder named `text`. Create a new file in Atom and save it as `Analysis.dn` within your new `text` directory (note that this is a different folder to the above one in `resources`).

Add the following code to this file in Atom, and save the changes:

```
component provides Analysis {

	int Analysis:countWords(char text[])
		{
		int count = 1
		for (int i = 0; i < text.arrayLength; i++) {
			if (text[i] == "." || text[i] == " ")
				count ++
			}
		return count
		}

	}
```

Finally we're going to modify our main program in `Main.dn` to the following:

```
component provides App requires io.Output out, io.Input in, data.IntUtil iu, text.Analysis analysis {

	int App:main(AppParam params[])
		{
		out.print("Type something: ")

		char txt[] = in.readln()

		out.println("You typed '$txt', which is $(iu.intToString(analysis.countWords(txt))) words")

		return 0
		}
}
```

Now we compile the entire project, using the command:

```
dnc .
```

This automatically compiles every component in the project for us. We can then run the new version of the program using the same command as before:

```
dana Main.o
```

This time, when we enter some text with spaces or full stops, we will see some data on how many words are in the string.

## Variations and manifest files

In a multi-component system it's possible write multiple different component variations which each implement the same interface but do so in a different way. Their implementations should all be semantically compatible, but may use different algorithms or resource levels.

As an example of this we'll create a new variant of the Analysis interface.

Create a new file in Atom and save it as `QuickAnalysis.dn` in your `text` directory.

In this file add the following code and save it:

```
component provides Analysis requires data.StringUtil stringUtil {

        int Analysis:countWords(char text[])
                {
                return stringUtil.explode(text, ". ").arrayLength
                }

        }
```

This component has exactly the same behaviour, but relies on another (existing) API to do the work. Compile everything again using the command:

```
dnc .
```

By default, when trying to resolve required interfaces against implementations, Dana will always link against a compiled component with the same name as the interface type. Advanced meta-composers (covered later in the course) will usually select dynamically between available implementations based on real-time learning, but it's also useful in some cases to change the default linking rules.

We can override Dana's default name equivalence with a `.manifest` file. We place this file in the directory of components it will affect, and then enter a simple set of rules expressed in JSON to map interfaces to default components. Following our above example, we can change the default implementation of `text.Analysis` by creating a `.manifest` file in the component `text` folder.

Create a new file in Atom and save it as `.manifest` in your `text` folder (the same folder that contains `QuickAnalysis.dn`). Add the following JSON to this file and save it:

```
{
"defaultLinks" :        [
                                {"interface" : "Analysis", "component", "QuickAnalysis"}
                        ]
}
```

The next time you run the program, it will use the *QuickAnalysis* component implementation for this interface rather than the original *Analysis* implementation. You can check that this really happens by adding some *println* statements to both implementation components to see which one gets used.

## Using APIs in the standard library

In completing the above exercises you have already used four different interfaces in Dana's standard library. This is simply a set of open-source interfaces and components in Dana's installation folder, which you can browse in your own time. The standard library is just a project like any other, with its own resources folder containing all interface type definitions, and other sub-folders of the root directory which contain corresponding implementation components of those interfaces.

The standard library is documented at http://www.projectdana.com/dana/api/ which is generated entirely from annotations in the source code within the central resources directory. Becoming familiar with browsing this API documentation will help with more advanced topics in the rest of the course.