



Clojure

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com



<https://www.facebook.com/tutorialspointindia>



<https://twitter.com/tutorialspoint>

About the Tutorial

Clojure is a high level, dynamic functional programming language. It is designed, based on the LISP programming language, and has compilers that makes it possible to be run on both Java and .Net runtime environment.

This tutorial is fairly comprehensive and covers various functions involved in Clojure. All the functions are explained using examples for easy understanding.

Audience

This tutorial is designed for all those software professionals who are keen on learning the basics of Clojure and how to put it into practice.

Prerequisites

Before proceeding with this tutorial, familiarity with Java and LISP programming language is preferred.

Copyright & Disclaimer

© Copyright 2016 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com

Table of Contents

About the Tutorial.....	i
Audience	i
Prerequisites	i
Copyright & Disclaimer.....	i
Table of Contents	ii
1. CLOJURE - OVERVIEW	1
2. CLOJURE – ENVIRONMENT	2
Leiningen Installation	2
Eclipse Installation	7
3. CLOJURE - BASIC SYNTAX.....	13
Hello World as a Complete Program	13
General Form of a Statement	13
Namespaces	14
Require Statement in Clojure	15
Comments in Clojure.....	16
Delimiters	16
Whitespaces.....	17
Symbols.....	17
Clojure Project Structure	17
4. CLOJURE - REPL	20
Starting a REPL Session.....	20
Special Variables in REPL.....	22

5.	CLOJURE - DATA TYPES	24
	Built-in Data Types	24
	Bound Values	25
	Class Numeric Types	25
6.	CLOJURE - VARIABLES	27
	Variable Declarations	27
	Naming Variables	28
	Printing variables	28
7.	CLOJURE - OPERATORS	30
	Arithmetic Operators	30
	Relational Operators	32
	Logical Operators	34
	Bitwise Operators	35
	Operator Precedence	36
8.	CLOJURE - LOOPS	37
	While Statement	37
	Doseq Statement	38
	Dotimes Statement	40
	Loop Statement	41
9.	CLOJURE - DECISION MAKING	44
	If Statement	44
	If/do Expression	45
	Nested If Statement	46
	Case Statement	47
	Cond Statement	48

10. CLOJURE - FUNCTIONS	50
Defining a Function	50
Anonymous Functions	51
Functions with Multiple Arguments	51
Variadic Functions	51
Higher Order Functions	52
11. CLOJURE - NUMBERS.....	53
Number Tests.....	54
12. CLOJURE - RECURSION	60
13. CLOJURE - FILE I/O.....	61
Reading the Contents of a File as an Entire String	61
Reading the Contents of a File One Line at a Time.....	62
Writing 'to' Files	62
Writing 'to' Files One Line at a Time	63
Checking to See If a File Exists	63
Reading from the Console	64
14. CLOJURE - STRINGS.....	65
Basic String Operations	65
15. CLOJURE - LISTS.....	76
list*	76
first	77
nth	78
cons.....	78
conj	79
rest.....	80

16. CLOJURE - SETS.....	81
sorted-set.....	81
get.....	82
contains?.....	82
conj.....	83
disj.....	84
union.....	84
difference.....	85
intersection.....	85
subset?.....	86
superset?.....	87
17. CLOJURE - VECTORS	88
vector-of	88
nth	89
get.....	90
conj.....	90
pop.....	91
subvec.....	92
18. CLOJURE - MAPS.....	93
Creation - HashMaps.....	93
Creation - SortedMaps	93
get.....	94
contains?.....	95
find.....	95
keys.....	96
vals.....	97

dissoc	97
merge.....	98
merge-with	99
select-keys	99
rename-keys	100
map-invert	101
19. CLOJURE - NAMESPACES	102
ns	102
ns	103
alias.....	103
all-ns	104
find-ns.....	105
ns-name	105
ns-aliases	106
ns-map	107
un-alias	108
20. CLOJURE - EXCEPTION HANDLING	109
Error.....	110
Multiple Catch Blocks.....	112
Finally Block	113
21. CLOJURE - SEQUENCES	116
cons.....	116
conj	117
concat	117
distinct	118
reverse	119

first	119
last	120
last	121
sort	121
drop	122
take-last	123
take	123
split-at	124
22. CLOJURE - REGULAR EXPRESSIONS	126
re-pattern	127
re-find	127
replace	128
replace-first	129
23. CLOJURE - PREDICATES	130
every-pred	131
every?	131
some	132
not-any?	133
24. CLOJURE - DESTRUCTURING	134
the-rest	135
Destructuring Maps	135
25. CLOJURE - DATE & TIME	137
java.util.Date	137
java.text.SimpleDateFormat	137
getTime	138

26. CLOJURE - ATOMS	139
reset!	139
compare-and-set!.....	140
swap!	141
27. CLOJURE - METADATA.....	143
meta-with	143
meta.....	144
vary-meta.....	144
28. CLOJURE - STRUCTMAPS	146
defstruct	146
struct.....	146
struct-map	147
Accessing Individual Fields	148
Immutable Nature.....	149
Adding a New Key to the Structure	150
29. CLOJURE - AGENTS	151
agent.....	151
send	152
shutdown-agents	153
send-off.....	154
await-for	155
await.....	156
agent-error.....	157
30. CLOJURE –WATCHERS	159
add-watch	159

remove-watch	160
31. CLOJURE - MACROS	161
defmacro	161
macro-expand	162
Macro with Arguments	163
32. CLOJURE - REFERENCE VALUES.....	164
ref	164
ref-set	165
alter	165
dosync.....	166
commute.....	167
33. CLOJURE - DATABASES	169
Database Connection	169
Querying Data	170
Inserting Data.....	171
Deleting Data	172
Updating Data	173
Transactions.....	174
34. CLOJURE - JAVA INTERFACE	175
Calling Java Methods.....	175
Calling Java Methods with Parameters.....	175
Creating Java Objects	176
Import Command	177
Running Code Using the Java Command.....	177
Java Built-in Functions.....	178

35. CLOJURE - CONCURRENT PROGRAMMING	179
36. CLOJURE - APPLICATIONS	183
Desktop – See-saw	183
Desktop – Changing the Value of Text	184
Desktop – Displaying a Modal Dialog Box.....	185
Desktop – Displaying Buttons.....	186
Desktop – Displaying Labels	187
Desktop – Displaying Text Fields	188
Web Applications - Introduction	188
Web Applications – Adding More Routes to Your Web Application	190
37. CLOJURE - AUTOMATED TESTING.....	192
Testing for Client Applications.....	192
Testing for Web-based Applications.....	193
38. CLOJURE - LIBRARIES	196
data.xml.....	196
data.json	196
data.csv.....	197

1. CLOJURE - OVERVIEW

Clojure is a high level, dynamic functional programming language. Clojure is designed based on the LISP programming language and has compilers which makes it run on both Java and .Net runtime environment.

Before we talk about Clojure, let's just have a quick description of LISP programming language. LISPs have a tiny language core, almost no syntax, and a powerful macro facility. With these features, you can bend LISP to meet your design, instead of the other way around. LISP has been there for a long time dating back to 1958.

Common LISP reads in an expression, evaluates it, and then prints out the result. For example, if you want to compute the value of a simple mathematical expression of 4+6 then you type in

```
USER(1) (+ 4 6)
```

Clojure has the following high-level key objectives as a programming language.

- It is based on the LISP programming language which makes its code statements smaller than traditional programming languages.
- It is a functional programming language.
- It focuses on immutability which is basically the concept that you should not make any changes to objects which are created in place.
- It can manage the state of an application for the programmer.
- It supports concurrency.
- It embraces existing programming languages. For example, Clojure can make use of the entire Java ecosystem for management of the running of the code via the JVM.

The official website for Clojure is <http://clojure.org/>



2.CLOJURE – ENVIRONMENT

There are a variety of ways to work with Clojure as a programming language. We will look at two ways to work with Clojure programming.

- **Leiningen** - Leiningen is an essential tool to create, build, and automate Clojure projects.
- **Eclipse Plugin** – There is a plugin called CounterClockwise, which is available for Eclipse for carrying out Clojure development in the Eclipse IDE.

Leiningen Installation

Ensure the following System requirements are met before proceeding with the installation.

System Requirements

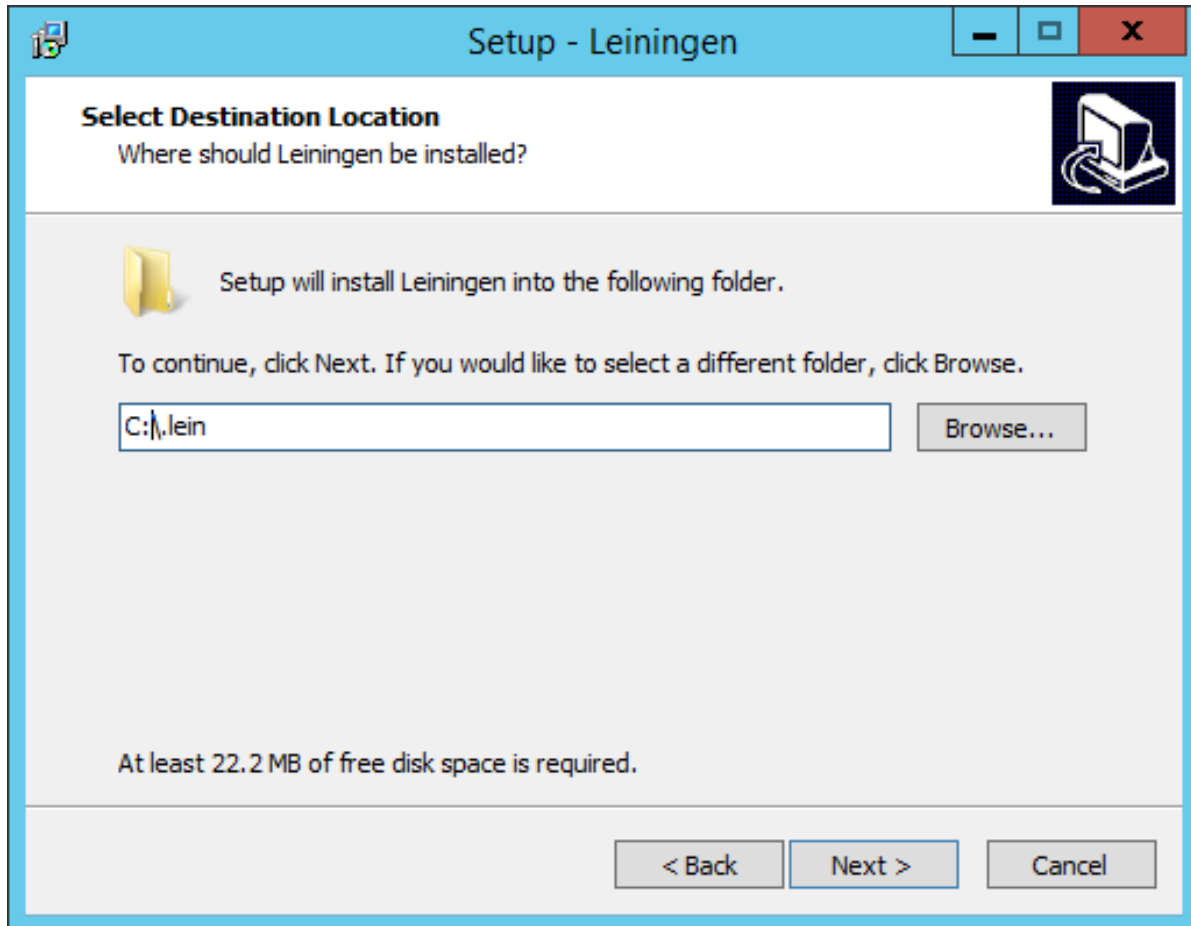
JDK	JDK 1.7 or above
Memory	2 GB RAM (recommended)

Step 1: Download the binary installation. Go to the link <http://leiningen-win-installer.djpowell.net/> to get the Windows Installer. Click on the option to start the download of the Groovy installer.

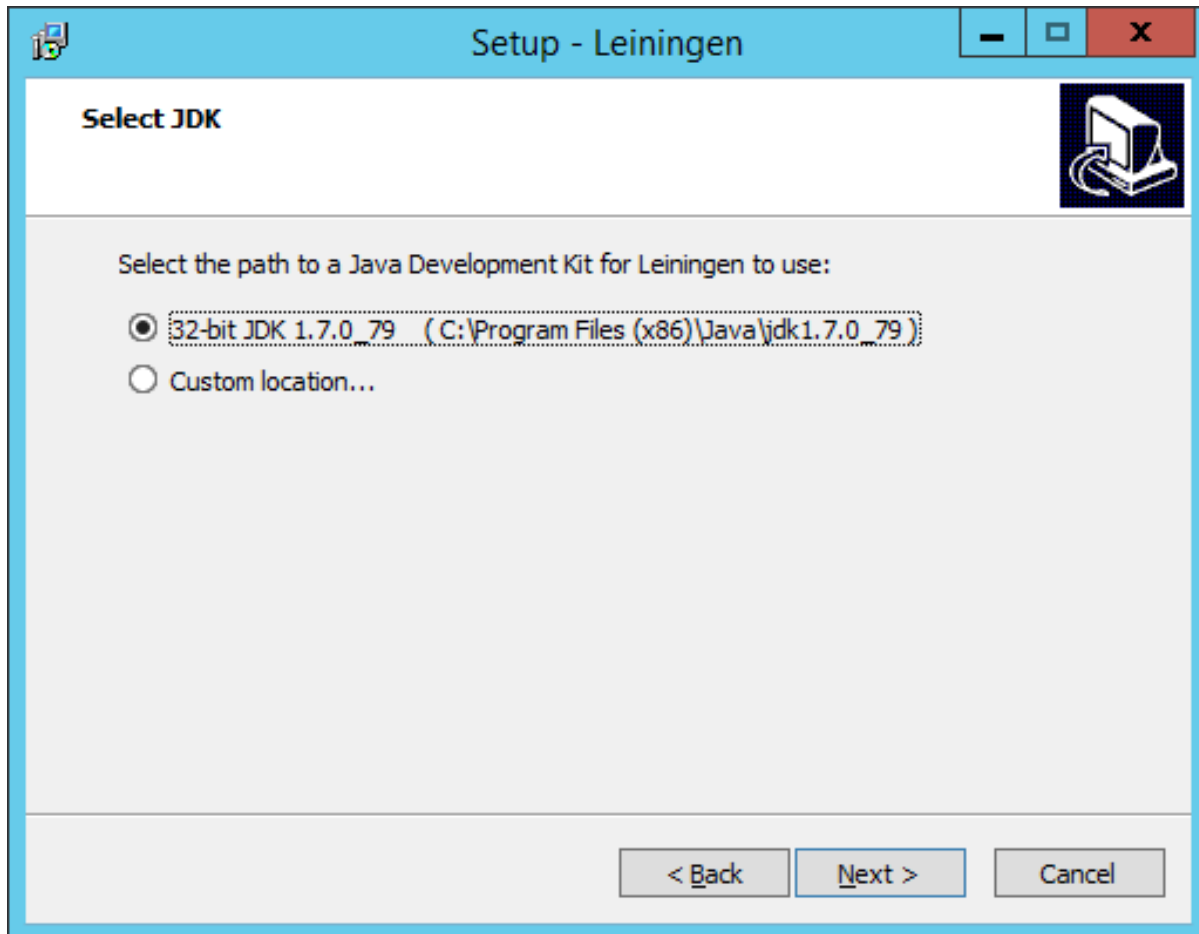
Step 2: Launch the Installer and click the Next button.



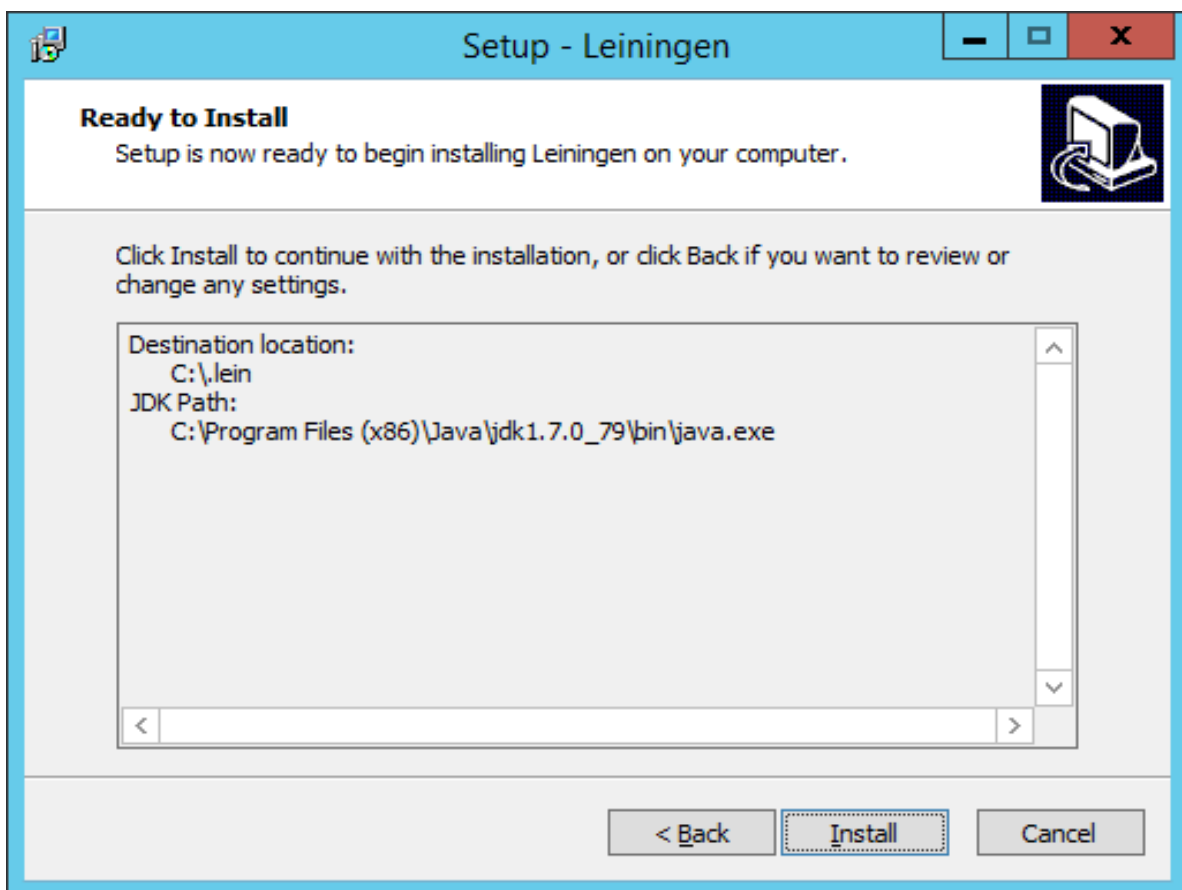
Step 3: Specify the location for the installation and click the Next button.



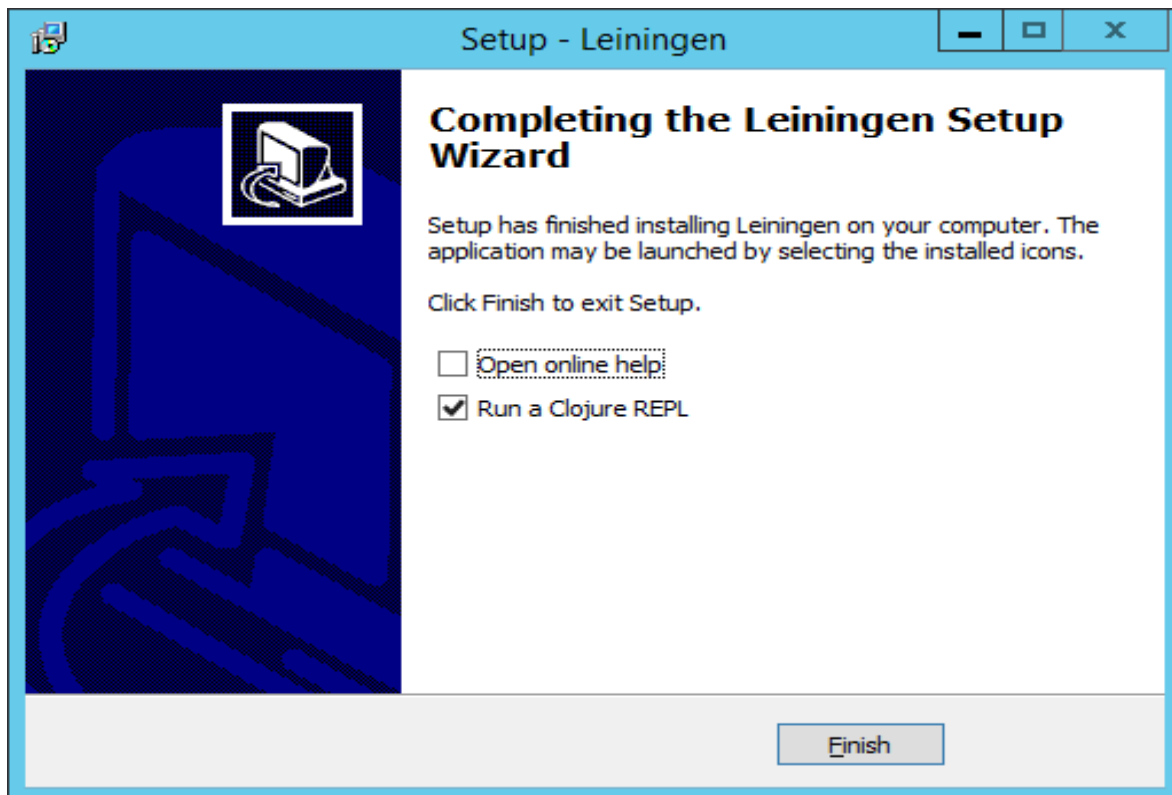
Step 4: The setup will detect the location of an existing Java installation. Click the Next button to proceed.



Step 5: Click the Install button to begin the installation.



After the installation is complete, it will give you the option to open a Clojure REPL, which is an environment that can be used to create and test your Clojure programs.



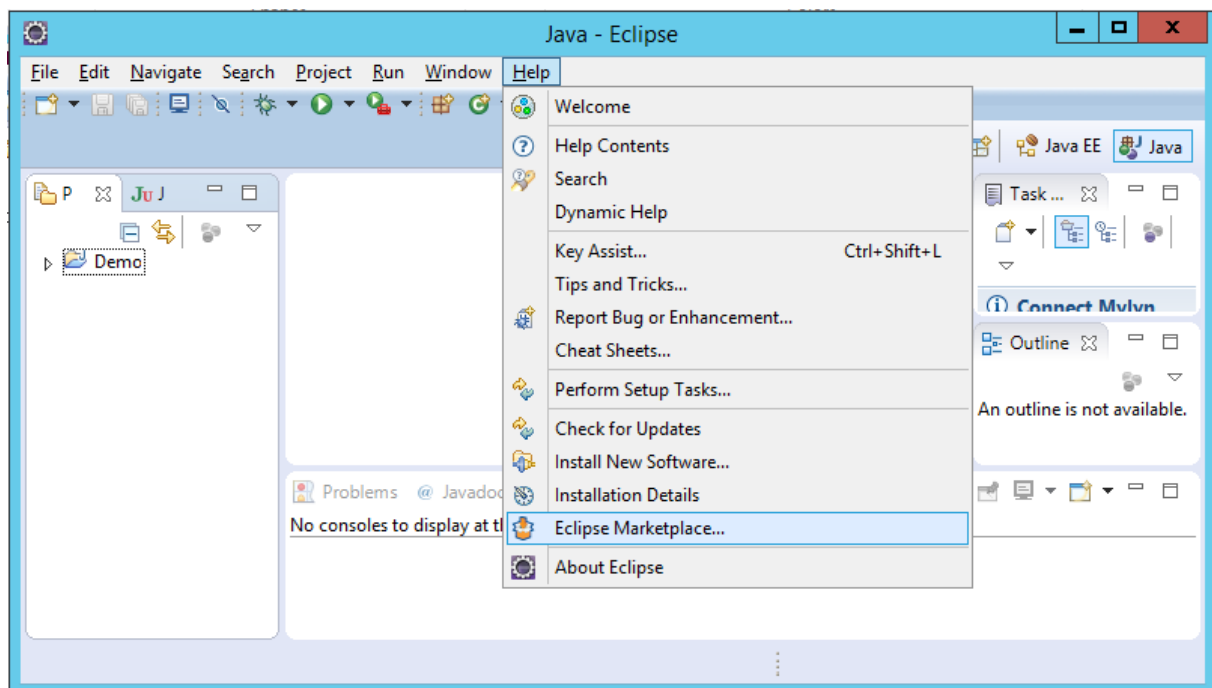
Eclipse Installation

Ensure the following System requirements are met before proceeding with the installation.

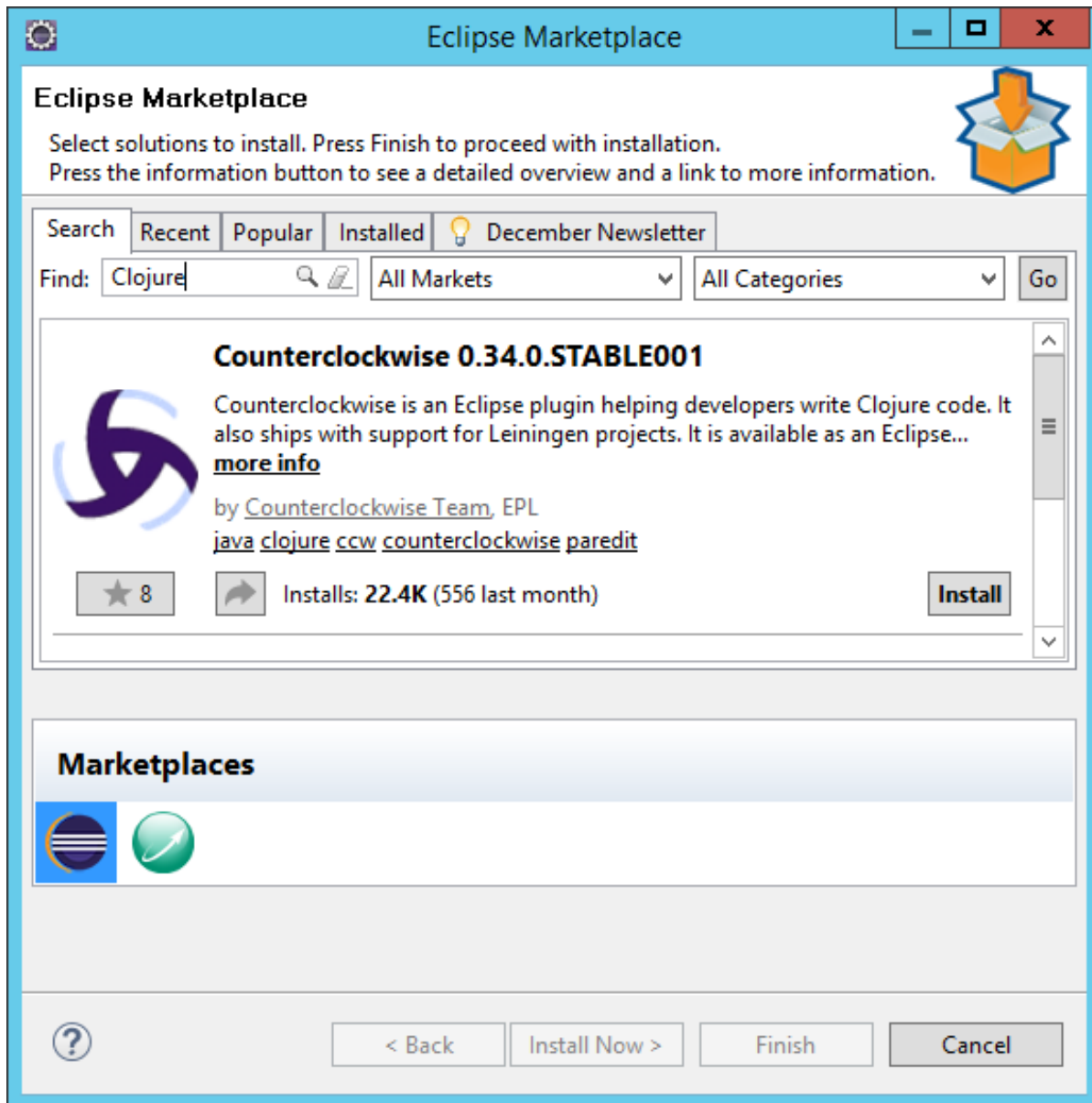
System Requirements

JDK	JDK 1.7 or above
Eclipse	Eclipse 4.5 (Mars)

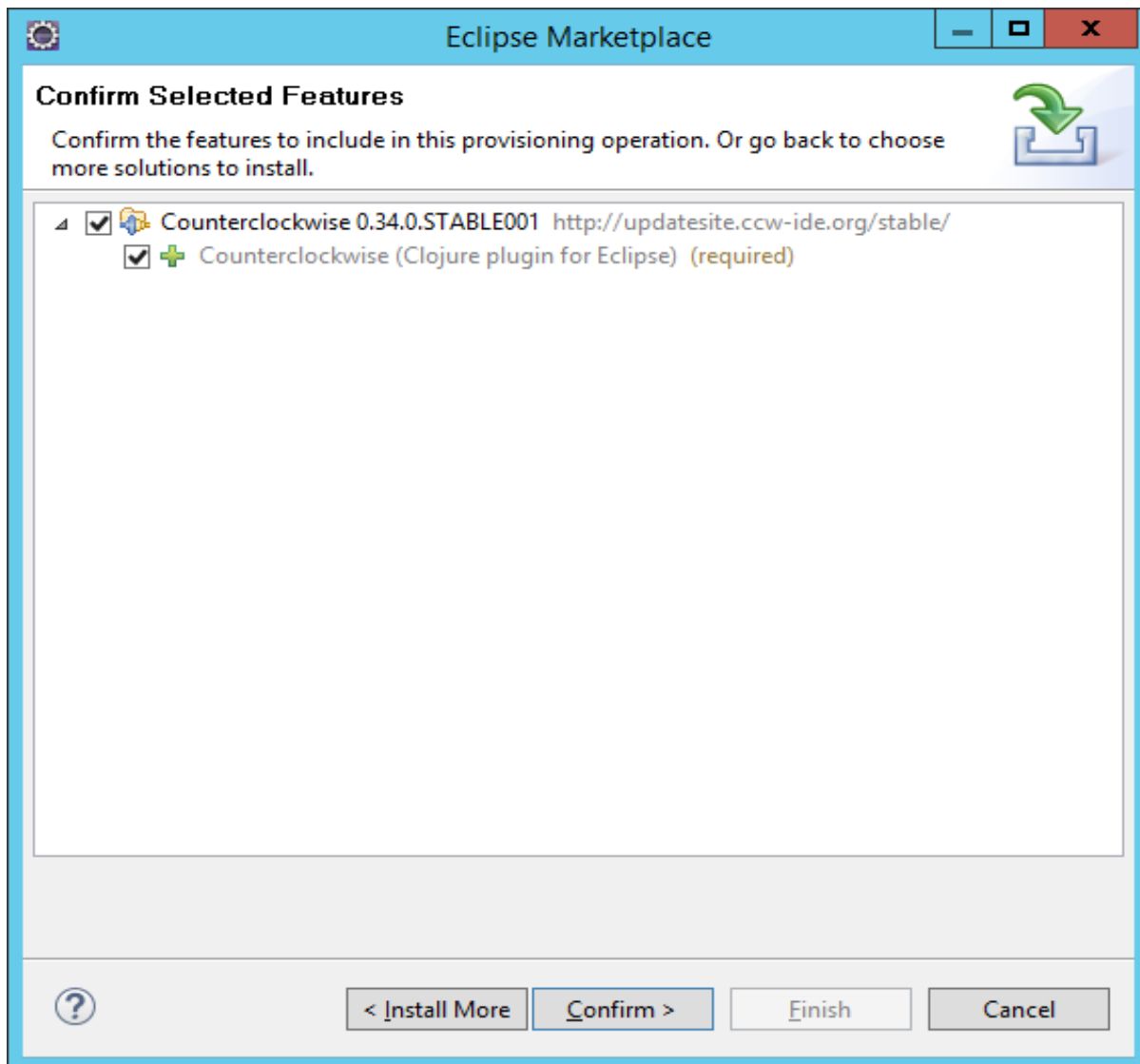
Step 1: Open Eclipse and click the Menu item. Click Help -> Eclipse Marketplace.



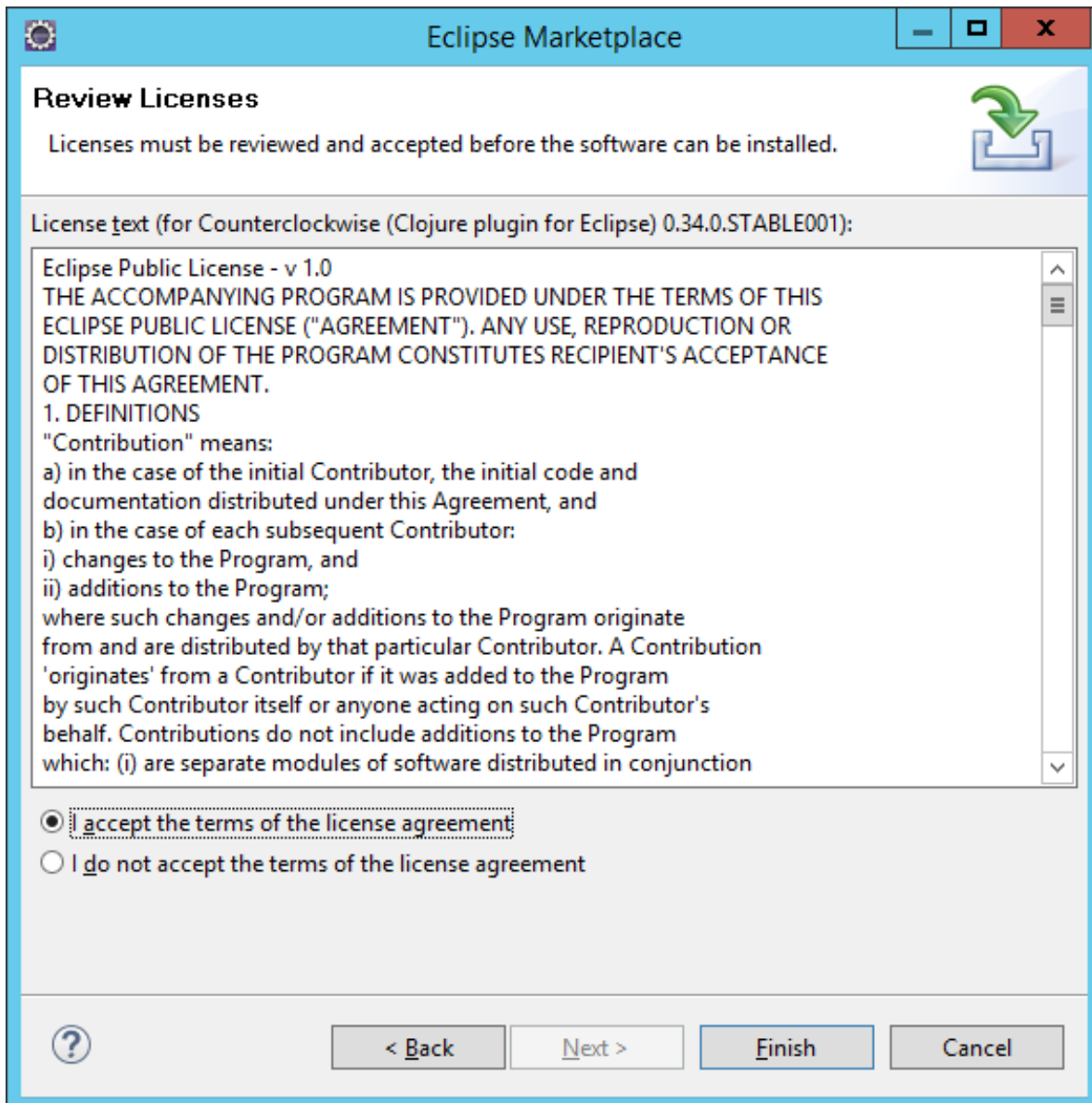
Step 2: Type in the keyword Clojure in the dialog box which appears and hit the 'Go' button. The option for counterclockwise will appear, click the Install button to begin the installation of this plugin.



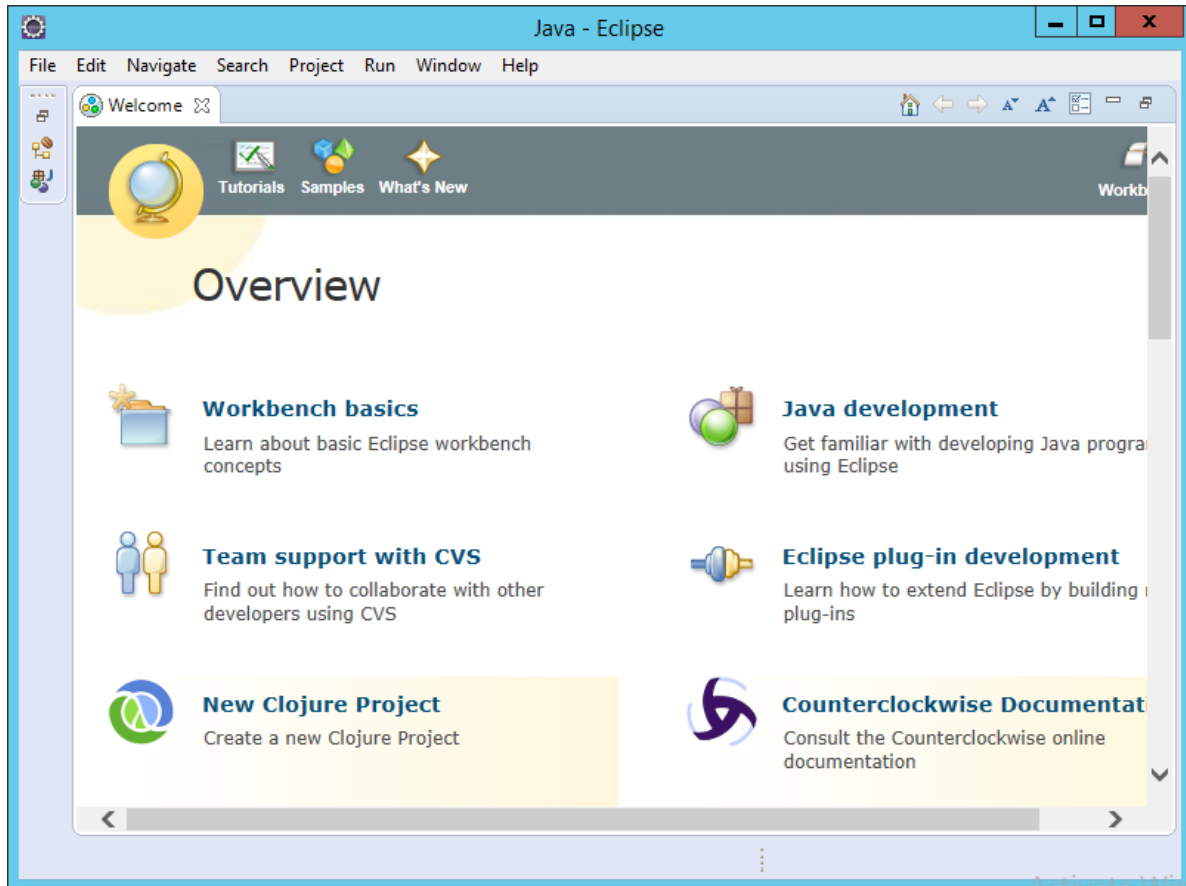
Step 3: In the next dialog box, click the Confirm button to begin the installation.



Step 4: In the next dialog box, you will be requested to accept the license agreement. Accept the license agreement and click the Finish button to continue with the installation.



The installation will begin, and once completed, it will prompt you to restart Eclipse.
Once Eclipse is restarted, you will see the option in Eclipse to create a new Clojure project.



3.CLOJURE - BASIC SYNTAX

In order to understand the basic syntax of Clojure, let's first look at a simple Hello World program.

Hello World as a Complete Program

Write 'Hello world' in a complete Clojure program. Following is an example.

```
(ns clojure.examples.hello
  (:gen-class))
(defn hello-world []
  (println "Hello World"))
(hello-world)
```

The following things need to be noted about the above program.

- The program will be written in a file called main.clj. The extension 'clj' is the extension name for a clojure code file. In the above example, the name of the file is called main.clj.
- The 'defn' keyword is used to define a function. We will see functions in details in another chapter. But for now, know that we are creating a function called hello-world, which will have our main Clojure code.
- In our Clojure code, we are using the 'println' statement to print "Hello World" to the console output.
- We then call the hello-world function which in turn runs the 'println' statement.

The above program produces the following output.

```
Hello World
```

General Form of a Statement

The general form of any statement needs to be evaluated in braces as shown in the following example.

```
(+ 1 2)
```

In the above example, the entire expression is enclosed in braces. The output of the above statement is 3. The + operator acts like a function in Clojure, which is used for the addition of numerals. The values of 1 and 2 are known as **parameters to the function**.

Let us consider another example. In this example, 'str' is the operator which is used to concatenate two strings. The strings "Hello" and "World" are used as parameters.

```
(str "Hello" "World")
```

If we combine the above two statements and write a program, it will look like the following.

```
(ns clojure.examples.hello
  (:gen-class))
(defn Example []
  (println (str "Hello World")))
  (println (+ 1 2)))
(Example)
```

The above code produces the following output.

```
Hello World
3
```

Namespaces

A namespace is used to define a logical boundary between modules defined in Clojure.

Current Namespace

This defines the current namespace in which the current Clojure code resides in.

Syntax

```
*ns*
```

Example

In the REPL command window run the following command.

```
*ns*
```

Output

When we run the above command, the output will defer depending on what is the current namespace. Following is an example of an output. The namespace of the Clojure code is:

```
clojure.examples.hello

(ns clojure.examples.hello
  (:gen-class))
(defn Example []
  (println (str "Hello World")))
  (println (+ 1 2)))
  (Example)
```

Require Statement in Clojure

Clojure code is packaged in libraries. Each Clojure library belongs to a namespace, which is analogous to a Java package. You can load a Clojure library with the 'Require' statement.

Syntax

```
(require quoted-namespace-symbol)
```

Following is an example of the usage of this statement.

```
(ns clojure.examples.hello
  (:gen-class))
(require 'clojure.java.io')
(defn Example []
  (.exists (file "Example.txt")))
```

(Example)

In the above code, we are using the 'require' keyword to import the namespace clojure.java.io which has all the functions required for input/output functionality. Since we not have the required library, we can use the 'file' function in the above code.

Comments in Clojure

Comments are used to document your code. Single line comments are identified by using the ;; at any position in the line. Following is an example.

```
(ns clojure.examples.hello
  (:gen-class))
;; This program displays Hello World
(defn Example []
  (println "Hello World"))
(Example)
```

Delimiters

In Clojure, statements can be split or delimited by using either the curved or square bracket braces.

Following are two examples.

```
(ns clojure.examples.hello
  (:gen-class))
;; This program displays Hello World
(defn Example []
  (println (+ 1 2
             3)))
(Example)
```

The above code produces the following output.

Following is another example.

```
(ns clojure.examples.hello    (:gen-class))
  ;; This program displays Hello World
(defn Example []
  (println [+ 1 2
              3]))
)
(Example)
```

The above code produces the following output.

```
[#object[clojure.core$_PLUS_ 0x10f163b "clojure.core$_PLUS_@10f163b"] 1 2 3]
```

Whitespaces

Whitespaces can be used in Clojure to split different components of a statement for better clarity. This can be done with the assistance of the comma (,) operator.

For example, the following two statements are equivalent and the output of both the statements will be 15.

```
(+ 1 2 3 4 5)
(+ 1, 2, 3, 4, 5)
```

Although Clojure ignores commas, it sometimes uses them to make things easier for the programmer to read.

For instance, if you have a hash map like the following (def a-map {:a 1 :b 2 :c 3}) and ask for its value in the REPL window, Clojure will print the output as {:a 1, :b 2, :c 3}.

The results are easier to read, especially if you're looking at a large amount of data.

Symbols

In Clojure, symbols are equivalent to identifiers in other programming languages. But unlike other programming languages, the compiler sees symbols as actual string values. As a symbol is a value, a symbol can be stored in a collection, passed as an argument to a function, etc., just like any other object.

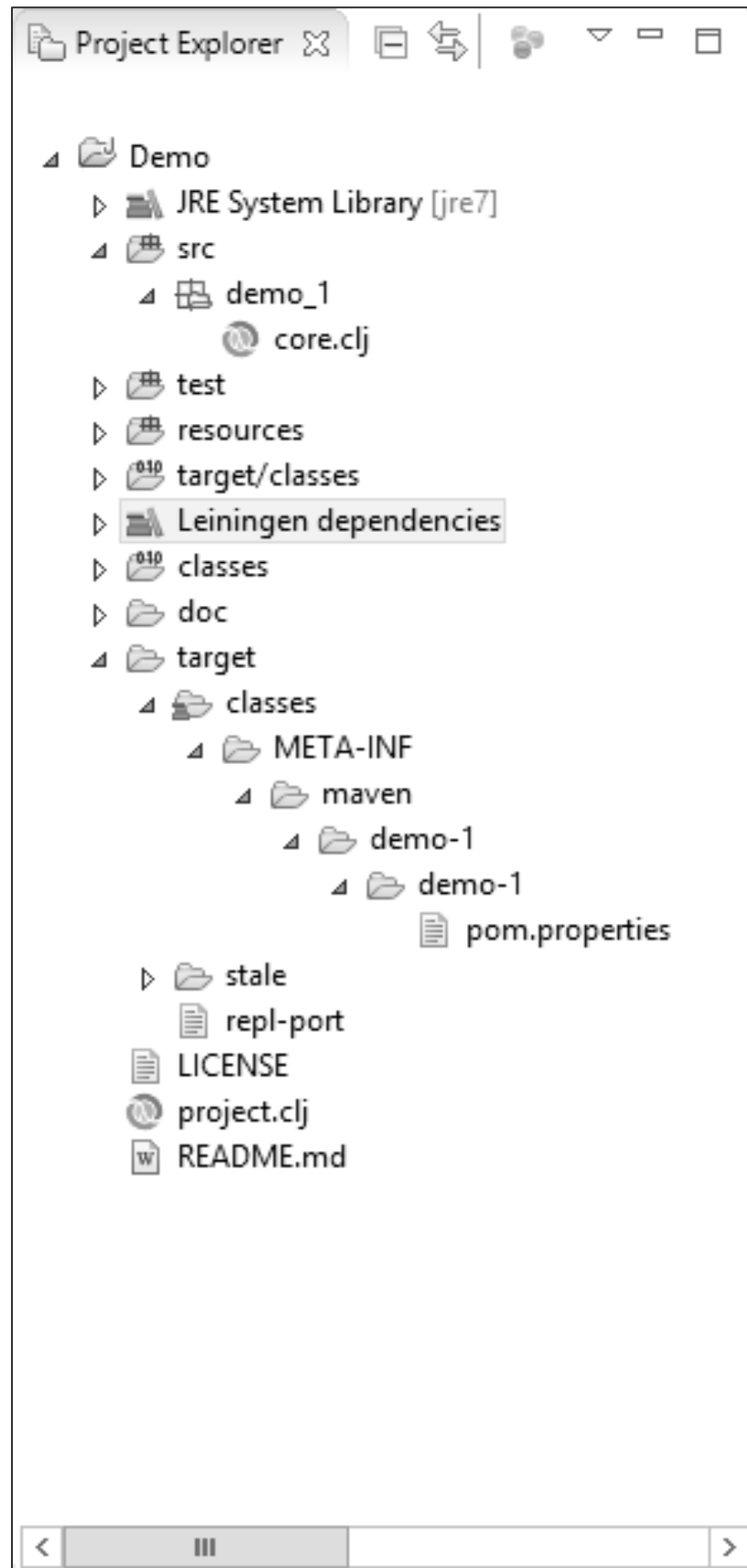
A symbol can only contain alphanumeric characters and `* + ! / . : - _ ?' but must not begin with a numeral or colon.

Following are valid examples of symbols.

```
tutorial-point!  
TUTORIAL  
+tutorial+
```

Clojure Project Structure

Finally let's talk about a typical project structure for a Clojure project. Since Clojure code runs on Java virtual machine, most of the project structure within Clojure is similar to what you would find in a java project. Following is the snapshot of a sample project structure in Eclipse for a Clojure project.



Following key things need to be noted about the above program structure.

- demo_1 – This is the package in which the Clojure code file is placed.
- core.clj – This is the main Clojure code file, which will contain the code for the Clojure application.
- The Leiningen folder contains files like clojure-1.6.0.jar which is required to run any Clojure-based application.
- The pom.properties file will contain information such as the groupId, artifactId and version of the Clojure project.
- The project.clj file contains information about the Clojure application itself. Following is a sample of the project file contents.

```
(defproject demo-1 "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.6.0"]])
```

4.CLOJURE - REPL

REPL (read-eval-print loop) is a tool for experimenting with Clojure code. It allows you to interact with a running program and quickly try out if things work out as they should. It does this by presenting you with a prompt where you can enter the code. It then reads your input, evaluates it, prints the result, and loops, presenting you with a prompt again.

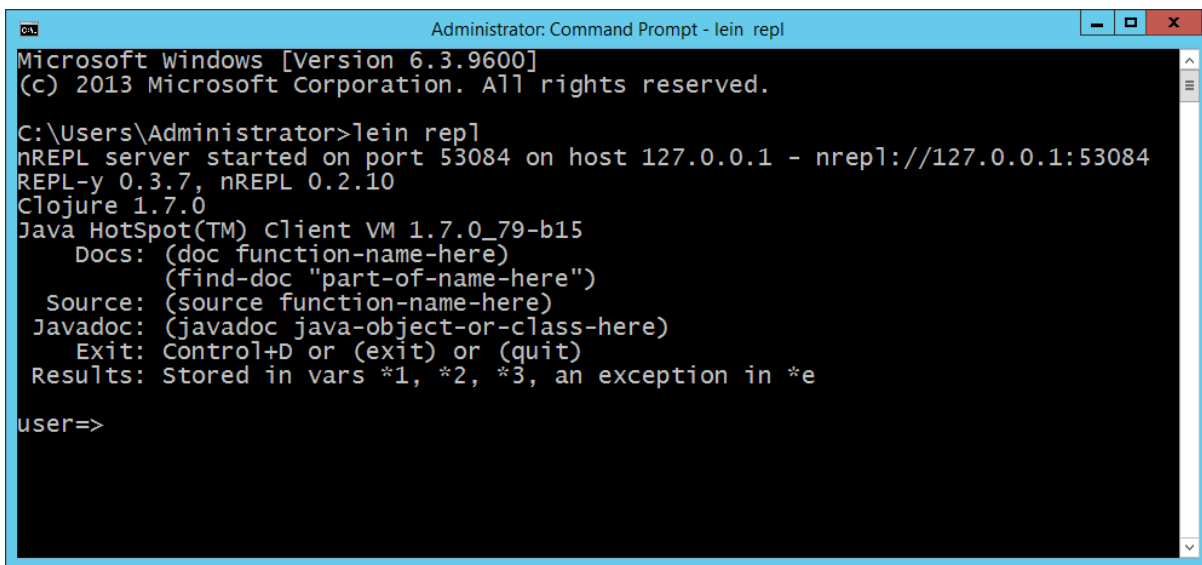
This process enables a quick feedback cycle that isn't possible in most other languages.

Starting a REPL Session

A REPL session can be started in Leiningen by typing the following command in the command line.

```
lein repl
```

This will start the following REPL window.

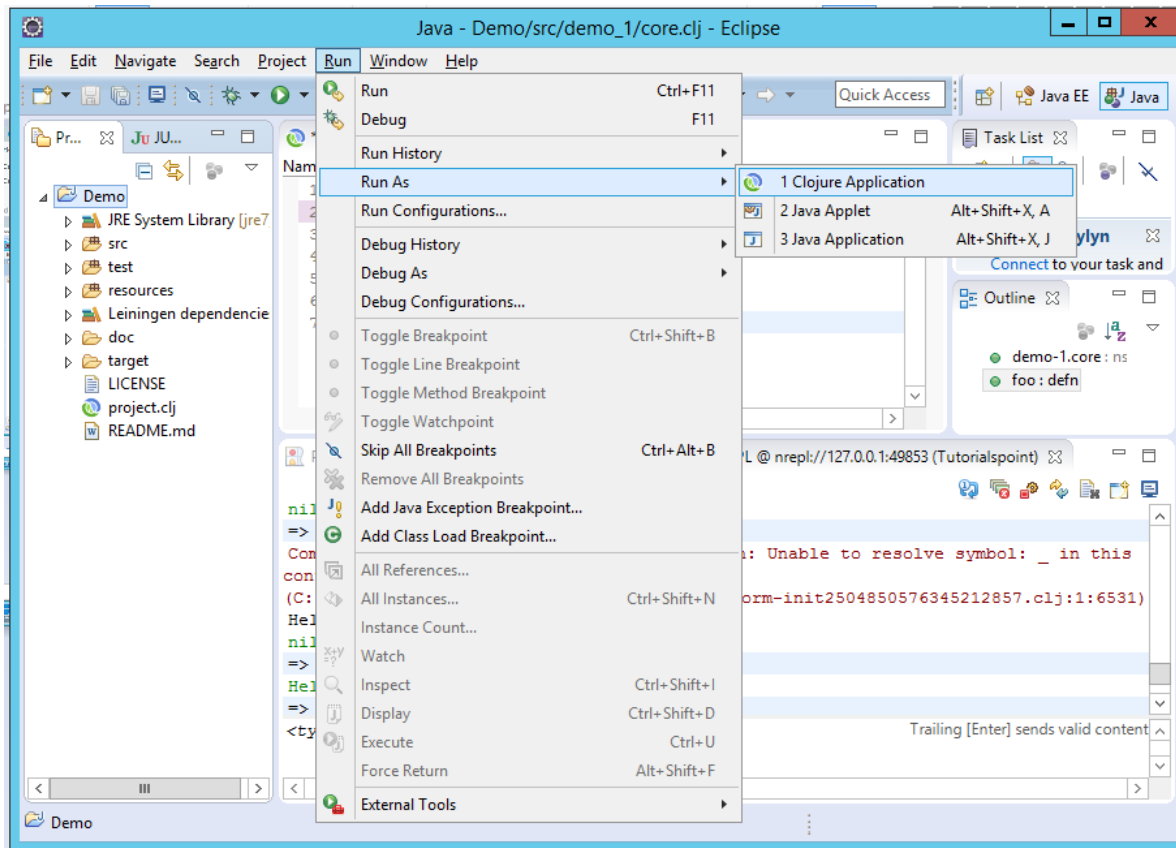


```
Administrator: Command Prompt - lein repl
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

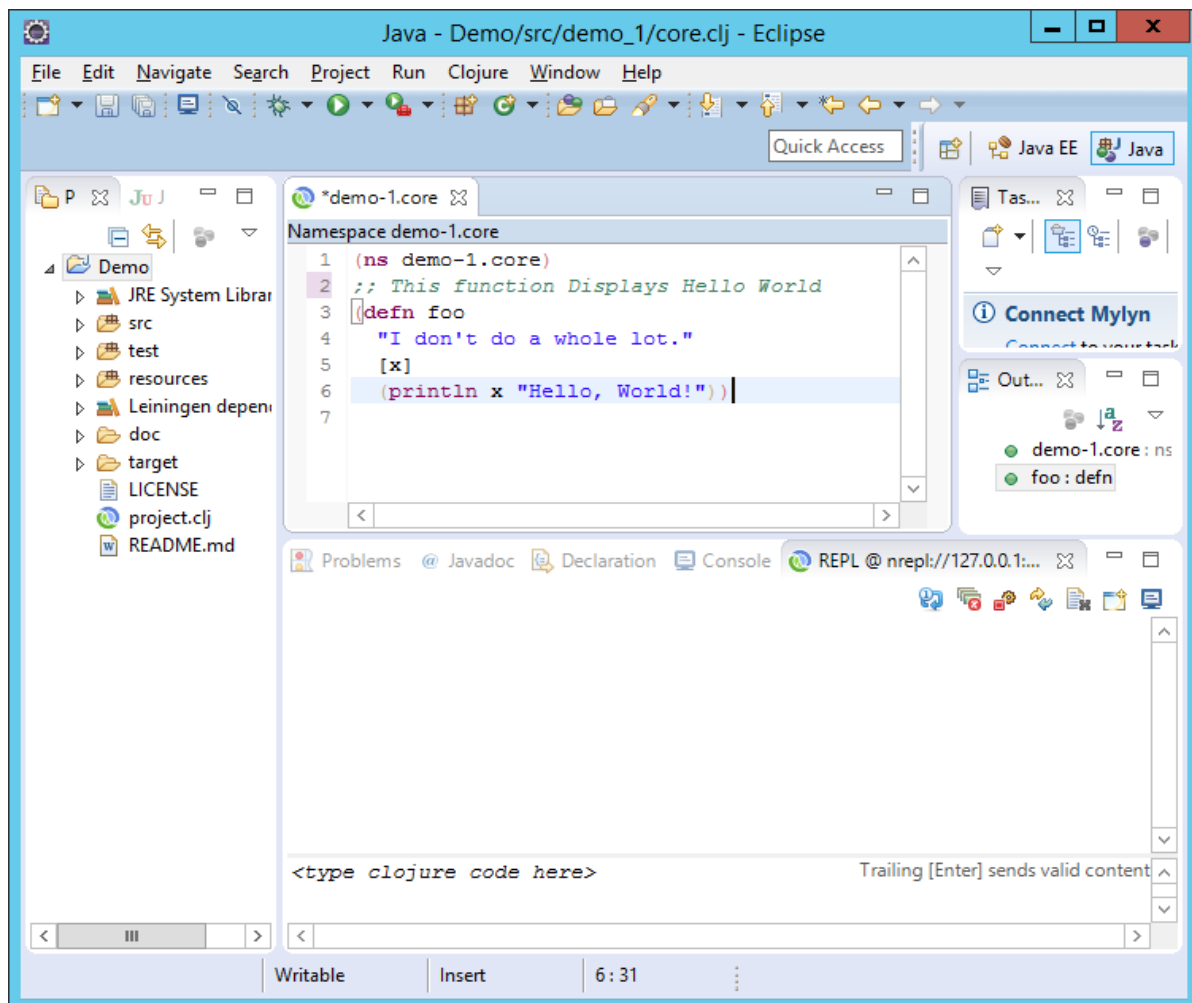
C:\Users\Administrator>lein repl
nREPL server started on port 53084 on host 127.0.0.1 - nrepl://127.0.0.1:53084
REPL-y 0.3.7, nREPL 0.2.10
Clojure 1.7.0
Java HotSpot(TM) Client VM 1.7.0_79-b15
  Docs: (doc function-name-here)
        (find-doc "part-of-name-here")
  Source: (source function-name-here)
  Javadoc: (javadoc java-object-or-class-here)
  Exit: Control+D or (exit) or (quit)
  Results: Stored in vars *1, *2, *3, an exception in *e
user=>
```

You then start evaluating Clojure commands in the REPL window as required.

To start a REPL session in Eclipse, click the Menu option, go to Run As -> Clojure Application.



This will start a new REPL session in a separate window along with the console output.



Conceptually, REPL is similar to Secure Shell (SSH). In the same way that you can use SSH to interact with a remote server, Clojure REPL allows you to interact with a running Clojure process. This feature can be very powerful because you can even attach a REPL to a live production app and modify your program as it runs.

Special Variables in REPL

REPL includes some useful variables, the one widely used is the special variable *1, *2, and *3. These are used to evaluate the results of the three most recent expressions.

Following example shows how these variables can be used.

```
user => "Hello"
Hello
user => "World"
World
user => (str *2 *1)
HelloWorld
```

In the above example, first two strings are being sent to the REPL output window as "Hello" and "World" respectively. Then the *2 and *1 variables are used to recall the last 2 evaluated expressions.

5.CLOJURE - DATA TYPES

Clojure offers a wide variety of **built-in data types**.

Built-in Data Types

Following is a list of data types which are defined in Clojure.

- **Integers** – Following are the representation of Integers available in Clojure.
 - Decimal Integers (Short, Long and Int) – These are used to represent whole numbers. For example, 1234.
 - Octal Numbers – These are used to represent numbers in octal representation. For example, 012.
 - Hexadecimal Numbers – These are used to represent numbers in hexadecimal representation. For example, 0xff.
 - Radix Numbers – These are used to represent numbers in radix representation. For example, 2r1111 where the radix is an integer between 2 and 36, inclusive.
- **Floating point.**
 - The default is used to represent 32-bit floating point numbers. For example, 12.34.
 - The other representation is the scientific notation. For example, 1.35e-12.
- **char** – This defines a single character literal. Characters are defined with the backlash symbol. For example, /e.
- **Boolean** – This represents a Boolean value, which can either be true or false.
- **String** – These are text literals which are represented in the form of chain of characters. For example, "Hello World".
- **Nil** – This is used to represent a NULL value in Clojure.
- **Atom** - Atoms provide a way to manage shared, synchronous, independent state. They are a reference type like refs and vars.

Bound Values

Since all of the datatypes in Clojure are inherited from Java, the bounded values are the same as in Java programming language. The following table shows the maximum allowed values for the numerical and decimal literals.

short	-32,768 to 32,767
int	-2,147,483,648 to 2,147,483,647
long	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
float	1.40129846432481707e-45 to 3.40282346638528860e+38
double	4.94065645841246544e-324d to 1.79769313486231570e+308d

Class Numeric Types

In addition to the primitive types, the following object types (sometimes referred to as wrapper types) are allowed.

Name
java.lang.Byte
java.lang.Short
java.lang.Integer
java.lang.Long
java.lang.Float
java.lang.Double

The following program shows a consolidated clojure code to demonstrate the data types in Clojure.

```
(ns clojure.examples.hello
  (:gen-class))
;; This program displays Hello World
(defn Example []
  ;; The below code declares a integer variable
  (def x 1)
  ;; The below code declares a float variable
  (def y 1.25))
```

```
;; The below code declares a string variable
(def str1 "Hello")
(println x)
(println y)
(println str1)
)
(Example)
```

The above program produces the following output.

```
1
1.25
Hello
```

6.CLOJURE - VARIABLES

In Clojure, **variables** are defined by the **'def'** keyword. It's a bit different wherein the concept of variables has more to do with binding. In Clojure, a value is bound to a variable. One key thing to note in Clojure is that variables are immutable, which means that in order for the value of the variable to change, it needs to be destroyed and recreated again.

Following are the basic types of variables in Clojure.

- **short** - This is used to represent a short number. For example, 10.
- **int** - This is used to represent whole numbers. For example, 1234.
- **long** - This is used to represent a long number. For example, 10000090.
- **float** - This is used to represent 32-bit floating point numbers. For example, 12.34.
- **char** - This defines a single character literal. For example, 'a'.
- **Boolean** - This represents a Boolean value, which can either be true or false.
- **String** - These are text literals which are represented in the form of chain of characters. For example, "Hello World".

Variable Declarations

Following is the general syntax of defining a variable.

```
(def var-name var-value)
```

Where 'var-name' is the name of the variable and 'var-value' is the value bound to the variable.

Following is an example of variable declaration.

```
(ns clojure.examples.hello
  (:gen-class))
;; This program displays Hello World
(defn Example []
  ;; The below code declares a integer variable
  (def x 1)
  ;; The below code declares a float variable
  (def y 1.25))
```

```
;; The below code declares a string variable
(def str1 "Hello")
;; The below code declares a boolean variable
(def status true)
)
(Example)
```

Naming Variables

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because Clojure, just like Java is a case-sensitive programming language.

Following are some examples of variable naming in Clojure.

```
(ns clojure.examples.hello
  (:gen-class))
;; This program displays Hello World
(defn Example []
  ;; The below code declares a Boolean variable with the name of status
  (def status true)
  ;; The below code declares a Boolean variable with the name of STATUS
  (def STATUS false)
  ;; The below code declares a variable with an underscore character.
  (def _num1 2)
  )
(Example)
```

Note: In the above statements, because of the case sensitivity, status and STATUS are two different variable defines in Clojure.

The above example shows how to define a variable with an underscore character.

End of ebook preview
If you liked what you saw...
Buy it from our store @ [**https://store.tutorialspoint.com**](https://store.tutorialspoint.com)