Clojure

tutorialspoint

SIMPLYEASYLEARNING

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

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.

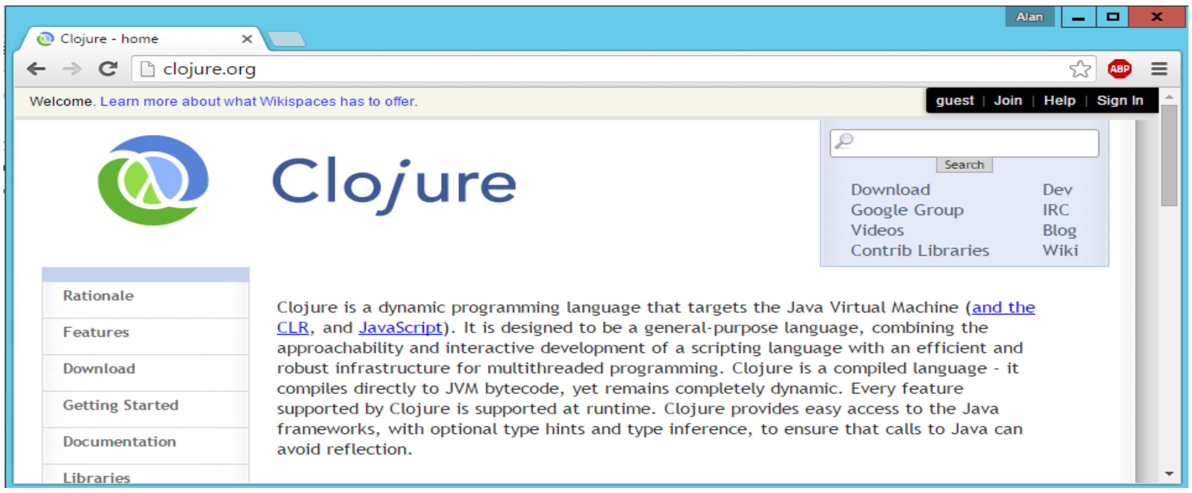
Common LISP reads in an expression, evaluates it, and then prints out the result. For example, if you want to compute the value of a simple mathematical expression of 4+6 then you type in

```
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/



Clojure is a dynamic programming language that targets the Java Virtual Machine (and the CLR, and JavaScript). It is designed to be a general-purpose language, combining the approachability and interactive development of a scripting language with an efficient and robust infrastructure for multithreaded programming. Clojure is a compiled language - it compiles directly to JVM bytecode, yet remains completely dynamic. Every feature supported by Clojure is supported at runtime. Clojure provides easy access to the Java frameworks, with optional type hints and type inference, to ensure that calls to Java can avoid reflection.

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

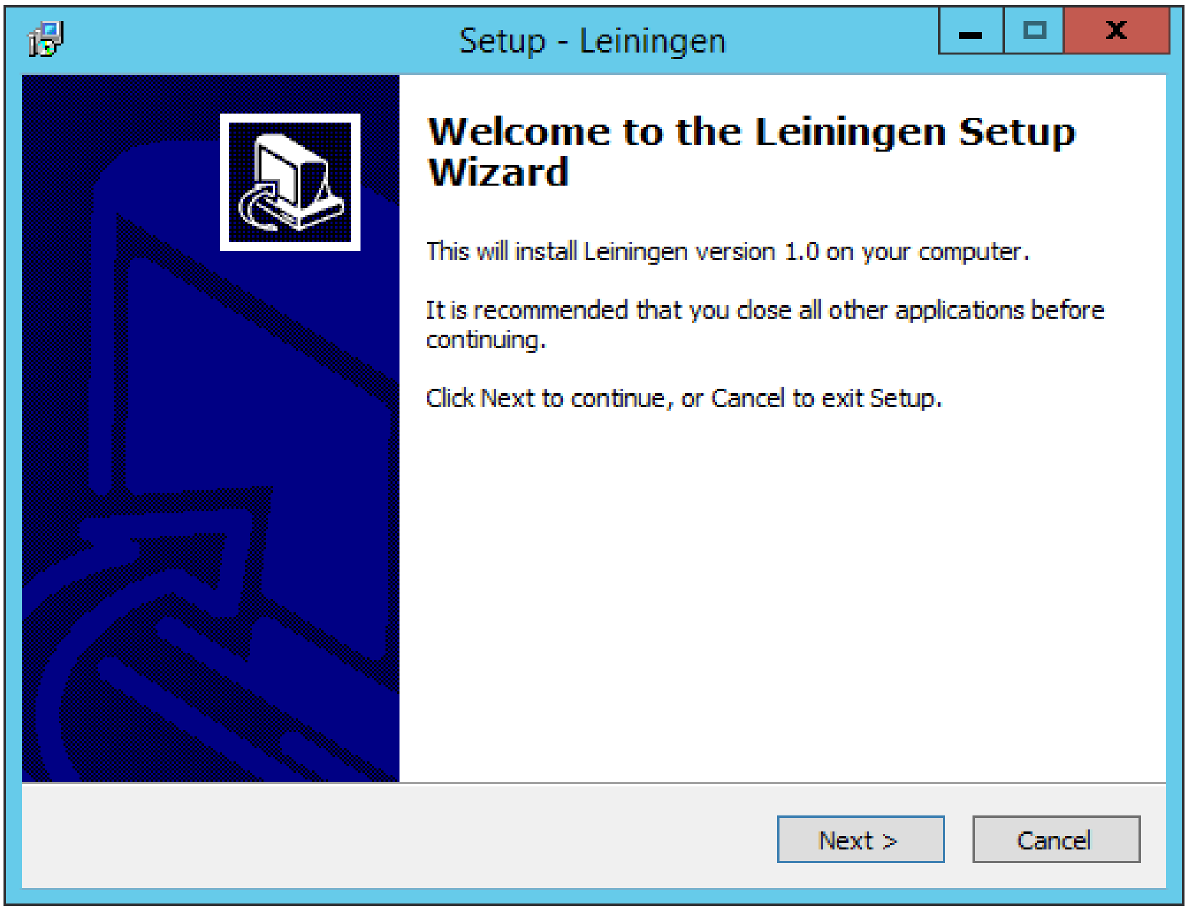
Ensure the following System requirements are met before proceeding with the installation.

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.



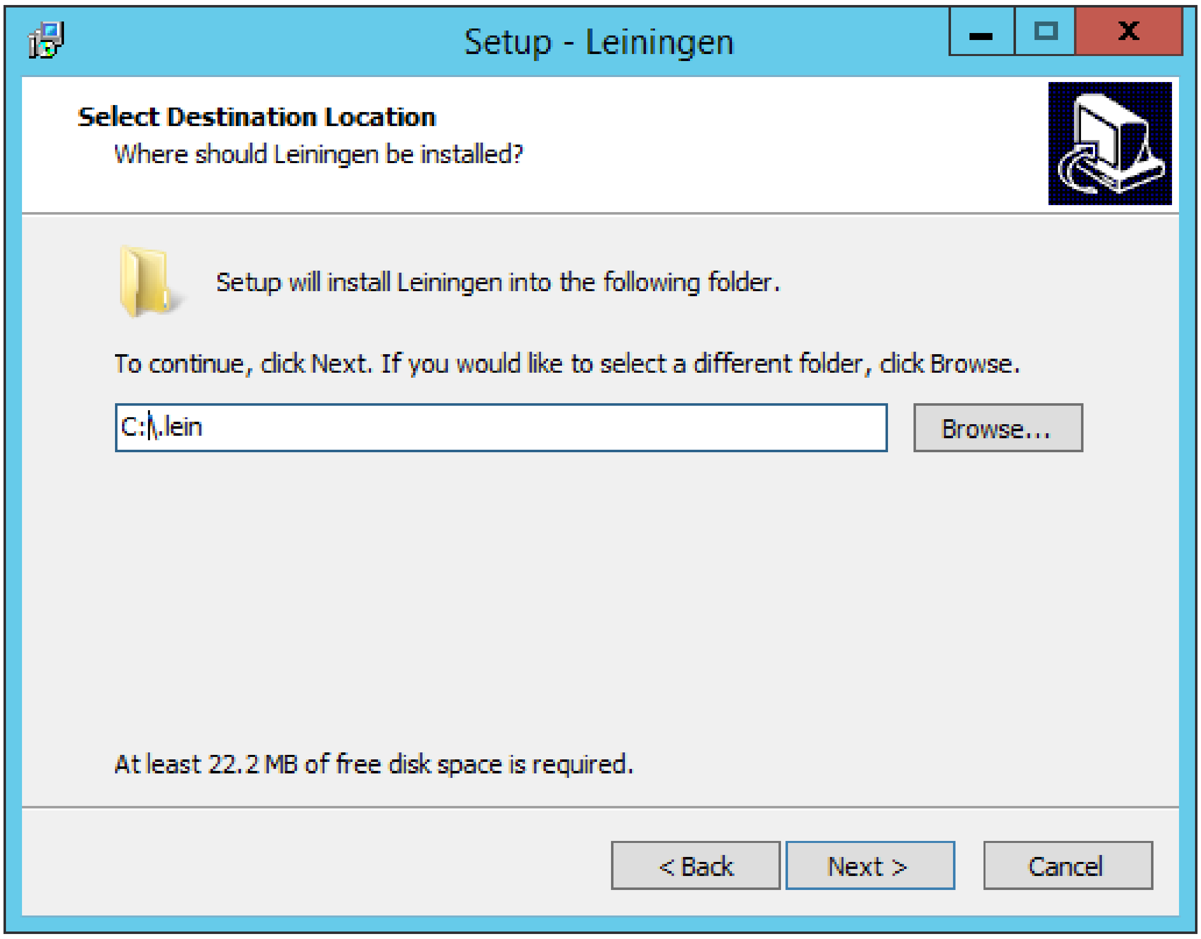
Welcome to the Leiningen Setup Wizard

This will install Leiningen version 1.0 on your computer.

It is recommended that you close all other applications before continuing.

Click Next to continue, or Cancel to exit Setup.

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



Select Destination Location

Where should Leiningen be installed?

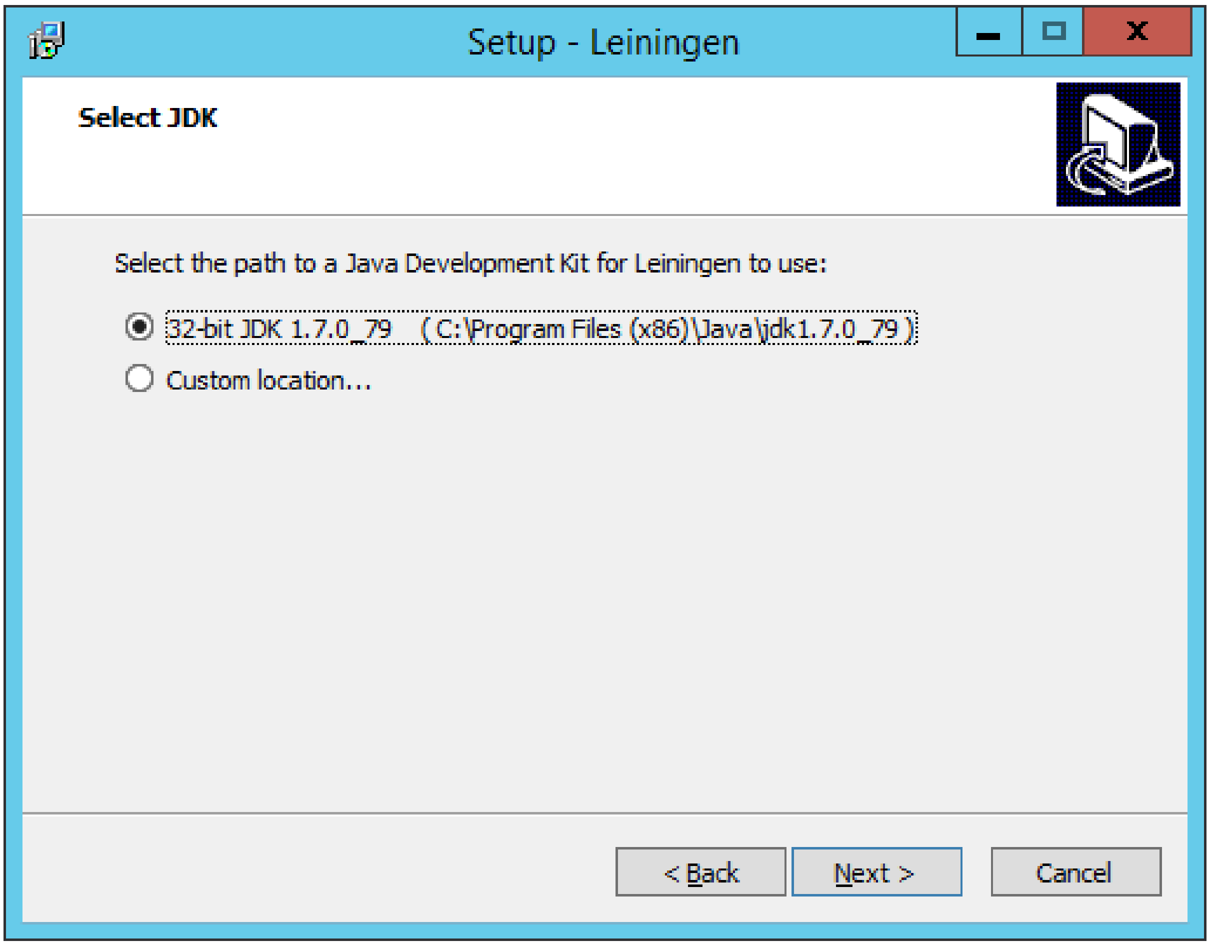
Setup will install Leiningen into the following folder.

To continue, click Next. If you would like to select a different folder, click Browse.

C:\.lein

At least 22.2 MB of free disk space is required.

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

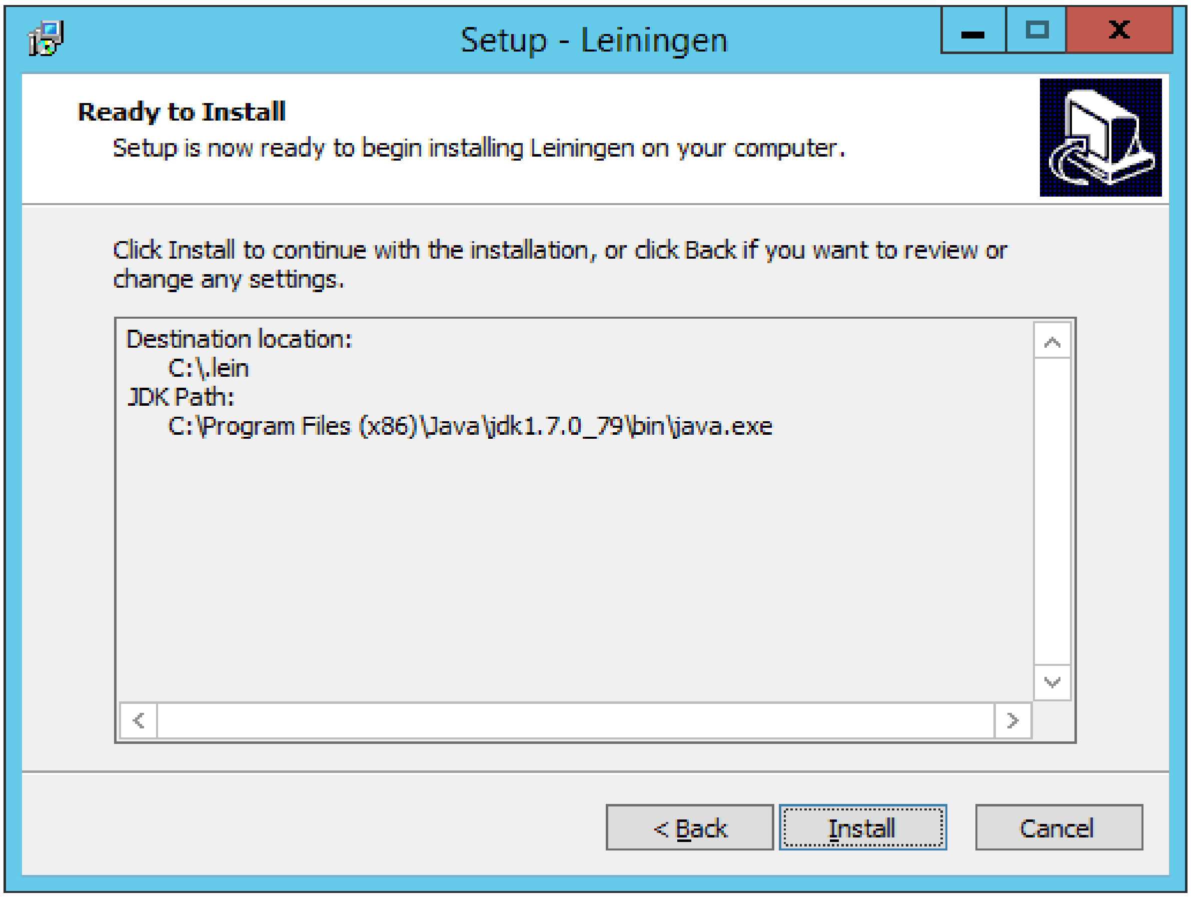


Select the path to a Java Development Kit for Leiningen to use:

[✓ 32-bit JDK 1.7.0_79 (C:\Program Files (x86)\Java\jdk1.7.0_79)]

Custom location...

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



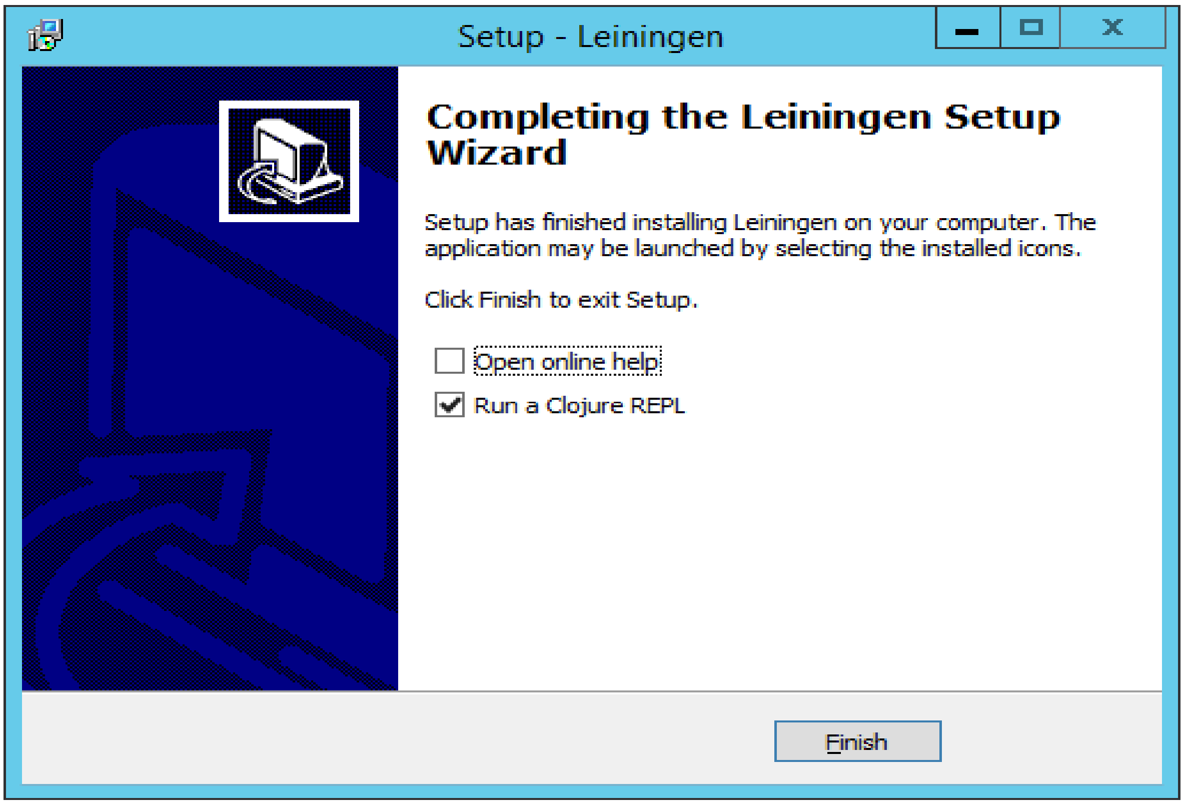
Ready to Install

Setup is now ready to begin installing Leiningen on your computer.

Click Install to continue with the installation, or click Back if you want to review or change any settings.

```
Destination location:
      C:\.lein
JDK Path:
      C:\Program Files (x86)\Java\jdk1.7.0_79\bin\java.exe
```

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.



Setup - Leiningen

Completing the Leiningen Setup Wizard

Setup has finished installing Leiningen on your computer. The application may be launched by selecting the installed icons.

Click Finish to exit Setup.

☐ Open online help

☑ Run a Clojure REPL

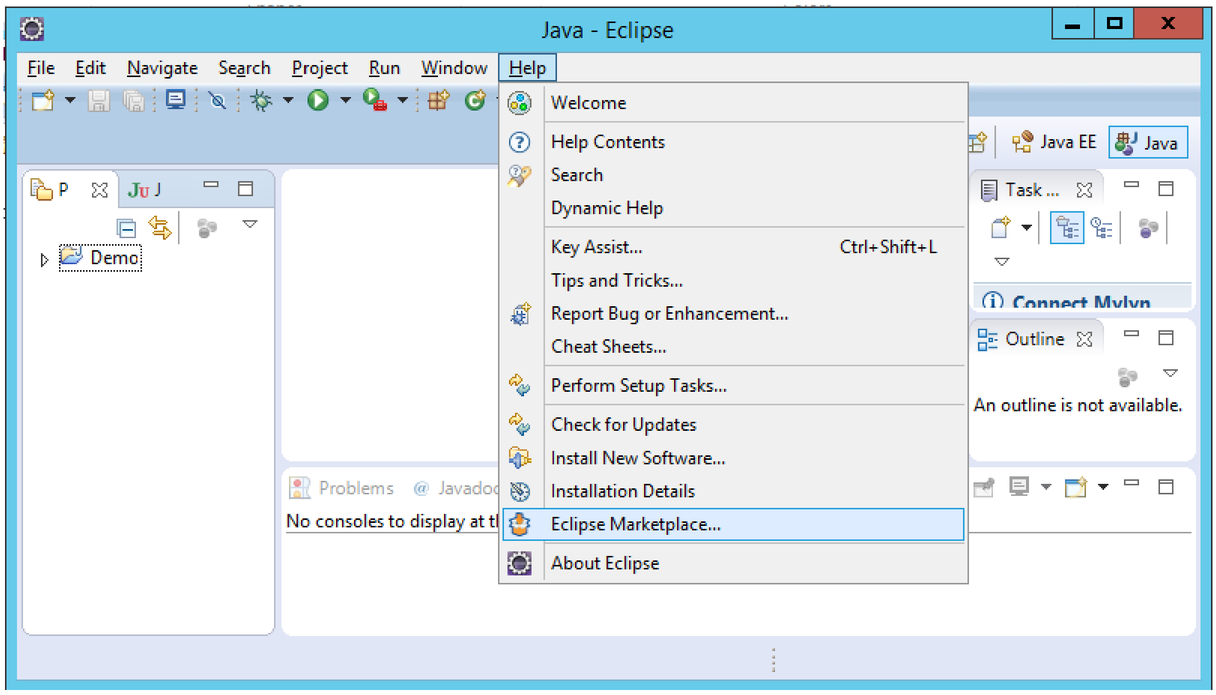
Eclipse Installation

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

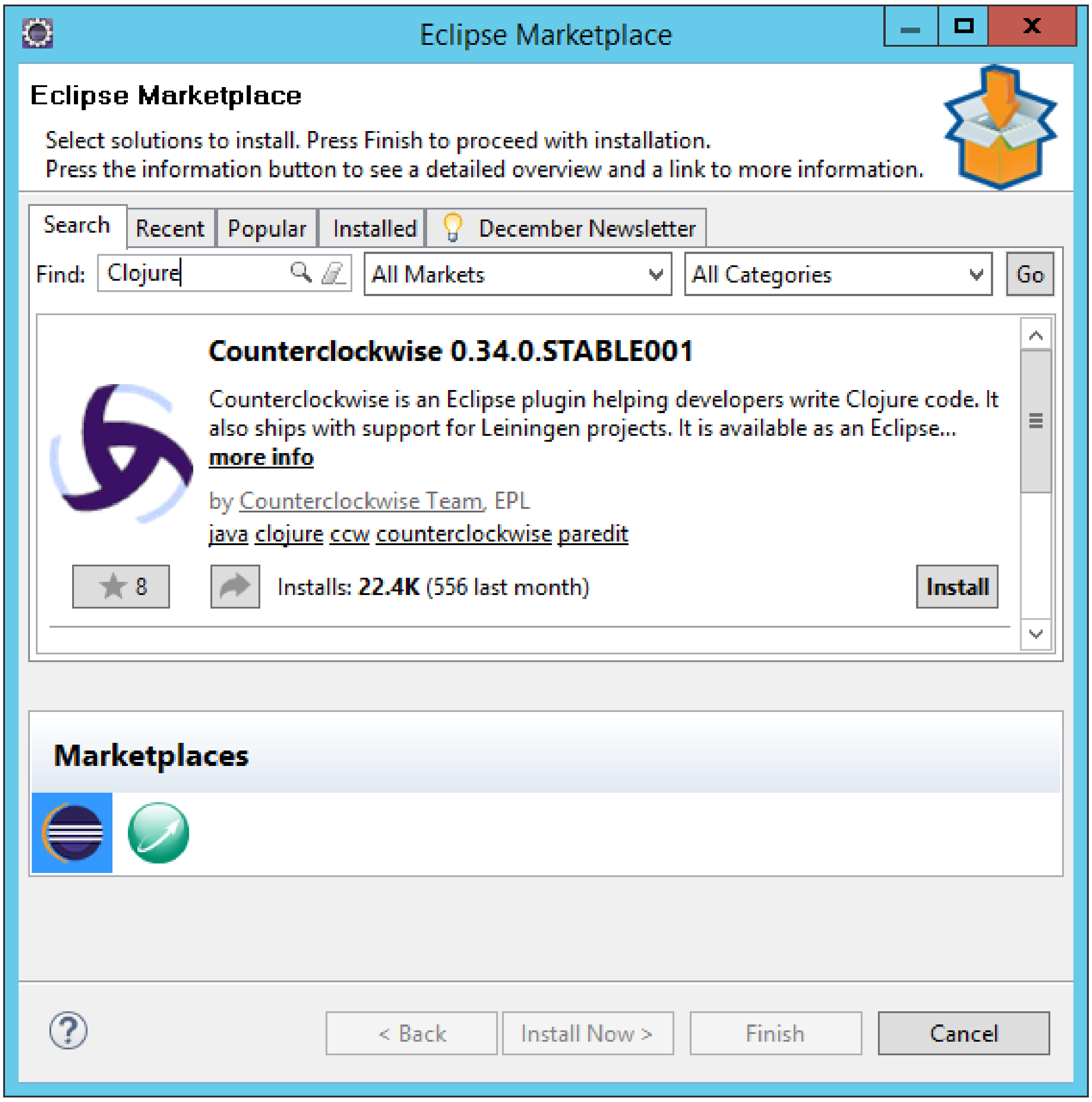
System Requirements

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.



Eclipse Marketplace

Eclipse Marketplace

Select solutions to install. Press Finish to proceed with installation.
Press the information button to see a detailed overview and a link to more information.

Find: Clojure | All Markets | All Categories | Go

Counterclockwise 0.34.0.STABLE001

Counterclockwise is an Eclipse plugin helping developers write Clojure code. It also ships with support for Leiningen projects. It is available as an Eclipse... **more info**

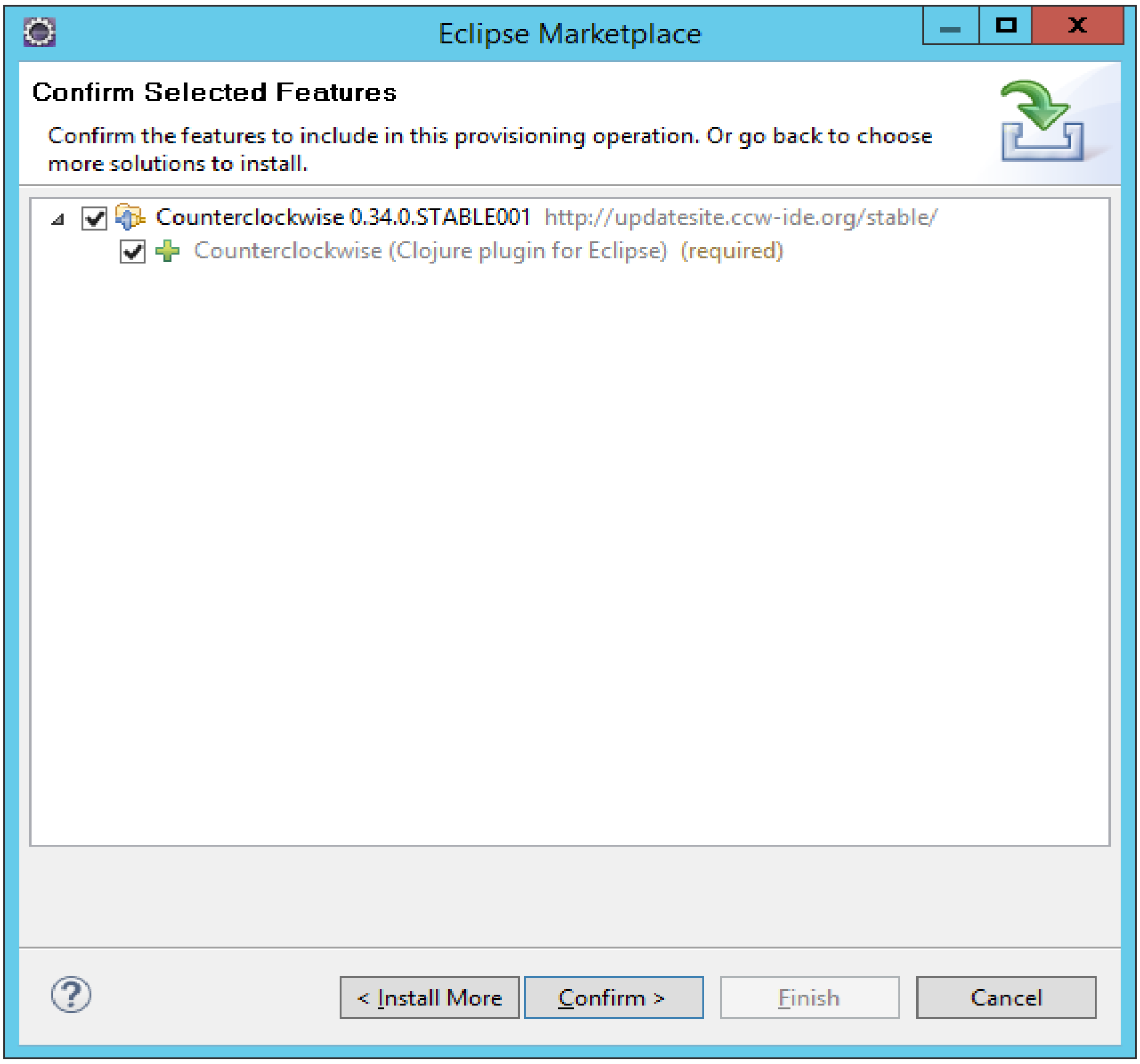
by Counterclockwise Team, EPL

java clojure ccw counterclockwise paredit

8 Installs: **22.4K** (556 last month) **Install**

Marketplaces

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

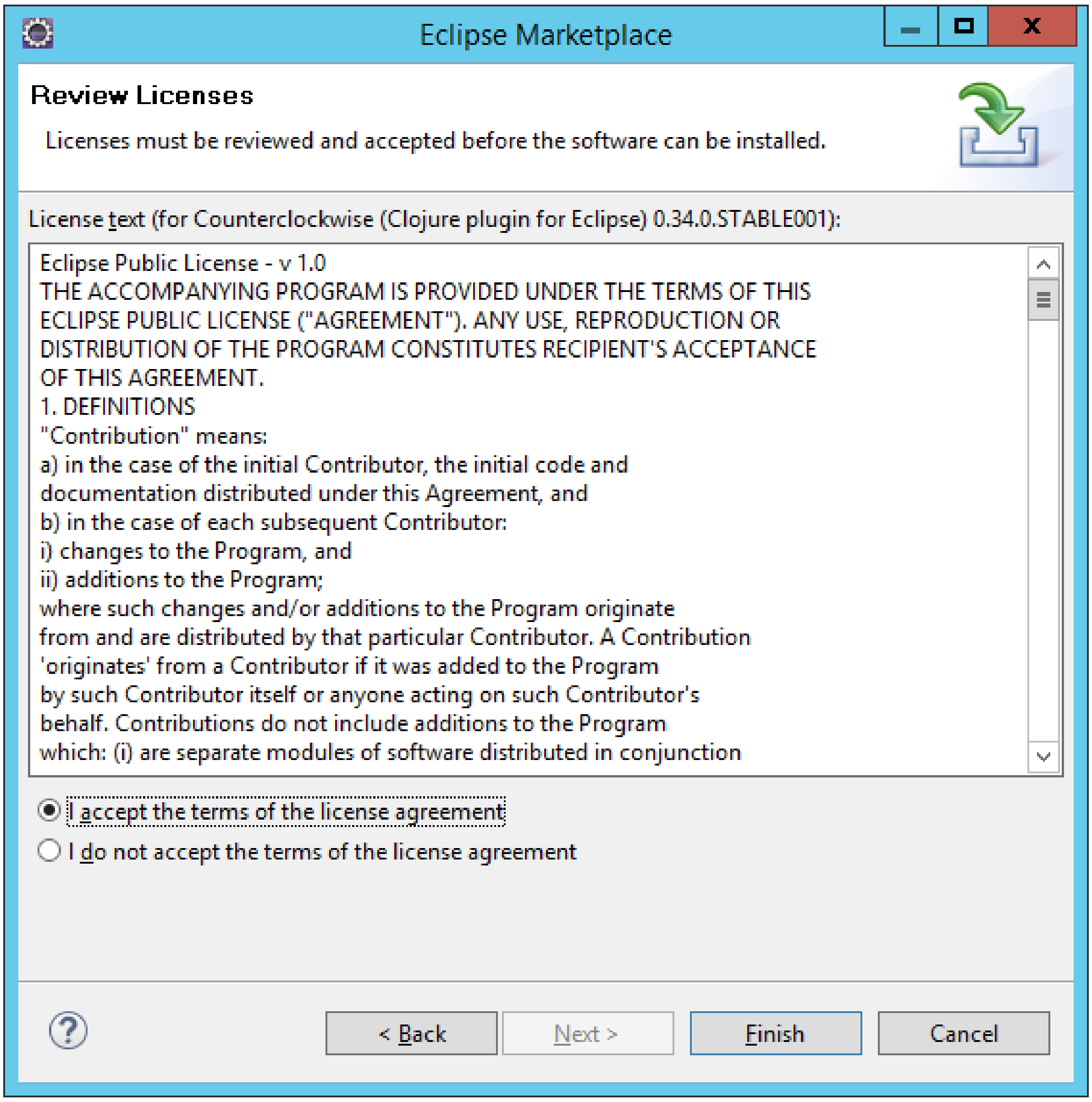


Confirm Selected Features

Confirm the features to include in this provisioning operation. Or go back to choose more solutions to install.

- ☑ Counterclockwise 0.34.0.STABLE001 http://updatesite.ccw-ide.org/stable/
  - ☑ Counterclockwise (Clojure plugin for Eclipse) (required)

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.



Licenses must be reviewed and accepted before the software can be installed.

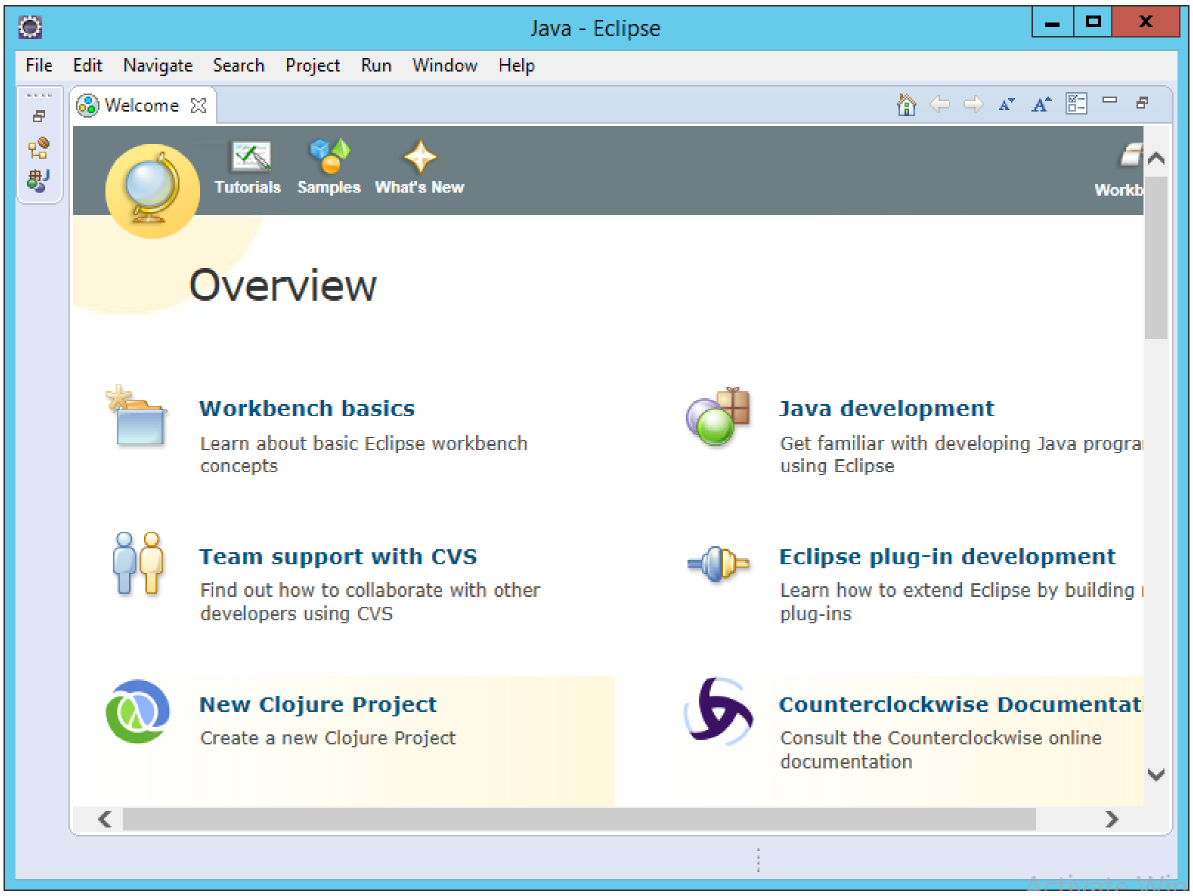
License text (for Counterclockwise (Clojure plugin for Eclipse) 0.34.0.STABLE001):

```
Eclipse Public License - v 1.0
THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS
ECLIPSE PUBLIC LICENSE ("AGREEMENT"). ANY USE, REPRODUCTION OR
DISTRIBUTION OF THE PROGRAM CONSTITUTES RECIPIENT'S ACCEPTANCE
OF THIS AGREEMENT.
1. DEFINITIONS
"Contribution" means:
a) in the case of the initial Contributor, the initial code and
documentation distributed under this Agreement, and
b) in the case of each subsequent Contributor:
i) changes to the Program, and
ii) additions to the Program;
where such changes and/or additions to the Program originate
from and are distributed by that particular Contributor. A Contribution
'originates' from a Contributor if it was added to the Program
by such Contributor itself or anyone acting on such Contributor's
behalf. Contributions do not include additions to the Program
which: (i) are separate modules of software distributed in conjunction
```

[✓ I accept the terms of the license agreement]
I do not accept the terms of the license agreement

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.



Overview

Workbench basics
Learn about basic Eclipse workbench concepts

Java development
Get familiar with developing Java progra using Eclipse

Team support with CVS
Find out how to collaborate with other developers using CVS

Eclipse plug-in development
Learn how to extend Eclipse by building plug-ins

New Clojure Project
Create a new Clojure Project

Counterclockwise Documentat
Consult the Counterclockwise online documentation

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.

```
6
```

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.

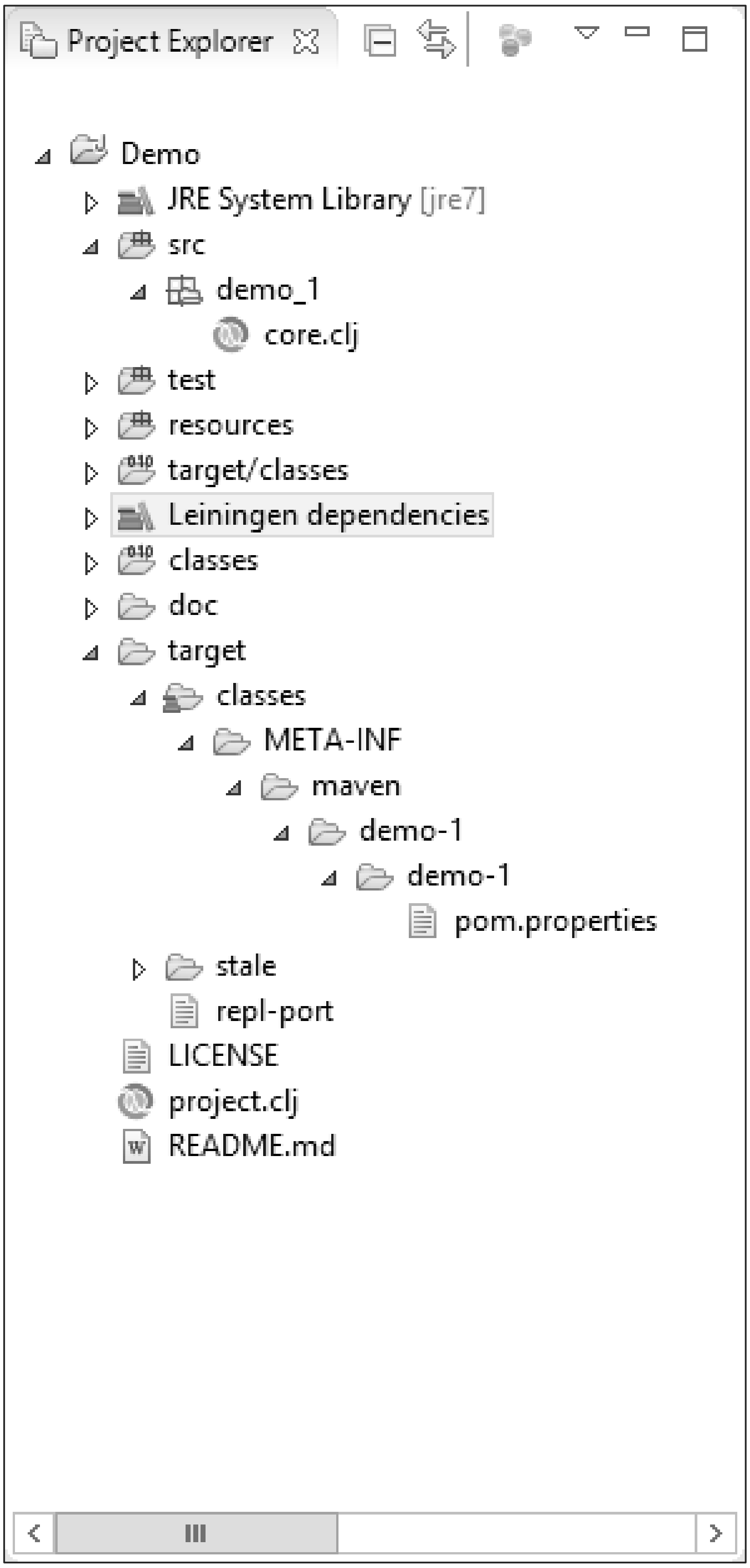
A symbol can only contain alphanumeric characters and '* + ! / . : - _ ?' but must not begin with a numeral or colon.

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.



```
Demo
  JRE System Library [jre7]
  src
    demo_1
      core.clj
  test
  resources
  target/classes
  Leiningen dependencies
  classes
  doc
  target
    classes
      META-INF
        maven
          demo-1
            demo-1
              pom.properties
    stale
    repl-port
  LICENSE
  project.clj
  README.md
```

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.

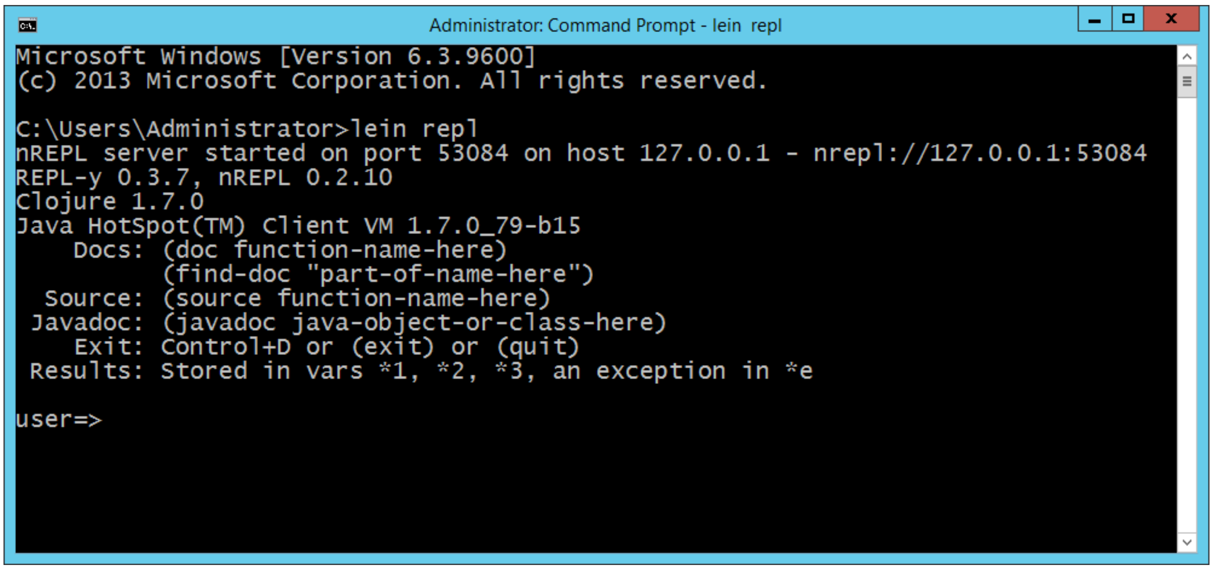
This process enables a quick feedback cycle that isn't possible in most other languages.

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.



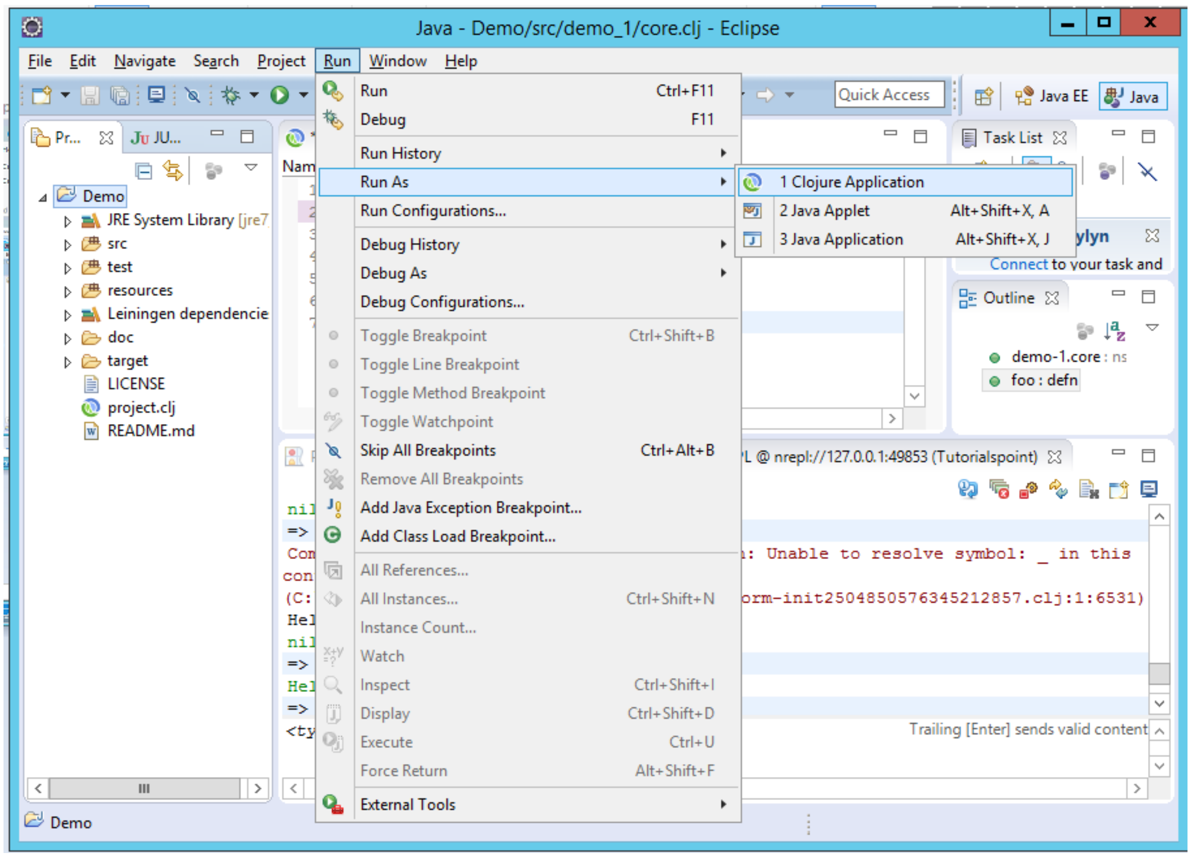
```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

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=>
```

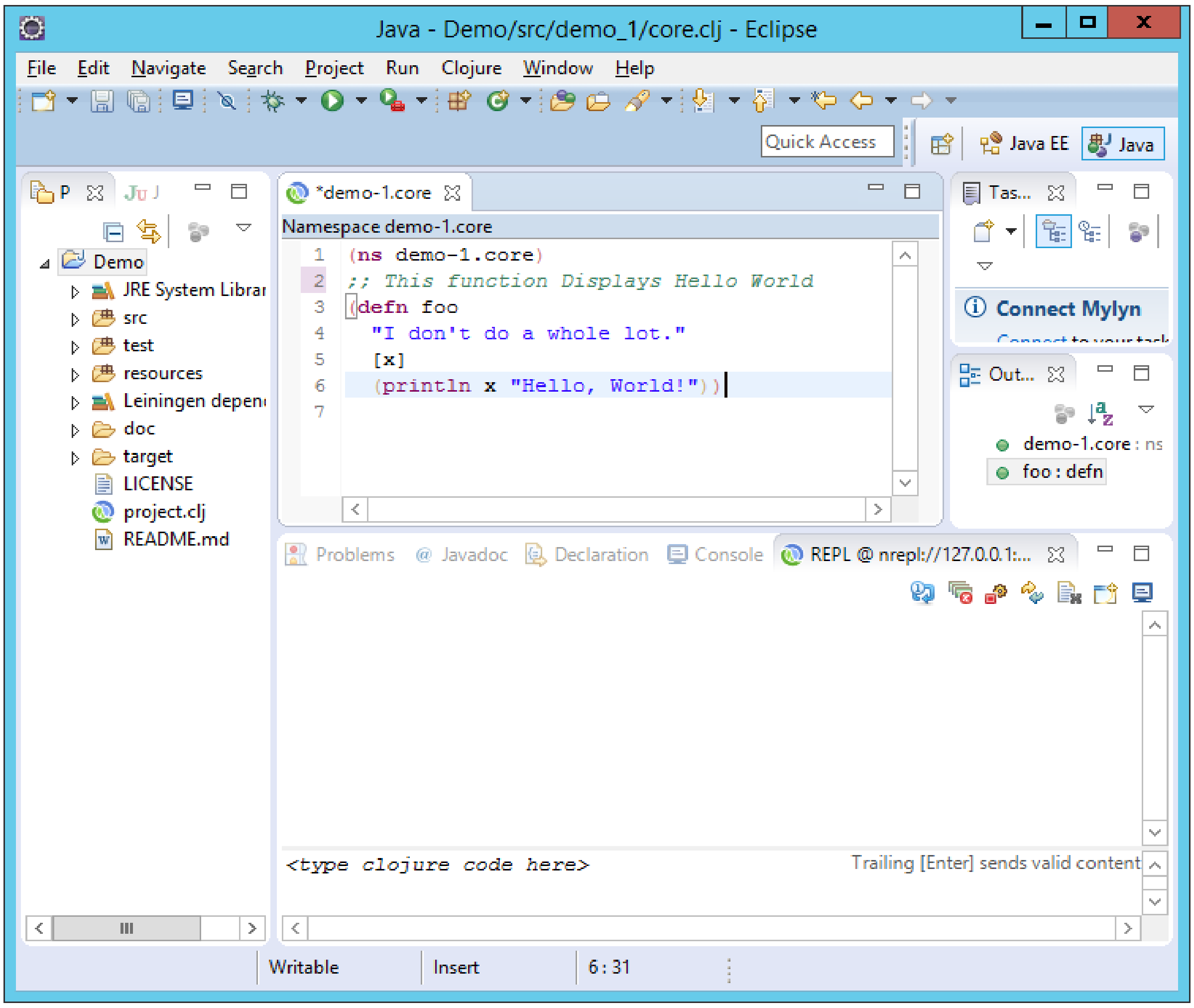
You then start evaluating Clojure commands in the REPL window as required.

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



```
: Unable to resolve symbol: _ in this
orm-init2504850576345212857.clj:1:6531)
```

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



```
(ns demo-1.core)
;; This function Displays Hello World
(defn foo
  "I don't do a whole lot."
  [x]
  (println x "Hello, World!"))
```

```
<type clojure code here>
```

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.

Printing variables

Since Clojure uses the JVM environment, you can also use the 'println' function. The following example shows how this can be achieved.

```
(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
```

7. Clojure - Operators

An **operator** is a symbol that tells the compiler to perform specific mathematical or logical manipulations.

Clojure has the following types of operators:

- Arithmetic operators
- Relational operators
- Logical operators
- Bitwise operators

Note: In Clojure, operators and operands work in the following syntax manner.

```
(operator operand1 operand2 operandn)
```

For example,

```
(+ 1 2)
```

The above example does an arithmetic operation on the numbers 1 and 2.

Arithmetic Operators

Clojure language supports the normal Arithmetic operators as any language. Following are the Arithmetic operators available in Clojure.

Operator	Description	Example
+	Addition of two operands	(+ 1 2) will give 3
−	Subtracts second operand from the first	(- 2 1) will give 1
*	Multiplication of both operands	(* 2 2) will give 4
/	Division of numerator by denominator	(float (/ 3 2)) will give 1.5

inc	Incremental operators used to increment the value of an operand by 1	inc 5 will give 6
dec	Incremental operators used to decrement the value of an operand by 1	dec 5 will give 4
max	Returns the largest of its arguments	max 1 2 3 will return 3
min	Returns the smallest of its arguments	min 1 2 3 will return 1
rem	Remainder of dividing the first number by the second	rem 3 2 will give 1

The following code snippet shows how the various operators can be used.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def x (+ 2 2))
(println x)
(def x (- 2 1))
(println x)
(def x (* 2 2))
(println x)
(def x (float(/ 2 1)))
(println x)
(def x (inc 2))
(println x)
(def x (dec 2))
(println x)
(def x (max 1 2 3))
(println x)
(def x (min 1 2 3))
(println x)
(def x (rem 3 2))
(println x)
)
(Example)
```

The above program produces the following output.

```
4
1
4
2.0
3
1
3
1
1
```

Relational Operators

Relational operators allow comparison of objects. Following are the relational operators available in Clojure.

Operator	Description	Example
=	Tests the equality between two objects	(= 2 2) will give true
not=	Tests the difference between two objects	(not= 3 2) will give true
<	Checks to see if the left object is less than the right operand	(< 2 3) will give true
<=	Checks to see if the left object is less than or equal to the right operand	(<= 2 3) will give true
>	Checks to see if the left object is greater than the right operand	(> 3 2) will give true
>=	Checks to see if the left object is greater than or equal to the right operand	(>= 3 2) will give true

The following code snippet shows how the various operators can be used.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def x (= 2  2))
(println x)
(def x (not= 3  2))
(println x)
(def x (< 2 3))
(println x)
(def x (<= 2 3))
(println x)
(def x (> 3 2))
(println x)
(def x (>= 3 2))
(println x)
)
(Example)
```

The above program produces the following output.

```
true
true
true
true
true
true
```

Logical Operators

Logical operators are used to evaluate Boolean expressions. Following are the logical operators available in Groovy.

Operator	Description	Example
or	This is the logical "and" operator	(or true true) will give true
and	This is the logical "or" operator	(and true false) will give false
not	This is the logical "not" operator	(not false) will give true

The following code snippet shows how the various operators can be used.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def x (or true true))
(println x)
(def x (and true false))
(println x)
(def x (not true))
(println x)
)
(Example)
```

The above program produces the following output.

```
true
false
false
```

Bitwise Operators

Groovy provides four bitwise operators. Following are the bitwise operators available in Groovy.

Operator	Description
bit-and	This is the bitwise "and" operator
bit-or	This is the bitwise "or" operator
bit-xor	This is the bitwise "xor" or Exclusive 'or' operator
bit-not	This is the bitwise negation operator

Following is the truth table showcasing these operators.

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

The following code snippet shows how the various operators can be used.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def x (bit-and 00111100 00001101))
(println x)
(def x (bit-or 00111100 00001101))
(println x)
(def x (bit-xor 00111100 00001101))
(println x)
) (Example)
```

The above program produces the following output.

```
576
37441
36865
```

Operator Precedence

As is the case with LISPs in general, there is no need to worry about operator precedence. This is one of the benefits of S-Expressions and prefix notation. All functions evaluate left to right and inside out. The operators in Clojure are just functions, and everything is fully parenthesized.

8. Clojure - Loops

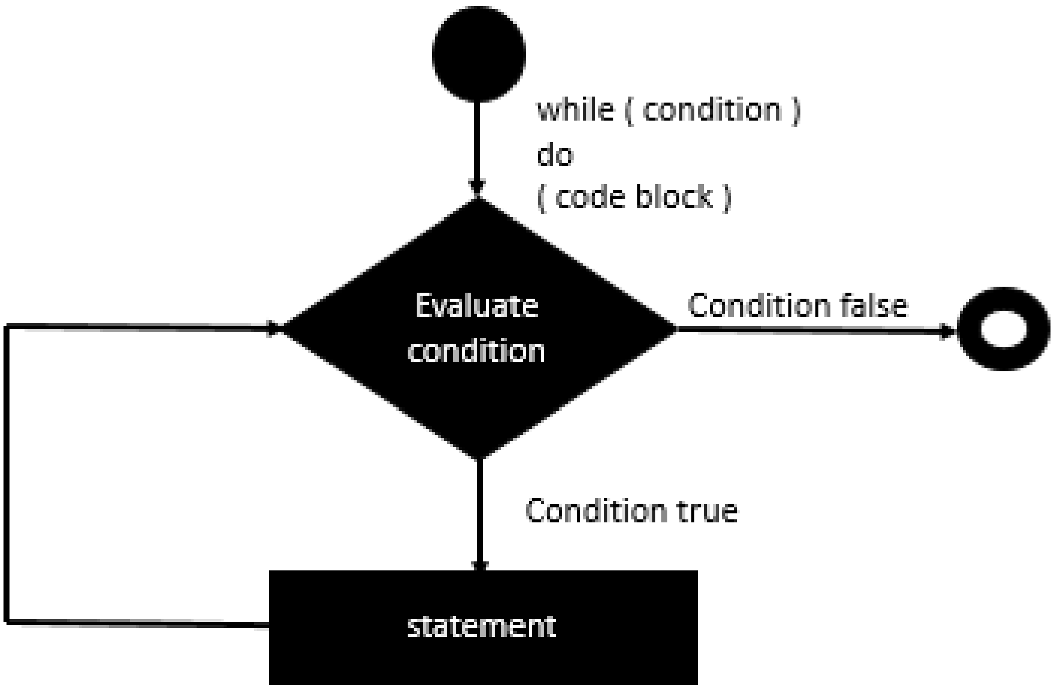
So far we have seen statements which are executed one after the other in a sequential manner. Additionally, statements are provided in Clojure to alter the flow of control in a program's logic. They are then classified into flow of control statements which we will see in detail.

While Statement

Following is the syntax of the '**while**' statement.

```
(while(expression)
(do
codeblock)
)
```

The while statement is executed by first evaluating the condition expression (a Boolean value), and if the result is true, then the statements in the while loop are executed. The process is repeated starting from the evaluation of the condition in the while statement. This loop continues until the condition evaluates to false. When the condition is false, the loop terminates. The program logic then continues with the statement immediately following the while statement. Following is the diagrammatic representation of this loop.



Following is an example of a while loop statement.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def x (atom 1))
(
        while ( < @x 5 )
        (do
                    (println @x)
                    (swap! x inc))
        )
)
(Example)
```

In the above example, we are first initializing the value of 'x' variable to 1. Note that we are using an atom value, which is a value which can be modified. Then our condition in the while loop is that we are evaluating the condition of the expression to be such as 'x' should be less than 5. Till the value of 'x' is less than 5, we will print the value of 'x' and then increase its value. The swap statement is used to populate the atom variable of 'x' with the new incremented value.

The above code produces the following output.

```
1
2
3
4
```

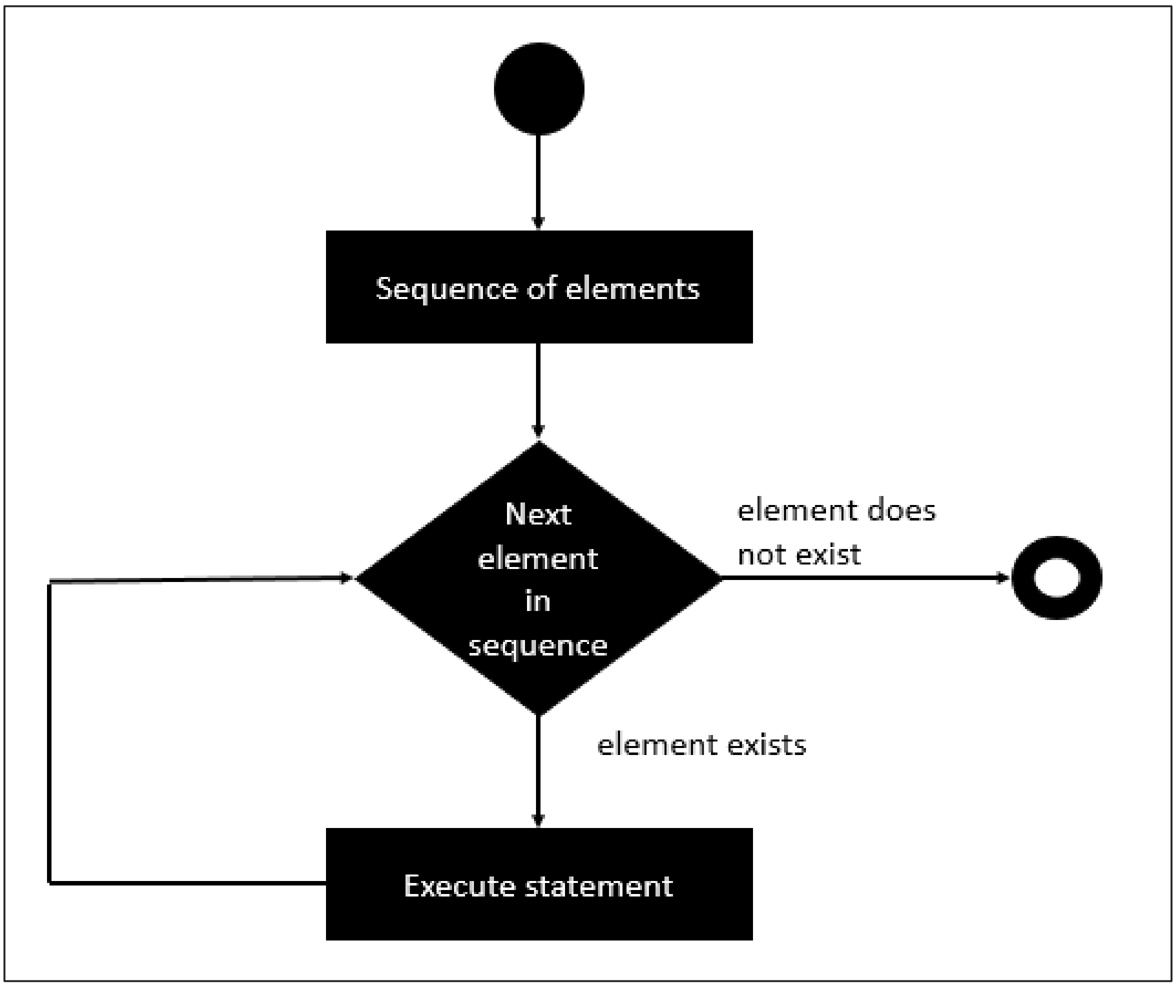
Doseq Statement

The '**doseq**' statement is similar to the 'for each' statement which is found in many other programming languages. The doseq statement is basically used to iterate over a sequence.

Following is the general syntax of the doseq statement.

```
(doseq (sequence)
statement#1
)
```

Following is the diagrammatic representation of this loop.



Following is an example of the doseq statement.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
        (doseq [n [0 1 2]]
                (println n))
)
(Example)
```

In the above example, we are using the doseq statement to iterate through a sequence of values 0, 1, and 2 which is assigned to the variable n. For each iteration, we are just printing the value to the console.

The above code produces the following output.

```
0
1
2
```

Dotimes Statement

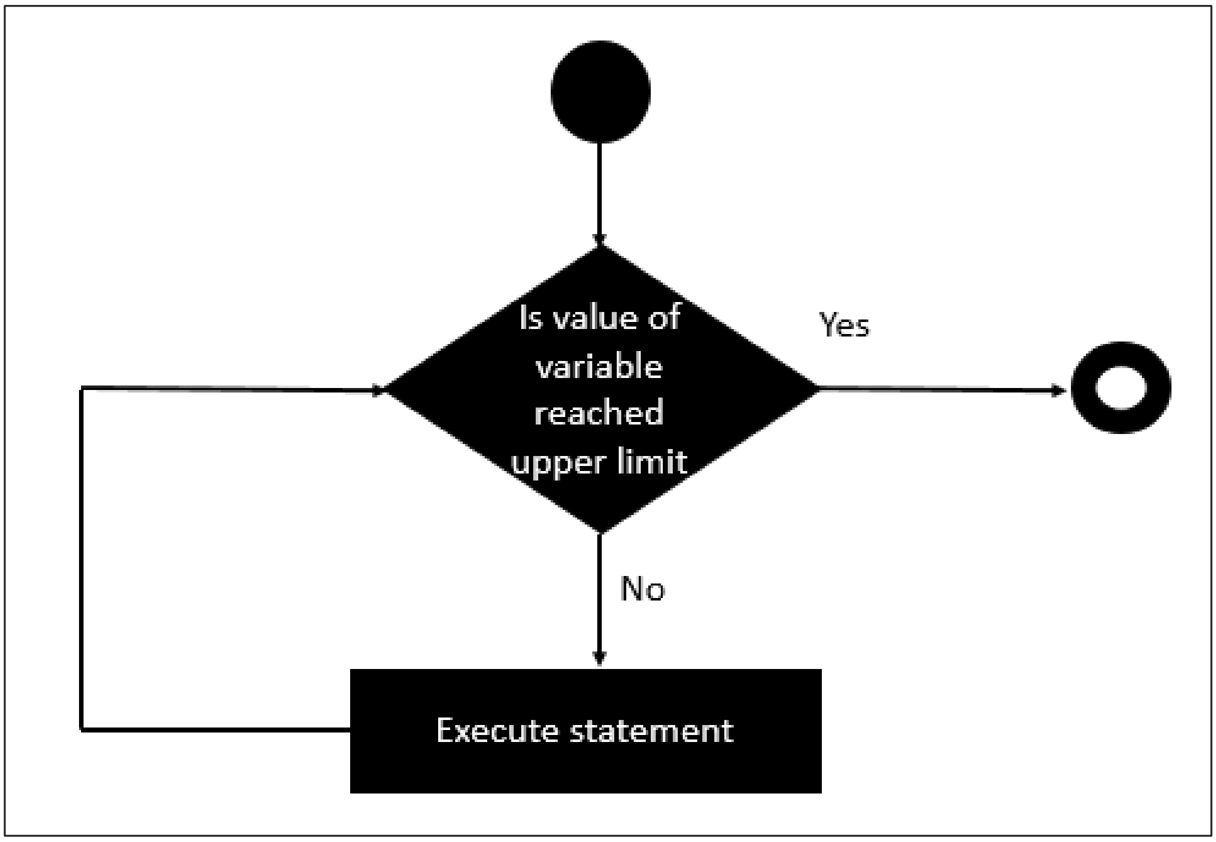
The '**dotimes'** statement is used to execute a statement 'x' number of times.

Following is the general syntax of the doseq statement.

```
(dotimes (variable value)
statement
)
```

Where value has to be a number which indicates the number of times the loop needs to be iterated.

Following is the diagrammatic representation of this loop.



Following is an example of the 'doseq' statement.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
        (dotimes [n 5]
                (println n))
)
(Example)
```

In the above example, we are using the dotimes statement to repeat the number if times the println statement is executed. And for each iteration it also increments the value the variable n.

The above code produces the following output.

```
0
1
2
3
4
```

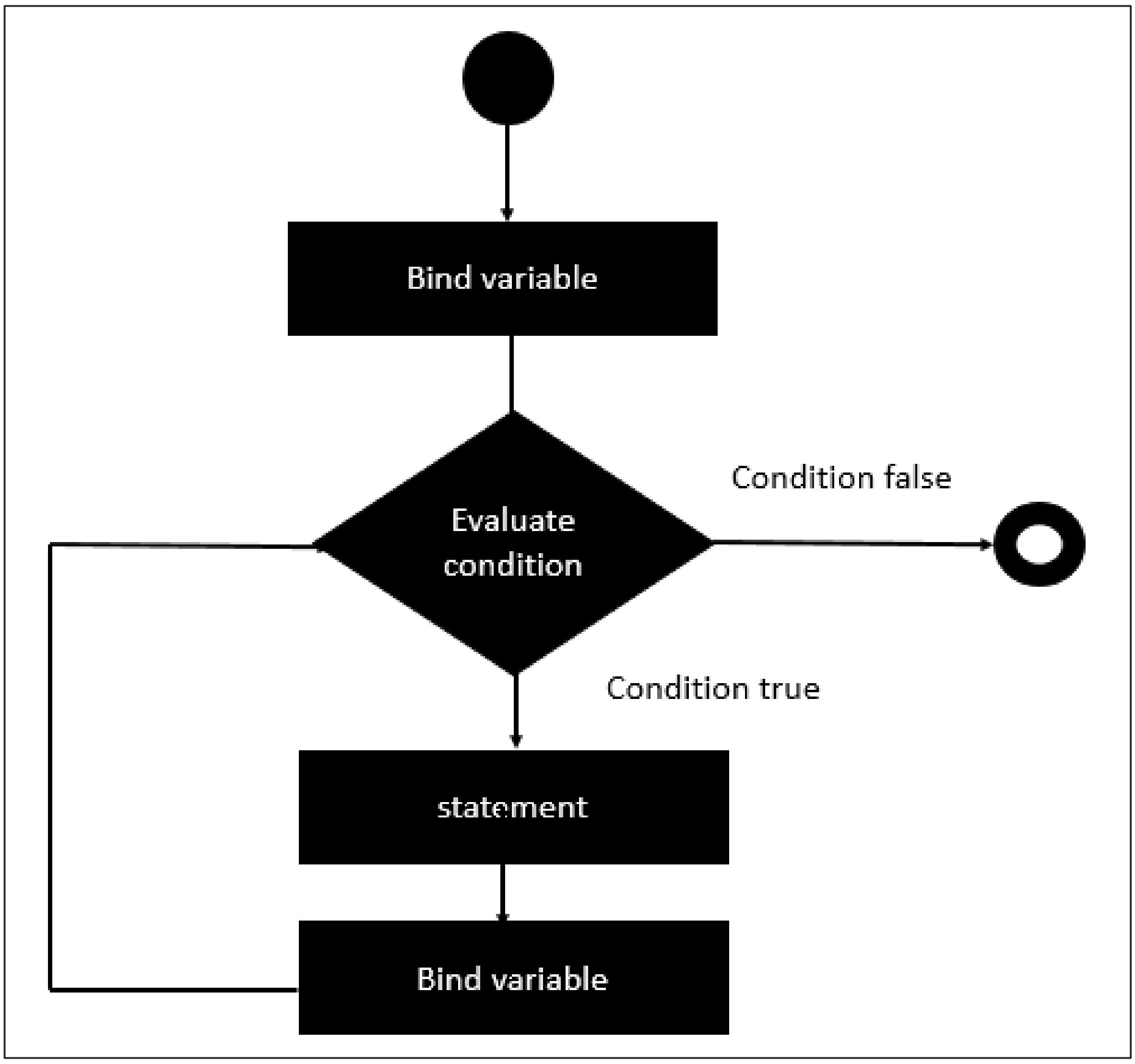
Loop Statement

The loop special form is not like a '**for'** loop. The usage of loop is the same as the let binding. However, loop sets a recursion point. The recursion point is designed to use with recur, which means loop is always used with **recur**. To make a loop happen, the number of arguments (arity) specified for recurs must coincide with the number of bindings for the loop. That way, recur goes back to the loop.

Following is the general syntax of the loop statement.

```
loop [binding]
  (condition
    (statement)
    (recur (binding))
```

Following is the diagrammatic representation of this loop.



Following is an example of a '**for-in**' statement.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(loop [x 10]
                (when (> x 1)
                (println x)
                        (recur (- x 2))
                )        ))      (Example)
```

In the above example, we are first binding the value of ‘x’ to 10 using the loop statement. We then use the **when condition clause** to see if the value of ‘x’ is less than 1. We then print the value of ‘x’ to the console and use the recur statement to repeat the loop. The loop is repeated after the value of ‘x’ is decremented by 2.

The above code produces the following output.

```
10

8

6

4

2
```

9. Clojure - Decision Making

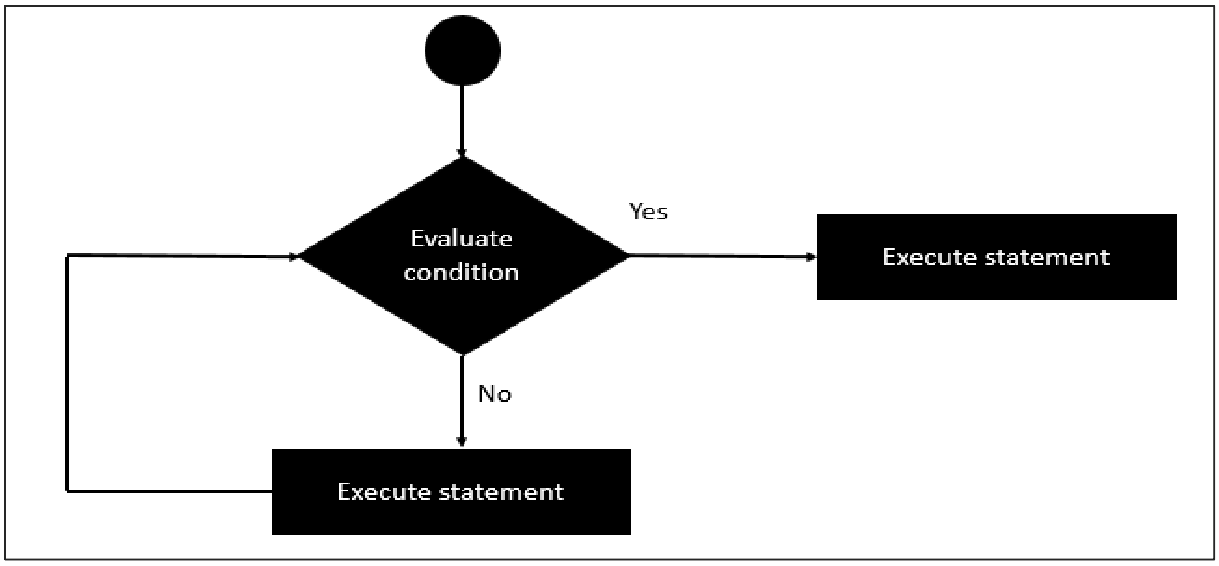
Decision-making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

If Statement

The first decision-making statement is the '**if**' statement. Following is the general form of this statement in Clojure.

```
if (condition) statement#1 statement #2
```

In Clojure, the condition is an expression which evaluates it to be either true or false. If the condition is true, then statement#1 will be executed, else statement#2 will be executed. The general working of this statement is that first a condition is evaluated in the 'if' statement. If the condition is true, it then executes the statements. Following diagram shows the flow of the 'if' statement.



Following is an example of the simple 'if' expression in Clojure.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(if ( = 2 2) (println "Values are equal") (println "Values are not equal")))
(Example)
```

The output of the above program will be "Values are equal". In the above code example, the 'if' condition is used to evaluate whether the values of 2 and 2 are equal. If they are, then it will print the value of "Values are equal" else it will print the value of "Values are not equal".

If/do Expression

The '**if-do**' expression in Clojure is used to allow multiple expressions to be executed for each branch of the 'if' statement. We have seen in the classic 'if' statement in Clojure that you can just have two statements, one which is executed for the true part and the other which is for the false part. But the 'if-do' expression allows you to use multiple expressions. Following is the general form of the 'if-do' expression.

```
if(condition)
(
statement #1
statement #1.1
)
(
statement #2
statement #2.1
}
```

Following is an example of a '**for if-do**' statement.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(        if ( = 2 2)
        (
            do(println "Both the values are equal")
            (println "true")
        )
        (            do(println "Both the values are not equal")
            (println "false")
        )
    )
)
(Example)
```

In the above example, the 'if' condition is used to evaluate whether the values of 2 and 2 are equal. If they are, then it will print the value of "Values are equal" and in addition we are printing the value of "true", else it will print the value of "Values are not equal" and the value of "false".

The above code produces the following output.

```
Both the values are equal

true
```

Nested If Statement

Sometimes there is a requirement to have multiple 'if' statement embedded inside of each other, as is possible in other programming languages. In Clojure, this is made possible with the help of using the logical 'and' when evaluating multiple expressions.

Following is the general form of this statement.

```
if(and condition1 condition2) statement #1 statement #2
```

Following is an example of how multiple conditions can be implemented.

```
(ns clojure.examples.hello

     (:gen-class))

     ;; This program displays Hello World

(defn Example []

(

                if ( and (= 2 2) (= 3 3)) (println "Values are equal") (println
"Values are not equal")

        )

)

(Example)
```

The above code produces the following output.

```
Values are equal
```

Case Statement

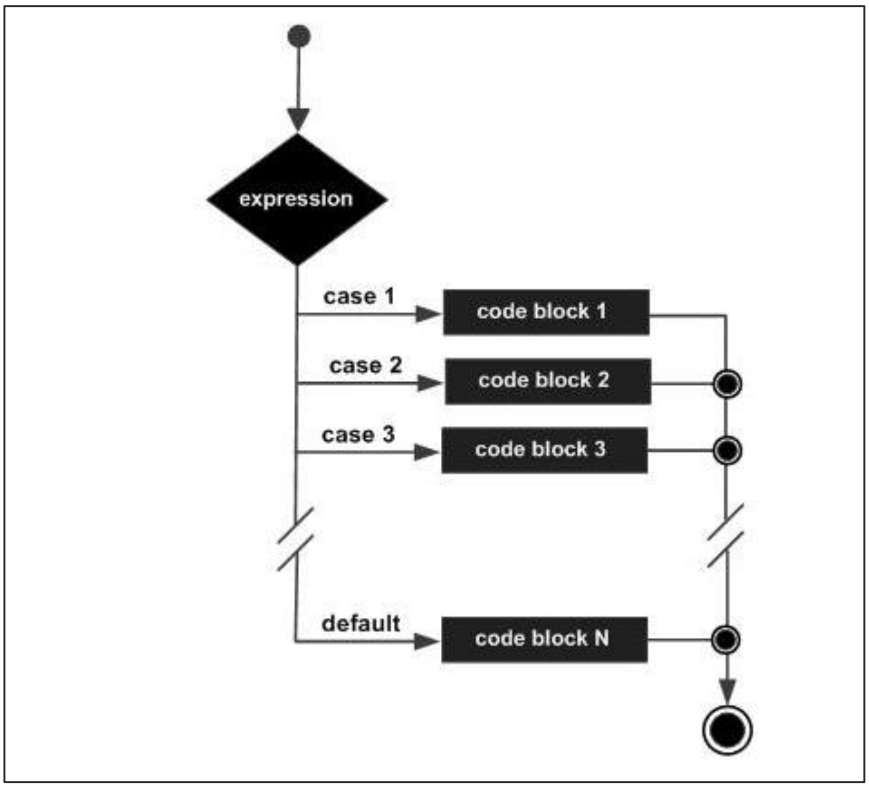
Clojure offers the '**case'** statement which is similar to the '**switch'** statement available in the Java programming language. Following is the general form of the case statement.

```
case expression
value1  statement #1
value2  statement #2
valueN  statement #N
statement #Default
```

The general working of this statement is as follows:

- The expression to be evaluated is placed in the 'case' statement. This generally will evaluate to a value, which is used in the subsequent statements.
- Each value is evaluated against that which is passed by the 'case' expression. Depending on which value holds true, subsequent statement will be executed.
- There is also a default statement which gets executed if none of the prior values evaluate to be true.

Following diagram shows the flow of the 'if' statement.



Following is an example of the 'case' statement in Clojure.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def x 5)
        ( case x
                5 (println "x is 5")
                10 (println "x is 10")
                (println "x is neither 5 nor 10")
        )
)
(Example)
```

In the above example, we are first initializing a variable 'x' to a value of 5. We then have a 'case' statement which evaluates the value of the variable 'x'. Based on the value of the variable, it will execute the relevant case set of statements. The last statement is the default statement, if none of the previous statements are executed.

The above code produces the following output.

```
x is 5
```

Cond Statement

Clojure offers another evaluation statement called the '**cond'** statement. This statement takes a set of test/expression pairs. It evaluates each test one at a time. If a test returns logical true, 'cond' evaluates and returns the value of the corresponding expression and doesn't evaluate any of the other tests or expressions. 'cond' returns nil.

Following is the general form of this statement.

```
cond
(expression evaluation1)  statement #1
(expression evaluation2)  statement #2
(expression evaluationN)  statement #N
:else statement #Default
```

The general working of this statement is as follows:

- There are multiple expression evaluation defined and for each there is a statement which gets executed.
- There is also a default statement, which gets executed if none of the prior values evaluate to be true. This is defined by the :else statement.

Following is an example of the 'cond' statement in Clojure.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def x 5)
        ( cond
                (= x 5) (println "x is 5")
                (= x 10)(println  "x is 10")
        :else (println "x is not defined")
)
)
(Example)
```

In the above example, we are first initializing a variable x to a value of 5. We then have a 'cond' statement which evaluates the value of the variable 'x'. Based on the value of the variable, it will execute the relevant set of statements.

The above code produces the following output.

```
x is 5
```

10. Clojure - Functions

Clojure is known as a functional programming language, hence you would expect to see a lot of emphasis on how functions work in Clojure. This chapter covers what all can be done with functions in Clojure.

Defining a Function

A function is defined by using the '**defn'** macro. Following is the general syntax of the definition of a function.

```
(defn functionname
“optional documentation string”
[arguments]
(code block))
```

Functions can have documentation strings, which is good to describe what the function actually does.

Following is a simple example of a function.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def x 1)
(def y 1.25)
(def str1 "Hello")
(println x)
(println y)
(println str1)
)
(Example)
```

In the above example, the name of the function is Example.

Anonymous Functions

An anonymous function is a function which has no name associated with it. Following is an example of an anonymous function.

```
(ns clojure.examples.hello

     (:gen-class))

     ;; This program displays Hello World

(defn Example []

((fn [x] (* 2 x)) 2)

)

(Example)
```

The above example defines a function which takes a value of 'x' as an argument and the function itself multiplies the value of the argument by 2.

Functions with Multiple Arguments

Clojure functions can be defined with zero or more parameters. The values you pass to functions are called **arguments**, and the arguments can be of any type. The number of parameters is the function's arity. This chapter discusses some function definitions with different arities.

In the following example, the function demo is defined with multiple arguments for each function definition.

```
(defn demo [] (* 2 2))
(defn demo [x] (* 2 x))
(defn demo [x y] (* 2 x y))
```

In the above example, the first function definition is a 0-arity function, since it has 0 arguements, one-param is 1-arity, and two-params is 2-arity and so on.

Variadic Functions

Variadic functions are functions that take varying number of arguments (some arguments are optional). Function can also specify the '&' ampersand symbol to take in an arbitrary number of arguments.

Following example shows how this can be achieved.

```
(defn demo [message & others] (str message (clojure.string/join " " others)))
```

The above function declaration has the '&' symbol next to the argument others, which means that it can take an arbitrary number of arguments.

If you invoke the above function as

```
(demo "Hello" "This" "is" "the" "message")
```

Following will be the output.

```
“HelloThis is the message”
```

The '**clojure.string/join**' is used to combine each individual string argument, which is passed to the function.

Higher Order Functions

Higher-order functions (HOFs) are functions that take other functions as arguments. HOFs are an important functional programming technique and are quite commonly used in Clojure. One example of an HOF is a function that takes a function and a collection and returns a collection of elements that satisfy a condition (a predicate). In Clojure, this function is called clojure.core/filter:

Following is an example code of the higher order function.

```
(filter even? (range 0 10))
```

The above program produces the following output.

```
(0 2 4 6 8)
```

11. Clojure - Numbers

Numbers datatype in Clojure is derived from Java classes.

Clojure supports integer and floating point numbers.

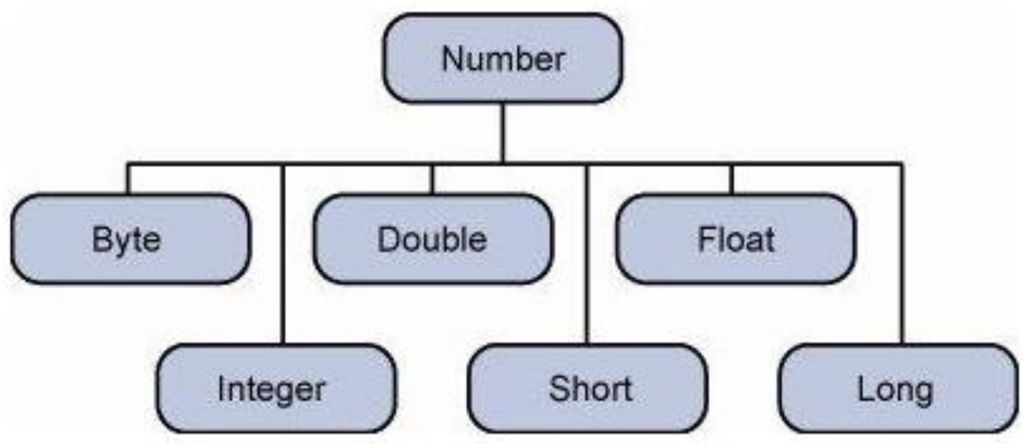
- An integer is a value that does not include a fraction.
- A floating-point number is a decimal value that includes a decimal fraction.

Following is an example of numbers in Clojure.

```
(def x 5)
(def y 5.25)
```

Where 'x' is of the type **Integer** and 'y' is the **float**.

In Java, the following classes are attached to the numbers defined in Clojure.



To actually see that the numbers in Clojure are derived from Java classes, use the following program to see the type of numbers assigned when using the 'def' command.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def x 5)
(def y 5.25)
(println (type x))
(println (type y))  )
(Example)
```

The '**type**' command is used to output the class associated with the value assigned to a variable.

The above code will produce the following output.

```
Java.lang.long
Java.lang.double
```

Number Tests

The following test functions are available for numbers.

zero?

Returns true if the number is zero, else false.

Following is the syntax.

```
(zero? number)
```

Following is an example of the zero test function.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def x (zero? 0))
(println x)
(def x (zero? 0.0))
(println x)
(def x (zero? 1))
(println x))
(Example)
```

The above program produces the following output.

```
true
true
false
```

pos?

Returns true if number is greater than zero, else false.

Following is the syntax.

```
(pos? number)
```

Following is an example for the pos test function.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def x (pos? 0))
(println x)
(def x (pos? -1))
(println x)
(def x (pos? 1))
(println x))
(Example)
```

The above program produces the following output.

```
false
false
true
```

neg?

Returns true if number is less than zero, else false.

Following is the syntax.

```
 (neg? number)
```

Following is an example of the neg test function.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
```

```
(defn Example []
(def x (neg? -1))
(println x)
(def x (neg? 0))
(println x)
(def x (neg? 1))
(println x))
(Example)
```

The above program produces the following output.

```
true
true
false
```

even?

Returns true if the number is even, and throws an exception if the number is not an integer.

Following is the syntax.

```
(even? number)
```

Following is an example of the even test function.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def x (even? 0))
(println x)
(def x (even? 2))
(println x)
(def x (even? 3))
(println x))
(Example)
```

The above program produces the following output.

```
true
true
false
```

odd?

Returns true if the number is odd, and throws an exception if the number is not an integer.

Following is the syntax.

```
(odd? number)
```

Following is an example of the odd test function.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def x (odd? 0))
(println x)
(def x (odd? 2))
(println x)
(def x (odd? 3))
(println x))
(Example)
```

The above program produces the following output.

```
false
false
true
```

number?

Returns true if the number is really a Number.

Following is the syntax.

```
(number? number)
```

Following is an example of the number test function.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def x (number? 0))
(println x)
(def x (number? 0.0))
(println x)
(def x (number? :a))
(println x))
(Example)
```

The above program produces the following output.

```
true
true
false
```

integer?

Returns true if the number is an integer.

Following is the syntax.

```
(integer? number)
```

Following is an example of the integer test function.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def x (integer? 0))
(println x)
(def x (integer? 0.0))
(println x)
(Example)
```

The above program produces the following output.

```
true
false
```

float?

Returns true if the number is a float.

Following is the syntax.

```
 (float? number)
```

Following is an example of the float test function.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def x (float? 0))
(println x)
(def x (float? 0.0))
(println x)
(Example)
```

The above program produces the following output.

```
false
true
```

12. Clojure - Recursion

We have seen the recur statement in an earlier topic and whereas the 'for' loop is somewhat like a loop, **recur** is a real loop in Clojure.

If you have a programming background, you may have heard of tail recursion, which is a major feature of functional languages. This recur special form is the one that implements tail recursion. As the word "tail recursion" indicates, recur must be called in the tail position. In other words, recur must be the last thing to be evaluated.

The simplest example of the recur statement is used within the 'for' loop. In the following example, the recur statement is used to change the value of the variable 'i' and feed the value of the variable back to the loop expression.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
        (loop [i 0]
                (when (< i 5)
                (println i)
                        (recur (inc i))
                ))
)
(Example)
```

The above program produces the following output.

```
0
1
2
3
4
```

13. Clojure - File I/O

Clojure provides a number of helper methods when working with I/O. It offers easier classes to provide the following functionalities for files.

- Reading files
- Writing to files
- Seeing whether a file is a file or directory

Let's explore some of the file operations Clojure has to offer.

Reading the Contents of a File as an Entire String

If you want to get the entire contents of the file as a string, you can use the **clojure.core.slurp** method. The slurp command opens a reader on a file and reads all its contents, returning a string.

Following is an example of how this can be done.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
      (def string1 (slurp "Example.txt"))
        (println string1)
)
(Example)
```

If the file contains the following lines, they will be printed as:

```
line : Example1
line : Example2
```

Reading the Contents of a File One Line at a Time

If you want to get the entire contents of the file as a string one line at a time, you can use the **clojure.java.io/reader** method. The clojure.java.io/reader class creates a reader buffer, which is used to read each line of the file.

Following is an example that shows how this can be done.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
         (with-open [rdr (clojure.java.io/reader "Example.txt")]
                (reduce conj [] (line-seq rdr)))

)
(Example)
```

If the file contains the following lines, they will be printed as:

```
line : Example1
line : Example2
```

The output will be shown as:

```
["line : Example1" "line : Example2"]
```

Writing 'to' Files

If you want to write 'to' files, you can use the **clojure.core.spit** command to spew entire strings into files. The spit command is the opposite of the slurp method. This method opens a file as a writer, writes content, then closes file.

Following is an example.

```
(ns clojure.examples.hello
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
           (spit "Example.txt"
               "This is
               a string"
        )
)
```

In the above example, if you see the contents of the Example.txt file , you will see the contents of "This is a string".

Writing 'to' Files One Line at a Time

If you want to write 'to' files one line at a time, you can use the **clojure.java.io.writer** class. The clojure.java.io.writer class is used to create a writer stream wherein bytes of data are fed into the stream and subsequently into the file.

Following is an example that shows how the spit command can be used.

```
(ns clojure.examples.hello

     (:gen-class))

     ;; This program displays Hello World

(defn Example []

(with-open [w (clojure.java.io/writer  "Example.txt" :append true)]

                (.write w (str "hello" "world"))

                )

        )

)(Example)
```

When the above code is executed, the line "hello world" will be present in the Example.txt file. The append:true option is to append data to the file. If this option is not specified, then the file will be overwritten whenever data is written to the file.

Checking to See If a File Exists

To check if a file exists, the **clojure.java.io.file** class can be used to check for the existence of a file. Following is an example that shows how this can be accomplished.

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

If the file Example.txt exists, the output will be true.

Reading from the Console

To read data from the console, the **read-line** statement can be used. Following is an example that shows how this can be used.

If you enter the (read-line) command in the REPL window, you will have the chance to enter some input in the console window.

```
user->(read-line)
Hello World
```

The above code will produce the following output.

```
“Hello World”
```

14. Clojure - Strings

A **String** literal is constructed in Clojure by enclosing the string text in quotations. Strings in Clojure need to be constructed using the double quotation marks such as “Hello World”.

Following is an example of the usage of strings in Clojure.

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

The above program produces the following output.

```
Hello World
This is a demo application
```

Basic String Operations

Clojure has a number of operations that can be performed on strings. Following are the operations.

str

The concatenation of strings can be done by the simple str function.

Following is the syntax.

```
str stringvar1 stringvar2 stringvarn
```

Parameters - You can enter any number of string parameters which need to be concatenated.

Return Value - The return value is a string.

Following is an example of the string concatenation in Clojure.

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

The above program will produce the following output.

```
HelloWorld
HelloWorldAgain
```

format

The formatting of strings can be done by the simple format function. The format function formats a string using **java.lang.String.format**.

Following is the syntax.

```
(format fmt args)
```

Parameters – 'fmt' is the formatting which needs to be applied. 'Args' is the parameter to which the formatting needs to be applied.

Return Value - The return value is a string.

Following is an example of the string formatting in Clojure.

```
(ns clojure.examples.hello
     (:gen-class))
(defn hello-world []
(println (format "Hello , %s" "World"))
(println (format "Pad with leading zeros %06d" 1234)))
(hello-world)
```

The above program produces the following output.

```
Hello , World
Pad with leading zeros 001234
```

count

To find the number of characters in a string, you can use the count function.

Following is the syntax.

```
(count stringvariable)
```

Parameters – Stringvariable, is the string in which the number of characters need to be determined.

Return Value - The number of characters in the string.

Following is an example of the string formatting in Clojure.

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

The above program produces the following output.

```
5
```

subs

Returns the substring of 's' beginning at start inclusive, and ending at end (defaults to length of string), exclusive.

Following is the syntax.

```
(subs s start end)
```

Parameters – 'S' is the input string. 'Start' is the index position where to start the substring from. 'End' is the index position where to end the substring.

Return Value - The substring.

Following is an example of the substrings in Clojure.

```
(ns clojure.examples.hello
     (:gen-class))
(defn hello-world []
(println (subs "HelloWorld" 2 5))
(println (subs "HelloWorld" 5 7)))
(hello-world)
```

The above program produces the following output.

```
llo
Wo
```

compare

Returns a negative number, zero, or a positive number when ‘x’ is logically 'less than', 'equal to', or 'greater than' ‘y’. It is similar to Java x.compareTo(y) except it also works for nil, and mpares numbers and collections in a type-independent manner.

Following is the syntax.

```
(compare x y)
```

Parameters - Where x and y are the 2 strings which need to be compared.

Return Value - Returns a negative number, zero, or a positive number when ‘x’ is logically 'less than', 'equal to', or 'greater than' ‘y’.

Following is an example of the string comparison in Clojure.

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

The above program produces the following output.

```
-32
0
```

lower-case

Converts string to all lower-case.

Following is the syntax.

```
(lower-case s)
```

Parameters - Where 's' is the string to be converted.

Return Value - The string in lowercase.

Following is an example of lower-case in Clojure.

```
(ns clojure.examples.hello
     (:gen-class))
(defn hello-world []
(println (clojure.string/lower-case "HelloWorld"))
(println (clojure.string/lower-case "HELLOWORLD")))
(hello-world)
```

The above program produces the following output.

```
helloworld
helloworld
```

upper-case

Converts string to all upper-case.

Following is the syntax.

```
(upper-case s)
```

Parameters - Where 's' is the string to be converted.

Return Value - The string in uppercase.

Following is an example of upper-case in Clojure.

```
(ns clojure.examples.hello
     (:gen-class))
(defn hello-world []
(println (clojure.string/upper-case "HelloWorld"))
(println (clojure.string/upper-case "helloworld")))
(hello-world)
```

The above program produces the following output.

```
HELLOWORLD
HELLOWORLD
```

join

Returns a string of all elements in collection, as returned by (seq collection), separated by an optional separator.

Following is the syntax.

```
(join sep col)
```

Parameters – 'sep' is the separator for each element in the collection. 'col' is the collection of elements.

Return Value - A joined string.

Following is an example of join in Clojure.

```
(ns clojure.examples.hello
     (:gen-class))
(defn hello-world []
(println (clojure.string/join ", " [1 2 3])))
(hello-world)
```

The above program produces the following output.

```
1 , 2  , 3
```

split

Splits string on a regular expression.

Following is the syntax.

```
(split str reg)
```

Parameters – ‘str’ is the string which needs to be split. ‘reg’ is the regular expression based on which the string split needs to happen.

Return Value - The split string.

Following is an example of split in Clojure.

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

The above program produces the following output.

```
[Hello World]
```

Note that in the above output, both the strings “Hello” and “World” are separate strings.

split-lines

Split strings is based on the escape characters \n or \r\n.

Following is the syntax.

```
(split-lines str)
```

Parameters – ‘str’ is the string which needs to be split.

Return Value - The split string.

Following is an example of split-lines in Clojure.

```
(ns clojure.examples.hello
     (:gen-class))
(defn hello-world []
(println (clojure.string/split-lines "Hello\nWorld")))
(hello-world)
```

The above program produces the following output.

```
[Hello World]
```

Note that in the above output, both the strings "Hello" and "World" are separate strings.

reverse

Reverses the characters in a string.

Following is the syntax.

```
(reverse str)
```

Parameters – 'str' is the string which needs to be reversed.

Return Value - The reversed string.

Following is an example of reverse in Clojure.

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

The above program produces the following output.

```
dlroW olleH
```

replace

Replaces all instance of a match in a string with the replacement string.

Following is the syntax.

```
(replace str match replacement)
```

Parameters – 'str' is the input string. 'match' is the pattern which will be used for the matching process. 'replacement' will be the string which will be replaced for each pattern match.

Return Value - The string which has the replaced value as per the pattern match.

Following is an example of replace in Clojure.

```
(ns clojure.examples.hello
     (:gen-class))
(defn hello-world []
(println (clojure.string/replace "The tutorial is about Groovy" #"Groovy"
"Clojure")))
(hello-world)
```

The above program produces the following output.

```
The tutorial is about clojure
```

trim

Removes whitespace from both ends of the string.

Following is the syntax.

```
(trim str)
```

Parameters – 'str' is the input string.

Return Value - The string which has the whitespaces removed.

Following is an example of trim in Clojure.

```
(ns clojure.examples.hello
     (:gen-class))
(defn hello-world []
(println (clojure.string/trim "    White spaces     ")))
(hello-world)
```

The above program produces the following output.

```
White spaces
```

triml

Removes whitespace from the left hand side of the string.

Following is the syntax.

```
 (triml str)
```

Parameters – 'str' is the input string.

Return Value - The string which has the whitespaces removed from the beginning of the string.

Following is an example of triml in Clojure.

```
(ns clojure.examples.hello
     (:gen-class))
(defn hello-world []
(println (clojure.string/triml "    White spaces     ")))
(hello-world)
```

The above program produces the following output.

```
White spaces
```

The above output will have the white spaces at the end of the string.

trimr

Removes whitespace from the right hand side of the string.

Following is the syntax.

```
(trimr str)
```

Parameters – 'str' is the input string.

Return Value - The string which has the white spaces removed from the end of the string.

Following is an example of trimr in Clojure.

```
(ns clojure.examples.hello
     (:gen-class))
(defn hello-world []
(println (clojure.string/trimr "    White spaces     ")))
(hello-world)
```

The above program produces the following output.

```
White spaces
```

The above output will have the white spaces at the beginning of the string.

15. Clojure - Lists

List is a structure used to store a collection of data items. In Clojure, the List implements the **ISeq** interface. Lists are created in Clojure by using the list function.

Following is an example of creating a list of numbers in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(println (list 1 2 3 4)))
(example)
```

The above code produces the following output.

```
(1 2 3 4)
```

Following is an example of creating a list of characters in Clojure.

```
(ns clojure.examples. example
     (:gen-class))
(defn example []
(println (list 'a 'b 'c 'd)))
(example)
```

The above code produces the following output.

```
(a b c d)
```

Following are the list methods available in Clojure.

list*

Creates a new list containing the items prepended to the rest, the last of which will be treated as a sequence.

Following is the syntax.

```
(list* listitems [lst])
```

Parameters – ‘listitems’ is the new list items which need to be appended. ‘lst’ is the list to which the items need to be appended to.

Return Value - The new list with the appended list items.

Following is an example of list* in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(println (list* 1 [2,3])))
(example)
```

The above program produces the following output.

```
(1 2 3)
```

first

This function returns the first item in the list.

Following is the syntax.

```
(first lst)
```

Parameters – ‘lst’ is the list of items.

Return Value - The first value from the list.

Following is an example of first in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(println (first (list 1 2,3))))
(example)
```

The above code produces the following output.

```
1
```

nth

This function returns the item in the 'nth' position in the list.

Following is the syntax.

```
(nth lst index)
```

Parameters – 'lst' is the list of items. 'index' is the index position of the element, which needs to be returned.

Return Value - The value at the index position from the list.

Following is an example of nth in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(println (nth (list 1 2,3) 0))
(println (nth (list 1 2,3) 2)))
(example)
```

The above code produces the following output.

```
1
3
```

cons

Returns a new list wherein an element is added to the beginning of the list.

Following is the syntax.

```
(cons element lst)
```

Parameters – 'element' is the element which needs to be added to the list. 'lst' is the list of items.

Return Value - The list with the appended value.

Following is an example of cons in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(println (cons 0 (list 1 2,3))))
(example)
```

The above code produces the following output.

```
(0 1 2 3)
```

conj

Returns a new list wherein the list is at the beginning and the elements to be appended are placed at the end.

Following is the syntax.

```
(conj lst elementlst)
```

Parameters – 'elementlst' is the list of items which needs to be added to the list. 'lst' is the list of items.

Return Value - The list with the appended values.

Following is an example of conj in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(println (conj (list 1 2,3) 4 5)))
(example)
```

The above code produces the following output.

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

rest

Returns the remaining items in the list after the first item.

Following is the syntax.

```
(rest lst)
```

Parameters – ‘lst’ is the list of items.

Return Value - A list of items with the first item removed.

Following is an example of rest in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(println (rest (list 1 2,3))))
(example)
```

The above code produces the following output.

```
(2 3)
```

16. Clojure - Sets

Sets in Clojure are a set of unique values. Sets are created in Clojure with the help of the set command.

Following is an example of the creation of sets in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(println (set '(1 1 2 2))))
(example)
```

The above code produces the following output.

```
{1,2}
```

Following are the methods available in Clojure for sets.

sorted-set

Returns a sorted set of elements.

Following is the syntax.

```
(sorted-set setofelements)
```

Parameters – 'setofelements' is the set of elements which need to be sorted.

Return Value - The sorted set of elements.

Following is an example of sorted-set in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(println (sorted-set 3 2 1)))
(example)
```

The above code produces the following output.

```
{1,2,3}
```

get

Returns the element at the index position.

Following is the syntax.

```
(get setofelements index)
```

Parameters – ‘setofelements’ is the set of elements. ‘index’ is the element at the index position, which needs to be returned.

Return Value - The value of the element at the index position.

Following is an example of get in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(println (get (set '(3 2 1)) 2))
(println (get (set '(3 2 1)) 1)))
(example)
```

The above code produces the following output.

```
2
1
```

contains?

Finds out whether the set contains a certain element or not.

Following is the syntax.

```
(contains? setofelements searchelement)
```

Parameters – ‘setofelements’ is the set of elements. ‘Searchelement’ is the element which needs to be searched for in the list.

Return Value - Returns true if the element exists in the set or false if it dosen't.

Following is an example of contains? in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(println (contains? (set '(3 2 1)) 2))
(println (contains? (set '(3 2 1)) 5)))
(example)
```

The above code produces the following output.

```
true
false
```

conj

Appends an element to the set and returns the new set of elements.

Following is the syntax.

```
(conj setofelements x)
```

Parameters – 'setofelements' is the set of elements. 'x' is the element which needs to be appended to the set of elements.

Return Value - Returns the new set with the appended element.

Following is an example of conj in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(println (conj (set '(3 2 1)) 5)))
(example)
```

The above code produces the following output.

```
{1 3 2 5}
```

disj

Disjoins an element from the set.

Following is the syntax.

```
(disj setofelements x)
```

Parameters – 'setofelements' is the set of elements. 'x' is the element which needs to be removed from the set.

Return Value - Returns the new set with the removed element.

Following is an example of disj in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(println (disj (set '(3 2 1)) 2)))
(example)
```

The above code produces the following output.

```
{1 3}
```

union

Disjoins an element from the set.

Following is the syntax.

```
(union set1 set2)
```

Parameters – 'set1' is the first set of elements. 'set2' is the second set of elements.

Return Value - The joined set of elements.

Following is an example of union in Clojure.

```
(ns clojure.examples.example
(:require [clojure.set :as set])
     (:gen-class))
(defn example []
(println (set/union #{1 2} #{3 4})))(example)
```

Note that in the above example, you have to use the require statement to include the clojure.set class which contains the union method.

The above code produces the following output.

```
{1 4 3 2}
```

difference

Return a set that is the first set without elements of the remaining sets.

Following is the syntax.

```
(difference set1 set2)
```

Parameters – 'set1' is the first set of elements. 'set2' is the second set of elements.

Return Value - The difference between the set of elements.

Following is an example of difference in Clojure.

```
(ns clojure.examples.example
(:require [clojure.set :as set])
     (:gen-class))
(defn example []
(println (set/difference #{1 2} #{2 3})))
(example)
```

The above code produces the following output.

```
{1}
```

intersection

Return a set that is the intersection of the input sets.

Following is the syntax.

```
(intersection set1 set2)
```

Parameters – 'set1' is the first set of elements. 'set2' is the second set of elements.

Return Value - The intersection between the different set of elements.

Following is an example of intersection in Clojure.

```
(ns clojure.examples.example
(:require [clojure.set :as set])
     (:gen-class))
(defn example []
(println (set/intersection #{1 2} #{2 3})))
(example)
```

The above code produces the following output.

```
{2}
```

subset?

Is set1 a subset of set2?

Following is the syntax.

```
(subset? set1 set2)
```

Parameters – ‘set1’ is the first set of elements. ‘set2’ is the second set of elements.

Return Value - True if set1 is a subset of set2, else false if not.

Following is an example of subset? in Clojure.

```
(ns clojure.examples.example
(:require [clojure.set :as set])
     (:gen-class))
(defn example []
(println (set/subset? #{1 2} #{2 3}))
(println (set/subset? #{1 2} #{1 2 3})))
(example)
```

The above code produces the following output.

```
false
true
```

superset?

Is set1 a superset of set2?

Following is the syntax.

```
(superset? set1 set2)
```

Parameters – 'set1' is the first set of elements. 'set2' is the second set of elements.

Return Value - True if set1 is a superset of set2, else false if not.

Following is an example of superset? in Clojure.

```
(ns clojure.examples.example
(:require [clojure.set :as set])
     (:gen-class))
(defn example []
(println (set/superset? #{1 2} #{1 2 3}))
(println (set/superset? #{1 2 3} #{1 2})))
(example)
```

The above code produces the following output.

```
false
true
```

17. Clojure - Vectors

A **Vector** is a collection of values indexed by contiguous integers. A vector is created by using the vector method in Clojure.

Following is an example of creating a vector in Clojure.

```
(ns clojure.examples.example
(:require [clojure.set :as set])
     (:gen-class))
(defn example []
(println (vector 1 2 3)))
(example)
```

The above code produces the following output.

```
[1 2 3]
```

Following are the methods available in Clojure.

vector-of

Creates a new vector of a single primitive type 't', where 't' is one of :int :long :float :double :byte :short :char or :boolean. The resulting vector complies with the interface of vectors in general, but stores the values unboxed internally.

Following is the syntax.

```
(vector-of t setofelements)
```

Parameters – 't' is the type which the vector elements should be. 'Setofelements' is the set of elements comprised in the vector.

Return Value - The vector set of elements of the required type.

Following is an example of vector-of in Clojure.

```
(ns clojure.examples.example
(:require [clojure.set :as set])
     (:gen-class))
```

```
(defn example []
(println (vector-of :int 1 2 3)))
(example)
```

The above code produces the following output.

```
[1 2 3]
```

nth

This function returns the item in the nth position in the vector.

Following is the syntax.

```
(nth vec index)
```

Parameters – 'vec' is the vector of items. 'index' is the index position of the element which needs to be returned.

Return Value - The value at the index position from the vector.

Following is an example of nth in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(println (nth (vector 1 2,3) 0))
(println (nth (vector 1 2,3) 2)))
(example)
```

The above code produces the following output.

```
1
3
```

get

Returns the element at the index position in the vector.

Following is the syntax.

```
(get vec index)
```

Parameters – 'vec' is the set of elements in the vector. 'index' is the element at the index position which needs to be returned.

Return Value - The value of the element at the index position.

Following is an example of get in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(println (get (vector 3 2 1) 2))
(println (get (vector 3 2 1) 1)))
(example)
```

The above code produces the following output.

```
1
2
```

conj

Appends an element to the vector and returns the new set of vector elements.

Following is the syntax.

```
(conj vec x)
```

Parameters – 'vec' is the vector set of elements. 'x' is the element which needs to be appended to the set of elements in the vector.

Return Value - Returns the new vector with the appended element.

Following is an example of conj in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(println (conj (vector 3 2 1) 5)))
(example)
```

The above code produces the following output.

```
[3 2 1 5]
```

pop

For a list or queue, returns a new list/queue without the first item, for a vector, returns a new vector without the last item.

Following is the syntax.

```
(pop vec)
```

Parameters – ‘vec’ is the vector set of elements.

Return Value - Returns the new vector without the last item.

Following is an example of pop in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(println (pop (vector 3 2 1))))
(example)
```

The above code produces the following output.

```
[3 2]
```

subvec

Returns a sub vector from a starting and ending index.

Following is the syntax.

```
(subvec vec start end)
```

Parameters – ‘vec’ is the vector set of elements. ‘start’ is the starting index. ‘end’ is the ending index.

Return Value - Returns the new vector from the starting to the ending index.

Following is an example of subvec in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(println (subvec (vector 1 2 3 4 5 6 7) 2 5)))
(example)
```

The above code produces the following output.

```
[3 4 5]
```

18. Clojure - Maps

A **Map** is a collection that maps keys to values. Two different map types are provided - hashed and sorted. **HashMaps** require keys that correctly support hashCode and equals. **SortedMaps** require keys that implement Comparable, or an instance of Comparator.

A map can be created in two ways, the first is via the hash-map method.

Creation - HashMaps

HashMaps have a typical key value relationship and is created by using hash-map function.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(def demokeys (hash-map "z" "1" "b" "2" "a" "3"))
(println demokeys))
(example)
```

The above code produces the following output.

```
{z 1, b 2, a 3}
```

Creation - SortedMaps

SortedMaps have the unique characteristic of sorting their elements based on the key element. Following is an example that shows how the sorted map can be created using the sorted-map function.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(def demokeys (sorted-map "z" "1" "b" "2" "a" "3"))
(println demokeys))
(example)
```

The above code produces the following output.

```
{a 3, b 2, z 1}
```

From the above program you can clearly see that elements in the maps are sorted as per the key value. Following are the methods available for maps.

get

Returns the value mapped to key, not-found or nil if key is not present.

Following is the syntax.

```
(get hmap key)
```

Parameters – 'hmap' is the map of hash keys and values. 'key' is the key for which the value needs to be returned.

Return Value - Returns the value of the key passed to the get function.

Following is an example of get in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(def demokeys (hash-map "z" "1" "b" "2" "a" "3"))
(println demokeys)
(println (get demokeys "b")))
(example)
```

The above code produces the following output.

```
{z 1, b 2, a 3}
2
```

contains?

See whether the map contains a required key.

Following is the syntax.

```
(contains hmap key)
```

Parameters – 'hmap' is the map of hash keys and values. 'key' is the key which needs to be searched in the map.

Return Value - Returns the value of true if the key is present, else returns false.

Following is an example of contains? in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(def demokeys (hash-map "z" "1" "b" "2" "a" "3"))
(println (contains? demokeys "b"))
(println (contains? demokeys "x")))
(example)
```

The above code produces the following output.

```
true
false
```

find

Returns the map entry for the key.

Following is the syntax.

```
(find hmap key)
```

Parameters – 'hmap' is the map of hash keys and values. 'key' is the key which needs to be searched in the map.

Return Value - Returns the key value pair for the desired key, else returns nil.

Following is an example of find in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(def demokeys (hash-map "z" "1" "b" "2" "a" "3"))
(println demokeys)
(println (find demokeys "b"))
(println (find demokeys "x")))
(example)
```

The above program produces the following output.

```
[b 2]
nil
```

keys

Returns the list of keys in the map.

Following is the syntax.

```
(keys hmap)
```

Parameters – 'hmap' is the map of hash keys and values.

Return Value - Returns the list of keys in the map.

Following is an example of keys in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(def demokeys (hash-map "z" "1" "b" "2" "a" "3"))
(println (keys demokeys)))
(example)
```

The above program produces the following output.

```
(z a b)
```

vals

Returns the list of values in the map.

Following is the syntax.

```
(vals hmap)
```

Parameters – ‘hmap’ is the map of hash keys and values.

Return Value - Returns the list of values in the map.

Following is an example of vals in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(def demokeys (hash-map "z" "1" "b" "2" "a" "3"))
(println (vals demokeys)))
(example)
```

The above program produces the following output.

```
(1 3 2)
```

dissoc

Dissociates a key value entry from the map.

Following is the syntax.

```
(dissoc hmap key)
```

Parameters – ‘hmap’ is the map of hash keys and values. ‘key’ is the key which needs to be dissociated from the HashMap.

Return Value - Returns a map with the dissociated key.

Following is an example of dissoc in Clojure.

```
 (ns clojure.examples.example
     (:gen-class))
(defn example []
(def demokeys (hash-map "z" "1" "b" "2" "a" "3"))
(println (dissoc demokeys "b")))
(example)
```

The above code produces the following output.

```
{z 1, a 3}
```

merge

Merges two maps entries into one single map entry.

Following is the syntax.

```
(merge hmap1 hmap2)
```

Parameters – 'hmap1' is the map of hash keys and values. 'hmap2' is the map of hash keys and values, which needs to be mapped with the first HashMap.

Return Value - Returns a combined HashMap of both hasmap1 and hasmap2.

Following is an example of merge in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(def demokeys (hash-map "z" "1" "b" "2" "a" "3"))
(def demokeys1 (hash-map "x" "4" "h" "5" "i" "7"))
(println (merge demokeys demokeys1)))
(example)
```

The above program produces the following output.

```
{z 1, x 4, a 3, i 7, b 2, h 5}
```

merge-with

Returns a map that consists of the rest of the maps conj-ed onto the first. If a key occurs in more than one map, the mapping(s) from the latter (left-to-right) will be combined with the mapping in the result.

Following is the syntax.

```
(merge-with f hmap1 hmap2)
```

Parameters – 'f' is the operator which needs to be applied to the hash maps. 'hmap1' is the map of hash keys and values. 'hmap2' is the map of hash keys and values, which needs to be mapped with the first HashMap.

Return Value - Returns a map that consists of the rest of the maps conj-ed onto the first.

Following is an example of merge-with in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(def demokeys (hash-map "z" 1 "b" 2 "a" 3))
(def demokeys1 (hash-map "a" 2 "h" 5 "i" 7))
(println (merge-with + demokeys demokeys1)))
(example)
```

The above program produces the following output.

```
{z 1, a 5, i 7, b 2, h 5}
```

Notice that in the output since the key 'a' occurs twice, the value is added from both HashMaps as per the operator +.

select-keys

Returns a map containing only those entries in map whose key is in keys.

Following is the syntax.

```
(select-keys hmap keys)
```

Parameters – 'hmap' is the map of hash keys and values. 'keys' is the list of keys which need to be selected from the HashMap.

Return Value - Returns the keys from the map as per the select clause of keys.

Following is an example of select-keys in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(def demokeys (hash-map "z" 1 "b" 2 "a" 3))
(println (select-keys demokeys ["z" "a"])))
(example)
```

The above program produces the following output.

```
{z 1, a 3}
```

rename-keys

Renames keys in the current HashMap to the newly defined ones.

Following is the syntax.

```
(rename-keys hmap keys)
```

Parameters – 'hmap' is the map of hash keys and values. 'keys' is the new list of keys which need to be replaced in the map.

Return Value - Returns a map with a new list of keys.

Following is an example of rename-keys in Clojure.

```
(ns clojure.examples.example
(:require [clojure.set :as set])
     (:gen-class))
(defn example []
(def demokeys (hash-map "z" 1 "b" 2 "a" 3))
(def demonew (set/rename-keys demokeys {"z" "newz" "b" "newb" "a" "newa"}))
(println demonew))
(example)
```

The above program produces the following output.

```
{newa 3, newb 2, newz 1}
```

map-invert

Inverts the maps so that the values become the keys and vice versa.

Following is the syntax.

```
(map-invert hmap)
```

Parameters – ‘hmap’ is the map of hash keys and values.

Return Value - Returns a map with the values inverted to the keys and the keys inverted to values.

Following is an example of map-invert in Clojure.

```
(ns clojure.examples.example
(:require [clojure.set :as set])
     (:gen-class))
(defn example []
(def demokeys (hash-map "z" 1 "b" 2 "a" 3))
(def demonew (set/map-invert demokeys))
(println demonew))
(example)
```

The above program produces the following output.

```
{1 z, 3 a, 2 b}
```

19. Clojure - Namespaces

Namespaces in Clojure are used to differentiate classes into separate logical spaces just like in Java. Consider the following statement.

```
(:require [clojure.set :as set])
```

In the above statement, 'clojure.set' is a namespace which contains various classes and methods to be used in the program. For example, the above namespace contains the function called map-invert, which is used to invert a map of key-values. We cannot use this function unless we explicitly tell our program to include this namespace.

Let's look at the different methods available for namespaces.

ns

This is used to look at your current namespace.

Following is the syntax.

```
(*ns*)
```

Parameters – None.

Return Value - Returns the namespace of the currently executing program.

Following is an example of namespace in Clojure.

```
(ns clojure.examples.example
(:require [clojure.set :as set])
     (:gen-class))
(defn example []
(println *ns*))
(example)
```

The above program produces the following output.

```
#object[clojure.lang.Namespace 0x50ad3bc1 clojure.examples.hello]
```

As you can see the output of the above program displays the namespace as clojure.examples.hello which is the name of the current namespace.

ns

This is used to create a new namespace and associate it with the running program.

Following is the syntax.

```
(ns namespace-name)
```

Parameters – ‘namespace-name’ is the namespace which needs to be associated with the running program.

Return Value – None.

Following is an example of ns in Clojure.

```
(ns clojure.myown
(:require [clojure.set :as set])
     (:gen-class))
(defn hello-world []
(println *ns*))
(hello-world)
```

The above program produces the following output.

```
#object[clojure.lang.Namespace 0x50ad3bc1 clojure.myown]
```

alias

Add an alias in the current namespace to another namespace. Arguments are two symbols: the alias to be used and the symbolic name of the target namespace.

Following is the syntax.

```
(alias aliasname namespace-name)
```

Parameters – ‘aliasname’ is the alias name which needs to be associated with the namespace. ‘namespace-name’ is the namespace which is associated with the running program.

Return Value – None.

Following is an example of alias in Clojure.

```
(ns clojure.examples.example
(:require [clojure.set :as set])
     (:gen-class))
(defn example []
(alias 'string 'clojure.examples.hello)
(example)
```

all-ns

Returns a list of all namespaces.

Following is the syntax.

```
(all-ns)
```

Parameters – None.

Return Value - The list of all namespaces.

Following is an example of all-ns in Clojure.

```
(ns clojure.examples.example
(:require [clojure.set :as set])
     (:gen-class))
(defn example []
(println (all-ns)))
(example)
```

The above program produces the following output.

```
(#object[clojure.lang.Namespace 0x6bedbc4d clojure.core]
#object[clojure.lang.Namespace 0x932bc4a clojure.uuid]
#object[clojure.lang.Namespace 0xd29f28 clojure.set] #object[clo

jure.lang.Namespace 0x2fd1433e clojure.examples.hello]
#object[clojure.lang.Namespace 0x29d89d5d clojure.java.io]
#object[clojure.lang.Namespace 0x3514a4c0 clojure.main] #objec

t[clojure.lang.Namespace 0x212b5695 user] #object[clojure.lang.Namespace
0x446293d clojure.core.protocols] #object[clojure.lang.Namespace 0x69997e9d
clojure.instant] #object[cl

ojure.lang.Namespace 0x793be5ca clojure.string])
```

Basically, all of the namespaces available in Clojure will be returned.

find-ns

Finds and returns a particular namespace.

Following is the syntax.

```
(find-ns namespace-name)
```

Parameters – 'namespace-name' is the namespace which needs to be found.

Return Value - The namespace, if it exists.

Following is an example of find-ns in Clojure.

```
(ns clojure.examples.example
(:require [clojure.set :as set])
     (:gen-class))
(defn example []
(println (find-ns 'clojure.string)))
(example)
```

The above program produces the following output.

```
#object[clojure.lang.Namespace 0x6d3a388c clojure.string]
```

ns-name

Returns the name of a particular namespace.

Following is the syntax.

```
(ns-name namespace-name)
```

Parameters – 'namespace-name' is the namespace which needs to be found.

Return Value - The actual name of the namespace.

Following is an example of ns-name in Clojure.

```
(ns clojure.examples.example
(:require [clojure.set :as set])
     (:gen-class))
```

```
(defn example []
(println (ns-name 'clojure.string)))
(example)
```

The above program produces the following output.

```
clojure.string
```

ns-aliases

Returns the aliases, which are associated with any namespaces.

Following is the syntax.

```
(ns-aliases namespace-name)
```

Parameters – 'namespace-name' is the namespace which needs to be found.

Return Value - Returns the aliases which are associated with any namespaces.

Following is an example of ns-aliases in Clojure.

```
(ns clojure.examples.example
(:require [clojure.set :as set])
     (:gen-class))
(defn example []
(println (ns-aliases 'clojure.core)))
(example)
```

The above program produces the following output.

```
{jio #object[clojure.lang.Namespace 0x2fd1433e clojure.java.io]}
```

ns-map

Returns a map of all the mappings for the namespace.

Following is the syntax.

```
(ns-map namespace-name)
```

Parameters – ‘namespace-name’ is the namespace which needs to be found.

Return Value - Returns a map of all the mappings for the namespace.

Following is an example of ns-map in Clojure.

```
(ns clojure.examples.example
(:require [clojure.set :as set])
     (:gen-class))
(defn example []
(println (count (ns-map 'clojure.core))))
(example)
```

The above program produces the following output.

```
848
```

The count method is used to provide the total count of maps returned.

un-alias

Removes the alias for the symbol from the namespace.

Following is the syntax.

```
(un-alias namespace-name aliasname)
```

Parameters – 'namespace-name' is the namespace which is attached to an alias.

'aliasname' is the alias name which has been mapped to the namespace.

Return Value - Removes the alias from the namespace.

Following is an example of un-alias in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(alias 'string 'clojure.core)
(ns-unalias 'clojure.core 'string))
(example)
```

20. Clojure - Exception Handling

Exception handling is required in any programming language to handle the runtime errors so that the normal flow of the application can be maintained. Exception usually disrupts the normal flow of the application, which is the reason why we need to use exception handling in our application.

Exception is broadly classified into the following categories:

- **Checked Exception** - The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions. E.g. IOException, SQLException, etc. Checked exceptions are checked at compile-time.

Let’s consider the following program which does an operation on a file called Example.txt. However, there can be always a case wherein the file Example.txt does not exist.

```
(ns clojure.examples.example
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
      (def string1 (slurp "Example.txt"))
        (println string1)
)
(Example)
```

If the file Example.txt does not exist, then the following exception will be generated by the program.

```
Caused by: java.io.FileNotFoundException: Example.txt (No such file or
directory)
        at java.io.FileInputStream.open0(Native Method)
        at java.io.FileInputStream.open(FileInputStream.java:195)
        at java.io.FileInputStream.<init>(FileInputStream.java:138)
        at clojure.java.io$fn__9185.invoke(io.clj:229)
        at clojure.java.io$fn__9098$G__9091__9105.invoke(io.clj:69)
        at clojure.java.io$fn__9197.invoke(io.clj:258)
        at clojure.java.io$fn__9098$G__9091__9105.invoke(io.clj:69)
```

From the above exception, we can clearly see that the program raised a FileNotFoundException.

- **Unchecked Exception** - The classes that extend RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

One classical case is the ArrayIndexOutOfBoundsException which happens when you try to access an index of an array which is greater than the length of the array. Following is a typical example of this sort of mistake.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
(try
      (aget (int-array [1 2 3]) 5)
         (catch Exception e (println (str "caught exception: " (.toString e))))
         (finally (println "This is our final block")))
         (println "Let's move on")
)
(Example)
```

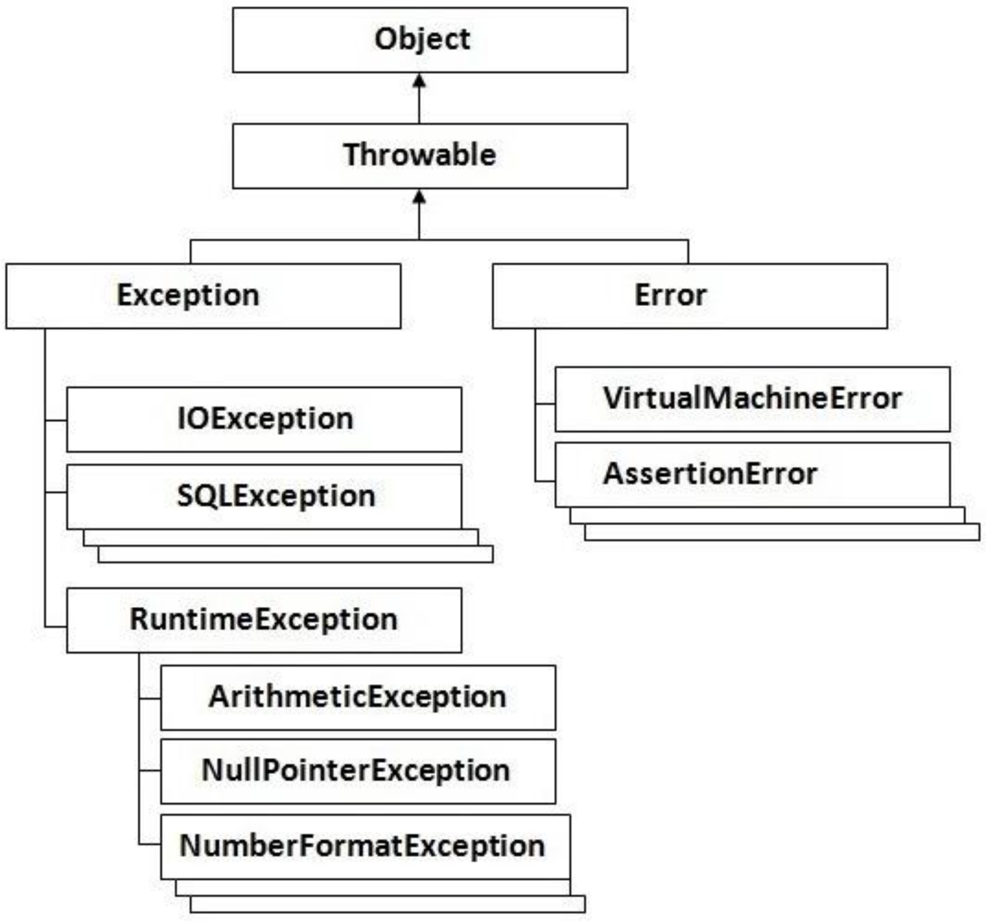
When the above code is executed, the following exception will be raised.

```
caught exception: java.lang.ArrayIndexOutOfBoundsException: 5
This is our final block
Let's move on
```

Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError, etc. These are errors which the program can never recover from and will cause the program to crash. We now need some mechanism to catch these exceptions so that the program can continue to run if these exceptions exist.

The following diagram shows how the hierarchy of exceptions in Clojure is organized. It's all based on the hierarchy defined in Java.



Catching Exceptions

Just like other programming languages, Clojure provides the normal 'try-catch' block to catch exceptions as and when they occur.

Following is the general syntax of the try-catch block.

```
(try
(
   //Protected code
)
catch Exception e1)
(
   //Catch block
) )
```

All of your code which could raise an exception is placed in the **Protected code block**.

In the **catch block**, you can write custom code to handle your exception so that the application can recover from the exception.

Let’s look at our earlier example which generated a file-not-found exception and see how we can use the try catch block to catch the exception raised by the program.

```
(ns clojure.examples.example
     (:gen-class))
 (defn Example []
(try
      (def string1 (slurp "Example.txt"))
        (println string1)
         (catch Exception e (println (str "caught exception: " (.getMessage
e)))))
)
(Example)
```

The above program produces the following output.

```
caught exception: Example.txt (No such file or directory)
```

From the above code, we wrap out faulty code in the **try block**. In the catch block, we are just catching our exception and outputting a message that an exception has occurred. So, we now have a meaningful way of capturing the exception, which is generated by the program.

Multiple Catch Blocks

One can have multiple catch blocks to handle multiple types of exceptions. For each catch block, depending on the type of exception raised you would write code to handle it accordingly.

Let’s modify our earlier code to include two catch blocks, one which is specific for our file not found exception and the other is for a general exception block.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
(try
      (def string1 (slurp "Example.txt"))
```

```
        (println string1)
         (catch java.io.FileNotFoundException e (println (str "caught file
exception: " (.getMessage e))))
         (catch Exception e (println (str "caught exception: " (.getMessage
e)))))
         (println "Let's move on")
)
(Example)
```

The above program produces the following output.

```
caught file exception: Example.txt (No such file or directory)
Let's move on
```

From the above output, we can clearly see that our exception was caught by the 'FileNotFoundException' catch block and not the general one.

Finally Block

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code. Following is the syntax for this block.

```
(try
(
   //Protected code
)
catch Exception e1)
(
   //Catch block
)
(finally
//Cleanup code
)
)
```

Let’s modify the above code and add the finally block of code. Following is the code snippet.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
(try
      (def string1 (slurp "Example.txt"))
        (println string1)
         (catch java.io.FileNotFoundException e (println (str "caught file
exception: " (.getMessage e))))
         (catch Exception e (println (str "caught exception: " (.getMessage
e))))
         (finally (println "This is our final block")))
         (println "Let's move on")
)
(Example)
```

The above program produces the following output.

```
caught file exception: Example.txt (No such file or directory)
This is our final block
Let's move on
```

From the above program, you can see that the final block is also implemented after the catch block catches the required exception.

Since Clojure derives its exception handling from Java, similar to Java, the following methods are available in Clojure for managing the exceptions.

- **public String getMessage() -** Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
- **public Throwable getCause()** - Returns the cause of the exception as represented by a Throwable object.
- **public String toString()** - Returns the name of the class concatenated with the result of getMessage().
- **public void printStackTrace()** - Prints the result of toString() along with the stack trace to System.err, the error output stream.
- **public StackTraceElement [] getStackTrace()** - Returns an array containing each element on the stack trace. The element at index 0 represents the top of the

call stack, and the last element in the array represents the method at the bottom of the call stack.

- **public Throwable fillInStackTrace()** - Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

Following is the example code that uses some of the methods listed above.

```
(ns clojure.examples.example
     (:gen-class))
      (defn Example []
(try
      (def string1 (slurp "Example.txt"))
        (println string1)
         (catch java.io.FileNotFoundException e (println (str "caught file
exception: " (.toString e))))
         (catch Exception e (println (str "caught exception: " (.toString e))))
         (finally (println "This is our final block")))
         (println "Let's move on")
)
(Example)
```

The above program produces the following output.

```
caught file exception: java.io.FileNotFoundException: Example.txt (No such file
or directory)
This is our final block
Let's move on
```

21. Clojure - Sequences

Sequences are created with the help of the '**seq'** command. Following is a simple example of a sequence creation.

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

The above program produces the following output.

```
(1 2 3)
```

Following are the various methods available for sequences.

cons

Returns a new sequence where 'x' is the first element and 'seq' is the rest.

Following is the syntax.

```
(cons x seq)
```

Parameters – 'x' is the element which needs to be added to the sequence. 'seq' is the sequence list of elements.

Return Value - The new sequence with the appended element.

Following is an example of con in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(println (cons 0 (seq [1 2 3] ) ))) (Example)
```

The above program produces the following output.

```
(0 1 2 3)
```

conj

Returns a new sequence where 'x' is the element that is added to the end of the sequence.

Following is the syntax.

```
(conj seq x)
```

Parameters – 'x' is the element which needs to be added to the sequence. 'seq' is the sequence list of elements.

Return Value - The new sequence with the appended element.

Following is an example of conj in Clojure. Note that in the following program, we are also seeing the shorter version of creating a sequence, which can simply be done by using the square brackets [].

```
(ns clojure.examples.example
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(println (conj [1 2 3] 4)))
(Example)
```

The above program produces the following output.

```
(1 2 3 4)
```

concat

This is used to concat two sequences together.

Following is the syntax.

```
 (concat seq1 seq2)
```

Parameters – 'seq1' is the first sequence list of elements. 'seq2' is the second sequence list of elements, which needs to be appended to the first.

Return Value - The combined sequence of elements.

Following is an example of concat in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def seq1 (seq [1 2]))
(def seq2 (seq [3 4]))
(println (concat seq1 seq2)))
(Example)
```

The above program produces the following output.

```
(1 2 3 4)
```

distinct

Used to only ensure that distinct elements are added to the sequence.

Following is the syntax.

```
(distinct seq1)
```

Parameters – ‘seq1’ is the sequence list of elements.

Return Value - The sequence of elements wherein only distinct elements are returned.

Following is an example of distinct in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def seq1 (distinct (seq [1 1 2 2])))
(println seq1))
(Example)
```

The above program produces the following output.

```
(1 2)
```

reverse

Reverses the elements in the sequence.

Following is the syntax.

```
(reverse seq1)
```

Parameters – 'seq1' is the sequence list of elements.

Return Value - The reverse sequence of elements is returned.

Following is an example of reverse in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def seq1 (reverse (seq [1 2 3])))
(println seq1))
(Example)
```

The above program produces the following output.

```
(3 2 1)
```

first

Returns the first element of the sequence.

Following is the syntax.

```
(first seq1)
```

Parameters – 'seq1' is the sequence list of elements.

Return Value - The first element of the sequence is returned.

Following is an example of first in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def seq1 (seq [1 2 3]))
(println (first seq1)))
(Example)
```

The above program produces the following output.

```
1
```

last

Returns the last element of the sequence.

Following is the syntax.

```
(last seq1)
```

Parameters – 'seq1' is the sequence list of elements.

Return Value - The last element of the sequence is returned.

Following is an example of last in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def seq1 (seq [1 2 3]))
(println (last seq1)))
(Example)
```

The above program produces the following output.

```
3
```

rest

Returns the entire sequence except for the first element

Following is the syntax.

```
(rest seq1)
```

Parameters – 'seq1' is the sequence list of elements.

Return Value - Returns the entire sequence except for the first element.

Following is an example of last in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def seq1 (seq [1 2 3 4 5]))
(println (rest seq1)))
(Example)
```

The above program produces the following output.

```
(2 3 4 5)
```

sort

Returns a sorted sequence of elements.

Following is the syntax.

```
(sort seq1)
```

Parameters – 'seq1' is the sequence list of elements.

Return Value - Returns a sorted sequence of elements.

Following is an example of sort.

```
(ns clojure.examples.example
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def seq1 (seq [5 4 3 2 1]))
(def seq2 (sort seq1))
(println seq2))
(Example)
```

The above program produces the following output.

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

drop

Drops elements from a sequence based on the number of elements, which needs to be removed.

Following is the syntax.

```
(drop num seq1)
```

Parameters – 'seq1' is the sequence list of elements. 'num' is the number of elements which need to be dropped.

Return Value - Returns the sequence of elements with the required elements dropped from the sequence.

Following is an example of drop in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def seq1 (seq [5 4 3 2 1]))
(def seq2 (drop 2 seq1))
(println seq2))
(Example)
```

The above program produces the following output.

```
(3 2 1)
```

take-last

Takes the last list of elements from the sequence.

Following is the syntax.

```
(take-last num seq1)
```

Parameters – 'seq1' is the sequence list of elements. 'num' is the number of elements which needs to be included in the sequence from the end.

Return Value - A new sequence of elements with only the end number of elements included.

Following is an example of take-last in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def seq1 (seq [5 4 3 2 1]))
(def seq2 (take-last 2 seq1))
(println seq2))
(Example)
```

The above program produces the following output.

```
(2 1)
```

take

Takes the first list of elements from the sequence.

Following is the syntax.

```
(take num seq1)
```

Parameters – 'seq1' is the sequence list of elements. 'num' is the number of elements which needs to be included in the sequence from the beginning.

Return Value - A new sequence of elements with only the beginning number of elements included.

Following is an example of take in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def seq1 (seq [5 4 3 2 1]))
(def seq2 (take 2 seq1))
(println seq2))
(Example)
```

The above program produces the following output.

```
(5 4)
```

split-at

Splits the sequence of items into two parts. A location is specified at which the split should happen.

Following is the syntax.

```
(split-at num seq1)
```

Parameters – 'seq1' is the sequence list of elements. 'num' is the index position at which the split should happen.

Return Value – Two sequence of elements which are split based on the location where the split should happen.

Following is an example of split-at in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def seq1 (seq [5 4 3 2 1]))
(println (split-at 2 seq1)))
(Example)
```

The above program produces the following output.

```
[(5 4) (3 2 1)]
```

22. Clojure - Regular Expressions

A **regular expression** is a pattern that is used to find substrings in text. Regular expressions are used in a variety of programming languages and used a lot in LISP type programming languages.

Following is an example of a regular expression.

```
//d+
```

The above regular expression is used to find one more occurrence of a digit in a string. The // characters are used to ensure that characters 'd' and '+' are used to represent a regular expression.

In general, regular expressions works with the following set of rules.

- There are two special positional characters that are used to denote the beginning and end of a line: caret (∧) and dollar sign ($):

- Regular expressions can also include quantifiers. The plus sign (+) represents one or more times, applied to the preceding element of the expression. The asterisk (*) is used to represent zero or more occurrences. The question mark (?) denotes zero or once.

- The metacharacter { and } is used to match a specific number of instances of the preceding character.

- In a regular expression, the period symbol (.) can represent any character. This is described as the wildcard character.

- A regular expression may include character classes. A set of characters can be given as a simple sequence of characters enclosed in the metacharacters [and] as in [aeiou]. For letter or number ranges, you can use a dash separator as in [a–z] or [a–mA–M]. The complement of a character class is denoted by a leading caret within the square brackets as in [∧a–z] and represents all characters other than those specified.

The following methods are available for regular expressions.

re-pattern

Returns an instance of java.util.regex.Pattern. This is then used in further methods for pattern matching.

Following is the syntax.

```
 (re-pattern pat)
```

Parameters – 'pat' is the pattern which needs to be formed.

Return Value - A pattern object of the type java.util.regex.Pattern.

Following is an example of re-pattern in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def pat (re-pattern "\\d+")))
(Example)
```

The above program will create a pattern object, which will be a pattern of one or more digits.

re-find

Returns the next regex match, if any, of string to pattern, using java.util.regex.Matcher.find()

Following is the syntax.

```
(re-find pat str)
```

Parameters – 'pat' is the pattern which needs to be formed. 'str' is the string in which text needs to be found based on the pattern.

Return Value - A string if a match is found based on the input string and pattern.

Following is an example of re-find in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def pat (re-pattern "\\d+"))
(println (re-find pat "abc123de")))
(Example)
```

The above program produces the following output.

```
123
```

replace

The replace function is used to replace a substring in a string with a new string value. The search for the substring is done with the use of a pattern.

Following is the syntax.

```
(replace str pat replacestr)
```

Parameters – 'pat' is the regex pattern. 'str' is the string in which a text needs to be found based on the pattern. 'replacestr' is the string which needs to be replaced in the original string based on the pattern.

Return Value - The new string in which the replacement of the substring is done via the regex pattern.

Following is an example of replace in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def pat (re-pattern "\\d+"))
(def newstr (clojure.string/replace "abc123de" pat "789"))
(println newstr))
(Example)
```

The above program produces the following output.

```
abc789de
```

replace-first

The replace function is used to replace a substring in a string with a new string value, but only for the first occurrence of the substring. The search for the substring is done with the use of a pattern.

Following is the syntax.

```
(replace-first str pat replacestr)
```

Parameters – 'pat' is the regex pattern. 'str' is the string in which a text needs to be found based on the pattern. 'replacestr' is the string which needs to be replaced in the original string based on the pattern.

Return Value - The new string in which the replacement of the substring is done via the regex pattern, but only with the first occurrence.

Following is an example of replace-first in Clojure.

```
 (ns clojure.examples.example
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def pat (re-pattern "\\d+"))
(def newstr1 (clojure.string/replace "abc123de123" pat "789"))
(def newstr2 (clojure.string/replace-first "abc123de123" pat "789"))
(println newstr1)
(println newstr2))
(Example)
```

The above example shows the difference between the replace and replace-first function.

The above program produces the following output.

```
abc789de789
abc789de123
```

23. Clojure - Predicates

Predicates are functions that evaluate a condition and provide a value of either true or false. We have seen predicate functions in the examples of the chapter on numbers. We have seen functions like 'even?' which is used to test if a number is even or not, or 'neg?' which is used to test if a number is greater than zero or not. All of these functions return either a true or false value.

Following is an example of predicates in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
     ;; This program displays Hello World
(defn Example []
(def x (even? 0))
(println x)
(def x (neg? 2))
(println x)
(def x (odd? 3))
(println x)
(def x (pos? 3))
(println x))
(Example)
```

The above program produces the following output.

```
true
false
true
true
```

In addition to the normal predicate functions, Clojure provides more functions for predicates. The following methods are available for predicates.

every-pred

Takes a set of predicates and returns a function '**f**' that returns true if all of its composing predicates return a logical true value against all of its arguments, else it returns false.

Following is the syntax.

```
(every-pred p1 p2 .. pn)
```

Parameters – 'p1 p2...pn' is the list of all predicates which need to be tested.

Return Value - Returns true if all of its composing predicates return a logical true value against all of its arguments, else it returns false.

Following is an example of every-pred in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
(println ((every-pred number? even?) 2 4 6))
(println ((every-pred number? odd?) 2 4 6)))
(Example)
```

The above program produces the following output.

```
true
false
```

every?

Returns true if the predicate is true for every value, else false.

Following is the syntax.

```
(every? p1 col)
```

Parameters – 'p1' is the predicate which need to be tested. 'col' is the collection of values which needs to be tested.

Return Value - Returns true if the predicate is true for every value, else false.

Following is an example of every? in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
(println (every? even? '(2 4 6)))
(println (every? odd? '(2 4 6))))
(Example)
```

The above program produces the following output.

```
true
false
```

some

Returns the first logical true value for any predicate value of x in the collection of values.

Following is the syntax.

```
(some p1 col)
```

Parameters – 'p1' is the predicate which needs to be tested. 'col' is the collection of values which needs to be tested.

Return Value - Returns true if the predicate is true for every value, else false.

Following is an example of some in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
(println (some even? '(1 2 3 4))))
(Example)
```

The above program produces the following output.

```
true
```

Note that in the above program, once the predicate reaches the value 2, which is even, the function will exit and the values of 3 and 4 will not be tested.

not-any?

Returns false if any of the predicates of the values in a collection are logically true, else true.

Following is the syntax.

```
(not-any? p1 col)
```

Parameters – 'p1'is the predicate which needs to be tested. 'col' is the collection of values which needs to be tested.

Return Value - Returns false if any of the predicates of the values in a collection are logically true, else true.

Following is an example of not-any? in Clojure.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
(println (not-any? even? '(2 4 6))))
(Example)
```

The above program produces the following output.

```
false
```

24. Clojure - Destructuring

Destructuring is a functionality within Clojure, which allows one to extract values from a data structure, such as a vector and bind them to symbols without having to explicitly traverse the datastructure.

Let's look at an example of what exactly Destructuring means and how does it happen.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (def my-vector [1 2 3 4])
     (let [[a b c d] my-vector]
     (println a b c d)))
(Example)
```

The above program produces the following output.

```
1 2 3 4
```

In the above example, following things are to be noted:

- We are defining a vector of integers as 1 ,2, 3 and 4.
- We are then using the '**let**' statement to assign 4 variables (a, b, c, and d) to the my-vector variable directly.
- If we run the '**println**' statement on the four variables, we can see that they have already been assigned to the values in the vector respectively.

So clojure has destructured the my-vector variable which has four values when it was assigned using the 'let' statement. The deconstructed four values were then assigned to the four parameters accordingly.

If there are excess variables which don't have a corresponding value to which they can be assigned to, then they will be assigned the value of nil. The following example makes this point clear.

```
(ns clojure.examples.hello
     (:gen-class))
     (defn Example []
     (def my-vector [1 2 3 4])
     (let [[a b c d e] my-vector]
     (println a b c d e)))
(Example)
```

The above program produces the following output. You can see from the output that since the last variable 'e' does not have a corresponding value in the vector, it accounts to nil.

```
1 2 3 4 nil
```

the-rest

The 'the-rest' variable is used to store the remaining values, which cannot get assigned to any variable.

An example on how this is used is shown in the following program.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (def my-vector [1 2 3 4])
     (let [[a b & the-rest] my-vector]
     (println a b the-rest)))
(Example)
```

The above program produces the following output. From the output, you can clearly see that the values of 3 and 4 cannot be assigned to any variable so they are assigned to the 'the-rest' variable.

```
1 2 (3 4)
```

Destructuring Maps

Just like vectors, maps can also be destructured. Following is an example of how this can be accomplished.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (def my-map {"a" 1 "b" 2})
     (let [{a "a" b "b"} my-map]
     (println a b))) (Example)
```

The above program produces the following output. From the program you can clearly see that the map values of "a" and "b" are assigned to the variables of a and b.

```
1 2
```

Similarly in the case of vectors, if there is no corresponding value in the map when the destructuring happens, then the variable will be assigned a value of nil.

Following is an example.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (def my-map {"a" 1 "b" 2})
     (let [{a "a" b "b" c "c"} my-map]
     (println a b c)))
(Example)
```

The above program produces the following output.

```
1 2 nil
```

25. Clojure - Date & Time

Since the Clojure framework is derived from Java classes, one can use the date-time classes available in Java in Clojure. The **class date** represents a specific instant in time, with millisecond precision.

Following are the methods available for the date-time class.

java.util.Date

This is used to create the date object in Clojure.

Following is the syntax.

```
java.util.Date.
```

Parameters – None.

Return Value - Allocates a Date object and initializes it so that it represents the time at which it was allocated, measured to the nearest millisecond.

An example on how this is used is shown in the following program.

```
(ns example)
     (defn Example []
     (def date (.toString (java.util.Date.)))
     (println date))
(Example)
```

The above program produces the following output. This will depend on the current date and time on the system, on which the program is being run.

```
Sat Feb 06 21:22:05 UTC 2016
```

java.text.SimpleDateFormat

This is used to format the date output

Following is the syntax.

```
(java.text.SimpleDateFormat. format dt)
```

Parameters – 'format' is the format to be used when formatting the date. 'dt' is the date which needs to be formatted.

Return Value - A formatted date output.

An example on how this is used is shown in the following program.

```
(ns example)
     (defn Example []
     (def date (.format (java.text.SimpleDateFormat. "MM/dd/yyyy") (new
java.util.Date)))
     (println date))
(Example)
```

The above program produces the following output. This will depend on the current date and time on the system, on which the program is being run.

```
02/06/2016
```

getTime

Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by this Date object.

Following is the syntax.

```
(.getTime)
```

Parameters – None.

Return Value - The number of milliseconds since January 1, 1970, 00:00:00 GMT represented by this date.

An example on how this is used is shown in the following program.

```
(ns example)
(import java.util.Date)
     (defn Example []
     (def date (.getTime (java.util.Date.)))
     (println date))
(Example)
```

The above program produces the following output. This will depend on the current date and time on the system, on which the program is being run.

```
1454794676995
```

26. Clojure - Atoms

Atoms are a data type in Clojure that provide a way to manage shared, synchronous, independent state. An atom is just like any reference type in any other programming language. The primary use of an atom is to hold Clojure's immutable data structures. The value held by an atom is changed with the **swap! method**.

Internally, swap! reads the current value, applies the function to it, and attempts to compare-and-set it in. Since another thread may have changed the value in the intervening time, it may have to retry, and does so in a spin loop. The net effect is that the value will always be the result of the application of the supplied function to a current value, atomically.

Atoms are created with the help of the atom method. An example on the same is shown in the following program.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(def myatom (atom 1))
(println @myatom))
(example)
```

The above program produces the following result.

```
1
```

The value of atom is accessed by using the @ symbol. Clojure has a few operations that can be performed on atoms. Following are the operations.

reset!

Sets the value of atom to a new value without regard for the current value.

Following is the syntax.

```
(reset! atom-name newvalue)
```

Parameters – 'atom-name' is the name of the atom whose value needs to be reset. 'newvalue' is the new value, which needs to be assigned to the atom.

Return Value - The atom with the new value set.

An example on how this is used is shown in the following program.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(def myatom (atom 1))
(println @myatom)
(reset! myatom 2)
(println @myatom))
(example)
```

The above program produces the following output.

```
1
2
```

compare-and-set!

Atomically sets the value of atom to the new value if and only if the current value of the atom is identical to the old value held by the atom. Returns true if set happens, else it returns false.

Following is the syntax.

```
(compare-and-set! atom-name oldvalue newvalue)
```

Parameters – 'atom-name' is the name of the atom whose value needs to be reset. 'oldvalue' is the current old value of the atom. 'newvalue' is the new value which needs to be assigned to the atom.

Return Value - The atom with the new value will be set only if the old value is specified properly.

An example on how this is used is shown in the following program.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(def myatom (atom 1))
(println @myatom)
(compare-and-set! myatom 0 3)
(println @myatom)
(compare-and-set! myatom 1 3)
(println @myatom)
)
(example)
```

The above program will produce the following output.

```
1
1
3
```

swap!

Atomically swaps the value of the atom with a new one based on a particular function.

Following is the syntax.

```
(swap! atom-name function)
```

Parameters – 'atom-name' is the name of the atom whose value needs to be reset. 'function' is the function which is used to generate the new value of the atom.

Return Value - The atom with the new value will be set based on the function provided.

An example on how this is used is shown in the following program.

```
(ns clojure.examples.example
     (:gen-class))
(defn example []
(def myatom (atom 1))
(println @myatom)
(swap! myatom inc)
(println @myatom)
)
(example)
```

The above program produces the following output.

```
1
2
```

From the above program you can see that the **'inc'** (Increment function) is used to increment the value of the atom and with the help of the swap! function, the new value is automatically associated with the atom.

27. Clojure - Metadata

In Clojure, **metadata** is used to annotate the data in a collection or for the data stored in a symbol. This is normally used to annotate data about types to the underlying compiler, but can also be used for developers. Metadata is not considered as part of the value of the object. At the same time, metadata is immutable.

The following operations are possible in Clojure with regards to metadata.

meta-with

This function is used to define a metadata map for any object.

Following is the syntax.

```
(with-meta obj mapentry)
```

Parameters – 'obj' is the object with which metadata needs to be associated with. 'mapentry' is the metadata which needs to be associated with the object.

Return Value - Returns an object of the same type and value as obj, with mapentry as its metadata.

An example on how this is used is shown in the following program.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (def my-map (with-meta [1 2 3] {:prop "values"}))
        (println (meta my-map)))
(Example)
```

The above program produces the following output.

```
{:prop values}
```

meta

This function is used to see if any metadata is associated with an object.

Following is the syntax.

```
(meta obj)
```

Parameters – 'obj' is the object which needs to be checked if any metadata is associated with it.

Return Value - Returns the metadata of obj, returns nil if there is no metadata.

An example on how this is used is shown in the following program.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (def my-map (with-meta [1 2 3] {:prop "values"}))
        (println (meta my-map)))
(Example)
```

The above program produces the following output.

```
{:prop values}
```

vary-meta

Returns an object of the same type and value as the original object, but with a combined metadata.

Following is the syntax.

```
(vary-meta obj new-meta)
```

Parameters – 'obj' is the object which needs to be checked if any metadata is associated with it. 'new-meta' is the metadata values which needs to be associated with the object.

Return Value - Returns an object of the same type and value as the original object, but with a combined metadata.

An example on how this is used is shown in the following program.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (def my-map (with-meta [1 2 3] {:prop "values"}))
        (println (meta my-map))
        (def new-map (vary-meta my-map assoc :newprop "new values"))
        (println (meta new-map)))
(Example)
```

The above program produces the following output.

```
{:prop values}
{:prop values, :newprop new values}
```

28. Clojure - StructMaps

StructMaps are used for creating structures in Clojure. For example, if you wanted to create a structure which comprised of an Employee Name and Employeeid, you can do that with StructMaps.

The following operations are possible in Clojure with regards to StructMaps.

defstruct

This function is used for defining the structure which is required.

Following is the syntax.

```
(defstruct structname keys)
```

Parameters – 'structname' is the name to be given to the structure. 'keys' is the keys which needs to be a part of the structure.

Return Value - Returns a structure object.

An example on how this is used is shown in the following program.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (println (defstruct Employee :EmployeeName :Employeeid)))
(Example)
```

Note that the above function is only used to create your structure, we will see more functions which can be used to work with structures.

The above program produces the following output.

```
#'clojure.examples.example/Employee
```

struct

This function is used to define a structure object of the type, which is created by the defstruct operation.

Following is the syntax.

```
(struct structname values)
```

Parameters – 'structname' is the name to be given to the structure. 'values' is the values which needs to be assigned to the key values of the structure.

Return Value - Returns a struct object with the values mapped to the keys of the structure.

An example on how this is used is shown in the following program.

```
 (ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (defstruct Employee :EmployeeName :Employeeid)
     (def emp (struct Employee "John" 1))
     (println emp))
(Example)
```

The above program produces the following output.

```
{:EmployeeName John, :Employeeid 1}
```

It can be clearly seen that the values supplied in the struct function was assigned to the keys for the Employee object.

struct-map

This function is used to specifically assign values to key values by explicitly defining which values get assigned to which keys in the structure.

Following is the syntax.

```
(struct-map structname keyn valuen …. )
```

Parameters – 'structname' is the name to be given to the structure. 'keyn and valuen' are the key values which needs to be assigned to the structure.

Return Value - Returns a struct object with the values mapped to the keys of the structure.

An example on how this is used is shown in the following program.

```
 (ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (defstruct Employee :EmployeeName :Employeeid)
     (def emp (struct-map Employee :EmployeeName "John" :Employeeid 1))
     (println emp))
(Example)
```

The above program produces the following output.

```
{:EmployeeName John, :Employeeid 1}
```

Accessing Individual Fields

Individual fields of the structure can be accessed by accessing the keys along with the structure object.

Following is the syntax.

```
:key structure-name
```

Parameters – 'key' is the keyvalue in the structure. 'structure-name' is the structure which is the respective key.

Return Value - The value associated with the key will be returned.

An example on how this is used is shown in the following program.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (defstruct Employee :EmployeeName :Employeeid)
     (def emp (struct-map Employee :EmployeeName "John" :Employeeid 1))
(println (:Employeeid emp))
     (println (:EmployeeName emp)))
(Example)
```

The above program produces the following output.

```
1
John
```

Immutable Nature

By default structures are also immutable, so if we try to change the value of a particular key, it will not change.

An example of how this happens is shown in the following program.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (defstruct Employee :EmployeeName :Employeeid)
     (def emp (struct-map Employee :EmployeeName "John" :Employeeid
1))
     (println (:EmployeeName emp))
     (assoc emp :EmployeeName "Mark")
     (println (:EmployeeName emp)))
(Example)
```

In the above example, we try to use the '**assoc'** function to associate a new value for the Employee Name in the structure.

The above program produces the following output.

```
John
John
```

This clearly shows that the structure is immutable. The only way to change the value is to create a new variable with the changed value as shown in the following program.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (defstruct Employee :EmployeeName :Employeeid)
     (def emp (struct-map Employee :EmployeeName "John" :Employeeid 1))
     (def newemp (assoc emp :EmployeeName "Mark"))
     (println newemp))
(Example)
```

The above program produces the following output.

```
{:EmployeeName Mark, :Employeeid 1}
```

Adding a New Key to the Structure

Since structures are immutable, the only way that another key can be added to the structure is via the creation of a new structure. An example on how this can be achieved is shown in the following program.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (defstruct Employee :EmployeeName :Employeeid)
     (def emp (struct-map Employee :EmployeeName "John" :Employeeid 1))
     (def newemp (assoc emp :EmployeeRank "A"))
     (println newemp))
(Example)
```

In the above example, we associate a new key called EmployeeRank to the structure, but by creating a new structure object.

The above program produces the following output.

```
{:EmployeeName John, :Employeeid 1, :EmployeeRank A}
```

29. Clojure - Agents

As pointed out many times, Clojure is a programming language wherein many of the data types are immutable, which means that the only way one can change the value of a variable is to create a new variable and assign the new value to it. However, Clojure does provide some elements, which can create an mutable state. We have seen that this can be achieved with the atom data type. The other way this can be achieved is via Agents.

Agents provide independent, asynchronous change of individual locations. Agents are bound to a single storage location for their lifetime, and only allow mutation of that location (to a new state) to occur as a result of an action. Actions are functions (with, optionally, additional arguments) that are asynchronously applied to an Agent's state and whose return value becomes the Agent's new state.

The following operations are possible in Clojure with regards to Agents.

agent

An agent is created by using the agent command.

Following is the syntax.

```
(agent state)
```

Parameters – 'state' is the initial state that should be assigned to the agent.

Return Value - Returns an agent object with a current state and value.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (def counter (agent 0))
     (println counter))
(Example)
```

The above program produces the following output.

```
#object[clojure.lang.Agent 0x371c02e5 {:status :ready, :val 0}]
```

Just like the atom data type, you can see that the agent also has a status and a value associated with it. To access the value of the agent directly you need to use the @symbol along with the variable name.

An example on how this is used is shown in the following program.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (def counter (agent 0))
     (println @counter))
(Example)
```

The above program produces the following output.

```
0
```

You can clearly see from the above program that if you have append the @ symbol like @counter, you will get access to the value of the agent variable.

send

This function is used to send across a value to the agent.

Following is the syntax.

```
(send agentname function value)
```

Parameters – 'agentname' is the agent to which the send function is being redirected to. The 'function' is used to determine which way the value of the agent will be changed. In our case, we will use the addition + symbol to add a value to the existing value of the agent. 'Value' is the value passed to the function, which in turn will be used to update the value of the agent accordingly.

Return Value - Returns an agent object with a new value.

An example on how this is used is shown in the following program.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (def counter (agent 0))
     (println @counter)
     (send counter + 100)
     (println "Incrementing Counter")
     (println @counter))
(Example)
```

The above program produces the following output.

```
0

Incrementing Counter
0
```

Please note the following about the above program.

- Since the send function is an asynchronous function, there is a time delay for when the value of the agent is updated. This is why we have added an extra 'println' statement to the program. This is to give the Clojure environment the time required to update the agent value accordingly.
- Secondly, when you run the above program, the program will not terminate immediately. This is because the Clojure environment does not know whether it is safe to shut down the agent. We will see how to shut down agents in the next function description.

shutdown-agents

This function is used to shut down any running agents.

Following is the syntax.

```
(shutdown-agents)
```

Parameters – None.

Return Value – None.

An example on how this is used is shown in the following program.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (def counter (agent 0))
     (println @counter)
     (send counter + 100)
     (println "Incrementing Counter")
     (println @counter)
     (shutdown-agents))
(Example)
```

The above program produces the following output.

```
0
Incrementing Counter
0
```

The key difference in the above program is that, the program will now terminate since all agents will shut down properly.

send-off

There are instances wherein an agent is assigned a function which is blocking in nature. A simple example is, consider you are reading contents from a file which itself is blocking in nature. So the send-off function will first immediately return the agent and continue with the file operation. When the file operation completes, it will automatically update the agent with the contents of the file.

Following is the syntax.

```
(send agentname function value)
```

Parameters – 'agentname' is the agent to which the send function is being redirected to. The 'function' is used to determine which way the value of the agent will be changed. In our case, we will use the addition + symbol to add a value to the existing value of the agent. 'Value' is the value passed to the function which in turn will be used to update the value of the agent accordingly.

Return Value - First returns the agent as it is, if there is a non-blocking function. In the end, returns an agent object with a new value.

An example on how this is used is shown in the following program.

```
(ns clojure.examples.example
      (:gen-class))
      (defn Example []
      (def counter (agent 0))
      (println @counter)
      (send-off counter + 100)
      (println @counter)
      (println @counter))
(Example)
```

We are looking at the same example of incrementing the value of the counter, but from the following output it will be clear what the send-off function does.

The above program produces the following output.

```
0
0
0
```

It can be seen that even though we have sent the agent a function to set the value to 100, it does not reflect immediately. The send-off function first returns the value of the agent as it is. Once the value of the agent has been set properly by Clojure, the value of the agent is then updated and we are able to see the new value of the agent.

await-for

Since there is a delay when a value of an agent is updated, Clojure provided a 'await-for' function which is used to specify time in milliseconds to wait for the agent to be updated.

Following is the syntax.

```
(await-for time agentname)
```

Parameters – 'agentname' is the agent for which the 'await-for' function should be set to. 'time' is the time in milliseconds to wait.

Return Value - Returns logical false if returning due to timeout, otherwise returns logical true.

An example on how this is used is shown in the following program.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (def counter (agent 0))
     (println @counter)
     (send-off counter + 100)
     (println (await-for 100 counter))
     (println @counter)
     (shutdown-agents))
(Example)
```

The above program produces the following output.

```
0
true
100
```

You can see from the above program that the value of the agent is printed to the screen immediately because the 'await-for' function incorporated a delay, which allowed Clojure to update the value of the agent.

await

Blocks the current thread (indefinitely!) until all actions dispatched thus far, from this thread or agent, to the agent(s) have occurred. Will block on failed agents.

Following is the syntax.

```
(await agentname)
```

Parameters – 'agentname' is the agent for which the await function should be set to.

Return Value – None.

An example on how this is used is shown in the following program.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (def counter (agent 0))
     (println @counter)
```

```
      (send-off counter + 100)
      (await counter)
      (println @counter)
      (shutdown-agents))
(Example)
```

The above program produces the following output.

```
0
100
```

You can see from the above program that the value of the agent is printed to the screen immediately because the await function will wait for Clojure to first update the value of the function and only then will return control to the calling program.

agent-error

Returns the exception thrown during an asynchronous action of the agent, if the agent fails. Returns nil if the agent does not fail.

Following is the syntax.

```
(agent-error agentname)
```

Parameters – 'agentname' is the agent for which the agent-error function should be set to.

Return Value - Returns the exception thrown during an asynchronous action of the agent if the agent fails. Returns nil if the agent does not fail.

An example on how this is used is shown in the following program.

```
(ns clojure.examples.example
      (:gen-class))
      (defn Example []
      (def my-date (agent(java.util.Date.)))
      (send my-date + 100)
      (await-for 100 my-date)
      (println (agent-error my-date))
      )
(Example)
```

In the above program we are forcefully causing an exception to occur by incrementing the value of a date variable which is wrong. This will cause an exception and with the help of the 'prinltn' statement, will be sent to the screen.

The above program produces the following output.

```
Exception in thread "main" java.lang.RuntimeException: Agent is failed, needs
start, compiling:(C:\Users\Administrator\demonew\src\demonew\main.clj:9:1)
        at clojure.lang.Compiler.load(Compiler.java:7239)
        at clojure.lang.Compiler.loadFile(Compiler.java:7165)
        at clojure.main$load_script.invoke(main.clj:275)
        at clojure.main$init_opt.invoke(main.clj:280)
        at clojure.main$initialize.invoke(main.clj:308)
        at clojure.main$null_opt.invoke(main.clj:343)
        at clojure.main$main.doInvoke(main.clj:421)
        at clojure.lang.RestFn.invoke(RestFn.java:421)
        at clojure.lang.Var.invoke(Var.java:383)
        at clojure.lang.AFn.applyToHelper(AFn.java:156)
        at clojure.lang.Var.applyTo(Var.java:700)
        at clojure.main.main(main.java:37)
Caused by: java.lang.RuntimeException: Agent is failed, needs restart
        at clojure.lang.Util.runtimeException(Util.java:225)
        at clojure.lang.Agent.dispatch(Agent.java:238)
        at clojure.core$send_via.doInvoke(core.clj:1995)
        at clojure.lang.RestFn.invoke(RestFn.java:445)
        at clojure.lang.AFn.applyToHelper(AFn.java:160)
        at clojure.lang.RestFn.applyTo(RestFn.java:132)
        at clojure.core$apply.invoke(core.clj:636)
        at clojure.core$send.doInvoke(core.clj:2006)
        at clojure.lang.RestFn.invoke(RestFn.java:425)
        at clojure.core$await_for.doInvoke(core.clj:3177)
        at clojure.lang.RestFn.invoke(RestFn.java:423)
        at clojure.examples.example$Example.invoke(main.clj:6)
        at clojure.examples.example$eval12.invoke(main.clj:9)
        at clojure.lang.Compiler.eval(Compiler.java:6782)
        at clojure.lang.Compiler.load(Compiler.java:7227)
        ... 11 more
```

30. Clojure –Watchers

Watchers are functions added to variable types such as atoms and reference variables which get invoked when a value of the variable type changes. For example, if the calling program changes the value of an atom variable, and if a watcher function is attached to the atom variable, the function will be invoked as soon as the value of the atom is changed.

The following functions are available in Clojure for Watchers.

add-watch

Adds a watch function to an agent/atom/var/ref reference. The watch **'fn'** must be a 'fn' of 4 args: a key, the reference, its old-state, its new-state. Whenever the reference's state might have been changed, any registered watches will have their functions called.

Following is the syntax.

```
(add-watch variable :watcher
       (fn [key variable-type old-state new-state])
```

Parameters – 'variable' is the name of the atom or reference variable. 'variable-type' is the type of variable, either atom or reference variable. 'old-state & new-state' are parameters that will automatically hold the old and new value of the variable. 'key' must be unique per reference, and can be used to remove the watch with remove-watch.

Return Value – None.

An example on how this is used is shown in the following program.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (def x (atom 0))
     (add-watch x :watcher
       (fn [key atom old-state new-state]
         (println "The value of the atom has been changed")
         (println "old-state" old-state)
         (println "new-state" new-state)))
     (reset! x 2)
     )
(Example)
```

The above program produces the following output.

```
The value of the atom has been changed
old-state 0
new-state 2
```

remove-watch

Removes a watch which has been attached to a reference variable.

Following is the syntax.

```
(remove-watch variable watchname)
```

Parameters – 'variable' is the name of the atom or reference variable. 'watchname' is the name given to the watch when the watch function is defined.

Return Value – None.

An example on how this is used is shown in the following program.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (def x (atom 0))
     (add-watch x :watcher
       (fn [key atom old-state new-state]
         (println "The value of the atom has been changed")
         (println "old-state" old-state)
         (println "new-state" new-state)))
     (reset! x 2)
     (remove-watch x :watcher)
     (reset! x 4) )
(Example)
```

The above program produces the following output.

```
The value of the atom has been changed
old-state 0
new-state 2
```

You can clearly see from the above program that the second reset command does not trigger the watcher since it was removed from the watcher's list.

31. Clojure - Macros

In any language, **Macros** are used to generate inline code. Clojure is no exception and provides simple macro facilities for developers. Macros are used to write code-generation routines, which provide the developer a powerful way to tailor the language to the needs of the developer.

Following are the methods available for Macros.

defmacro

This function is used to define your macro. The macro will have a macro name, a parameter list and the body of the macro.

Following is the syntax.

```
(defmacro name [params*] body)
```

Parameters – 'name' is the name of the macro. 'params' are the parameters assigned to the macro. 'body' is the body of the macro.

Return Value – None.

An example on how this is used is shown in the following program.

```
(ns clojure.examples.example
      (:gen-class))
      (defn Example []
      (defmacro Simple []
(println "Hello"))
      (macroexpand '(Simple))
      )
(Example)
```

The above program produces the following output.

```
Hello
```

From the above program you can see that the macro 'Simple' is expanded inline to 'println' "Hello". Macros are similar to functions, with the only difference that the arguments to a form are evaluated in the case of macros.

macro-expand

This is used to expand a macro and place the code inline in the program.

Following is the syntax.

```
(macroexpand macroname)
```

Parameters – ‘macroname’ is the name of the macro which needs to be expanded.

Return Value - The expanded macro.

An example on how this is used is shown in the following program.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (defmacro Simple []
(println "Hello"))
     (macroexpand '(Simple))
     )
(Example)
```

The above program produces the following output.

```
Hello
```

Macro with Arguments

Macros can also be used to take in arguments. The macro can take in any number of arguments. Following example showcases how arguments can be used.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (defmacro Simple [arg]
(list 2 arg))
     (println (macroexpand '(Simple 2)))
     )(Example)
```

The above example places an argument in the Simple macro and then uses the argument to add argument value to a list.

The above program produces the following output.

```
(2 2)
```

32. Clojure - Reference Values

Reference values are another way Clojure can work with the demand to have mutable variables. Clojure provides mutable data types such as atoms, agents, and reference types. They mostly all work in the same way. This chapter focuses on how reference types can be used.

Following are the operations available for reference values.

ref

This is used to create a reference value. When creating a reference value, there is an option to provide a validator function, which will validate the value created.

Following is the syntax.

```
(ref x options)
```

Parameters – 'x' is the value which needs to be provided to the reference. 'Options' is a set of options that can be provided, such as the validate command.

Return Value - The reference and its corresponding value.

An example on how this is used is shown in the following program.

```
(ns clojure.examples.example
      (:gen-class))
      (defn Example []
      (def my-ref (ref 1 :validator pos?))
      (println @my-ref)
      )
(Example)
```

To access the value of the reference value, you can use the @ symbol.

The above program produces the following output.

```
1
```

ref-set

This function is used to set the value of a reference to a new value irrespective of whatever is the older value.

Following is the syntax.

```
(ref-set refname newvalue)
```

Parameters – 'refname' is the name of the variable holding the reference value. 'newvalue' is the new value that needs to be associated with the reference type.

Return Value - The reference and its corresponding new value.

An example on how this is used is shown in the following program.

```
(ns clojure.examples.example
      (:gen-class))
      (defn Example []
      (def my-ref (ref 1 :validator pos?))
      (dosync
          (ref-set my-ref 2))
      (println @my-ref)
      )
(Example)
```

The above program produces the following output.

```
2
```

alter

This function is used to alter the value of a reference type but in a safe manner. This is run in a thread, which cannot be accessed by another process. This is why the command needs to be associated with a '**dosync'** method always. Secondly, to change the value of a reference type, a function needs to be called to make the necessary change to the value.

Following is the syntax.

```
(alter refname fun)
```

Parameters – 'refname' is the name of the variable holding the reference value. 'fun' is the function which is used to change the value of the reference type.

Return Value - The reference and its corresponding new value.

An example on how this is used is shown in the following program.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (def names (ref []))
     (defn change [newname]
       (dosync
         (alter names conj newname)))
     (change "John")
     (change "Mark")
     (println @names)
     )
(Example)
```

The above program produces the following output.

```
[John Mark]
```

dosync

Runs the expression (in an implicit do) in a transaction that encompasses expression and any nested calls. Starts a transaction if none is already running on this thread. Any uncaught exception will abort the transaction and flow out of dosync.

Following is the syntax.

```
(dosync expression)
```

Parameters – 'expression' is the set of expressions, which will come in the dosync block.

Return Value – None.

An example on how this is used is shown in the following program.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (def names (ref []))
     (defn change [newname]
       (dosync
         (alter names conj newname)))
     (change "John")
     (change "Mark")
     (println @names)
     )
(Example)
```

The above program produces the following output.

```
[John Mark]
```

commute

Commute is also used to change the value of a reference type just like alter and ref-set. The only difference is that this also needs to be placed inside a 'dosync' block. However, this can be used whenever there is no need to know which calling process actually changed the value of the reference type. Commute also uses a function to change the value of the reference variable.

Following is the syntax.

```
(commute refname fun)
```

Parameters – 'refname' is the name of the variable holding the reference value. 'fun' is the function which is used to change the value of the reference type.

Return Value - The reference and its corresponding new value.

An example on how this is used is shown in the following program.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (def counter (ref 0))
     (defn change [counter]
       (dosync
         (commute counter inc)))
     (change counter)
     (println @counter)
     (change counter)
     (println @counter)
     )
(Example)
```

The above program produces the following output.

```
1
2
```

33. Clojure - Databases

In order to use the database functionality, please ensure to first download the **jdbc files** from the following url - https://codeload.github.com/clojure/java.jdbc/zip/master

You will find a zip file which has the necessary drivers for Clojure to have the ability to connect to databases. Once the zip file is extracted, ensure to add the unzipped location to your classpath.

The main file for database connectivity is a file called **jdbc.clj** in the location clojure/java.

The clojure jdbc connector supports a wide variety of databases, some of which are the following.

- H2Database
- Oracle
- Microsoft SQL Server
- MySQL
- PostgreSQL

In our example, we are going to use MySQL DB as an example.

The following operations are possible in Clojure with regards to Databases.

Database Connection

Before connecting to a MySQL database, make sure of the following:

- You have created a database TESTDB.
- You have created a table EMPLOYEE in TESTDB.
- This table has fields FIRST_NAME, LAST_NAME, AGE, SEX and INCOME.
- User ID "testuser" and password "test123" are set to access TESTDB.
- Ensure you have downloaded the 'mysql jar file' and added the file to your classpath.
- You have gone through MySQL tutorial to understand MySQL Basics

Following is the syntax to create a connection in Clojure.

```
(def connection_name
{ :subprotocol “protocol_name”
:subname “Location of mysql DB”
:user “username” :password “password” })
```

Parameters – 'connection_name' is the name to be given to the connection. 'subprotocol' is the protocol to be used for the connection. By default we will be using the mysql protocol. 'subname' is the url to connect to the mysql db along with the database name. 'user' is the username used to connect to the database. 'password' is the password to be used to connect to the database.

Return Value - This will provide a connection string, which can be used in subsequent mysql operations.

The following example shows how to connect to the tables in the information schema and retrieve all the data in the table.

```
(ns test.core
(:require [clojure.java.jdbc :as sql]))
     (defn -main []
(def mysql-db {:subprotocol "mysql"
               :subname "//127.0.0.1:3306/information_schema"
               :user "root"
               :password "shakinstev"})
(println (sql/query mysql-db
  ["select table_name from tables"]
:row-fn :table_name)
))
```

Querying Data

Querying data on any database means to fetch some useful information from the database. Once a database connection is established, you are ready to make a query into this database. Following is the syntax by which data can be queried using Clojure.

```
clojure.java.jdbc/query dbconn
  ["query"]
:row-fn :sequence
```

Parameters – 'dbconn' is the name of the connection used to connect to the database. 'query' is the query string used to fetch data from the database. ':sequence' is by default all the rows of data fetched from the database and is returned as a sequence. The necessary operations on the sequence can then be done to see what data has been fetched.

Return Value - This will return a sequence, which will have the rows of data from the query operation.

The following example shows how to connect to the employee table and fetch the first_name column of the rows in the table.

```
(ns test.core
(:require [clojure.java.jdbc :as sql]))
     (defn -main []
(def mysql-db {:subprotocol "mysql"
               :subname "//127.0.0.1:3306/testdb"
               :user "root"
               :password "shakinstev"})
(println (sql/query mysql-db
  ["select first_name from employee"]
:row-fn :first_name)
))
```

From the above code, we can see that:

- The query of "select first_name from employee" is passed as the query string.
- The :first_name is the sequence, which is returned as a result of the fetch operation.

If we assume that there is just one row in our database which contains a first_name value of John, following will be the output of the above program.

```
(John)
```

Inserting Data

It is required when you want to create your records into a database table. Following is the syntax by which data can be inserted using Clojure. This is done by using the '**insert!**' function.

```
clojure.java.jdbc/insert!
 :table_name {:column_namen columnvalue}
```

Parameters – ':table_name' is the name of the table in which the insertion needs to be made. '{:column_namen columnvalue }' is a map of all the column names and values, which need to be added as a row in the table.

Return Value - This will return nil if the insertion is made successfully.

The following example shows how to insert a record into the employee table in the testdb database.

```
(ns test.core
(:require [clojure.java.jdbc :as sql]))
     (defn -main []
(def mysql-db {:subprotocol "mysql"
              :subname "//127.0.0.1:3306/testdb"
              :user "root"
              :password "shakinstev"})
(sql/insert! mysql-db
  :employee {:first_name "John" :last_name "Mark" :sex "M" :age 30 :income 30})
)
```

If you now check your MySQL database and the employee table, you will see that the above row will be successfully inserted in the table.

Deleting Data

Rows can be deleted from a table by using the '**delete!'** function. Following is the syntax on how this operation can be performed.

```
clojure.java.jdbc/delete!
 :table_name [condition]
```

Parameters – ':table_name' is the name of the table in which the insertion needs to be made. 'condition' is the condition used to determine which row needs to be deleted from the table.

Return Value - This will return the number of rows deleted.

The following example shows how to delete a record from the employee table in the testdb database. The example deletes a row from the table based on the condition that the age is equal to 30.

```
(ns test.core
(:require [clojure.java.jdbc :as sql]))
     (defn -main []
(def mysql-db {:subprotocol "mysql"
              :subname "//127.0.0.1:3306/testdb"
              :user "root"
              :password "shakinstev"})
(println (sql/delete! mysql-db
  :employee ["age = ? " 30])
))
```

If you had a record which had a row with age equal to the value of 30, that row will be deleted.

Updating Data

Rows can be updated from a table by using the '**update!**' function. Following is the syntax on how this operation can be performed.

```
clojure.java.jdbc/update!
 :table_name
{setcondition}
[condition]
```

Parameters – ':table_name' is the name of the table in which the insertion needs to be made. 'setcondition' is the column which needs to be updated as mentioned in terms of a map. 'condition' is the condition which is used to determine which row needs to be deleted from the table.

Return Value - This will return the number of rows updated.

The following example shows how to delete a record from the employee table in the testdb database. The example updates a row from the table based on the condition that the age is equal to 30 and updates the value of income to 40.

```
(ns test.core
(:require [clojure.java.jdbc :as sql]))
     (defn -main []
(def mysql-db {:subprotocol "mysql"
               :subname "//127.0.0.1:3306/testdb"
               :user "root"
               :password "shakinstev"})
(println (sql/update! mysql-db
  :employee
{:income 40}
["age = ? " 30])
))
```

If you had a record which had a row with age equal to the value of 30, that row will be updated wherein the value of income will be set to 40.

Transactions

Transactions are mechanisms that ensure data consistency. Transactions have the following four properties:

- **Atomicity**: Either a transaction completes or nothing happens at all.
- **Consistency**: A transaction must start in a consistent state and leave the system in a consistent state.
- **Isolation**: Intermediate results of a transaction are not visible outside the current transaction.
- **Durability**: Once a transaction was committed, the effects are persistent, even after a system failure.

The following example shows how to implement transactions in Clojure. Any operations which needs to be performed in a transaction needs to be embedded in the '**with-db-transaction**' clause.

```
(ns test.core
(:require [clojure.java.jdbc :as sql]))
     (defn -main []
(def mysql-db {:subprotocol "mysql"
               :subname "//127.0.0.1:3306/testdb"
               :user "root"
               :password "shakinstev"})
(sql/with-db-transaction [t-con mysql-db]
(sql/update! t-con
  :employee
{:income 40}
["age = ? " 30]))
)
```

34. Clojure - Java Interface

As we already know, Clojure code runs on the Java virtual environment at the end. Thus it only makes sense that Clojure is able to utilize all of the functionalities from Java. In this chapter, let's discuss the correlation between Clojure and Java.

Calling Java Methods

Java methods can be called by using the dot notation. An example is strings. Since all strings in Clojure are anyway Java strings, you can call normal Java methods on strings.

An example on how this is done is shown in the following program.

```
 (ns Project
     (:gen-class))
(defn Example []
(println (.toUpperCase "Hello World")))
(Example)
```

The above program produces the following output. You can see from the code that if you just call the dot notation for any string method, it will also work in Clojure.

```
HELLO WORLD
```

Calling Java Methods with Parameters

You can also call Java methods with parameters. An example on how this is done is shown in the following program.

```
 (ns Project
     (:gen-class))
(defn Example []
(println (.indexOf "Hello World","e"))
)
(Example)
```

The above program produces the following output. You can see from the above code, that we are passing the parameter "e" to the indexOf method. The above program produces the following output.

```
1
```

Creating Java Objects

Objects can be created in Clojure by using the 'new' keyword similar to what is done in Java.

An example on how this is done is shown in the following program.

```
 (ns Project
     (:gen-class))
(defn Example []
(def str1 (new String "Hello"))
(println str1)
)
(Example)
```

The above program produces the following output. You can see from the above code, that we can use the 'new' keyword to create a new object from the existing String class from Java. We can pass the value while creating the object, just like we do in Java. The above program produces the following output.

```
Hello
```

Following is another example which shows how we can create an object of the Integer class and use them in the normal Clojure commands.

```
 (ns Project
     (:gen-class))
(defn Example []
(def my-int(new Integer 1))
(println (+ 2 my-int))
)
(Example)
```

The above program produces the following output.

```
3
```

Import Command

We can also use the import command to include Java libraries in the namespace so that the classes and methods can be accessed easily.

The following example shows how we can use the import command. In the example we are using the import command to import the classes from the **java.util.stack** library. We can then use the push and pop method of the stack class as they are.

```
(ns Project
     (:gen-class))
(import java.util.Stack)
(defn Example []
(let [stack (Stack.)]
  (.push stack "First Element")
  (.push stack "Second Element")
(println (first stack))))
(Example)
```

The above program produces the following output.

```
First Element
```

Running Code Using the Java Command

Clojure code can be run using the Java command. Following is the syntax of how this can be done.

```
java -jar clojure-1.2.0.jar -i main.clj
```

You have to mention the Clojure jar file, so that all Clojure-based classes will be loaded in the JVM. The 'main.clj' file is the Clojure code file which needs to be executed.

Java Built-in Functions

Clojure can use many of the built-in functions of Java. Some of them are:

Math PI function - Clojure can use the Math method to the value of PI. Following is an example code.

```
(ns Project
     (:gen-class))
(defn Example []
(println (. Math PI))
)
(Example)
```

The above code produces the following output.

```
3.141592653589793
```

System Properties - Clojure can also query the system properties. Following is an example code.

```
(ns Project
     (:gen-class))
(defn Example []
(println (.. System getProperties (get "java.version")))
)
(Example)
```

Depending on the version of Java on the system, the corresponding value will be displayed. Following is an example output.

```
1.7.0_79
```

35. Clojure - Concurrent Programming

In Clojure programming most data types are immutable, thus when it comes to concurrent programming, the code using these data types are pretty safe when the code runs on multiple processors. But many a times, there is a requirement to share data, and when it comes to shared data across multiple processors, it becomes necessary to ensure that the state of the data is maintained in terms of integrity when working with multiple processors. This is known as **concurrent programming** and Clojure provides support for such programming.

The software transactional memory system (STM), exposed through dosync, ref, set, alter, etc. supports sharing changing state between threads in a synchronous and coordinated manner. The agent system supports sharing changing state between threads in an asynchronous and independent manner. The atoms system supports sharing changing state between threads in a synchronous and independent manner. Whereas the dynamic var system, exposed through def, binding, etc. supports isolating changing state within threads.

Other programming languages also follow the model for concurrent programming.

- They have a direct reference to the data which can be changed.
- If shared access is required, the object is locked, the value is changed, and the process continues for the next access to that value.

In Clojure there are no locks, but Indirect references to immutable persistent data structures.

There are three types of references in Clojure.

- **Vars** – Changes are isolated in threads.
- **Refs** – Changes are synchronized and coordinated between threads.
- **Agents** – Involves asynchronous independent changes between threads.

The following operations are possible in Clojure with regards to concurrent programming.

Transactions

Concurrency in Clojure is based on transactions. References can only be changed within a transaction. Following rules are applied in transactions.

- All changes are atomic and isolated.
- Every change to a reference happens in a transaction.
- No transaction sees the effect made by another transaction.
- All transactions are placed inside of dosync block.

We already seen what the dosync block does, let's look at it again.

dosync

Runs the expression (in an implicit do) in a transaction that encompasses expression and any nested calls. Starts a transaction if none is already running on this thread. Any uncaught exception will abort the transaction and flow out of dosync.

Following is the syntax.

```
(dosync expression)
```

Parameters – 'expression' is the set of expressions which will come in the dosync block.

Return Value – None.

Let's look at an example wherein we try to change the value of a reference variable.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (def names (ref []))
     (alter names conj "Mark")
     )
(Example)
```

The above program produces the following error.

```
Caused by: java.lang.IllegalStateException: No transaction running
        at clojure.lang.LockingTransaction.getEx(LockingTransaction.java:208)
        at clojure.lang.Ref.alter(Ref.java:173)
        at clojure.core$alter.doInvoke(core.clj:1866)
        at clojure.lang.RestFn.invoke(RestFn.java:443)
        at clojure.examples.example$Example.invoke(main.clj:5)
        at clojure.examples.example$eval8.invoke(main.clj:7)
        at clojure.lang.Compiler.eval(Compiler.java:5424)
        ... 12 more
```

From the error you can clearly see that you cannot change the value of a reference type without first initiating a transaction.

In order for the above code to work, we have to place the alter command in a dosync block as done in the following program.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
     (def names (ref []))
     (defn change [newname]
       (dosync
         (alter names conj newname)))
     (change "John")
     (change "Mark")
     (println @names)
     )
(Example)
```

The above program produces the following output.

```
[John Mark]
```

Let’s see another example of dosync.

```
(ns clojure.examples.example
     (:gen-class))
     (defn Example []
(def var1 (ref 10))
(def var2 (ref 20))
(println @var1 @var2)

(defn change-value [var1 var2 newvalue]
  (dosync
    (alter var1 - newvalue)
    (alter var2 + newvalue)
    ))
(change-value var1 var2 20)
(println @var1 @var2)
     )
(Example)
```

In the above example, we have two values which are being changed in a dosync block. If the transaction is successful, both values will change else the whole transaction will fail.

The above program produces the following output.

```
10 20
-10 40
```

36. Clojure - Applications

Clojure has some contributed libraries which have the enablement for creating **Desktop** and **Web-based applications**. Let's discuss each one of them.

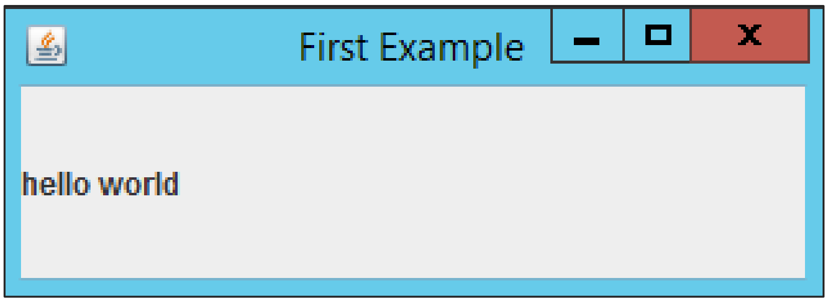
Desktop – See-saw

See-saw is a library which can be used for creating desktop applications. In order to use See-saw, first download the **.clj** file from the following github link https://github.com/daveray/seesaw

Then create a sample desktop application. Following is the code for the same.

```
(ns web.core
 (:gen-class)
 (:require [seesaw.core :as seesaw]))
(def window (seesaw/frame
 :title "First Example"
 :content "hello world"
 :width 200
 :height 50))
(defn -main
 [& args]
 (seesaw/show! window))
```

When the above code is run, you will get the following window.



The code is pretty self-explanatory.

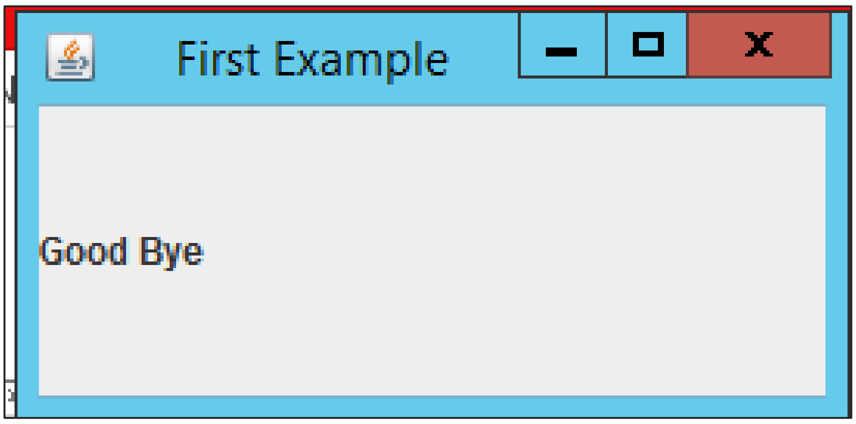
- First you need to ensure you use the **seesaw.core** library so all of the available methods can be used.
- The attributes of frame and content can be used to define the title and what content needs to be shown in the window.
- Finally the '**show!**' function is used to show the window.

Desktop – Changing the Value of Text

The value of the content in the window can be changed by using the '**config!**' option. In the following example the config! option is used to change the window content to the new value of "Good Bye".

```
(ns web.core
 (:gen-class)
 (:require [seesaw.core :as seesaw]))
(def window (seesaw/frame
 :title "First Example"
 :content "hello world"
 :width 200
 :height 50))
(defn -main
 [& args]
 (seesaw/show! window)
(seesaw/config! window :content "Good Bye"))
```

When the above code is run, you will get the following window.



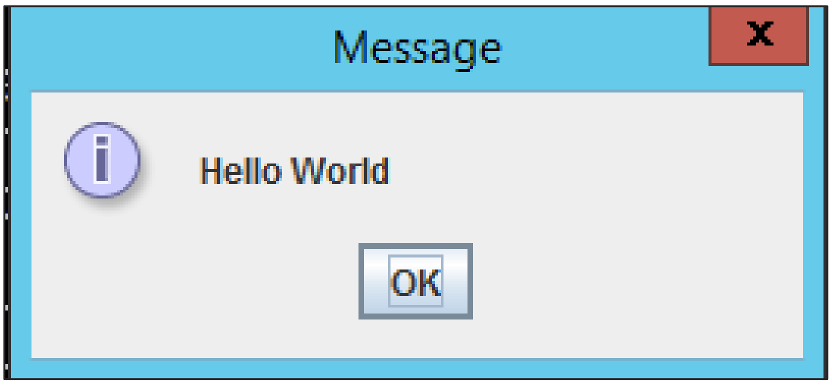
Desktop – Displaying a Modal Dialog Box

A modal dialog box can be shown by using the alert method of the see-saw class. The method takes the text value, which needs to be shown in the modal dialog box.

An example on how this is used is shown in the following program.

```
(ns web.core
 (:gen-class)
 (:require [seesaw.core :as seesaw]))
(def window (seesaw/frame
 :title "First Example"
 :content "hello world"
 :width 200
 :height 50))
(defn -main
 [& args]
 (seesaw/show! window)
(seesaw/alert "Hello World"))
```

When the above code is run, you will get the following window.



Hello World

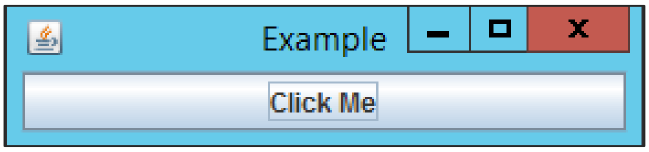
Desktop – Displaying Buttons

Buttons can be displayed with the help of the button class. An example on how this is used is shown in the following program.

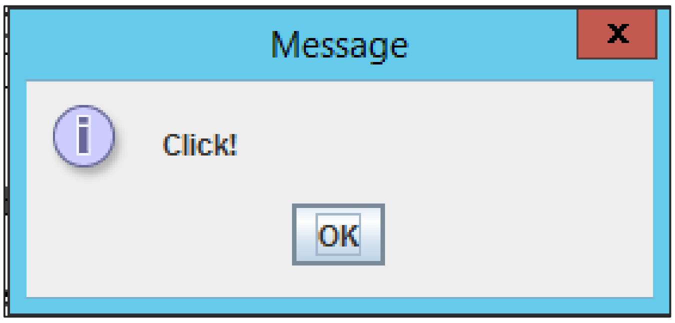
```
(ns web.core
 (:gen-class)
 (:require [seesaw.core :as seesaw]))
(defn -main  [& args]
(defn display
 [content]
 (let [window (seesaw/frame :title "Example")]
 (-> window
 (seesaw/config! :content content)
 (seesaw/pack!)
 (seesaw/show!))))
(def button
 (seesaw/button
 :text "Click Me"
 :listen [:action (fn [event](seesaw/alert "Click!" ))]))
(display button))
```

In the above code, first a button variable is created which is from the button class of the see-saw library. Next, the text of the button is set to “Click Me”. Then an event is attached to the button so that whenever the button is clicked, it will show an alert dialog box.

When the above code is run, you will get the following window.



When you click the button, you will get the following dialog box.



Click!

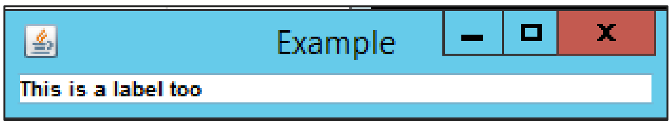
Desktop – Displaying Labels

Labels can be displayed with the help of the label class. An example on how this is used is shown in the following program.

```
(ns web.core
 (:gen-class)
 (:require [seesaw.core :as seesaw]))
(defn -main  [& args]
(defn display
 [content]
 (let [window (seesaw/frame :title "Example")]
 (-> window
 (seesaw/config! :content content)
 (seesaw/pack!)
 (seesaw/show!))))
(def label (seesaw/label
 :text "This is a label too"
 :background :white
 :foreground :black
 :font "ARIAL-BOLD-10"))
(display label))
```

In the above code, first a label variable is created which is from the label class of the seesaw library. Next, the text of the label is set to "This is a label too". Then, the background, foreground color, and font are set accordingly.

When the above code is run, you will get the following window.



This is a label too

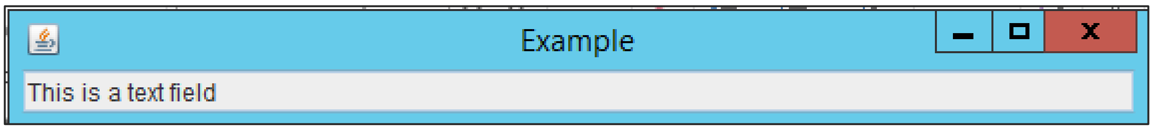
Desktop – Displaying Text Fields

Text Fields can be displayed with the help of the text class. An example on how this is used is shown in the following program.

```
(ns web.core
 (:gen-class)
 (:require [seesaw.core :as seesaw]))
(defn -main  [& args]
(defn display
 [content]
 (let [window (seesaw/frame :title "Example")]
 (-> window
 (seesaw/config! :content content)
 (seesaw/pack!)
 (seesaw/show!))))
(def textfield
 (seesaw/text
 :text "This is a text field"
 :editable? false
 :columns 50))
(display textfield))
```

In the above code, first a text field variable is created which is from the text class of the seesaw library. Next, the text of the text field is set to "This is a text field". Then the text field is made a static field by setting the editable attribute to false.

When the above code is run, you will get the following window.



This is a text field

Web Applications - Introduction

To create a web application in Clojure you need to use the Ring application library, which is available at the following link https://github.com/ring-clojure/ring

You need to ensure you download the necessary jars from the site and ensure to add it as a dependency for the Clojure application.

The **Ring framework** provides the following capabilities:

- Sets things up such that an http request comes into your web application as a regular Clojure HashMap, and likewise makes it so that you can return a response as a HashMap.
- Provides a specification describing exactly what those request and response maps should look like.
- Brings along a web server (Jetty) and connects your web application to it.

The Ring framework automatically can start a web server and ensures the Clojure application works on this server. Then one can also use the **Compojure framework**. This allows one to create routes which is now how most modern web applications are developed.

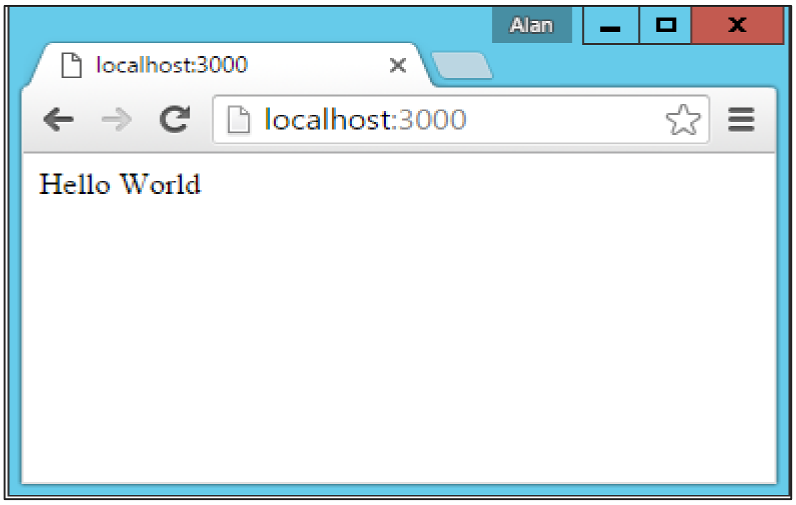
Creating your first Clojure application - The following example shows how you can create your first web application in Clojure.

```
(ns my-webapp.handler
  (:require [compojure.core :refer :all]
            [compojure.route :as route]
            [ring.middleware.defaults :refer [wrap-defaults site-defaults]]))
(defroutes app-routes
  (GET "/" [] "Hello World")
  (route/not-found "Not Found"))
(def app
  (wrap-defaults app-routes site-defaults))
```

Let’s look at the following aspects of the program:

- The ‘**defroutes**’ is used to create routes so that request made to the web application to different routes can be directed to different functions in your Clojure application.
- In the above example, the “/” is known as the default route, so when you browse to the base of your web application, the string “Hello World” will be sent to the web browser.
- If the user hits any url which cannot be processed by the Clojure application, then it will display the string “Not Found”.

When you run the Clojure application, by default your application will be loaded as localhost:3000, so if you browse to this location, you will receive the following output.



Hello World

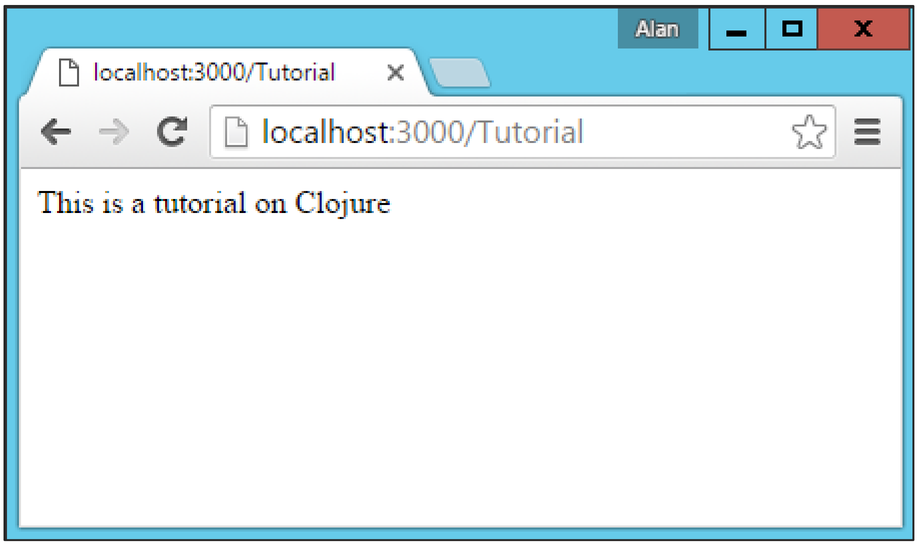
Web Applications – Adding More Routes to Your Web Application

You can also add more routes to your web application. The following example shows how to achieve this.

```
(ns my-webapp.handler
  (:require [compojure.core :refer :all]
            [compojure.route :as route]
            [ring.middleware.defaults :refer [wrap-defaults site-defaults]]))
(defroutes app-routes
  (GET "/" [] "Hello World")
  (GET "/Tutorial" [] "This is a tutorial on Clojure")
  (route/not-found "Not Found"))
(def app
  (wrap-defaults app-routes site-defaults))
```

You can see that adding a route in the application is as easy as just adding another GET function with the url route. (GET "/Tutorial" [] "This is a tutorial on Clojure")

If you browse to the location http://localhost:3000/Tutorial, you will receive the following output.



This is a tutorial on Clojure

37. Clojure - Automated Testing

In this chapter, let's discuss automated testing options provided by Clojure.

Testing for Client Applications

In order to use testing for Clojure framework, you have to use the dependencies located at https://github.com/slagyr/speclj#manual-installation

This URL provides the **speclj** framework, which is used as a Test data driven or Behaviour driven test framework for Clojure. You have to ensure that you use the Clojure 1.7.0 framework when using any of the 'speclj' libraries. By default, the test files will be different from the Clojure code files and need to be placed in a 'spec' directory.

Following is a sample code for a test file.

```
(ns change.core-spec
 (:require [speclj.core :refer :all]
            ))

(describe "Truth"

  (it "is true"
    (should true))

  (it "is not false"
    (should-not false)))

(run-specs)
```

Following things need to be noted about the above code:

- We first have to ensure to use the 'require' statement to include all the core libraries in the 'speclj' framework.
- Next is the 'describe' function. This is used to provide a description for the test case being created.
- Next function is the 'it' function, which is the actual test case. In the first test case, the "is true" string is the name given to the test case.

- Should and should-not are known as **assertions**. All assertions begin with should. Should and should-not are just two of the many assertions available. They both take expressions that they will check for truthy-ness and falsy-ness respectively.

If you run the test case, you will get the following output. The output shows the time taken in milliseconds for the test case to run.

```
←[32m.←[0m←[32m.←[0m


Finished in 0.00014 seconds
```

Testing for Web-based Applications

Selenium is one of the key frameworks used for testing modern day web-based applications. Clojure libraries are also available which can be used for testing web-based applications.

Let's look at how we can use the Selenium libraries for testing Clojure web-based applications.

Step 1: The first step is to ensure we are using the Ring and Compojure framework to create a web-based application, which needs to be tested. Let's use one of the examples from our earlier chapters. The following code is a simple web application, which displays "Hello World" in the browser.

```
(ns my-webapp.handler
  (:require [compojure.core :refer :all]
            [compojure.route :as route]
            [ring.middleware.defaults :refer [wrap-defaults site-defaults]]))

(defroutes app-routes
  (GET "/" [] "Hello World")
  (route/not-found "Not Found"))

(def app
  (wrap-defaults app-routes site-defaults))
```

Step 2: Next make sure to download the selenium jar file http://mvnrepository.com/artifact/org.seleniumhq.selenium/selenium-server/2.47.0 and include it in your classpath.

Step 3: Also ensure to download the 'clj' web driver, which will be used for running the web test from the following location.

https://clojars.org/clj-webdriver/versions/0.7.1

Step 4: In your project directory, create another directory called features and create a file called 'config.clj'.

Step 5: Next add the following code to the 'config.clj' file created in the earlier step.

```
ns clj-webdriver-tutorial.features.config)
(def test-port 3000)
(def test-host "localhost")
(def test-base-url (str "http://" test-host ":" test-port "/"))
```

The above code basically tells the web test framework to test the application, which gets loaded at the URL http://localhost:3000

Step 6: Finally, let's write our code to carry out our test.

```
(ns clj-webdriver-tutorial.features.homepage
  (:require [clojure.test :refer :all]
            [ring.adapter.jetty :refer [run-jetty]]
            [clj-webdriver.taxi :refer :all]
            [clj-webdriver-tutorial.features.config :refer :all]
            [clj-webdriver-tutorial.handler :refer [app-routes]]))

(ns clj-webdriver-tutorial.features.homepage
  (:require [clojure.test :refer :all]
            [ring.adapter.jetty :refer [run-jetty]]
            [clj-webdriver.taxi :refer :all]
            [clj-webdriver-tutorial.features.config :refer :all]
            [clj-webdriver-tutorial.handler :refer [app-routes]]))

(defn start-server []
  (loop [server (run-jetty app-routes {:port test-port, :join? false})]
    (if (.isStarted server)
      server
      (recur server))))

(defn stop-server [server]
  (.stop server))

(defn start-browser []
  (set-driver! {:browser :firefox}))
```

```
(defn stop-browser []
  (quit))

(deftest homepage-greeting
  (let [server (start-server)]
    (start-browser)
    (to test-base-url)
    (is (= (text "body") "Hello World"))
    (stop-browser)
    (stop-server server)))
```

The above code is going to take the following actions:

- Start the server for the application.
- Open the root path in the browser.
- Check if the "Hello World" message is present on the page.
- Close the browser.
- Shut down the server.

38. Clojure - Libraries

One thing which makes the Clojure library so powerful is the number of libraries available for the Clojure framework. We have already seen so many libraries used in our earlier examples for web testing, web development, developing swing-based applications, the jdbc library for connecting to MySQL databases. Following are just a couple of examples of few more libraries.

data.xml

This library allows Clojure to work with XML data. The library version to be used is org.clojure/data.xml "0.0.8". The data.xml supports parsing and emitting XML. The parsing functions will read XML from a Reader or InputStream.

Following is an example of the data processing from a string to XML.

```
(ns clojure.examples.example
(use 'clojure.data.xml)
     (:gen-class))
     (defn Example []
     (let [input-xml (java.io.StringReader. "<?xml version=\"1.0\"
encoding=\"UTF-8\"?><example><clo><Tutorial>The Tutorial
value</Tutorial></clo></example>")]
  (parse input-xml))
#clojure.data.xml.Element{:tag :example, :attrs {},
:content (#clojure.data.xml.Element{:tag :clo,:attrs {},
:content (#clojure.data.xml.Element{:tag :Tutorial,
:attrs {},:content ("The Tutorial value")})})}
(Example)
```

data.json

This library allows Clojure to work with JSON data. The library version to be used is org.clojure/data.json "0.2.6".

Following is an example on the use of this library.

```
(ns clojure.examples.example
(:require [clojure.data.json :as json])
     (:gen-class))
     (defn Example []
(println (json/write-str {:a 1 :b 2})) (Example)
```

The above program produces the following output.

```
{\"a\":1,\"b\":2}
```

data.csv

This library allows Clojure to work with **'csv'** data. The library version to be used is org.clojure/data.csv "0.1.3".

Following is an example on the use of this library.

```
(ns clojure.examples.example
(require '[clojure.data.csv :as csv]
         '[clojure.java.io :as io])
     (:gen-class))
     (defn Example []
(with-open [in-file (io/reader "in-file.csv")]
  (doall
    (csv/read-csv in-file)))
(with-open [out-file (io/writer "out-file.csv")]
  (csv/write-csv out-file
                 [[":A" "a"]
                  [":B" "b"]]))
(Example)
```

In the above code, the 'csv' function will first read a file called **in-file.csv** and put all the data in the variable in-file. Next, we are using the write-csv function to write all data to a file called **out-file.csv**.